The Perspective-n-Point (PnP) problem

Contents

The goal of this laboratory session is to implement the Direct Linear Transform (DLT) algorithm to estimate the pose of a camera, given a set of 2D-3D correspondences.

1 Preliminaries

1.1 Outline of the exercise

You will be given a dataset of images, with a set of 2D-3D correspondences for each image, as well as the camera matrix K. Your goal will be to implement the DLT algorithm described during the course to estimate the camera pose $[R|t]$ for each image.

1.2 Description of the input data

The ***data/*** folder contains the inputs that you will need to complete these exercises.

- ***images_undistorted/*** contains images that have been processed to compensate for lens distortion (using the function `undistortImage` from the previous exercise). This allows you to completely ignore the effect of lens distortion when solving the PnP problem.
- ***K.txt*** contains the camera matrix
- ***p_W_corners.txt*** contains the positions of the n reference 3D points (illustrated in Figure 1), given in the world coordinate system (defined below), in centimeters. $n = 12$ in this exercise.
- ***detected_corners.txt*** contains m lines (where m is the number of images). Each line i gives the 2D coordinates $p_i = (u_i, v_i)$ of the projections of the reference 3D points in the undistorted image i given as a tuple: $(u_1, v_1, ..., u_n, v_n)$.

1.3 Notations and coordinate systems

In this exercise, we use the following conventions:

- $\mathbf{P}_A$ denotes that the point $\mathbf{P}$ is expressed in the coordinate frame A.
- ${}^BT_A$ denotes the transformation that maps points in frame A to frame B, such that:

$$\mathbf{P}_B = {}^BT_A\mathbf{P}_A$$

The reference (or world) coordinate system, denoted W, is right-handed, and illustrated in Figure 1.

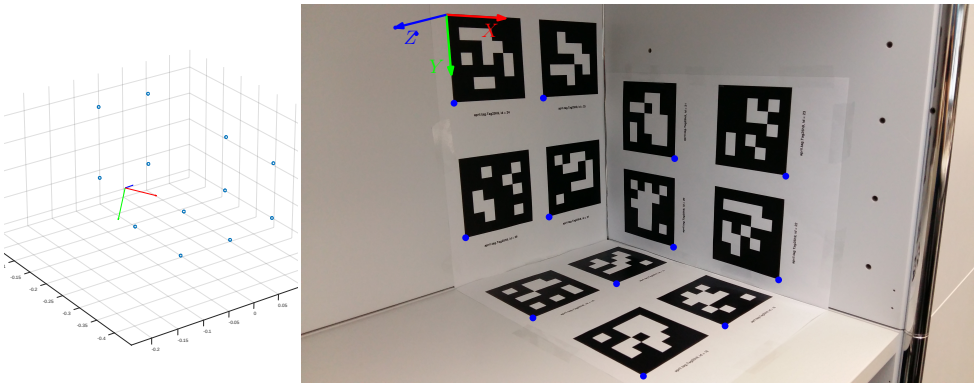

Figure 1: Left: 3D world points P_w^i and the camera pose in space. Right: Image from a camera with the corresponding 2D projections p_i marked in blue.

2 Implementing DLT

In this section, you will implement the DLT algorithm to determine the camera pose for each image in the dataset, using the 2D-3D correspondences provided for each image.

The DLT algorithm is described in the lecture slides. We first briefly remind here how it works (and define the notations that we use).

2.1 Reminder: The DLT algorithm.

2.1.1 Derivation of the linear system of equations to solve

As opposed to the lecture slides where the camera calibration matrix K was not given (and therefore had to be estimated jointly with the camera pose R, t), in this exercise K is given. We briefly derive here a slightly modified version of the DLT algorithm presented in the course that takes into account the fact that K is known.

