In [125]:

```
import cv2
import numpy as np
import random
from math import hypot
```

1 Calibration

The goal is to compute the projection matrix that goes from world 3D coordinates to 2D image coordinates.

Recall you solve for the 3x4 matrix M using either SVD to solve the homogeneous version of the equations or by setting m4,4 to 1 and then using the a normal least squares method. Remember that M is only known up to a scale factor.

1.1 Create the least squares function that will solve for the 3x4 matrix MnormA given the normalized 2D and 3D lists, namely ./pts2d-norm-pic_a.txt and ./pts3d-norm.txt. Test it on the normalized 3D points by multiplying those points by your M matrix and comparing the resulting the normalized 2D points to the normalized 2D points given in the file. Remember to divide by the homogeneous value to get an inhomogeneous point. You can do the comparison by checking the residual between the predicted location of each test point using your equation and the actual location given by the 2D input data. The residual is just the distance (square root of the sum of squared differences in u and v).

Output: code that does the solving, the matrix M you recovered from the normalized points, the < u, v > projection of the last point given your M matrix, and the residual between that projected location and the actual one given in the file.

In [2]:

```
f3d = open("./pts3d-norm.txt","r")
f2d = open("./pts2d-norm-pic_a.txt", "r")
lines3d = f3d.readlines()
lines2d = f2d.readlines()
f3d.close()
f2d.close()
n = len(lines3d)
A = np.zeros((2*n, 12))
for idx in range(n):
    X, Y, Z = list(map(float, lines3d[idx].split()))
    u, v = list(map(float, lines2d[idx].split()))
    A[idx*2] = [X, Y, Z, 1, 0, 0, 0, 0, -1*u*X, -1*u*Y, -1*u*Z, -1*u]
    A[idx*2+1] = [0, 0, 0, 0, X, Y, Z, 1, -1*v*X, -1*v*Y, -1*v*Z, -1*v]
ATA = np.transpose(A).dot(A)
V, D2, VT = np.linalg.svd(ATA)
M = V[:,-1]
M = np.reshape(M, (3, 4))
print(M)
```

```
[[ 0.45827554 -0.29474237 -0.01395746  0.0040258 ]
 [-0.05085589 -0.0545847  -0.54105993 -0.05237592]
 [ 0.10900958  0.17834548 -0.04426782  0.5968205 ]]
```

In [3]:

```
last_p3d = np.hstack((list(map(float, lines3d[-1].split())), [1]))
last_p2d = np.array(list(map(float, lines2d[-1].split())))
u, v, w = M.dot(last_p3d)
residual = (u/w - last_p2d[0]) ** 2 + (v/w - last_p2d[1]) ** 2
print(residual)
```

```
2.440269026888714e-06
```

1.2 For the three point set sizes k of 8, 12, and 16, repeat 10 times:

1. Randomly choose k points from the 2D list and their corresponding points in the 3D list.
2. Compute the projection matrix M on the chosen points.
3. Pick 4 points not in your set of k and compute the average residual.
4. Save the M that gives the lowest residual.

Output: code that does the computation, and the average residual for each trial of each k (so that would be 10 x 3 = 30 numbers). Explain any difference you see between the results for the different k.

Output: Best M

In [19]:

```
def calculate_M_res(K):
    if K > n - 4: return
    p_index = range(n)
    random_index = random.sample(p_index, K+4)
    A = np.zeros((2*K, 12))
    for idx in range(K):
        X, Y, Z = list(map(float, lines3d[random_index[idx]].split()))
        u, v = list(map(float, lines2d[random_index[idx]].split()))
        A[idx*2] = [X, Y, Z, 1, 0, 0, 0, 0, -1*u*X, -1*u*Y, -1*u*Z, -1*u]
        A[idx*2+1] = [0, 0, 0, 0, X, Y, Z, 1, -1*v*X, -1*v*Y, -1*v*Z, -1*v]
    ATA = np.transpose(A).dot(A)
    V, D2, VT = np.linalg.svd(ATA)
    M = V[:,-1]
    M = np.reshape(M, (3, 4))
    # print(M)

    residual = 0
    for idx in range(4):
        p3d = np.hstack((list(map(float, lines3d[random_index[-1*idx-1]].split
())), [1]))
        p2d = np.array(list(map(float, lines2d[random_index[-1*idx-1]].split
())))
        u, v, w = M.dot(p3d)
        residual += (u/w - p2d[0]) ** 2 + (v/w - p2d[1]) ** 2
    return residual, M
```

