Assignment 2: Multi-View Geometry

Comment: most of the written problems (except the first one) are designed to help with the coding part (structure-from-motion). Thus, they should be solved first. Your coding part require slicing, broadcasting, and other standard operations with numpy arrays. This could be tricky if you are new to numpy. I recommend working on small test cases. For debugging, print arrays before and after slicing (etc.) to verify that the result is correct.

Problem 1 (Frobenius norm)

Compute Frobenius norm of $n \times n$ matrix $xx^\top$ assuming that (Euclidean) norm $\|x\| = \sqrt{x^\top x}$ of vector $x \in R^n$ is given. That is, express $\|xx^\top\|_F$ in terms of $\|x\|$. The answer should not depend on n. Comment: this "random math" excerise is only meant to encourage you to look up the definition of Frobenius norm; the proof is only a couple of lines of simple algebra.

Solution: We have $(xx^T)^2_{ij} = (x_i x_j)^2 = x_i^2 x_j^2 \Rightarrow$

$$\|xx^\top\|_F = \sqrt{\sum_{i,j}((xx^T)_{i,j})^2} = \sqrt{\sum_{i,j} x_i^2 x_j^2}$$

$$= \sqrt{\sum_i x_i^2 \sum_j x_j^2} = \sqrt{(\sum_i x_i^2)^2} = |x^T x|$$

$$= \|x\|^2$$

Problem 2

Assuming a $calibrated$ camera (that is, $K = I$) and its two views corresponding to projection matrices $P_1 = [I|0]$ and $P_2 = [R|T]$ w.r.t. some world coordinate system, show formulas for coordinates of the following 3D points (in the same world coordinate system):

(a) optical center for the first view: $C_1 = (0, 0, 0)$

(b) image center for the first view: $Q_1 = (0, 0, 1)$

(c) optical center for the second view: $C_2 = -R^{-1}T$

(d) image center for the second view: $Q_2 = R^{-1}(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - T)$

Problem 3

Using the same set up as in problem 2, show formulas for normalized coordinates of the following image points:

(a) epipole in the first camera image: $e_1 = P_1 C_2 = [I|0](-R^{-1}T) = -R^{-1}T$

(b) epipole in the second camera image: $e_2 = P_2 C_1 = [R|T] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = T$

Problem 4 (homogeneous and non-homogeneous line representations)

Lines in 2D images can be represented "homogeneously" as 3-vectors $l = [l_1, l_2, l_3]^T$ that give equation $l^T x = 0$ for homogeneous points $x = [x_1, x_2, x_3]^T \in \mathcal{P}^2$ forming a line. Given l, what are the values of scalar parameters a, b in the line equation $v = au + b$ for

the same 2D points based on their regular (nonhomogeneous) representation $(u, v) = (\frac{x_1}{x_3}, \frac{x_2}{x_3})$ in $\mathcal{R}^2$?

$a = -\frac{l_1}{l_2}$

$b = -\frac{l_3}{l_2}$

Problem 5 (epipolar lines in normalized and non-normalized images)

Given a matrix of intrinsic camera parameters K and essential matrix E between two views (A) and (B) such that $x_A^T E x_B = 0$ for any corresponding points, write expressions for the following:

(a) given homogeneous normalized point x_B^n in image B, specify 3-vector l_A^n describing the corresponding epipolar line of normalized points in image A:

$l_A^n = E x_B^n$

(b) given homogeneous normalized point x_A^n in image A, specify 3-vector l_B^n describing the corresponding epipolar line of normalized points in image B:

$l_B^n = E^T x_A^n$

(c) assuming line (3-vector) l^n of normalized image points, what is a 3-vector representation l for the line formed by the corresponding points on the real (unnormalized) camera image:

$l = l^n K^{-1}$

Problem 6 (least squares for triangulation)

Describe your approach to triangulating two matched feature points $x_a = [u_a, v_a, 1]^T$ and $x_b = [u_b, v_b, 1]^T$ in two views with given projection matrices P_a and P_b. You should find 3D point $X = [X_1, X_2, X_3, 1]^T$ and two scalars w_a, w_b such that $P_a X \approx w_a x_a$ and $P_b X \approx w_b x_b$. Be specific as you will need this for your programming part below. Use notation $M[i]$ to denote the i-th row vector of matrix M.

You should use the first approach described for homograpy estimation in topic 6. In particualr, you can formulate the problem as $AX \approx 0$, define elements of $4x4$ matrix A, convert the problem to an overdetermined system of 4 linear equations $A_{1:3}[X_1, X_2, X_3]^T \approx -A_4$, and specify its solution minimizing the sum of squared errors.

Can you characterize geometrically the case when your solution satisfies $A_{1:3}[X_1, X_2, X_3]^T = -A_4$ exactly?

Solution:

$$A = \begin{bmatrix} P_a[1,1] - u_a P_a[3,1] & P_a[1,2] - u_a P_a[3,2] & P_a[1,3] - u_a P_a[3,3] & P_a[1,4] - u_a P_a[3,4] \\ P_a[2,1] - v_a P_a[3,1] & P_a[2,2] - v_a P_a[3,2] & P_a[2,3] - v_a P_a[3,3] & P_a[2,4] - v_a P_a[3,4] \\ P_b[1,1] - u_b P_b[3,1] & P_b[1,2] - u_b P_b[3,2] & P_b[1,3] - u_b P_b[3,3] & P_b[1,4] - u_b P_b[3,4] \\ P_b[2,1] - v_b P_b[3,1] & P_b[2,2] - v_a P_b[3,2] & P_b[2,3] - v_b P_b[3,3] & P_b[2,4] - v_b P_b[3,4] \end{bmatrix} \text{ So that } AX \approx 0$$

$$\Rightarrow A_{1:3}[X_1, X_2, X_3]^T = \begin{bmatrix} P_a[1,1] - u_a P_a[3,1] & P_a[1,2] - u_a P_a[3,2] & P_a[1,3] - u_a P_a[3,3] \\ P_a[2,1] - v_a P_a[3,1] & P_a[2,2] - v_a P_a[3,2] & P_a[2,3] - v_a P_a[3,3] \\ P_b[1,1] - u_b P_b[3,1] & P_b[1,2] - u_b P_b[3,2] & P_b[1,3] - u_b P_b[3,3] \\ P_b[2,1] - v_b P_b[3,1] & P_b[2,2] - v_a P_b[3,2] & P_b[2,3] - v_b P_b[3,3] \end{bmatrix} [X_1, X_2, X_3]^T \approx - \begin{bmatrix} P_a[1,4] - u_a \\ P_a[2,4] - v_a \\ P_b[1,4] - u_b \\ P_b[2,4] - v_b \end{bmatrix}$$

So the solution of $[X_1, X_2, X_3]^T$ would be generated by solving $\min \|A_{1:3} X - A_4\|^2 \Rightarrow X = (A_{1:3}^T \cdot A_{1:3})^{-1} \cdot A_{1:3}^T \cdot (-A_4)$

