Two-view Geometry

Contents

1 Linear Triangulation **1**
- 1.1 Linear system of equations . . . 1
- 1.2 Solving the system . . . 2
- 1.3 Writing `linearTriangulation` . . . 2

2 Eight-point algorithm **3**
- 2.1 Derivation of the eight-point algorithm . . . 3
- 2.2 Implementation . . . 4
  - 2.2.1 Implementation of the eight-point algorithm . . . 4
  - 2.2.2 Normalized eight-point algorithm . . . 5
- 2.3 Extracting E from F . . . 5

3 Putting things together: Structure from Motion **6**
- 3.1 Decomposing the essential matrix E into (R, T) . . . 6
- 3.2 Visualization . . . 7

4 Numerical Exercises **9**

The goal of this laboratory session is to practice with several techniques used in two-view geometry such as: linear triangulation of 3-D points from their projections, estimation of the fundamental matrix via the 8-point algorithm, and extraction of relative camera motion from the essential matrix.

In the first two sections, we will write functions to perform linear triangulation and fundamental matrix estimation, and verify they work correctly using synthetic data. In the last section, we will use these functions to build a minimal structure from motion pipeline that will estimate two cameras' relative poses, as well as a sparse point cloud of a scene, given two images and a set of noise-free correspondences between these images. The expected output of the pipeline is shown in Fig. 5.

1 Linear Triangulation

1.1 Linear system of equations

We will first derive a way to triangulate the position of a 3D point $P^i = (X^i, Y^i, Z^i, 1)^T$ in the scene, given its projections $\mathbf{p}_1^i = (u_1^i, v_1^i, 1)^T$ and $\mathbf{p}_2^i$ on two images (expressed in homogeneous coordinates), and the projection matrices M_1 and M_2 (see Fig. 1).

As seen in the lecture slides, the scale factor λ can be removed from the projection equation $\lambda\mathbf{p} = MP$ by taking the cross product of both sides with $\mathbf{p}$:

$$\mathbf{p} \times (\lambda\mathbf{p}) = \mathbf{p} \times (MP) \rightarrow \mathbf{p} \times (MP) = 0 \tag{1}$$

Using this trick for both projections, a linear system of equations can be set up to estimate P^i:

$$\begin{bmatrix} [\mathbf{p}_1^i]_\times M_1 \\ [\mathbf{p}_2^i]_\times M_2 \end{bmatrix} \cdot P^i = 0 \tag{2}$$

$$\iff A \cdot P^i = 0 \tag{3}$$

where $[\mathbf{x}]_\times = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$, which is the matrix representation of the cross product.

1.2 Solving the system

(2) has the general form $AX = 0$, which we already encountered in a previous exercise. As a reminder, we look for a non-trivial solution X that minimizes $||A \cdot X||^2$ subject to the constraint $||X||^2 = 1$ (to avoid the trivial solution $X = 0$). This can be done using the Singular Value Decomposition (SVD) of A: $A = USV^T$ where U,V are unitary matrices and S is diagonal. The solution of the problem $AX = 0$ is the eigenvector corresponding to the smallest eigenvalue of A^TA, which simply corresponds to the last column of V (remember that both Matlab or Pyhthon's `numpy.linalg.svd` sorts the singular values by decreasing order).

1.3 Writing `linearTriangulation`

Fill in the code of the function `linearTriangulation` that takes as inputs a set of correspondences $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1..N}$, the projection matrices M_1 and M_2, and returns $\mathbf{P} = \begin{bmatrix} P^1 & \cdots & P^N \end{bmatrix}$, a $4 \times N$ matrix containing the triangulated points (in homogeneous coordinates). To achieve this, iterate over all the points using a `for` loop and incrementally build $\mathbf{P}$. Do not forget to dehomogeneize the triangulated points P_i at the end of the function, i.e. divide each row of $\mathbf{P}$ by the last row. Then run the script `run_test_triangulation.m` in Matlab or `run_test_triangulation.py` to check if your function works correctly. The errors for each 3D point should be very close to zero, as illustrated in Fig. 2.