In [20]:

```
res_8p = []
res_12p = []
res_16p = []
min_err = np.inf

for i in range(10):
    res8, M8 = calculate_M_res(8)
    res12, M12 = calculate_M_res(12)
    res16, M16 = calculate_M_res(16)
    res_8p.append(res8)
    res_12p.append(res12)
    res_16p.append(res16)
    if res8 < min_err:
        min_err = res8
        Mbest = M8
        Kbest = 8
    if res12 < min_err:
        min_err = res12
        Mbest = M12
        Kbest = 12
    if res16 < min_err:
        min_err = res12
        Mbest = M16
        Kbest = 16

print("Residual of K=8\n", res_8p)
print("Residual of K=12\n", res_12p)
print("Residual of K=16\n", res_16p)
```

```
Residual of K=8
 [1.5403231279332638e-05, 0.00020829962735620134, 0.0001535689334013
0723, 0.006285376218659766, 0.0004683075867407076, 0.000115021122483
44843, 7.994084421543738e-05, 0.000757176598804571, 0.00018956300580
333466, 9.796841879938926e-05]
Residual of K=12
 [6.681075877420733e-05, 4.3287440824124105e-05, 0.00015214872458628
963, 6.099688170306107e-05, 5.7562396969582235e-05, 1.60168781132416
7e-05, 0.00016965963711785165, 5.372086604757769e-05, 0.000135718455
69749786, 0.00044668599260293935]
Residual of K=16
 [0.00014041446505455433, 1.9417953022738675e-05, 0.0002440152292446
4908, 9.046838501746815e-05, 0.00018395333085309722, 0.0002894927178
932297, 6.971714341831692e-05, 1.6988438581738974e-05, 6.38426565868
9622e-05, 6.123446179177929e-05]
```

In [21]:

```
print("Avg Residual of K=8", np.mean(res_8p))
print("Avg Residual of K=12", np.mean(res_12p))
print("Avg Residual of K=16", np.mean(res_16p))
```

```
Avg Residual of K=8 0.0008370625587543498
Avg Residual of K=12 0.00012026080324363726
Avg Residual of K=16 0.00011795447814644686
```

In [22]:

```
print("The best M is from", Kbest, "random points.", "\nThe best M is: \n", Mbes
t)
```

```
The best M is from 8 random points.
The best M is:
 [[ 0.45844672 -0.29434899 -0.01667359  0.00418657]
 [-0.05082033 -0.05488169 -0.54128986 -0.05233743]
 [ 0.10852825  0.17834604 -0.04448086  0.59665474]]
```

Answer:

With more points, we cannot see a significant improvement in the residual value. And the best M can come from a K=8 matrix. I think that 6 points are enough for solving this equation; the error is low if we have more than 6 points. And all the points have a slight difference after they are translated to numeric numbers. More points could not avoid this problem.

1.3 Given the best M from the last part, compute C.

Output: code that does the computation, and the location of the camera in real 3D world coordinates.

