ENV-408: Ex3-Absolute Orientation (Camera Pose)

Jan Skaloud

05/03/2025

Objectives

Understand how to orient a single image in space, i.e. determine its *pose* (position and attitude) from image observations of points with known coordinates in object (*world*) space.

Methodology: Given a set of n-points in undistorted perspective-centered coordinates (**Lab 01**) and their corresponding ground coordinates (mapping frame) implement the DLT algorithm in the simplified setup where camera calibration parameters (c, c_x, c_y and thus K) are known. It will be used to estimate the camera *pose* $\Pi = [R, t]$.

Overview

Input data

1. `gcps.txt`: mapping coordinates (`No  X  Y  Z`) of Ground Control Points (GCPs), units: meters
2. `cam_param.txt`: image dimensions, camera constant and coordinates of perspective point (c, c_x, c_y), units: pixels, as well the unitless distortion coefficients (k_1, k_2, p_1, p_2).
3. `id_xy_corrected.txt`: undistorted image coordinates (`No  x  y`) of GCPs, unitless in perspective-centered coordinate system (you can use provided or your own output from **Lab 01** or provided measurements.

Functions to implement:

- `Q = build_Q(p, P)`: builds the system of equations to be solved by the DLT algorithm
- `PI_prime = estimatePoseDLT(Q)`: estimates the camera rotation matrix and camera translation vector that fit as much as possible the perspective projection for the set of points.
- `r = reprojectPoints(P, PI_prime)`: re-projects 3D points P_i to image plane using estimated projection matrix `PI_prime`.

Notation and coordinate systems

- P^A denotes the point P expressed in the coordinate frame A
- T_A^B dentotes the transformation that maps points in frame A to frame B, such that: $P^B = T_A^B \cdot P^A = [R|t]_A^B \cdot P^A$

Task 1 : Building DLT's system of equations

Formulation

The DLT algorithm is presented in the lecture (slides). Here we consider the simplified case that **K** is (approximately) known, as would be the case for a pre-calibrated camera.

Our goal is to estimate R and t (rotation matrix & translation vector) that satisfy the perspective projection:

$$f_{undistort}\left(\mathbf{K}^{-1}\begin{bmatrix}u'\\v'\\1\end{bmatrix}\right)=\begin{bmatrix}x\\y\\1\end{bmatrix}=\frac{1}{\mu}[\mathbf{R}|\mathbf{t}]^c_m\cdot\begin{bmatrix}X^m\\Z^m\\Z^m\\1\end{bmatrix} \tag{1}$$

Note that using the **topleft2perspective** function to convert (u,v) from top left to (x,y) perspective centered coordinate, you implicitly used K. Denoting the *projection matrix* for normalized image coordinates $\Pi=[\mathbf{R}|\mathbf{t}]^c_m$, the problem reduces to finding $\hat{\Pi}$ and scale factors μ_i satisfying:

$$\mu_i p_i = \Pi P_i$$

For each 2D-3D correspondence $i=1,\dots,n$, where $p_i=(x,y,1)^T$ and $P_i=(X^m_i,Y^m_i,Z^m_i,1)^T$ are the respective i^{th} image & object point *homogeneous* coordinates.

As shown in the lecture, the scale factors μ_i can be canceled by dividing the first two equations by the 3rd, e.g. $\frac{\mu_i x_i}{\mu_i}=\frac{m_1^T\cdot P_i}{m_3^T\cdot Pi}$; and afterwards the problem reduces to finding Π alone[1]. Π can be estimated by stacking its *rows* into a (12×1) vector:

$$\texttt{vec}(\Pi)=\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} & m_{21} & m_{22} & m_{23} & m_{24} & m_{31} & m_{32} & m_{33} & m_{34}\end{bmatrix}$$

and solving the following homogeneous system of linear equations:

$$Q\cdot\texttt{vec}(\Pi)=0$$

where

$$Q=\begin{bmatrix}
X^m_1 & Y^m_1 & Z^m_1 & 1 & 0 & 0 & 0 & 0 & -x_1X^m_1 & -x_1Y^m_1 & -x_1Z^m_1 & -x_1\\
0 & 0 & 0 & 0 & X^m_1 & Y^m_1 & Z^m_1 & 1 & -y_1X^m_1 & -y_1Y^m_1 & -y_1Z^m_1 & -y_1\\
 & & & & \cdots & \cdots & \cdots & & & & & \\
X^m_n & Y^m_n & Z^m_n & 1 & 0 & 0 & 0 & 0 & -x_nX^m_n & -x_nY^m_n & -x_nZ^m_n & -x_n\\
0 & 0 & 0 & 0 & X^m_n & Y^m_n & Z^m_n & 1 & -y_nX^m_n & -y_nY^m_n & -y_nZ^m_n & -y_n
\end{bmatrix} \tag{2}$$

Implementation

Input

1. Measurements normalized undistorted coordinates, Nx2 array : $p_i=\begin{bmatrix}x\\y\end{bmatrix}$
2. Measurements GCPs 3D coordinates (shifted) Nx3 array : $P_i=\begin{bmatrix}X\\Y\\Z\end{bmatrix}$

[1]The scale factors can be recovered once Π is determined.

