Prof. Dr. Maren Bennewitz
M.Sc. Stefan Oßwald
M.Sc. AbdElMoniem Bayoumi

Bonn, 27 April 2017

Humanoid Robots
Exercise Sheet 2 - Odometry Calibration and Projective Geometry

Note: Due to changes in the schedule of the lecture, you have two weeks to complete this exercise sheet. Deadline: May 11.

Exercise 3 (20 points)

Mobile robots typically execute motion commands only inaccurately due to slippage on the ground, uneven terrain, or hardware issues. For example, if one of the knee joints of a humanoid robot is weaker than the other one, then it will walk on a circular trajectory although it intends to walk straight ahead. In order to account for such systematic errors, odometry calibration can be applied to learn and correct the drift.

In the lecture a system was introduced for calibrating odometry using a least squares approach.

The repository contains the following files:

- `src/03_odometry_calibration/data/calib.dat` contains odometry data recorded by a real robot. The first three columns contain the odometry data $(u_x^\star, u_y^\star, u_\theta^\star)$ measured with an external tracking system and the last three columns contain the odometry (u_x, u_y, u_θ) measured by the robot. θ is measured in radians counterclockwise.
- `src/03_odometry_calibration/src/OdometryCalibration.cpp` contains the functions that you have to implement.
- `src/03_odometry_calibration/include/odometry_calibration/CalibrationData.h` contains the data structures for the odometry, calibration data, and 2D poses.

The `main` function already loads the data file into the `MeasurementData` data structure (i.e., both the ground truth odometry data observed externally, and the robot's observed odometry).

Exercise steps:

a) Implement `errorFunction(groundTruth,observation,calibrationMatrix)`, which calculates the error $\mathbf{e}_i(x)$ between the ground truth odometry $\mathbf{u}_i^\star$ and the corrected observed odometry (which is corrected by the current estimate of the calibration matrix).

b) Calculate the Jacobian $\mathbf{J}_i$ of the error function for a given odometry measurement by implementing `jacobian(observation)`. The Jacobian is defined as

$$\mathbf{J}_i := \frac{\partial \mathbf{e}_i\left(\mathbf{x}\right)}{\partial \mathbf{x}}.$$

c) Calculate the calibration matrix by implementing `calibrateOdometry(measurements)`. The function uses the loaded data passed as a parameter and should return the calibration matrix. In our case, the weight matrix $\mathbf{\Omega}_i$ is the identity matrix. One iteration is sufficient, because the error function is linear.

d) Use the calibration matrix (computed in the previous step) to correct the robot's odometry observations in `applyOdometryCorrection(uncalibratedOdometry,calibrationMatrix)`.

e) Compute the robot's trajectory based on the corrected odometry observations by implementing `calculateTrajectory(calibratedOdometry)`:

   1) Assume that the robot starts at the position $(x, y, \theta) = (0, 0, 0)$.
   2) Transform the robot's pose using the corrected odometry by implementing `odometryToAffineTransformation(odometry)` (Check the lecture slides 34–35 for more information about affine transformations).
   3) Chain the affine transformation to get the next pose. (See lecture slides 36–38).
   4) Convert the chained affine transformation back to a robot pose by implementing `affineTransformationToPose(transformation)`.
   5) Store the pose in the trajectory vector.

When you push your code to the server, the server will plot a figure and save it to the result page in the Wiki. The figure shows the calibrated trajectory computed by your program in blue. Compare it to the ground truth trajectory (in red). The trajectories should match approximately, but there will still be a small error accumulating over time.

If you have Gnuplot installed, you can generate the same figure on your computer using the `scripts/plot.gp` script (see the Wiki for instructions).

Exercise 4 (20 points)

You install a surveillance camera on a flag pole in front of the building. The camera's data sheet specifies the following parameters:

principal point	$\begin{pmatrix} x_H \\ y_H \end{pmatrix} = \begin{pmatrix} 400 \\ 300 \end{pmatrix}$
camera constant	$c = 550$
scale difference	$m = 0.0025$

Let the origin of the world coordinate system be at the bottom of the flag pole. The projection center of the camera is located $Z_0 = 10\,\text{m}$ above the ground and $X_0 = 40\,\text{cm}$ in front of the flag pole in X direction.

The camera can be rotated vertically in the (Z, X) plane around its projection center. Let α be the current rotation angle in radians. The rotation matrix is then

$$R = R_2(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}.$$

a) Implement the function `euclideanToHomogeneous` that converts Euclidean coordinates in 3D to homogeneous coordinates.

b) Implement the function `homogeneousToEuclidean` that converts the homogeneous coordinates back to Euclidean coordinates. You may assume that the last component of the homogeneous coordinates is $\neq 0$.

c) Implement the function `setCameraParameters` that returns a data structure that contains the camera parameters.

d) Implement the function `calibrationMatrix` that returns the calibration matrix K for the camera parameters given as an argument.