Our goal is to compute R, t that satisfy the perspective projection equation:

$$\begin{bmatrix}\tilde{u}\\ \tilde{v}\\ \tilde{w}\end{bmatrix} = \lambda \begin{bmatrix}u\\ v\\ 1\end{bmatrix} = K[R|t] \begin{bmatrix}X_w\\ Y_w\\ Z_w\\ 1\end{bmatrix}$$

Multiplying each side by K^{-1} on the left, we get:

$$\lambda K^{-1} \begin{bmatrix}u\\ v\\ 1\end{bmatrix} = [R|t] \begin{bmatrix}X_w\\ Y_w\\ Z_w\\ 1\end{bmatrix}$$

where we can identify $[x, y, 1]^T = K^{-1}[u, v, 1]^T$ as being the *normalized coordinates* (sometimes also called *calibrated coordinates)* corresponding to pixel (u, v). Denoting $\tilde{M} = [R|t]$ the *projection matrix* (for normalized coordinates), the problem amounts to finding $\tilde{M}$ and the scale factors λ_i that satisfy:

$$\lambda_i \tilde{p}_i = \tilde{M} P_i$$

for every 2D-3D correspondence $i = 1, \ldots, n$, where $\tilde{p_i} = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$ and $P_i = \begin{bmatrix} X_w^i \\ Y_w^i \\ Z_w^i \\ 1 \end{bmatrix}$ are respectively the i^{th} corresponding 2D and 3D points.

However, as it was shown in the lecture, by considering ratios of pairs of equations (such as $\frac{\tilde{u_i}}{\tilde{w_i}} = \frac{\tilde{m}_1^T \cdot P_i}{\tilde{m}_2 \cdot P_i}$), the scale factors λ_i are cancelled, i.e. removed from the equations and the problem reduces to finding $\tilde{M}$ alone. The scale factors can be recovered, if desired, once $\tilde{M}$ is known. It is shown in the lecture that $\tilde{M}$ can be recovered by solving the following homogeneous system of linear equations for the unknown $[12 \times 1]$ matrix $\tilde{M}$:

$$Q \cdot \tilde{M} = 0 \tag{1}$$

where

$$Q = \begin{bmatrix} X_w^1 & Y_w^1 & Z_w^1 & 1 & 0 & 0 & 0 & 0 & -x_1 X_w^1 & -x_1 Y_w^1 & -x_1 Z_w^1 & -x_1 \\ 0 & 0 & 0 & 0 & X_w^1 & Y_w^1 & Z_w^1 & 1 & -y_1 X_w^1 & -y_1 Y_w^1 & -y_1 Z_w^1 & -y_1 \\ & & & & & & \cdots & \cdots & \cdots & & & \\ X_w^n & Y_w^n & Z_w^n & 1 & 0 & 0 & 0 & 0 & -x_n X_w^n & -x_n Y_w^n & -x_n Z_w^n & -x_n \\ 0 & 0 & 0 & 0 & X_w^n & Y_w^n & Z_w^n & 1 & -y_n X_w^n & -y_n Y_w^n & -y_n Z_w^n & -y_n \end{bmatrix}$$

can be built from the known 2D-3D point correspondences and

$$\tilde{M} = \begin{bmatrix} \tilde{m}_{11} & \tilde{m}_{12} & \tilde{m}_{13} & \tilde{m}_{14} & \tilde{m}_{21} & \tilde{m}_{22} & \tilde{m}_{23} & \tilde{m}_{24} & \tilde{m}_{31} & \tilde{m}_{32} & \tilde{m}_{33} & \tilde{m}_{34} \end{bmatrix}^T$$

is an unknown projection matrix which we wish to recover. *Hint:* You may use Matlab's function `kron` to build Q (although that is not strictly necessary). Note that $\tilde{M}$ here is a $[12 \times 1]$ vector obtained by unrolling the matrix:

$$\begin{bmatrix} \tilde{m}_{11} & \tilde{m}_{12} & \tilde{m}_{13} & \tilde{m}_{14} \\ \tilde{m}_{21} & \tilde{m}_{22} & \tilde{m}_{23} & \tilde{m}_{24} \\ \tilde{m}_{31} & \tilde{m}_{32} & \tilde{m}_{33} & \tilde{m}_{34} \end{bmatrix}$$

in a *row-wise* fashion, i.e. by unrolling it row by row. Keep that in mind when you later convert $\tilde{M}$ back to a $[3 \times 4]$ matrix, since Matlab usually reshapes matrices in a *column-wise* fashion.