Steps

1. Express P in homogeneous coordinates

$$P_i = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \Rightarrow P_i = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

2. Build the Q matrix as expressed in Eq. 3 above from p and P

   Hint : Using np array advanced indexing and broadcasting will help a lot, see below

   Example 1 : `Q[ i:j:k, :]` allows to access the lines i to j every k lines.

   You might consider using it when building Q matrix columns 0 to 7

   Example 2 : Broadcasting line array with column array

   `col` is a Nx1 column array, `row` is a 1xM array. By doing `col*row`, you will end up with a NxM matrix corresponding to the matrix multiplication of the two array. You might consider this when building Q matrix columns 8 to 11

Output

`Q matrix` 2*N x 12 matrix of the DLT algorithm

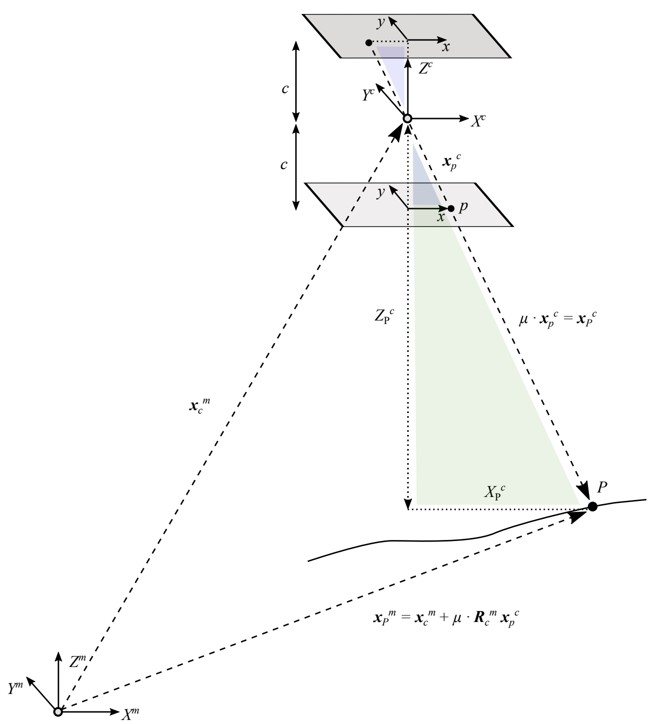

Figure 1: The goal is to estimate the camera pose x_c^m and R_m^c from n pairs $[p_i - P_i]$ of 2D-3D *point correspondences*, via the PnP (DLT) solver.

Task 2 : Solving the over-determined system of equations and extracting $\boldsymbol{R}$, $\boldsymbol{t}$ from Π

Formulation

Each 2D-3D correspondence provides 2 independent equations, we need at least 6 points to resolve the twelve unknowns $m_{i,j}$.

However, the distribution of points has to avoid degenerate configurations (such as all points lying on a plane), in other words, Q needs to be at least of rank 11 (solution up to an unknown scale factor).

Ideally, the system is over-determined ($n > 6$) and we look for a solution that minimizes $||Q \cdot \Pi||$ subject to the constraint $||\Pi|| = 1$. This constraint can be enforced by Singular Value Decomposition (SVD) of Q: $Q = USV^T$, where U, V are unitary matrices and S is diagonal.[2]

1. The best (approximate) solution $\texttt{vec}(\tilde{\Pi})$ is the eigen-vector corresponding to the smallest eigen-value of $Q^T Q$, i.e. the last column of V if S has its diagonal entries sorted in descending order. After solving the linear system $Q \cdot \texttt{vec}(\Pi)$, the obtained vector $\texttt{vec}(\tilde{\Pi})'(12 \times 1)$ must be reshaped to the projection matrix $\tilde{\Pi}(3 \times 4) = [\tilde{R}|\tilde{t}]$.
2. Because SVD operation does not preserve the sign of its solution, we must ensure that the sign of $\tilde{\Pi}$ is correct. Since the object are lying in front of the camera and since the camera axis points forward, the translation vector must be positive. Inspect the z component of the recovered translation: $t_z = \tilde{\Pi}_{34}$ and ensure that it is positive. If not, you need to multiply $\tilde{\Pi}$ by (-1).

In summary after this operation we obtain the approximate translation vector $\tilde{t}$ as the last column of Π: $\tilde{t} = \Pi(:, 4)$, while the first three columns of Π correspond to the approximated rotation matrix $\tilde{R}$.