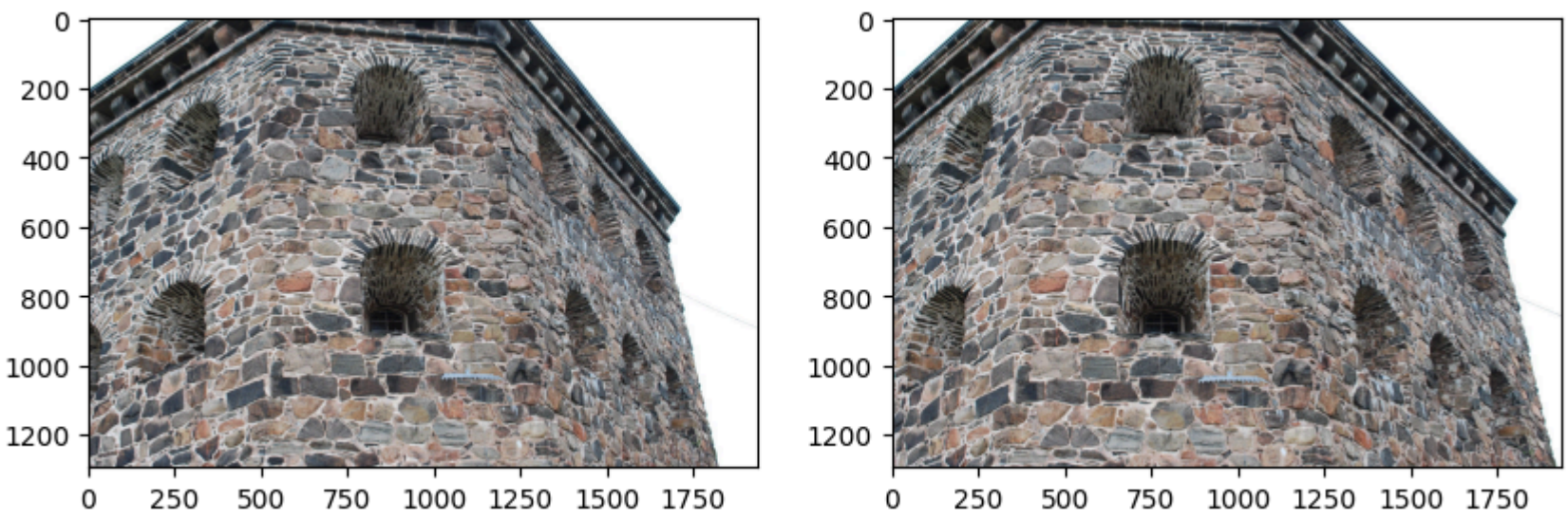
Probelm 7 (the programming part)

Structure from Motion

NOTE: Steps 0-3 and 10 are given, other steps are to be implemented.

Step 0: Loading two camera views and camera's intrinsic matrix K

In [1]:
```
import numpy as np
import numpy.linalg as la
import matplotlib
import matplotlib.image as image
import matplotlib.pyplot as plt
from skimage.feature import (corner_harris, corner_peaks, plot_matches, BRIEF, match_descriptors)
from skimage.transform import warp, ProjectiveTransform, EssentialMatrixTransform, FundamentalMatrixTransform
from skimage.color import rgb2gray
from skimage.measure import ransac

# Indicate (E) inlier matches in image 1 and image 2
# loading two images (two camera views) and the corresponding matrix K (intrinsic parameters)
imL = image.imread("images/kronan1.jpg")
imR = image.imread("images/kronan2.jpg")
imLgray = rgb2gray(imL)
imRgray = rgb2gray(imR)

K = 1.0e+03 * np.array([[2.3940, -0.0000,    0.9324],
                        [     0,  2.3981,    0.6283],
                        [     0,       0,    0.0010]])


plt.figure(0,figsize = (10, 4))
ax81 = plt.subplot(121)
plt.imshow(imL)
ax82 = plt.subplot(122)
plt.imshow(imR)
plt.show()
print(imR.shape)
```



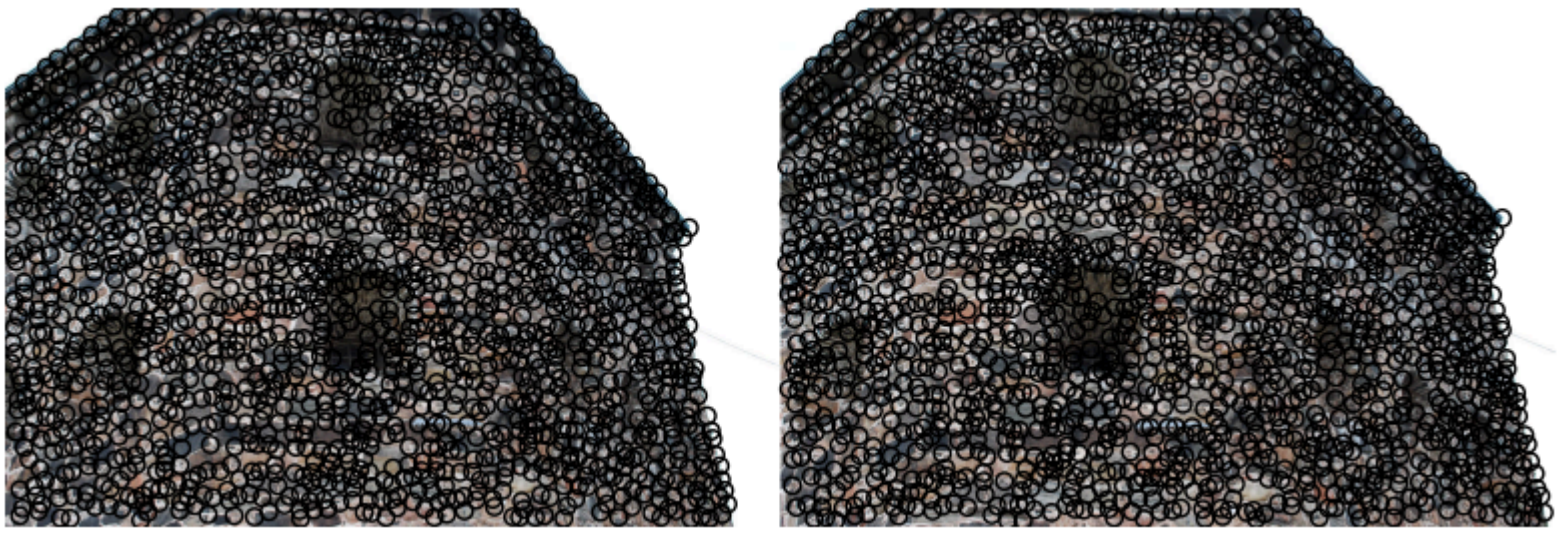

```
(1296, 1936, 3)
```

Step 1: Feature detection (e.g. corners)

In [2]:
```
# NOTE: corner_peaks and many other feature extraction functions return point coordinates as (y,x), that is (rows,cols)
keypointsL = corner_peaks(corner_harris(imLgray), threshold_rel=0.001, min_distance=15)
keypointsR = corner_peaks(corner_harris(imRgray), threshold_rel=0.001, min_distance=15)


print ('the number of features in images 1 and 2 are {:5d} and {:5d}'.format(keypointsL.shape[0],keypointsR.shape[0]))

fig = plt.figure(1,figsize = (10, 4))
axA = plt.subplot(111)
plt.gray()
matchesLR = np.empty((0,2))
plot_matches(axA, imL, imR, keypointsL, keypointsR, matchesLR)
axA.axis('off')

plt.show()
```

```
the number of features in images 1 and 2 are  1578 and  1660
```

```
C:\Users\韩佳芮\AppData\Local\Temp\ipykernel_13048\2030401058.py:12: FutureWarning: `plot_matches` is deprecated since version
0.23 and will be removed in version 0.25. Use `skimage.feature.plot_matched_features` instead.
  plot_matches(axA, imL, imR, keypointsL, keypointsR, matchesLR)
```