In [12]:

```
Q = M[:, 0:3]
print(Q)
```

```
[[ 0.45827554 -0.29474237 -0.01395746]
 [-0.05085589 -0.0545847  -0.54105993]
 [ 0.10900958  0.17834548 -0.04426782]]
```

In [13]:

```
C = (-1 * np.linalg.inv(Q)).dot(M[:, 3])
print(C)
```

```
[-1.51267725 -2.35168754  0.28262819]
```

2 Fundamental Matrix Estimation

We now wish to estimate the mapping of points in one image to lines in another by means of the fundamental matrix. This will require you to use similar methods to those in Problem 1. We will make use of the corresponding point locations listed in pts2d-pic_a.txt and pts2d-pic_b.txt.

2.1 Create the least squares function that will solve for the 3x3 matrix F˜ that satisfies the epipolar constraints defined by the sets of corresponding points. Solve this function to create your least squares estimate of the 3x3 transform F˜.

Output: code that does the solving. The matrix F˜ generated from your least squares function.

In [94]:

```
f2da = open("./pts2d-pic_a.txt","r")
f2db = open("./pts2d-pic_b.txt", "r")
lines2da = f2da.readlines()
lines2db = f2db.readlines()
f2da.close()
f2db.close()
n2d = len(lines2da)
AF = np.zeros((n2d, 9))
for idx in range(n2d):
    u, v = list(map(float, lines2da[idx].split()))
    up, vp = list(map(float, lines2db[idx].split()))
    AF[idx] = [up*u, up*v, up, vp*u, vp*v, vp, u, v, 1]
U, D, V = np.linalg.svd(AF)
F = V[-1,:]
F = np.reshape(F, (3, 3))
print(F)
```

```
[[-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]]
```

2.2 The linear squares estimate of F˜ is full rank; however, the fundamental matrix is a rank 2 matrix. As such we must reduce its rank. In order to do this we can decompose F˜ using singular value decomposition into the matrices UΣV T = F˜. We can then estimate a rank 2 matrix by setting the smallest singular value in Σ to zero thus generating Σ′. The fundamental matrix is then easily calculated as F = UΣ′V T . Use the SVD function to do, well, the SVD. Duh.

Output: Code and fundamental matrix F.

In [99]:

```
U, D, VT = np.linalg.svd(F)
D[-1] = 0
D = np.diag(D)
F_ = U.dot(D).dot(VT)
print(F_)
```

```
[[-5.36264198e-07  7.90364771e-06 -1.88600204e-03]
 [ 8.83539184e-06  1.21321685e-06  1.72332901e-02]
 [-9.07382264e-04 -2.64234650e-02  9.99500092e-01]]
```

2.3 Now you can use your matrix F to estimate an epipolar line lb in image 'b' corresponding to point pa in image 'a':

lb =Fpa

Similarly, epipolar lines in image a corresponding to points in image b are related by the transpose of F.