2.1.2 Solving the over-determined system of equations

Since Q should have rank 11, and each 2D-3D point correspondence provides 2 independent equations, at least 6 point correspondences (in general position, thus avoiding degenerate configurations, such as all 3D points lying on a plane) are needed. In this exercise, $n = 12$ point correspondences are provided, thus the system of equations is over-determined. The trivial solution $\tilde{M} = 0$ is obviously of no interest for us. We can further observe that if $\tilde{M}$ is a solution of (1), then $\alpha \cdot \tilde{M}$ is also a solution (for any scalar α; thus we will recover the projection matrix $\tilde{M}$ up to an unknown scale factor). So, we look for a solution that minimizes $||Q \cdot \tilde{M}||$ subject to the constraint $||\tilde{M}|| = 1$. This can be done using the Singular Value Decomposition (SVD) of Q: $Q = USV^T$ where U,V are unitary matrices and S is diagonal.

It can be shown that the solution of this problem is the eigenvector corresponding to the smallest eigenvalue of $Q^T Q$, which simply corresponds to that the last column of V if S has its diagonal entries sorted in descending order. The `svd` function from Matlab provides such a guarantee.

2.1.3 Extracting R,t from $\tilde{M}$ with the correct scale

Enforcing $\det R = 1$ After solving the linear system (1), you will need to convert back the $[12 \times 1]$ vector $\tilde{M}$ to its corresponding $[3 \times 4]$ projection matrix $\tilde{M} = [R|t]$. Make sure that the z component of the recovered translation: $t_z = \tilde{m}_{34}$ is positive. If that is not the case, you will need to multiply $\tilde{M}$ by -1. This is to ensure that the rotation matrix R that will be extracted from $\tilde{M}$ is a *proper* rotation matrix with determinant $+1$.

Extracting a rotation matrix from R When we solved the system of linear equations (1), we did not impose any constraint on R to ensure it is actually a rotation matrix (i.e. we have no guarantee that $R \in SO(3)$, the space of rotation matrices). This means what we actually estimated is an approximation $\bar{R}$. In this step, we will extract a true rotation matrix $\tilde{R} \in SO(3)$ from our current estimate R. To do that, we will compute the matrix $\tilde{R} \in SO(3)$ which is the closest to R (in the sense of the Frobenius norm). This is known as the Orthogonal Procrustes Problem. $\tilde{R}$ can be obtained by first decomposing R using the SVD: $R = U\Sigma V^T$, and then forcing all the eigenvalues to be 1 as follows: $\tilde{R} = UIV^T = UV^T$.

Recovering the scale of the projection matrix $\tilde{M}$ As explained above, by solving the system of linear equations above we can only compute $\tilde{M}$, up to a scale. Thus the true projection matrix must be $\tilde{M} = [\tilde{R}|\tilde{t}] = [\alpha R|\alpha t]$, where α is an unknown scale factor and $[R|t]$ are the values computed from DLT. In the previous step, we have computed the nearest (true) rotation matrix $\tilde{R}$ from the given matrix R, which can be used to improve our estimate of the projection matrix: $\tilde{M} = [\tilde{R}|\alpha t]$. The projection of R on $SO(3)$ implicitly recovered the unknown scale factor α by ensuring that R is an orthogonal matrix. We can take advantage of this to explicitly estimate α from $\tilde{R} = \alpha R$. The α which comes closest to solving this is: $\alpha = \frac{||\tilde{R}||}{||R||}$ where $||A||$ is any matrix norm of A - which can be computed in Matlab using the command `norm(A)`.

Wrapping up Finally, the final projection matrix can be recovered as: $\tilde{M} = [\tilde{R}|\alpha t]$.

2.2 Implementation of DLT

- Write a function `M = estimatePoseDLT(p, P, K)` that implements the steps of the DLT algorithm as described above, to solve for the projection matrix $\tilde{M} = [R|t]$, given the n 2D-3D point correspondences p_i and P_i. **Pay a particular attention to the fact that the matrix Q is built using the *calibrated coordinates* $\tilde{p}_i$ and not directly the pixel coordinates p_i.** After you have computed the $[12 \times 1]$ matrix $\tilde{M}$, you will need to reshape it into a $[3 \times 4]$ matrix. You can use the `reshape` function from Matlab to achieve this. Pay attention, however, that this function works *column-wise* and not *row-wise*, so will need to first reshape $\tilde{M}$ to a $[4 \times 3]$ matrix and then take its transpose. *Check* that the resulting rotation R is a valid rotation matrix (i.e. $\det R = 1$ and $R^T R = I$).