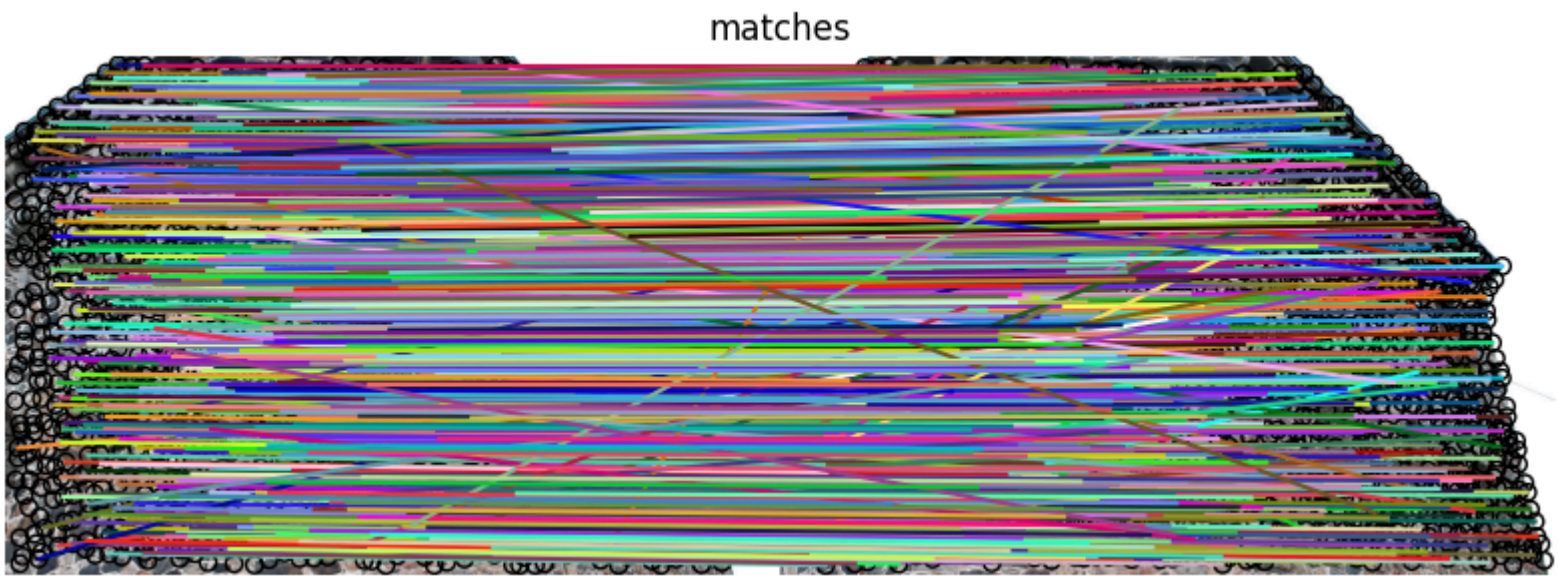


Step 2: Feature matching (e.g. BRIEF descriptor, a variant of SURF, SIFT, etc)

In [3]:
```
extractor = BRIEF()

extractor.extract(imLgray, keypointsL)
keypointsL = keypointsL[extractor.mask]
descriptorsL = extractor.descriptors

extractor.extract(imRgray, keypointsR)
keypointsR = keypointsR[extractor.mask]
descriptorsR = extractor.descriptors

matchesLR = match_descriptors(descriptorsL, descriptorsR, cross_check=True)

print ('the number of matches is {:2d}'.format(matchesLR.shape[0]))

fig = plt.figure(2,figsize = (10, 4))
axA = plt.subplot(111)
axA.set_title("matches")
plt.gray()
plot_matches(axA, imL, imR, keypointsL, keypointsR, matchesLR) #, matches_color = 'r')
axA.axis('off')

plt.show()
```

```
the number of matches is 965
```

```
C:\Users\韩佳芮\AppData\Local\Temp\ipykernel_13048\2891731649.py:19: FutureWarning: `plot_matches` is deprecated since version
0.23 and will be removed in version 0.25. Use `skimage.feature.plot_matched_features` instead.
  plot_matches(axA, imL, imR, keypointsL, keypointsR, matchesLR) #, matches_color = 'r')
```



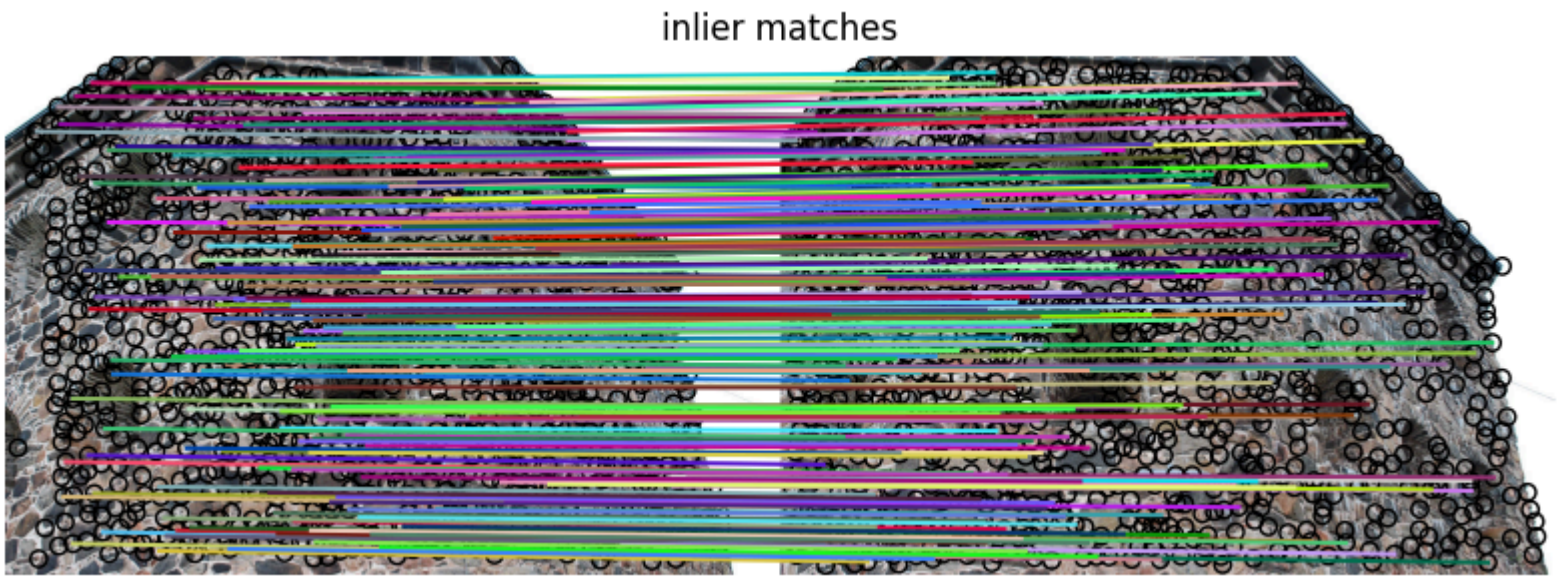
Step 3: Fundamental Matrix estimation using RANSAC

In [5]:
```
ptsL1 = []
ptsR1 = []
for i in matchesLR:
    ptsL1.append(keypointsL[i[0]])
    ptsR1.append(keypointsR[i[1]])
ptsL1 = np.array(ptsL1)
ptsR1 = np.array(ptsR1)

# swapping columns using advanced indexing https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing
# This changes point coordinates from (y,x) in ptsL1/ptsR1 to (x,y) in ptsL/ptsR
ptsL = ptsL1[:,[1, 0]]
ptsR = ptsR1[:,[1, 0]]

# robustly estimate fundamental matrix using RANSAC
F_trans, F_inliers = ransac((ptsL, ptsR), FundamentalMatrixTransform, min_samples=8, residual_threshold=0.1, max_trials=1500)
print ('the number of inliers is {:2d}'.format(np.sum(F_inliers)))

ind = np.ogrid[:ptsL.shape[0]]
```

```
FmatchesRansac = np.column_stack((ind[F_inliers],ind[F_inliers]))

fig = plt.figure(3,figsize = (10, 4))
axA = plt.subplot(111)
axA.set_title("inlier matches")
plt.gray()
# NOTE: function "plot matches" expects that keypoint coordinates are given as (y,x), that is (row, col)
plot_matches(axA, imL, imR, ptsL1, ptsR1, FmatchesRansac) #, matches_color = 'r')
axA.axis('off')
plt.show()
```

```
the number of inliers is 199
```

```
C:\Users\韩佳芮\AppData\Local\Temp\ipykernel_13048\2110071511.py:26: FutureWarning: `plot_matches` is deprecated since version
0.23 and will be removed in version 0.25. Use `skimage.feature.plot_matched_features` instead.
  plot_matches(axA, imL, imR, ptsL1, ptsR1, FmatchesRansac) #, matches_color = 'r')
```