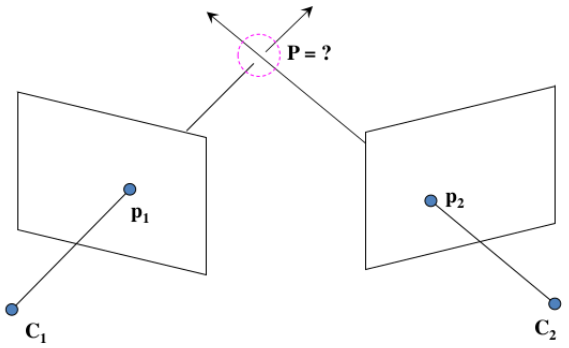


Figure 1: The goal of triangulation is to estimate P given $\mathbf{p_1}, \mathbf{p_2}$ and the projection matrices M_1, M_2.

```
P_est-P=

ans =

   1.0e-12 *

    0.0013   -0.0138    0.0056    0.0093   -0.0061   -0.0034    0.0209   -0.1075   -0.1181    0.0152
    0.0019   -0.0101    0.0041    0.0061   -0.0047   -0.0024    0.0123   -0.0797   -0.0873    0.0131
    0.0036    0.0213   -0.0124   -0.0098    0.0089    0.0036   -0.0018    0.1705    0.1528   -0.0089
         0         0         0         0         0         0         0         0         0         0
```

Figure 2: Result of running the script `run_test_triangulation.m`

2 Eight-point algorithm

During the course, the *essential matrix* E, which encapsulates the geometry of two views in a calibrated framework (i.e. when both cameras are calibrated), was presented. Specifically, given *two calibrated (sometimes also called normalized)* point correspondences $\overline{\mathbf{p}}_1 \leftrightarrow \overline{\mathbf{p}}_2$, where $\overline{\mathbf{p}}_j = K_j^{-1}\mathbf{p}_j$, $j = 1, 2$:

$$\overline{\mathbf{p}}_2^T E \overline{\mathbf{p}}_1 = 0 \tag{4}$$

where $E = [T]_\times R$ encodes the relative pose between the two cameras. Replacing the normalized coordinates with pixel coordinates:

$$\begin{aligned} (K_2^{-1}\mathbf{p}_2)^T E (K_1^{-1}\mathbf{p}_1) &= 0 \\ \iff \mathbf{p}_2^T (K_2^{-T} E K_1^{-1}) \mathbf{p}_1 &= 0 \\ \iff \mathbf{p}_2^T F \mathbf{p}_1 &= 0 \end{aligned}$$

where $F = K_2^{-T} E K_1^{-1}$ is the *fundamental matrix*. It is more general than the essential matrix because the equation $\mathbf{p}_2^T F \mathbf{p}_1 = 0$ holds also in the uncalibrated case, i.e. when the calibration matrix of the cameras is unknown. The fundamental matrix F can be estimated exactly like the essential matrix E, using the 8-point algorithm as shown in the lecture slides. However, we will now derive the same equations in a different way, that will allow for an elegant implementation of the eight-point algorithm.

2.1 Derivation of the eight-point algorithm

As shown during the course, the epipolar constraints (Fig. 3) can be written:

$$\mathbf{p}_2^{i\top} F \mathbf{p}_1^i = 0 \tag{5}$$

where $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}_{i=1..N}$ are a set of point correspondences in both images. Using a useful property of the Kronecker product, (5) can be rewritten as:

$$(\mathbf{p}_1^i \otimes \mathbf{p}_2^i)^T \operatorname{vec}(F) = 0$$

where $\operatorname{vec}(F)$ denotes the vectorization of the matrix F formed by stacking its columns into a single column vector, and $\otimes$ the Kronecker product. We can now stack the N constraints into a single $N \times 9$ matrix Q:

$$Q = \begin{bmatrix} (\mathbf{p}_1^1 \otimes \mathbf{p}_2^1)^T \\ \vdots \\ (\mathbf{p}_1^N \otimes \mathbf{p}_2^N)^T \end{bmatrix}$$

and then solve for $\operatorname{vec}(F)$ which satisfies:

$$Q \cdot \operatorname{vec}(F) = 0 \tag{6}$$

We solve (6) in the least-squares sense (i.e. we seek F that minimizes the epipolar constraints (5) in a least-squares sense). The fundamental matrix F has rank 2 (like the essential matrix E), therefore a valid fundamental matrix must satisfy $\det(F) = 0$.

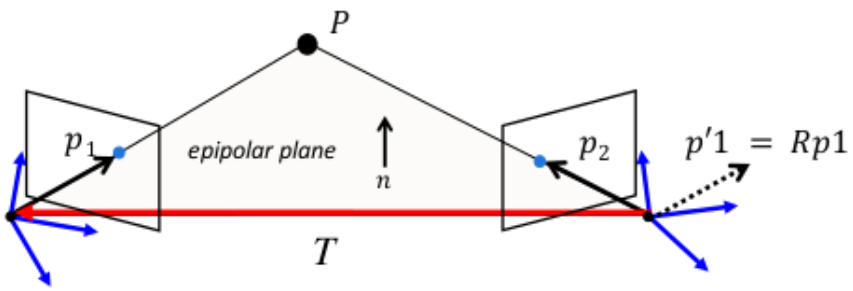


Figure 3: Two-view geometry: writing that $\overline{\mathbf{p}}_1$,$\overline{\mathbf{p}}_2$ and T are coplanar vectors yields the essential matrix equation (4)

2.2 Implementation

In this section, you will implement the eight-point algorithm to estimate F. You will then extract the essential matrix E from F, using the calibration matrices K_1 and K_2.

2.2.1 Implementation of the eight-point algorithm

Complete the function `fundamentalEightPoint`, which receives a set of point correspondences $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\}$ (in homogeneous coordinates) and returns the 3×3 fundamental matrix F that minimizes the epipolar constraints $\mathbf{p}_2^{iT} F \mathbf{p}_1^i = 0$ in a least-squares sense.

A few tips

- Matlab's `kron` or Python's `numpy.kron` function (which implements the Kronecker product) will be useful to build the matrix Q.
- The problem $Q \cdot \text{vec}(F) = 0$ has again the form $QX = 0$. To solve it you can therefore proceed exactly like in section 1.2. You will first compute the SVD: $Q = U\Sigma V^T$. Then the solution $\text{vec}(F)$ will be given by the last column of V.
- Once you have estimated $\text{vec}(F)$, you can use the `reshape` function to convert $\text{vec}(F)$ back to F.
- You will need to correct the estimated matrix F to enforce that $\det(F) = 0$. This can be done by first computing the SVD of F:

$$F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T$$

  and then forcing the smallest singular value to be 0 by updating F as follows:

$$F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T$$

  . This constraint is required for all the epipolar lines in an image to intersect at a single point, the epipole.

Error measures The quality of the estimated fundamental matrix F can be measured using different cost functions. For example, we provide code to compute the algebraic error given by the sum of squared epipolar constraints $\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{p}_2^{iT} F \mathbf{p}_1^i\right)^2}$. A better quality criterion is given by the function `distPoint2EpipolarLine` provided, since it measures a geometric quantity in the image plane: the Euclidean distance from points to their epipolar lines given by the estimated fundamental matrix F. Specifically, this function computes the Root-Mean-Square error

$$\Big(\frac{1}{N}\sum_{i=1}^{N}\left(d_\perp^2(\mathbf{p}_1^i, \boldsymbol{\ell}_1^i) + d_\perp^2(\mathbf{p}_2^i, \boldsymbol{\ell}_2^i)\right)\Big)^{\frac{1}{2}},$$

where $\boldsymbol{\ell}_1^i = F^T \mathbf{p}_2^i$ and $\boldsymbol{\ell}_2^i = F \mathbf{p}_1^i$ are the epipolar lines in images 1 and 2, respectively, and $d_\perp(\mathbf{p}, \boldsymbol{\ell})$ measures the point-to-line distance in the image planes.

Testing Run the first section of the MATLAB script `run_test_8point.m` or the Python script `run_test_8point.py` to test your function `fundamentalEightPoint`. The script generates a number of exact (i.e., noise-free) point correspondences, and computes the two errors described in the previous section. With such a data set you should get exact results (up to machine precision), i.e. a value $< 10^{-10}$ for both the algebraic and geometric errors.

Now run the second section, which runs the function `fundamentalEightPoint` on a set of noisy point correspondences (a small amount of Gaussian noise has been added compared to the first section). You will see that both the resulting algebraic error and the geometric error are now quite high, which means something is wrong with the function `fundamentalEightPoint.` Can you see what?

2.2.2 Normalized eight-point algorithm

It turns out, that (as seen in class), if there is a significant difference between the orders of magnitude of the individual 2D points $\{p_1^i = (u_1^i, v_1^i)^T\}_{i=1..N}$ and $\{p_2^i = (u_2^i, v_2^i)^T\}_{i=1..N}$ on each image plane, or if there are significant offsets, the numerical conditioning of the system of equations in the eight-point algorithm is poor, which makes the algorithm unstable, very sensitive to noise.

This can be fixed using a normalized eight-point algorithm, which estimates the fundamental matrix on a set of normalized correspondences (with better numerical properties) and then unnormalizes the result to obtain the fundamental matrix for the given (unnormalized) correspondences.

A classical way to do this is by building two normalized sets of 2D points $\{\widetilde{p}_1^i\}$ and $\{\widetilde{p}_2^i\}$ by scaling and applying an offset to (respectively) $\{p_1^i\}$ and $\{p_2^i\}$ in such a way that the centroid of each set is 0 and the average distance of a point on each set to the centroid is equal to $\sqrt{2}$. This can be done for every point as follows:

$$\widetilde{p}_j^i = \frac{\sqrt{2}}{\sigma_j}(p_j^i - \mu_j) \tag{7}$$

where $\mu_j = \frac{1}{N}\sum_i p_j^i$ is the centroid of set $\{p_j^i\}$ and $\sigma_j^2 = \frac{1}{N}\sum_i ||p_j^i - \mu_j||^2$ is the mean squared magnitude of the centered points (alternatively, one can use the mean distance $\frac{1}{N}\sum_i ||p_j^i - \mu_j||$ to normalize the points instead of σ_j; it does not matter much). Conveniently, (7) can be expressed linearly (i.e., as a matrix product) by using the homogeneous coordinates $\mathbf{p}_j^i = (u_j^i, v_j^i, 1)^T = (p_j^i, 1)^T$ and $\widetilde{\mathbf{p}}_j^i = (\widetilde{u}_j^i, \widetilde{v}_j^i, 1)^T = (\widetilde{p}_j^i, 1)^T$:

$$\begin{aligned} \widetilde{\mathbf{p}}_j^i &= \begin{bmatrix} s_j & 0 & -s_j\mu_j^x \\ 0 & s_j & -s_j\mu_j^y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_j^i \\ \widetilde{\mathbf{p}}_j^i &\triangleq T_j \mathbf{p}_j^i \end{aligned}$$

where $s_j \triangleq \frac{\sqrt{2}}{\sigma_j}$.

The normalized eight-point algorithm can therefore be summarized in three steps:

1. Normalize point correspondences: $\{\mathbf{p}_1^i \leftrightarrow \mathbf{p}_2^i\} \longrightarrow \{\tilde{\mathbf{p}}_1^i \leftrightarrow \tilde{\mathbf{p}}_2^i\}$, where $\tilde{\mathbf{p}}_j^i = T_j \mathbf{p}_j^i$ for $j = 1, 2$
2. Estimate the fundamental matrix using the eight-point algorithm: $\{\tilde{\mathbf{p}}_1^i \leftrightarrow \tilde{\mathbf{p}}_2^i\} \longrightarrow \tilde{F}$
3. Unnormalize the fundamental matrix: $\tilde{F} \longrightarrow F = T_2^T \widetilde{F} T_1$.

Complete the function `normalise2dpts()`. Remember that *normalization is carried out in Euclidean coordinates*, not in homogeneous coordinates (the last coordinate of $\mathbf{p}_j^i$ may not be 1). Use it to complete `fundamentalEightPoint_Normalized` which should implement the previous three steps. Finally, run the last section of the script `run_test_8point.m` and check that, this time, the resulting errors are much lower than when using the unnormalized eight-point algorithm. You should obtain an *algebraic error* of 2.20 and a *geometric error* of 41.51 pixels.

2.3 Extracting E from F

We established above a relationship between E and F:

$$F = K_2^{-T} E K_1^{-1} \tag{8}$$

Use (8) to write E in terms of F, K_1 and K_2 and complete the code of the function `estimateEssentialMatrix()` which should call `fundamentalEightPoint_Normalized()` and then convert the estimated fundamental matrix F to the essential matrix E, given the calibration matrices.

3 Putting things together: Structure from Motion

In this section, we use the functions written in the first two sections to implement of simple *Structure from Motion* pipeline, as seen during the course. We will assume that a set of noise-free point correspondences is provided (in the next exercise you will learn how to filter out outlier correspondences). A template code to complete is given in the script `run_sfm.m` in Matlab or `run_sfm.py` in Python. In this section, we assume: $K_1 = K_2 \triangleq K$.

First, we will estimate the essential matrix E using the point correspondences and K. Second, we will extract the relative camera positions R, T from E and use them to build the projection matrices $M_1$and M_2. Finally, we will use linear triangulation to build a sparse point cloud of the scene.

3.1 Decomposing the essential matrix E into (R, T)

In the rest of the exercise, we choose camera 1 as the origin of the world coordinates, which means we define the first projection matrix as $M_1 = K_1[I|0]$.

We now turn to extracting the relative transformation between camera 1 and camera 2 from the essential matrix $E = [T]_\times R$. As seen in the lecture, there are four possible solutions (R_i, T_i) that are consistent with the essential matrix E, namely:

$$\begin{aligned}
[R_1|T_1] &= [UWV^T|u_3] \\
[R_2|T_2] &= [UW^TV^T|u_3] \\
[R_3|T_3] &= [UWV^T|-u_3] \\
[R_4|T_4] &= [UW^TV^T|-u_3]
\end{aligned}$$

where $W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and u_3 is the last column of U in the SVD: $E = U\Sigma V^T$. However, only one of those solutions is physically feasible in the sense that the triangulated 3-D points using the camera motion are in front of both cameras (see Fig. 4).

- Complete the function `decomposeEssentialMatrix` that returns the two possible rotations UWV^T, UW^TV^T and the translation u_3 from E. **Important note!** You need to make sure that the returned rotation matrices are valid (i.e. their determinant is 1, and not -1). If the determinant of UWV^T or UW^TV^T is -1, simply invert the sign of the matrix.

- Complete the function `disambiguateRelativePose` that selects the correct relative pose among the four possible configurations. To do that, you can, for each configuration (R_i, T_i), compute the projection matrices $M_1 = K_1[I|0]$ and $M_2 = K_2 \begin{bmatrix} R_i & T_i \end{bmatrix}$, then triangulate the points using `linearTriangulation`. Among the possible configurations, keep the one that yields the highest number of triangulated points lying *in front* of the image plane, i.e. with positive depth.

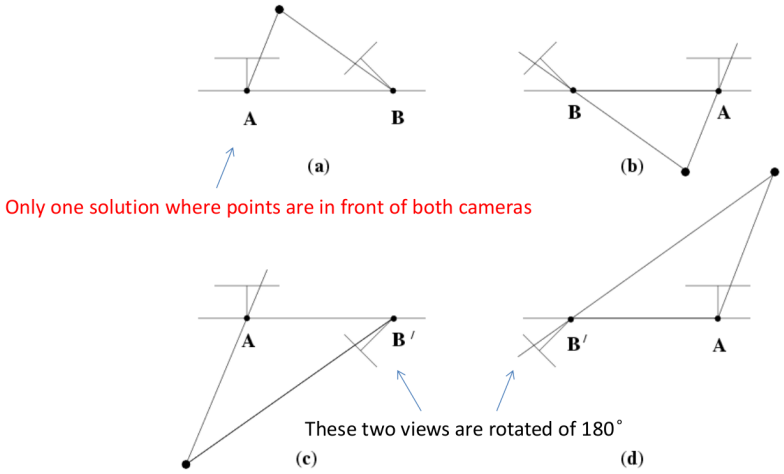


Figure 4: The four possible configurations that are consistent with a given essential matrix E

3.2 Visualization

Now that you have found the correct projection matrix $M_2 = K_2[R|T]$, as well as a point cloud of triangulated points $\mathbf{P}$, run the provided code to visualize the reconstructed 3-D points and camera poses. Your results should be similar to those in Figs. 5 and 6.

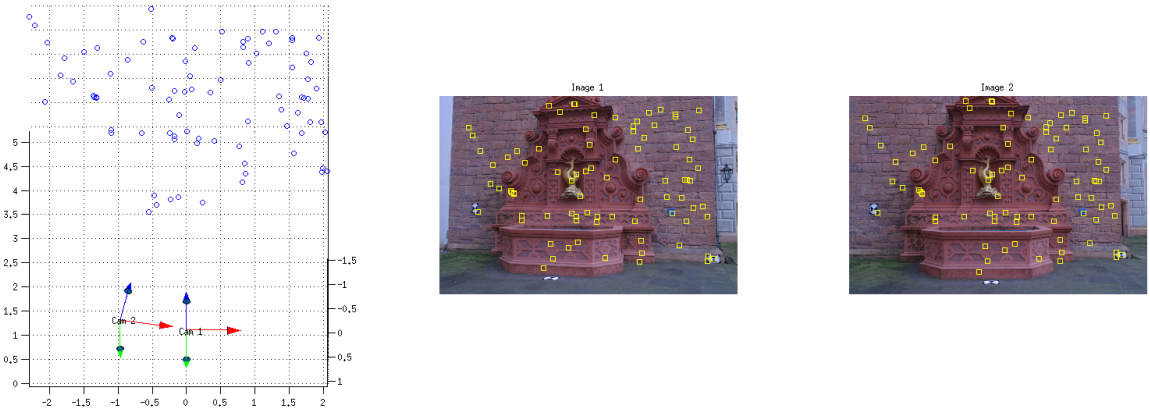


Figure 5: 3-D scene (sparse points and camera poses) and point correspondences.

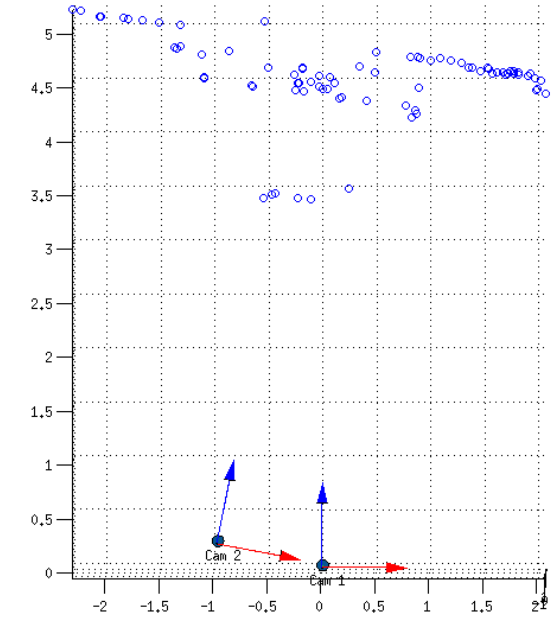


Figure 6: Top view of the reconstructed scene.

4 Numerical Exercises

1. Prove that for a matrix $R \in SO(3)$ and $a, b \in \mathbb{R}^3$

$$R(a \times b) = (Ra) \times (Rb)$$

2. Suppose we know the camera always moves (rotation R and translation t) in a plane parallel to the image plane. Show that

   - The essential matrix $E = [T_\times]R$ is of special form

$$E = \begin{bmatrix} 0 & 0 & a \\ 0 & 0 & b \\ c & d & 0 \end{bmatrix}, \qquad a, b, c, d \in \mathbb{R}$$

     where the cross-product matrix $[T]_\times$ associated with the vector T is defined as shown below 1.1.

   - Without using the SVD-based decomposition, find a solution to $[R|T]$ in terms of (a, b, c, d).

3. Show that for a line l in the image plane defined as the set of points in homogeneous coordinates $\{x \in \mathbb{P}^2 | x^\top l = 0\}$, the squared distance of a point $y \in \mathbb{R}^2$ to the line in the image plane is the following:

$$d^2 = \frac{(\tilde{y}^\top l)^2}{||[e_3]_\times l||^2}$$

   where $\tilde{y} = (y^\top, 1)^\top$ and e_3 is the unit vector in z-direction.

4. Consider a given essential matrix E and two matched points x_1 and x_2. In the noise-free case where the matched points x_1 and x_2 are perfect and the essential matrix is correct the point x_2 lies exactly on the epipolar line computed from E and x_1, so the distance of x_2 to the epipolar of x_1 is zero. However as soon as the matched points x_1 and x_2 are not noise-free this distance is non-zero.

   In the programming exercise you have used the function `distPoint2EpipolarLine`. Prove that the quantity d calculated by this function

$$d(x_1, x_2)^2 = \frac{(x_2^\top E x_1)^2}{||[e_3]_\times E x_1||^2}$$

   corresponds to the squared distance of a matched point to the epipolar line.