3. Elements in Q are built from observations affected by errors, and nothing ensures that $m_{i,j}$ with $(i, j) = 1, \cdots, 3$ have the properties of a rotation matrix. To guarantee that $R \in SO(3)$[3] we want to extract the true rotation matrix $\hat{R}$ from $\tilde{R}$, which is the closest matrix in the sense of the Frobenius norm, with all eigen values equal to one. The $\hat{R}$ matrix can be found by doing the SVD decomposition of $\tilde{R}$, and forcing all eigen values to one. The estimation of $\hat{R}$ follows : $\hat{R} = UIV^T = UV^T$
4. The applied solution of $Q \cdot \texttt{vec}(\Pi)'$ provides the projection matrix up to a scale, i.e. its approximation $\hat{\Pi}' = [\hat{R}|\hat{t}] = [\mu\tilde{R}|\mu\tilde{t}]$. The nearest (true) rotation matrix $\hat{R}$ was obtained from its approximate $\tilde{R}$, which implicitly recovered the unknown scale factor μ when ensuring $\hat{R} \in SO(3)$. One can take advantage of the relation $\hat{R} = \mu\tilde{R}$, to estimate $\mu = \frac{||\hat{R}||}{||\tilde{R}||}$, where $|| \cdot ||$ is any matrix norm, e.g. the Frobenius norm.

Implementation

Input

`Q matrix` 2*N x 12 matrix of the DLT algorithm

Steps

1. Perform the SVD decomposition of Q to obtain $\texttt{vec}(\tilde{\Pi})$ and reshape $\texttt{vec}(\tilde{\Pi})$ into $\tilde{\Pi}$ of shape (3x4)

   Hint `numpy.linalg.svd()`[4] will perform the svd with **S** values sorted by descending order. `np.reshape` is usefull to recover the matrix form
2. **Enforce** $det(R) = 1$ **property:** Implement an **if condition** that multiply $\tilde{\Pi}$ by -1 if $t_z = \tilde{\Pi}_{34}$ <0
3. **Extract rotation matrix** R**:** Perform SVD$(\tilde{R}) = U\Sigma V^T$. You can then estimate $\hat{R} = UIV^T = UV^T$ which is equivalent to forcing $\tilde{R}$ eigenvalue to one.

[2] Note that under the orthogonality properties of U, V matrices, the decomposition of Q is sufficient if we want to recover last column of V, as described in the Step 1, because the SVD$(Q^T Q) = V^T S^T U^T USV = V^T S^T SV$ yields the same answer.

[3] The space of rotation matrix in 3D is the special orthogonal group of $dim = 3$.

[4] In Matlab: `[U,S,V]=svd(Q'*Q)`.

4. **Recover the scale μ:** The scale is defined by $\mu = \frac{||\hat{R}||}{||\tilde{R}||}$, where $||\cdot||$ is any matrix norm, such as the Frobenius norm.

   Hint `numpy.linalg.norm()` using Frobenius norm is available to you for this task.[5]

Output

$\hat{\Pi} = [\hat{R}|\hat{t}]$, 3x4 matrix. The rotation matrix and translation vector that projects 3D points into the image plane

Check that the resulting R (within `PI_prime`) is a valid rotation matrix (i.e. $det(R) = 1$ and $R^T R = I$).

> **Hint** .T (matrice transpose operator) and np.det() function might prove useful.

Task 3 : Reproject 3D coordinates into the image

Thanks to the extraction of $\hat{\Pi} = [\hat{R}|\hat{t}]$, it is now possible to reproject any 3D point into the image plane.

1. Define `rp = reprojectPoints(P, PI_prime)` that re-projects the 3D points P_i to the image space using the estimated projection matrix `PI_prime`. Check that the re-projected points $p_{rep,i}$ are close to the points p_i.

The `plot_reprojection_error` and `plot_reprojection_error_norm` have been implemented for you. You can use them to estimate the reprojection error of your 3D points with respect to the initial values. The camera position and orientation is then calculated from your estimated $[\hat{\mathbf{R}}|\hat{\mathbf{t}}]_m^c$.

Your image is now oriented :)

> Note: The camera position error with the full set of provided point should lead to an error below 1m.

Task 4 : Numerical analysis

Calculate the camera poses and plot the re-projection error to simulate fewer GCPs, degenerate case and faulty measurements:

1. Full set of points
2. A minimum set of points:
   - `1092311568.jpg: [140,142,143,147,150,151]`
3. A subset of points (de-generate case):
   - `1092311568.jpg: [143,144,146,147,149,150]`
4. A faulty measurement:
   - `+10 pixels on (u,v) coordinates of GCP 149`

Complete the following table with your results:

Configuration	Mean rep. error (pix)	Max. rep. error (pix)	Cam Pos. X(m)	Cam Pos. Y(m)	Cam Pos. Z(m)
1. Full set					
2. Min. set					
3. Subset					
4. Faulty GCP					

[5] In Matlab `norm()`.

Discussion

Answer the following questions:

1. **Impact**. Do you consider the differences (i.e. in the obtained camera position and/or re-projection error) between the three cases significant? Justify your opinion.

2. **Cause**. According to your opinion what is (are) the main factor(s) affecting the differences between the cases 1.-2. and 1.-3.?