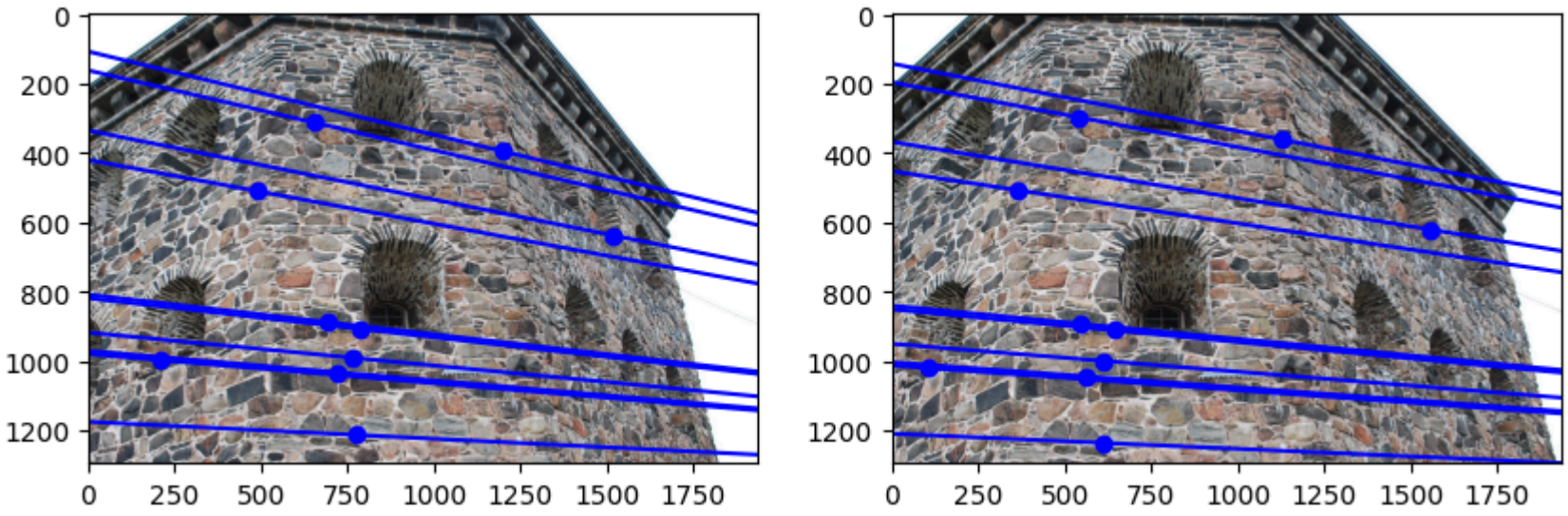
singular values for F

In [6]:
```
F = F_trans.params
Uf,Sf,Vf = la.svd(F, full_matrices=False)
print (Sf)
```

```
[7.37365114e-02 5.89395513e-05 1.40906246e-19]
```

Step 4: Epipolar lines from F

In [9]:
```
# Randomly select 10 matches (paris of features in two images) from the set of inliers for F
ind_sample = np.random.choice(ind[F_inliers], 10, replace = False)

# Indicate these matching features in image 1 and image 2
plt.figure(4,figsize = (10, 4))
ax41 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind_sample, 0], ptsL[ind_sample, 1], 'ob')
# print(ptsL[ind_sample, 0], ptsL[ind_sample, 1])
ax42 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind_sample, 0], ptsR[ind_sample, 1], 'ob')
# print(ptsL[ind_sample])
# generate epipolar line equations in image 2 (homoheneous 3-vectors l2 representing lines l2 x  = 0)
# a. create an array of points sampled in images 1 and 2
# b. create an array of homogeneous points sampled in images 1 and 2
# c. create an array of the corresponding epipolar lines in images 1 and 2
x = np.arange(0,imL.shape[1])
for i in range(10):
    # print(ptsL[ind_sample][0], ptsL[ind_sample][1])
    x1 = np.array([ptsL[ind_sample][i][0], ptsL[ind_sample][i][1], 1])
    # print(ptsL[ind_sample][i],"x1: ", x1)
    l = F@x1
    # print(l)
    y1 = (-l[0]*x-l[2])/l[1]

    x2 = np.array([ptsR[ind_sample][i][0], ptsR[ind_sample][i][1], 1])
    # print(F, x1, F*x1)
    l = np.transpose(x2)@F
    # print(l)
    y2 = (-l[0]*x-l[2])/l[1]

    ax42.plot(x, y1, 'b')
    ax41.plot(x, y2, 'b')

# for each feature (in both images) draw a correspoindiung epipolar line in the other image
# see Assignment 1 (line fitting part 1) for inspiration on how to visualize lines
# use ax41.plot and ax42.plot


plt.show()
```