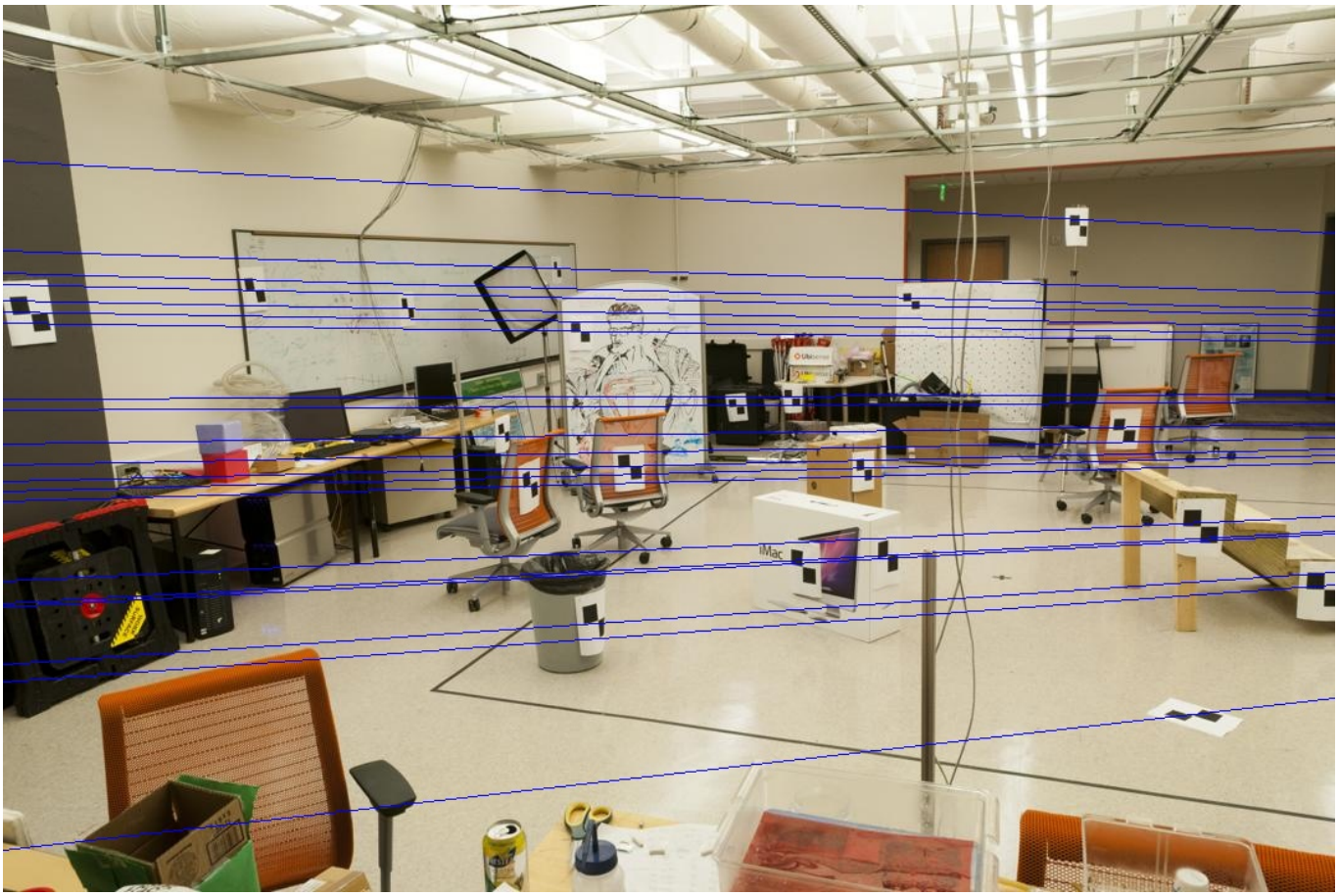
Output: Code to perform the estimation and line drawing. Images with the estimated epipolar lines drawn on them.

In [100]:

```
img_b = cv2.imread('pic_b.jpg')
hb, wb = img_b.shape[:2]
Ll = np.cross([0, 0, 1], [0, hb, 1])
Lr = np.cross([wb, 0, 1], [wb, hb, 1])
for idx in range(n2d):
    u, v = list(map(float, lines2da[idx].split()))
    lb = F_.dot([u, v, 1.0])
    Pl = np.cross(Ll, lb)
    Pl = list(map(int, Pl[:2]/Pl[2]))
    Pr = np.cross(Lr, lb)
    Pr = list(map(int, Pr[:2]/Pr[2]))
    img_b = cv2.line(img_b, (0, Pl[1]), (Pr[0], Pr[1]), (255, 0, 0), 1)
cv2.imwrite('ps3-2-3.jpg', img_b)
```

Out[100]:

```
True
```



2.4 Create a two matrics Ta and Tb for the set of points defined in the files ./pts2d-pic_a.txt and ./pts2d-pic_b.txt respectively. Use these matrices to transform the two sets of points. Then use these normalized points to create a new Fundamental matrix Fˆ. Compute it as above including making the smaller singular value zero.