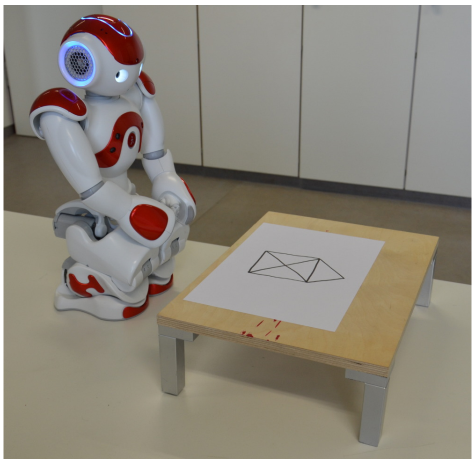
e) Implement the function `projectionMatrix` that returns the projection matrix P given the calibration matrix and the rotation angle α.

f) Implement the function `projectPoint` that projects a point in 3D coordinates to image coordinates given the projection matrix P.

Reprojection from image to 3D

In the lecture, we discussed how to project a given point in 3D space onto a camera image. Many robotics applications require the reverse: The robot detects a feature in the camera image, for example an object that the robot should grasp. It then has to compute the corresponding point in 3D space so that it can move its hand there to grasp the object. Unfortunately, projecting from 3D to the 2D image plane removes one dimension, so reconstructing the 3D point is only possible if an additional piece of information is available, for example the size of the object or a second image from a slightly different perspective.

In the following example, the Nao robot observes a drawing on a table and it is supposed to compute the 3D coordinates of the corners of the house in the drawing. The robot knows that the height of the table is h, so all points of the drawing will have $Z = h$ in world coordinates. By exploiting this piece of information, it is possible to reconstruct the full 3D world coordinates of any point on the drawing.


(a) Scene

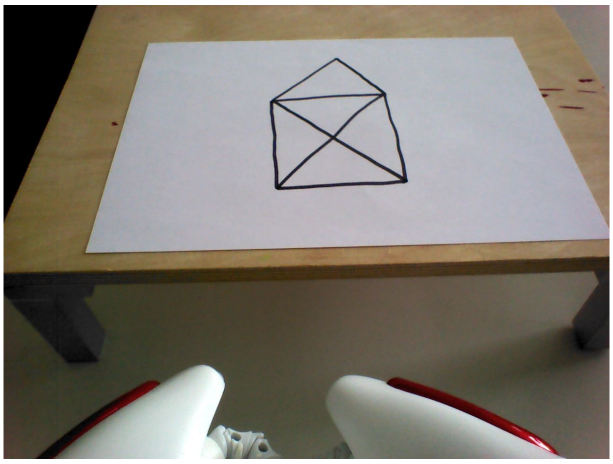

(b) Robot's view

Figure 1: Nao robot playing a board game.

g) Implement the function `reprojectTo3D`. Given an input point in image coordinates, the camera parameters, and the table height in meters, the method should calculate and return the corresponding 3D point. *Hint*: Start with the camera equation

$$ {}^s\mathbf{x} = \mathbf{K} \cdot \mathbf{R} \cdot [\mathbf{I}_3 \mid -\mathbf{X}_O] \cdot \mathbf{X} \tag{1}$$

$$\Leftrightarrow \begin{pmatrix} {}^s x \\ {}^s y \\ 1 \end{pmatrix} = \mathbf{K} \cdot \mathbf{R} \cdot \begin{pmatrix} 1 & 0 & 0 & -X_O \\ 0 & 1 & 0 & -Y_O \\ 0 & 0 & 1 & -Z_O \end{pmatrix} \cdot \begin{pmatrix} U \\ V \\ W \\ T \end{pmatrix}. \tag{2}$$

There are three equations for the four unknown components U, V, W, T of the 3D point $\mathbf{X}$ in homogeneous coordinates, so the system is underdetermined. Additionally, the table height is given, which yields the equation

$$\frac{W}{T} = h. \tag{3}$$

Solve Eq. 3 for T, substitute T in Eq. 2 and rewrite the equation in the form

$$\begin{pmatrix} {}^s x \\ {}^s y \\ 1 \end{pmatrix} = \mathbf{K} \cdot \mathbf{R} \cdot \mathbf{A} \cdot \begin{pmatrix} U \\ V \\ W \end{pmatrix} \tag{4}$$

where $\mathbf{A}$ is a 3×3 matrix. Solve the linear system, compute T and convert back from homogeneous to Euclidean coordinates.

Deadline: 11 May 2017, 11:59 am