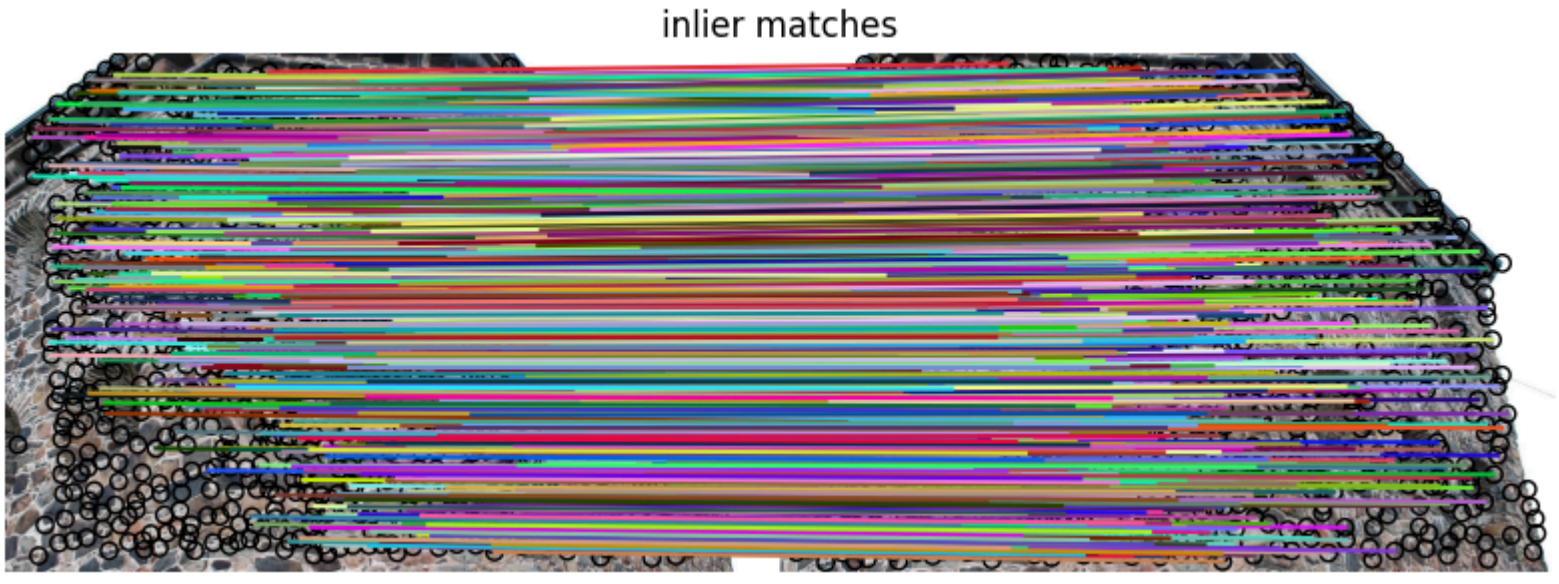


Step 5: Camera Normalization and Essential Matrix estimation using RANSAC

In [10]:
```
# normalization of points in two images using K (intrinsic parameters) e.g. in the following three steps
# a. convert original points to homogeneous 3-vectors (append "1" as a 3rd coordinate using np.append function)
# b. transform the point by applying the inverse of K
# c. convert homogeneous 3-vectors to 2-vectors (in R2)
Ones = np.ones((ptsL.shape[0], 1))
n_ptsL_temp = la.inv(K)@np.transpose(np.append(ptsL, Ones, axis=1))
n_ptsL = np.delete(np.transpose(n_ptsL_temp), 2, 1)

n_ptsR_temp = la.inv(K)@np.transpose(np.append(ptsR, Ones, axis=1))
n_ptsR = np.delete(np.transpose(n_ptsR_temp), 2, 1)

# robustly estimate essential matrix using normalized points and RANSAC
E_trans, E_inliers = ransac((n_ptsL, n_ptsR), EssentialMatrixTransform, min_samples=8, residual_threshold=0.0005, max_trials=5
num_inliers = np.sum(E_inliers)
print ('the number of inliers is {:2d}'.format(num_inliers))

ind = np.ogrid[:n_ptsL.shape[0]]
EmatchesRansac = np.column_stack((ind[E_inliers],ind[E_inliers]))

fig = plt.figure(5,figsize = (10, 4))
axA = plt.subplot(111)
axA.set_title("inlier matches")
plt.gray()
# NOTE: function "plot matches" expects that keypoint coordinates are given as (y,x), that is (row, col)
plot_matches(axA, imL, imR, ptsL1, ptsR1, EmatchesRansac) #, matches_color = 'r')
axA.axis('off')
plt.show()
```

```
the number of inliers is 735
```

```
C:\Users\韩佳芮\AppData\Local\Temp\ipykernel_13048\1168297276.py:25: FutureWarning: `plot_matches` is deprecated since version
0.23 and will be removed in version 0.25. Use `skimage.feature.plot_matched_features` instead.
  plot_matches(axA, imL, imR, ptsL1, ptsR1, EmatchesRansac) #, matches_color = 'r')
```



singular values for E

Hint: function svd from $linalg$ returns transpose V^T, not V.

In [11]:
```
E = E_trans.params
Ue,Se,Ve = la.svd(E)
print (Se)
```

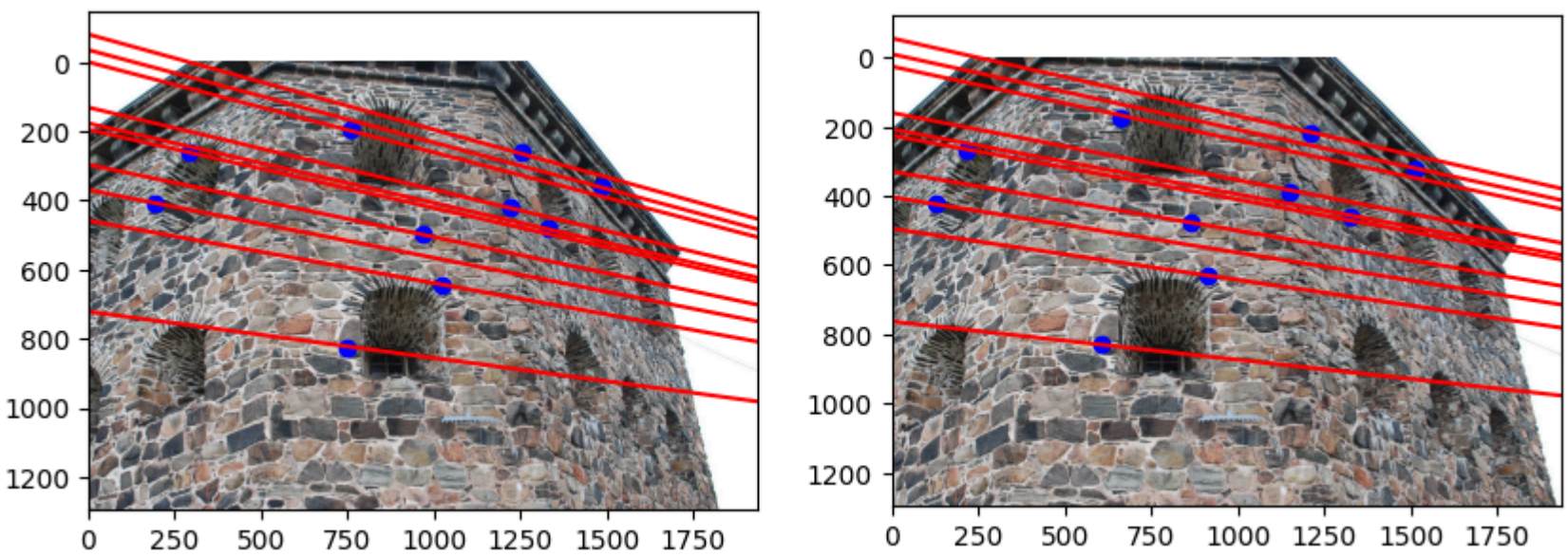
```
[4.91364874e+00 4.79728793e+00 3.85898759e-16]
```

Step 6: Epipolar Lines from E

In [12]:
```
plt.clf()
```

```
# Randomly select 10 matches (paris of features in two images) from the set of inliers for E
ind_sample = np.random.choice(ind[E_inliers], 10, replace = False)

# Indicate these matching features in image 1 and image 2
plt.figure(6,figsize = (10, 4))
ax61 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind_sample, 0], ptsL[ind_sample, 1], 'ob')
ax62 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind_sample, 0], ptsR[ind_sample, 1], 'ob')

# generate epipolar line equations in image 2 (homogeneous 3-vectors l2 representing lines l2 x  = 0)
# a. create an array of normalized points sampled in image 1
# b. create an array of homogeneous normalized points sampled in image 1
# c. create an array of the corresponding (uncalibrated) epipolar lines in image 2
n_ptsL_sample = n_ptsL[ind_sample,:]
n_ptsR_sample = n_ptsR[ind_sample,:]

# for each feature (in both images) draw a corresponding epipolar line in the other image
# use ax61.plot and ax62.plot
x = np.arange(0,imL.shape[1])
for i in range(10):
    x1 = np.array([n_ptsL_sample[i][0], n_ptsL_sample[i][1], 1])
    l1 = la.inv(K).T@E@x1
    y1 = (-l1[0]*x-l1[2])/l1[1]

    x2 = np.array([n_ptsR_sample[i][0], n_ptsR_sample[i][1], 1])
    l2 = la.inv(K.T)@(np.transpose(x2)@E)
    y2 = (-l2[0]*x-l2[2])/l2[1]

    ax62.plot(x, y1, 'r')
    ax61.plot(x, y2, 'r')
ax62.plot(n_ptsL[0, 0], n_ptsL[0, 1], 'r')
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



Step 7: Camera rotation and translation (four solutions)

Factorize essential matrix $E = [T]_x R$ where R is rotation and T is a translation. Find solutions R_1, R_2 and T_1, T_2. Use camera 1 for world coordinates. Define projection matrix for camera 1 as $P_w = [I|0]$ and compute four projection matrices for the second camera P_a, P_b, P_c, P_d.

Hint 1: for array multiplication use dot or $matmul$, never $*$.

Hint 2: function svd from $linalg$ returns V^T rather than V (the 2nd orthogonal matrix in svd decomposition $E = USV^T$).

Warning: remember that python uses 0 as a starting index for the rows or columns in arrays. For example, $A[0]$ denotes the first row of matrix A, while $P_w[2]$ stands for the 3rd row of the corresponding projection matrix and $E[:, [1]]$ is the second column of the essential matrix.

In [15]:
```
W = np.array([[0,-1,0], [1, 0, 0], [0, 0, 1]])

if la.det(Ue@Ve) < 0:
    Ve[:, -1] = -Ve[:, -1]
R1 = Ue @ W @ Ve
R2 = Ue @ np.transpose(W) @ Ve
T1 = Ue[:, 2]
T2 = -Ue[:, 2]

# first camera matrix
Pw = np.append(np.identity(3), np.zeros((3, 1)), axis=1)
```

```
# # four possible matrices for the second camera
Pa = np.append(R1, T1.reshape((3, 1)), axis=1)
Pb = np.append(R2, T1.reshape((3, 1)), axis=1)
Pc = np.append(R1, T2.reshape((3, 1)), axis=1)
Pd = np.append(R2, T2.reshape((3, 1)), axis=1)
```

Summary of Structure-from-Motion (the remaining steps 8-11):

In these 3D reconstruction steps you should use the world coordinate system consistent with the projection matrices estimated in step 7. In all steps you should obtain solutions for all four distinct cases of the second camera: P_a, P_b, P_c, P_d. First, step 8 is to Implement least squares (you can use svd or inv functions) for "triangulating" 3D points corresponding to pairs of matched features that are inliers for estimated E (i.e. consistent with the epipolar geometry). Make sure to use normalized coordinates for image points. Then (step 9) you will compute camera positioning (optical centers and calibrated image centers as 3D points) in the world coordinate system. This is used in data visualsation step 10 (fully implemented). That step visulaizes in 3D both camera positions (red - optcal centers, green - image centers) and triangulated points (blue) for four possible cases of the second camera. You should identify one case when solution has 3D points in front of both cameras. In the last step 11 you will project 3D points onto each camera, convert to uncalibrated coordinates, and display these projected points (use red) together with the original features (use blue). Observe if the are red and blue points are close in each image.

Step 8: Triangulation (four solutions)

In [17]:
```
Aa = []
Ab = []
Ac = []
Ad = []
EinliersL = n_ptsL[ind[E_inliers], :]
# print(EinliersL.shape)

EinliersR = n_ptsR[ind[E_inliers], :]
# print(EinliersR.shape)

for i in range(num_inliers):
    ua = EinliersL[i, 0]
    va = EinliersL[i, 1]
    ub = EinliersR[i, 0]
    vb = EinliersR[i, 1]

    # axis1 = np.array([ua, va])
    # print([Pw[0]-ua*Pw[2], Pw[1]-va*Pw[2], Pa[0]-ub*Pa[2], Pa[1]-vb*Pa[2]])
    # Aa.append([])
    arr = np.array([[ua], [va]])
    # print(arr, R1[2:3])
    Pw12 = Pw[0:2]-arr@Pw[2:3]
    Pa12 = Pa[0:2]-np.array([[ub], [vb]])@Pa[2:3]
    Pb12 = Pb[0:2]-np.array([[ub], [vb]])@Pb[2:3]
    Pc12 = Pc[0:2]-np.array([[ub], [vb]])@Pc[2:3]
    Pd12 = Pd[0:2]-np.array([[ub], [vb]])@Pd[2:3]

    tempAa = np.append(Pw12, Pa12, axis = 0)
    tempAb = np.append(Pw12, Pb12, axis = 0)
    tempAc = np.append(Pw12, Pc12, axis = 0)
    tempAd = np.append(Pw12, Pd12, axis = 0)

    Aa.append(tempAa)
    Ab.append(tempAb)
    Ac.append(tempAc)
    Ad.append(tempAd)

Aa = np.array(Aa)
Ab = np.array(Ab)
Ac = np.array(Ac)
Ad = np.array(Ad)
```

Solution using least squares: assume homogeneous 3D point $X = [X_1, X_2, X_3, 1]$. Then, $AX = 0$ gives 4 equations for 3 unknowns. Use approach 1 (inhomogeneous least squares) discussed for homography estimation (Topic 6).

In [18]:
```
# least squares for solving linear system A_{0:2} X_{0:2} = - A_3
Xa, Xb, Xc, Xd = [], [], [], []
for i in range(num_inliers):
    Aa_02 = Aa[i, :, 0:3]       # the first 3 columns of 4x4 matrix A

    Aa_3  = Aa[i, :, 3:4]       # the last column on 4x4 matrix A
    # print(Aa_3)
    Ab_02 = Ab[i, :, 0:3]
    # print(Ab_02)
    Ab_3  = Ab[i, :, 3]
    Ac_02 = Ac[i, :, 0:3]
    Ac_3  = Ac[i, :, 3]
    Ad_02 = Ad[i, :, 0:3]
    Ad_3  = Ad[i, :, 3]
    Aa_02T = np.transpose(Aa_02)
```

```
    Ab_02T = np.transpose(Ab_02)
    Ac_02T = np.transpose(Ac_02)
    Ad_02T = np.transpose(Ad_02)
    # print("original:", la.det(Aa[i]))
    # print("inverse: ", la.det(la.inv(Aa_02T @ Aa_02)))

#     # print(Aa_02, Aa_3)
#     # Nx3 matrices: N rows with 3D point coordinates for N reconstructed points (N=num_inliers)
    Xa.append(la.inv(Aa_02T @ Aa_02) @ Aa_02T @ (-Aa_3))
    Xb.append(la.inv(Ab_02T @ Ab_02) @ Ab_02T @ (-Ab_3.T))
    Xc.append(la.inv(Ac_02T @ Ac_02) @ Ac_02T @ (-Ac_3.T))
    Xd.append(la.inv(Ad_02T @ Ad_02) @ Ad_02T @ (-Ad_3.T))
Xa, Xb, Xc, Xd = np.array(Xa), np.array(Xb), np.array(Xc), np.array(Xd)
```

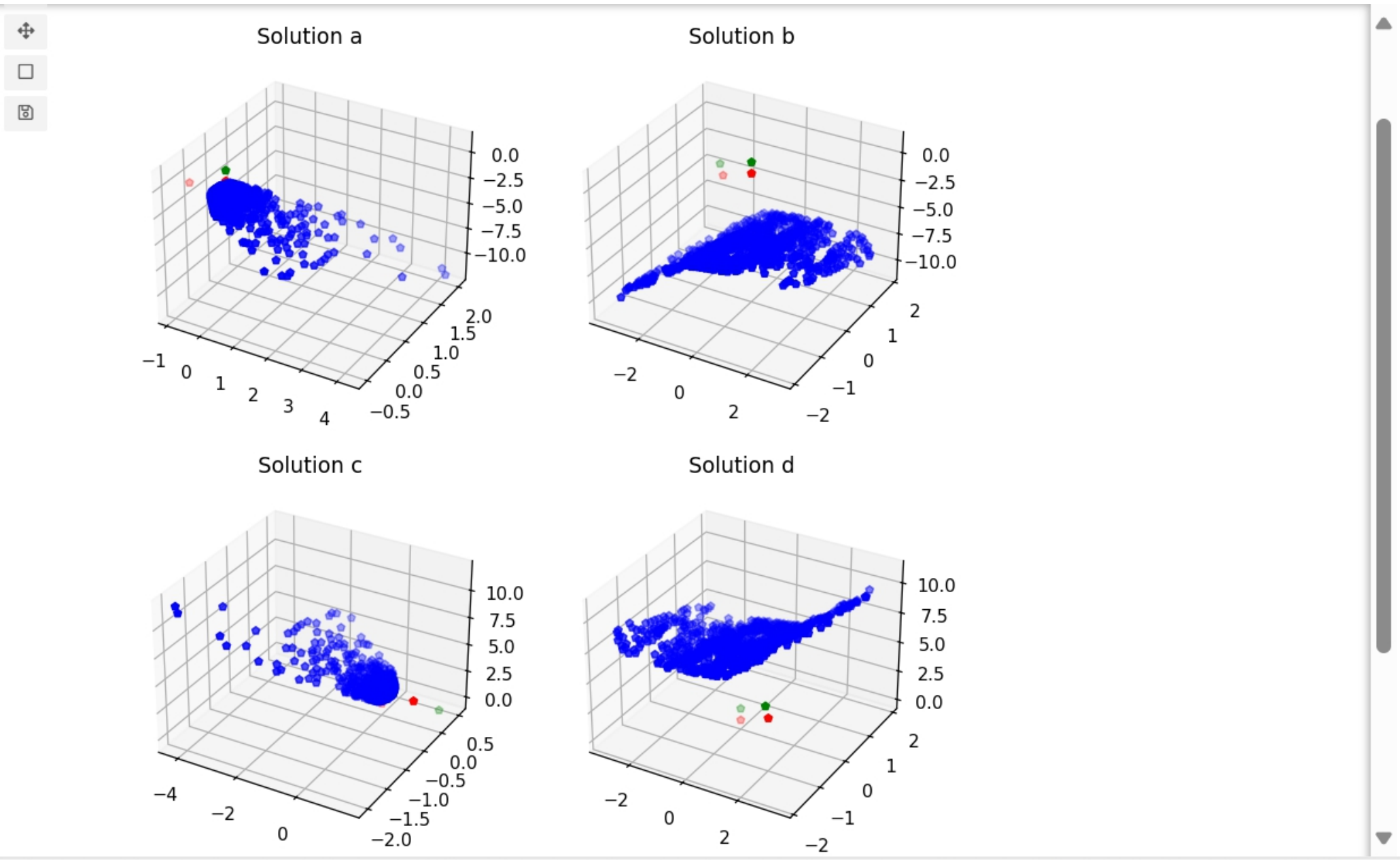
Step 9: Camera positioning in 3D (four solutions)

In this step you will compute location of each cameras' optical center and its (calibrated) image center as points in 3D (world coordinate system). The next step 10 visulaizes the computed cameras' optical centers in red and image centers in green.

In [19]:
```
# camera's optical centers (for pair of cameras) as points in 3D world coordinate system
# 2x3 matrices: two rows with 3D point coordinates for the first and second camera
la.inv(Pa[:, 0:3])
Cw = -la.inv(Pw[:, 0:3])@Pw[:,3]
Ca = np.array([Cw, -la.inv(Pa[:, 0:3])@Pa[:,3]])
Cb = np.array([Cw, -la.inv(Pb[:, 0:3])@Pb[:,3]])
Cc = np.array([Cw, -la.inv(Pc[:, 0:3])@Pc[:,3]])
Cd = np.array([Cw, -la.inv(Pd[:, 0:3])@Pd[:,3]])