Output: The matrixes Ta, Tb and Fˆ

In [159]:

```
points_a = np.zeros((n2d, 2))
points_b = np.zeros((n2d, 2))
for idx in range(n2d):
    ua, va = list(map(float, lines2da[idx].split()))
    ub, vb = list(map(float, lines2db[idx].split()))
    points_a[idx] = [ua, va]
    points_b[idx] = [ub, vb]

cua, cva = np.mean(points_a, axis=0)
dist = points_a - [cua, cva]
h_list = 0
for i in range(len(dist)):
    h = hypot(dist[i,0], dist[i,1])
    h_list += h

sa = h_list/n2d
scalea = np.diag([sa, sa, 1])
offseta = np.identity(3)
offseta[0,-1] = -1*cua
offseta[1,-1] = -1*cva
Ta = scalea.dot(offseta)
print(Ta)

cub, cvb = np.mean(points_b, axis=0)

distb = points_b - [cub, cvb]
hb_list = 0
for i in range(len(distb)):
    h = hypot(distb[i,0], distb[i,1])
    hb_list += h

sb = hb_list/n2d
scaleb = np.diag([sb, sb, 1])
offsetb = np.identity(3)
offsetb[0,-1] = -1*cub
offsetb[1,-1] = -1*cvb
Tb = scaleb.dot(offsetb)
print(Tb)
```

```
[[ 2.38345394e+02  0.00000000e+00 -1.33223158e+05]
 [ 0.00000000e+00  2.38345394e+02 -7.76052604e+04]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
[[ 2.34353982e+02  0.00000000e+00 -1.44526101e+05]
 [ 0.00000000e+00  2.34353982e+02 -8.12973965e+04]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

In [165]:

```
AF = np.zeros((n2d, 9))
for idx in range(n2d):
    ua, va = list(map(float, lines2da[idx].split()))
    nua, nva, w = Ta.dot([ua, va, 1])
    ub, vb = list(map(float, lines2db[idx].split()))
    nub, nvb, w = Tb.dot([ub, vb, 1])
    AF[idx] = [nub*nua, nub*nva, nub, nvb*nua, nvb*nva, nvb, nua, nva, 1]
U, D, V = np.linalg.svd(AF)
nF = V[-1,:]
nF = np.reshape(nF, (3, 3))

U, D, VT = np.linalg.svd(nF)
D[-1] = 0
D = np.diag(D)
nF = U.dot(D).dot(VT)
print(nF)