- Write a function `[p_reprojected] = reprojectPoints(P, M, K)` that reprojects the 3D points P_i in the current image using the estimated projection matrix $\tilde{M}$ and camera matrix K. Check that the reprojected points p'_i fall close to the points p_i. The output should look like Figure 2.

- Write a function that estimates the camera pose (using your implementation of DLT) for each image in the dataset, and create an animation which shows the motion of the camera. You may use the provided function `plotTrajectory3D` and the utility function `rotMatrix2Quat`. You can use a framerate of 30 frames per second. *Warning:* the function `plotTrajectory3D` expects to be given the transformation $[{}^W R_C | {}^W t_C]$ that maps points from the camera coordinate frame to the world coordinate frame. In this exercise, you actually estimated its inverse: $\tilde{M} = [{}^C R_C | {}^C t_C]$. Do not forget to inverse $\tilde{M}$ to get the correct transformation matrix before feeding it into `plotTrajectory3D`. Finally, **the units (for the translational part and the 3D points) should be meters (whereas you worked with centimeters throughout this exercise)**. Do not forget to convert to meters before calling the function.

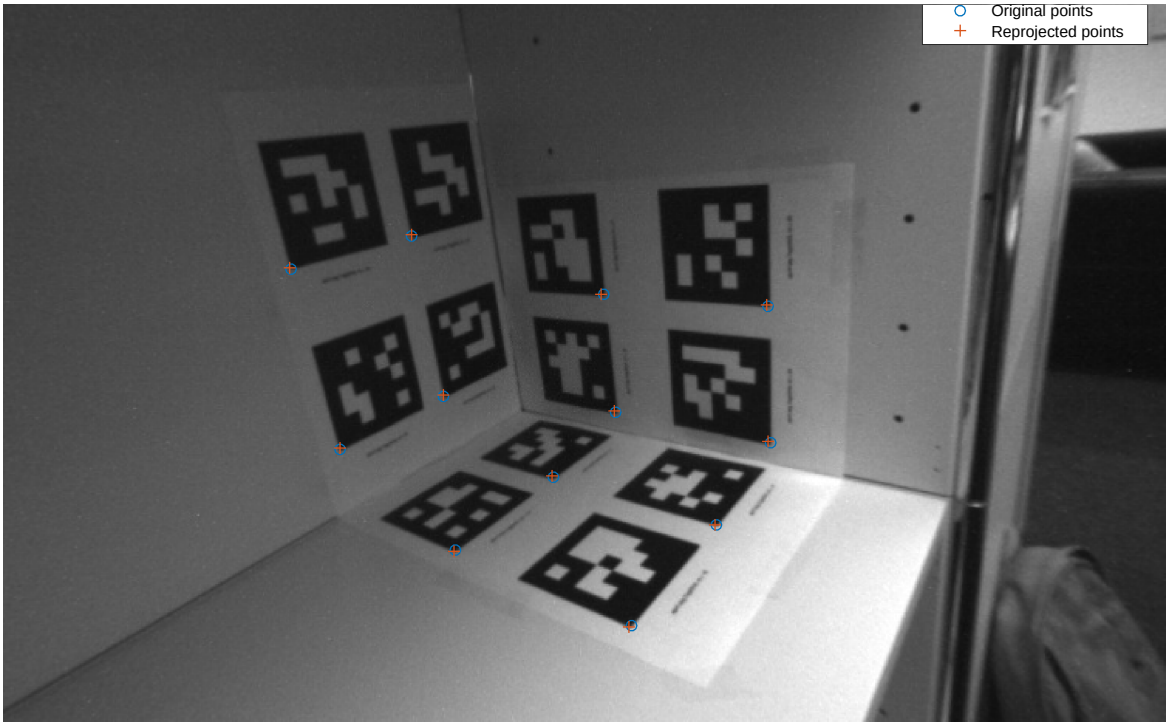

Figure 2: Original 2D points p_i and reprojected points p_i' using the estimated R,t