I = np.array([0,0,1])
# calibrated/normalized image centers (for pair of cameras) as points in 3D world coordinate system
# 2x3 matrices: two rows with 3D po### Step 9: Cameras positions/orientations in 3D  (four solutions)int coordinates for the f
Qw = la.inv(Pw[:, 0:3])@(I-Pw[:,3])
Qa = np.array([Qw, la.inv(Pa[:, 0:3])@(I-Pa[:,3])])
Qb = np.array([Qw, la.inv(Pb[:, 0:3])@(I-Pb[:,3])])
Qc = np.array([Qw, la.inv(Pc[:, 0:3])@(I-Pc[:,3])])
Qd = np.array([Qw, la.inv(Pd[:, 0:3])@(I-Pd[:,3])])

# print (Cc)
# print (Qc)
```

Step 10 (fully implemented): 3D visualization of cameras and triangulated points (four solutions)

In [27]:
```
# NOTE: WIDGETS MAGIC ALLOWS INTERACTIVE VISUALIZATION OF RECONSTRUCTED 3D POINT CLOUD
%matplotlib widget
plt.clf()
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(10,figsize = (8, 8))

ax10_1 = plt.subplot(221, projection='3d')
plt.title('Solution a')
ax10_1.scatter(Xa[:,0],Xa[:,1],Xa[:,2], c='b', marker='p')
ax10_1.scatter(Ca[:,0],Ca[:,1],Ca[:,2], c='r', marker='p')
ax10_1.scatter(Qa[:,0],Qa[:,1],Qa[:,2], c='g', marker='p')

ax10_2 = plt.subplot(222, projection='3d')
plt.title('Solution b')
ax10_2.scatter(Xb[:,0],Xb[:,1],Xb[:,2], c='b', marker='p')
ax10_2.scatter(Cb[:,0],Cb[:,1],Cb[:,2], c='r', marker='p')
ax10_2.scatter(Qb[:,0],Qb[:,1],Qb[:,2], c='g', marker='p')

ax10_3 = plt.subplot(223, projection='3d')
plt.title('Solution c')
ax10_3.scatter(Xc[:,0],Xc[:,1],Xc[:,2], c='b', marker='p')
ax10_3.scatter(Cc[:,0],Cc[:,1],Cc[:,2], c='r', marker='p')
ax10_3.scatter(Qc[:,0],Qc[:,1],Qc[:,2], c='g', marker='p')

ax10_4 = plt.subplot(224, projection='3d')
plt.title('Solution d')
ax10_4.scatter(Xd[:,0],Xd[:,1],Xd[:,2], c='b', marker='p')
ax10_4.scatter(Cd[:,0],Cd[:,1],Cd[:,2], c='r', marker='p')
ax10_4.scatter(Qd[:,0],Qd[:,1],Qd[:,2], c='g', marker='p')

plt.show()
```



x=3.7378, y pane=2.2185, z=−11.4339

Step 11: Reprojection errors

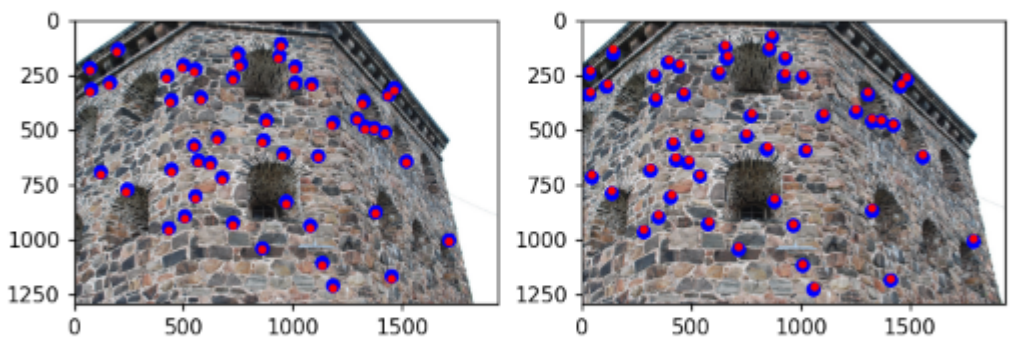
In [21]:
```
plt.clf()
# Randomly select N=50 matches (pairs of features in two images) from the set of inliers for E
N = 50
ind_sample2 = np.random.choice(num_inliers, N, replace = False)
# Indicate (E) inlier matches in image 1 and image 2\

# # project reconstructed 3D points onto both images and display them in red color
# # a. convert correct points (Xa, Xb, Xc, or Xd) to homogeneous 4 vectors
# # b. project homogeneous 3D points (onto uncalibrated cameras) using correct Projection matrices (KPw and, e.g. KPa)
# # c. convert to regular (inhomogeneous) point
uvLa = []
uvRa = []
uvLb = []
uvRb = []
uvLc = []
uvRc = []
uvLd = []
uvRd = []
Xac = Xa[ind_sample2]
Xbc = Xb[ind_sample2]
Xcc = Xc[ind_sample2]
Xdc = Xd[ind_sample2]

for i in range(50):
    # pw = np.append(Xw[i], 1)
    pa = np.append(Xac[i], 1)
    pb = np.append(Xbc[i], 1)
    pc = np.append(Xcc[i], 1)
    pd = np.append(Xdc[i], 1)
    uvLa.append(K@Pw@pa)
    uvRa.append(K@Pa@pa)
    uvLb.append(K@Pw@pb)
    uvRb.append(K@Pb@pb)
    uvLc.append(K@Pw@pc)
    uvRc.append(K@Pc@pc)
    uvLd.append(K@Pw@pd)
    uvRd.append(K@Pd@pd)

uvLa = np.array(uvLa)
uvRa = np.array(uvRa)
uvLb = np.array(uvLb)
uvRb = np.array(uvRb)
uvLc = np.array(uvLc)
uvRc = np.array(uvRc)
uvLd = np.array(uvLd)
uvRd = np.array(uvRd)
```

```
ptsL_proja = uvLa[:, :2]/uvLa[:, [2]]
ptsR_proja = uvRa[:, :2]/uvRa[:, [2]]
ptsL_projb = uvLb[:, :2]/uvLb[:, [2]]
ptsR_projb = uvRb[:, :2]/uvRb[:, [2]]
ptsL_projc = uvLc[:, :2]/uvLc[:, [2]]
ptsR_projc = uvRc[:, :2]/uvRc[:, [2]]
ptsL_projd = uvLd[:, :2]/uvLd[:, [2]]
ptsR_projd = uvRd[:, :2]/uvRd[:, [2]]
```

reprojection with Xa:

In [22]:
```
%matplotlib widget
plt.clf()
plt.figure(11,figsize = (8, 4))
ax11_1 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind[E_inliers][ind_sample2], 0], ptsL[ind[E_inliers][ind_sample2], 1], 'ob')
ax11_2 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind[E_inliers][ind_sample2], 0], ptsR[ind[E_inliers][ind_sample2], 1], 'ob')

ax11_1.plot(ptsL_proja[:,0], ptsL_proja[:,1], '.r')
ax11_2.plot(ptsR_proja[:,0], ptsR_proja[:,1], '.r')

plt.show()
```

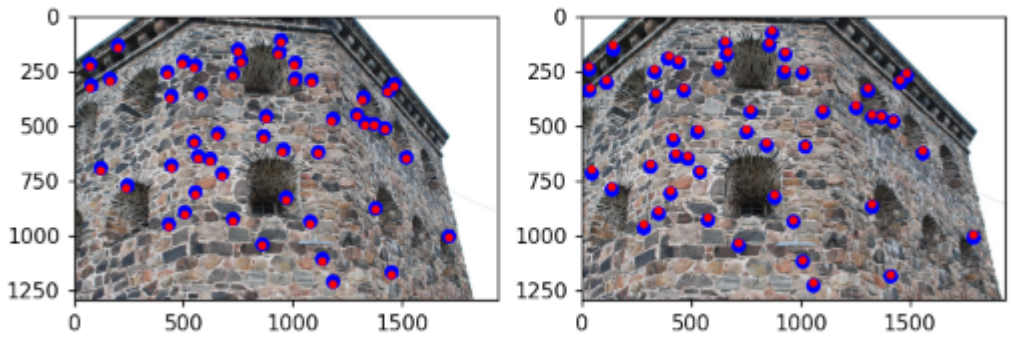
Figure 11



reprojection with Xb:

In [23]:
```
plt.clf()
plt.figure(12,figsize = (8, 4))
ax12_1 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind[E_inliers][ind_sample2], 0], ptsL[ind[E_inliers][ind_sample2], 1], 'ob')
ax12_2 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind[E_inliers][ind_sample2], 0], ptsR[ind[E_inliers][ind_sample2], 1], 'ob')

ax12_1.plot(ptsL_projb[:,0], ptsL_projb[:,1], '.r')
ax12_2.plot(ptsR_projb[:,0], ptsR_projb[:,1], '.r')
plt.show()
```

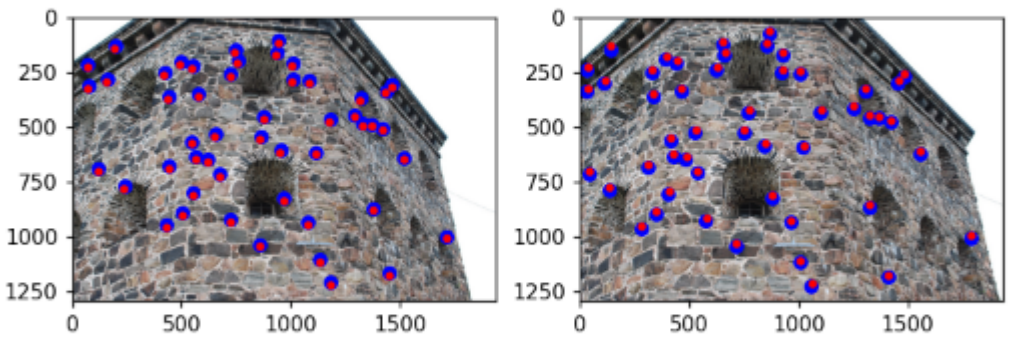
Figure 12



reprojection with Xc:

In [24]:
```
plt.clf()
plt.figure(13,figsize = (8, 4))
ax13_1 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind[E_inliers][ind_sample2], 0], ptsL[ind[E_inliers][ind_sample2], 1], 'ob')
ax13_2 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind[E_inliers][ind_sample2], 0], ptsR[ind[E_inliers][ind_sample2], 1], 'ob')
ax13_1.plot(ptsL_projc[:,0], ptsL_projc[:,1], '.r')
ax13_2.plot(ptsR_projc[:,0], ptsR_projc[:,1], '.r')

plt.show()
```

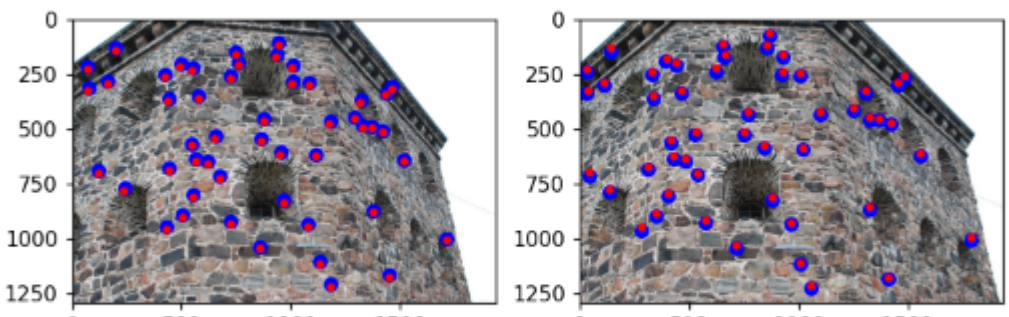
Figure 13



reprojection with Xd:

In [25]:
```
plt.clf()
plt.figure(14,figsize = (8, 4))
ax14_1 = plt.subplot(121)
plt.imshow(imL)
plt.plot(ptsL[ind[E_inliers][ind_sample2], 0], ptsL[ind[E_inliers][ind_sample2], 1], 'ob')
ax14_2 = plt.subplot(122)
plt.imshow(imR)
plt.plot(ptsR[ind[E_inliers][ind_sample2], 0], ptsR[ind[E_inliers][ind_sample2], 1], 'ob')
ax14_1.plot(ptsL_projd[:,0], ptsL_projd[:,1], '.r')
ax14_2.plot(ptsR_projd[:,0], ptsR_projd[:,1], '.r')
plt.show()
```

Figure 14



Question: how different are projected points for SfM solutions a, b, c, and d? Explain.

Answer: Solutions a, b, c and d all reproject to the same points on the image because the corresponding Pa, Pb, Pc, and Pd are the four different possible extraction from essential matrix E that determines the position of the points; however, according to the 3d visualization of cameras and triangulated points, only solution d has reasonable camera positions (optical center and image center), so only solution d is valid.