nF = np.transpose(Tb).dot(nF).dot(Ta)
print(nF)
```

```
[[-1.31693406e-10  1.72868520e-09  1.26619468e-05]
 [ 1.08510261e-09 -3.30616563e-10  9.35472674e-04]
 [ 7.19719801e-05 -8.65535308e-04 -9.99999185e-01]]
[[-7.35602391e-06  9.65595015e-05 -2.43607465e-02]
 [ 6.06107852e-05 -1.84673129e-05  1.91366305e-01]
 [ 6.64768548e-04 -2.59438288e-01  5.22006037e+00]]
```

2.5 Using the new F redraw the epipolar lines of 2.3. They should be better.

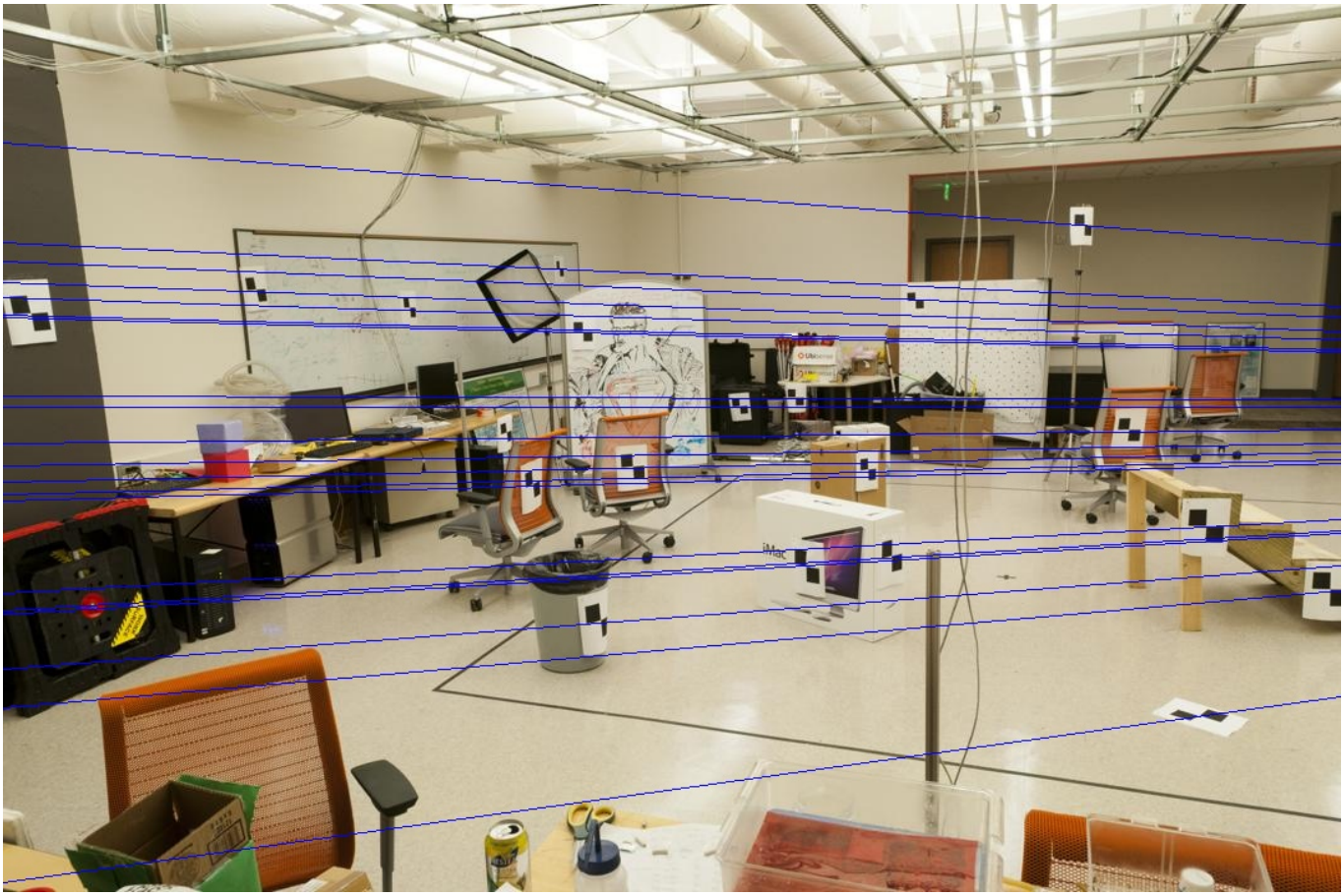
Output: The new F and the images with the "better" epipolar lines drawn.

In [167]:

```
img_b2 = cv2.imread('pic_b.jpg')
hb, wb = img_b.shape[:2]
Ll = np.cross([0, 0, 1], [0, hb, 1])
Lr = np.cross([wb, 0, 1], [wb, hb, 1])
for idx in range(n2d):
    u, v = list(map(float, lines2da[idx].split()))
    lb = nF.dot([u, v, 1.0])
    Pl = np.cross(Ll, lb)
    Pl = list(map(int, Pl[:2]/Pl[2]))
    Pr = np.cross(Lr, lb)
    Pr = list(map(int, Pr[:2]/Pr[2]))
    img_b2 = cv2.line(img_b2, (0, Pl[1]), (Pr[0], Pr[1]), (255, 0, 0), 1)
cv2.imwrite('ps3-2-5-b.jpg', img_b2)
```

Out[167]:

```
True
```



In [ ]: