Computer Vision Lab

Labs 1-2: Calibration and pose computation

1 Content of this lab

The goal is this lab is to perform camera calibration and pose computation. Do to so, we will use two C++ libraries:

- OpenCV[1] to read images from files or camera, and perform low-level image processing
- ViSP[2] to manipulate 3D points, camera pose and use vectors and matrices.

The actual classes that are used are detailed in Appendix A.

1.1 The cameras

Several IEEE-1394 Firewire cameras are available for this lab. Some of them have a low distortion while others are fish-eye cameras. We will consider the same framework to calibrate and perform pose computation for both, that is virtual visual servoing (VVS).

1.2 Calibration landmarks

Camera calibration and pose computation usually require to observe known objects. In our case we will consider a grid composed of 6×6 points and the OpenCV chessboard (see Fig. 1).

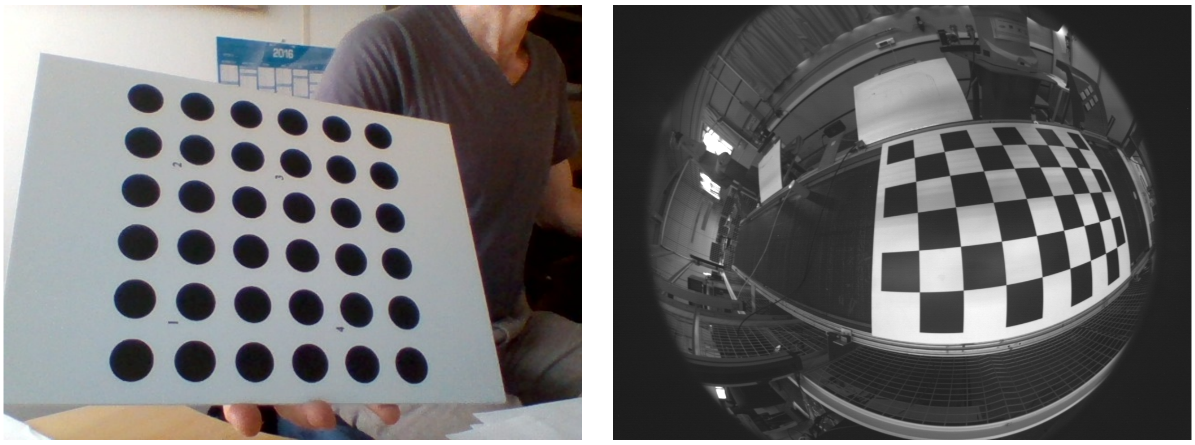

Figure 1: Calibration landmarks, point grid with a low distortion camera (left) and OpenCV chessboard seen by a fish-eye camera (right).

In both cases, the interest points lie on a grid pattern of known dimension (rows, columns and inter-point distance). The 3D coordinates of those points are thus known. The goal is to

[1]Open source Computer Vision, http://opencv.org
[2]Visual Servoing Platform, http://visp.inria.fr

estimate the camera parameters $\boldsymbol{\xi}$ and the camera poses $\mathbf{M}_i$ (one pose for each image) that minimize the reprojection error between the known 3D positions and the extracted 2D (pixel) positions.

1.3 Calibration procedure

Calibration will begin by saving several images acquired from the camera. The calibration program will then have to read them and extract the interest points. A virtual visual servoing scheme will then be used to minimize the error. Assuming we are using n images and that the calibration landmark is using m points:

- the reprojection error is of dimension $(2 \times n \times m)$ (2 coordinates per point per image).
- the unknown is of dimension $(n_{\boldsymbol{\xi}} + 6 \times n)$ where $n_{\boldsymbol{\xi}}$ is the number of parameters of the camera model.

The VVS is an optimization problem written as:

$$\boldsymbol{\xi}, (\mathbf{M}_i)_i = \arg\min \|\mathbf{s}(\boldsymbol{\xi}, (\mathbf{M}_i)_i) - \mathbf{s}^*\|^2 \tag{1}$$

where $\mathbf{s}^*$ regroups the desired pixel coordinates of the interest points in each image and correspond to the extracted pixel positions, and where $\mathbf{s}(\boldsymbol{\xi}, (\mathbf{M}_i)_i)$ is the pixel coordinates of the known 3D points when projected with intrinsic parameters $\boldsymbol{\xi}$ and camera poses $(\mathbf{M}_i)_i$.

From an estimation of $\boldsymbol{\xi}$ and $(\mathbf{M}_i)_i$, the 3D point $\mathbf{X}({}^oX, {}^oY, {}^oZ)$ whom coordinates are expressed in the object frame $\mathcal{F}_o$ can easily be projected in pixel coordinates. The coordinates of $\mathbf{X}$ in the camera frame yield:

$$\begin{bmatrix} {}^c\mathbf{X} \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{M}_i \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \tag{2}$$

For a classical perspective camera model, we have $\boldsymbol{\xi} = (p_x, p_y, u_0, v_0)$ and the perspective projection:

$$\begin{cases} u &= p_x.X/Z + u_0 \\ v &= p_y.Y/Z + v_0 \end{cases} \tag{3}$$

From (2) and (3) the full projection function $\mathbf{s}(\boldsymbol{\xi}, (\mathbf{M}_i)_i)$ can be defined.

(1) can then be resolved from an initial guess on $\boldsymbol{\xi}$ and $(\mathbf{M}_i)_i$ with a simple gradient descent. The error derivative with regards to intrinsic parameters and camera poses has the following structure:

$$\mathrm{d}\mathbf{s} = \begin{bmatrix} \mathbf{J}_1 & \mathbf{L}_1 & 0 & \dots & 0 \\ \mathbf{J}_i & 0 & \mathbf{L}_i & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \mathbf{J}_n & 0 & \dots & 0 & \mathbf{L}_n \end{bmatrix} \begin{bmatrix} \mathrm{d}\boldsymbol{\xi} \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_n \end{bmatrix} = \mathbf{J}\mathrm{d}\mathbf{x} \tag{4}$$

where:

- $\mathbf{J}_i$ is the derivative of the reprojection of the points of image i with regards to the intrinsic parameters (dimension $(2m) \times n_{\boldsymbol{\xi}}$)

- $\mathbf{L}_i$ is the interaction matrix of the reprojection of the points of image i (dimension $(2m)\times 6$)
- $\mathrm{d}\boldsymbol{\xi}$ is the change in the intrinsic parameters (dimension $n_{\boldsymbol{\xi}}$)
- $\mathbf{v}_i$ is the velocity screw of the estimated camera pose of image i (dimension 6)
- As a consequence, $\mathbf{J}$ is of dimension $(2mn) \times (n_{\boldsymbol{\xi}} + 6n)$ and $\mathrm{d}\mathbf{x}$ is of dimension $(n_{\boldsymbol{\xi}} + 6n)$

The gradient descent allows computing iteratively new values for $\boldsymbol{\xi}$ and $(\mathbf{M}_i)_i$ with:

$$\mathrm{d}\mathbf{x} = \begin{bmatrix} \mathrm{d}\boldsymbol{\xi} \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_n \end{bmatrix} = -\lambda \mathbf{J}^+(\mathbf{s} - \mathbf{s}^*) \tag{5}$$

$\boldsymbol{\xi}$ can then be updated simply with $\boldsymbol{\xi}^{\text{new}} = \boldsymbol{\xi} + \mathrm{d}\boldsymbol{\xi}$.
$\mathbf{M}_i$ is updated with $\mathbf{v}_i$ through the exponential map: $\mathbf{M}_i^{\text{new}} = \exp(-\mathbf{v}_i) \times \mathbf{M}_i$.

Matrices $\mathbf{J}_i$ and $\mathbf{L}_i$ are of course not constant and depend on the several considered points.

1.4 Pose computation procedure

In pose computation we assume the camera is already calibrated and we are dealing from the image stream of the camera. We are just looking for the current pose $\mathbf{M}$ of the camera. The minimization problem is thus the same as (1) but is highly simplified:

$$\mathbf{M} = \arg\min \|\mathbf{s}(\mathbf{M}) - \mathbf{s}^*\|^2 \tag{6}$$

The exact same algorithm as for the calibration can thus be used, without having to consider intrinsic parameters or several images. (4) is thus simplified to :

$$\mathrm{d}\mathbf{s} = \mathbf{L}\mathbf{v} \tag{7}$$

The initial guess for $\mathbf{M}$ still has to be done, but when a new image comes the guess is of course the previous estimation of $\mathbf{M}$ as we assume the image stream correspond to very close poses of the camera.

2 Calibration implementation

2.1 Expected work

During the lab the files will be modified and others will be created. At the end of the lab, please send by email a zip file allowing to compile and test the program.

You may answer the questions by inserting comments in the code at the corresponding lines.

3 Building the `Robot` class

3.1 Incomplete methods

Q1 Compile and execute the program. According to the `main.cpp` file, the robot is following some velocity setpoints defined in the (x, y, θ) space. In which files are the following elements defined:

1. `DefaultEnvironment`
2. `DefaultCartesianSetpoint`
3. `Environment`

Q2 Explain the signature of `DefaultCartesianSetpoint`, especially the way to pass arguments. From the `main()` function, can `i` be modified in `DefaultCartesianSetpoint`? What about `vx`?

Q3 In practice it is often impossible to control a ground robot by sending (x, y, θ) velocities. A classical way to control such a robot is to send a setpoint with a linear velocity v and an angular velocity ω, expressed in the robot frame as shown in Fig. ??.

The corresponding model is quite simple:

$$\left\{\begin{array}{ll} \dot{x} & = v\cos\theta \\ \dot{y} & = v\sin\theta \\ \dot{\theta} & = \omega \end{array}\right. \tag{8}$$

Implement such a function in `Robot::MoveVW`. This method should compute the (x, y, θ) velocities from (v, ω) and then call the `Robot::MoveXYT` method.

Q4 Now that a realistic way to control the robot is possible, should the `Robot::MoveXYT` method stay available for external use? What can we do in the `robot.h` file to make it impossible to use it from outside the `Robot` class?

Q5 When a robot is equipped with two actuated wheels, a simple model is the differential drive model, as shown in Fig. ??. Assuming the two wheels have a radius r and are separated with a distance b, then the kinematic model yields:

$$\left\{\begin{array}{ll} v & = r\dfrac{\omega_l + \omega_r}{2} \\ \omega & = r\dfrac{\omega_l - \omega_r}{2b} \end{array}\right. \tag{9}$$

As in question 3, implement such a function in the `Robot::RotateWheels` method, so that it can be possible to control the robot by sending wheel velocities. You may want to define new attributes in the `Robot` class in order to initialize the radius and base distance through the `Robot::InitWheel` method. We use the following values:

$$\left\{\begin{array}{ll} r & = 0.05m \\ b & = 0.3m \end{array}\right. \tag{10}$$

The method should rely on `Robot::MoveXYT` after having computed the (x, y, θ) velocities from (ω_l, ω_r).

Q6 By using the `wheels_init_` attribute, make sure that it is impossible to do anything in `Robot::RotateWheels` if the radius and base have not been initialized.

3.2 Velocity limits

With the current simulation, we can control the robot:

- by sending linear and angular velocity setpoint with `Robot::MoveVW`
- or by sending wheel velocities with `Robot::RotateWheels`

These two methods call `Robot::MoveXYT`[3] and the robot can reach any velocity. In practice, the wheels have a limited velocity at ± 80 round per minute (rpm).

Q1 Modify the `Robot::InitWheels` method in order to pass a new argument that defines the wheel angular velocity limit. You may need to define a new attribute of the `Robot` class to store this limit. Update the `main.cpp` file with the value in radian per second corresponding to 80 rpm.

Q2 Modify the `Robot::RotateWheels` method in order to ensure that the applied velocities (ω_l, ω_r) are within the bounds. The method should also print a message if the velocity setpoint is too high. Note that if you just saturate the velocities, the robot motion will be different. A scaling is a better strategy, in this case we keep the same ratio between ω_l and ω_r.

Q3 Although the robot actually moves by having its wheels rotate, it would be easier to be able to send linear and angular velocity setpoints. Modify the `Robot::MoveVW` method so that a (v, ω) setpoint is changed to a (ω_l, ω_r) setpoint that will then be called through `Robot::RotateWheels`. Apply the default (v, ω) trajectory in the simulator. The inverse of model (9) yields:

$$\left\{\begin{array}{ll} \omega_l & = \dfrac{v + b\omega}{r} \\ \omega_r & = \dfrac{v - b\omega}{r} \end{array}\right. \tag{11}$$

4 Sensors

The `sensor.h` files defines a `Sensor` class that has four methods:

- `void Init`: initializes the relative pose between the sensor and the robot
- `virtual void Update`: updates the measurement from the robot current position
- `void Print`: prints the current measurement

[3]which should not be calable anymore from outside the `Robot` class

- `void Plot`: plots the measurement history

The `Update` method is defined as a pure virtual function, which makes the `Sensor` class a abstract class. It is thus impossible to declare a variable to be of `Sensor` type, as this class is only designed to build daughter-classes depending on the sensor type.

The `Robot` class already has a attribute called `sensors_` which is a vector of `Sensor*`. As the `Sensor` class is abtract it is forbidden to use it by itself, but pointers are still possible.

4.1 Range sensors

Q1 Create a `sensor_range.h` file that defines a `SensorRange` class that is derived from `Sensor`. The `Update` method has to be defined so that the code compiles. For now, just make the method print something to the screen.

Q2 Include this file in `main.cpp` and declare a `SensorRange` variable. We will use a front range sensor placed at $(0.1, 0, 0)$ in the robot frame. Call the `Update` method at the beginning of the `for` loop. Run the program and ensure that the sensor is updated.

Q3 In this question we will build the `Update` function. This sensor should return the distance to the first wall in its x-axis. The sensor can thus be simulated in two steps:

1. Compute the absolute position and orientation of the sensor. As the robot is passed to the `Update` method, we can use its own (x_r, y_r, θ_r) position and the relative position (x_s, y_s, θ_s) of the sensor to get the absolute sensor position:
$$\begin{cases} x &= x_r + x_s \cos\theta_r - y_s \sin\theta_r \\ y &= y_r + x_s \sin\theta_r + y_s \cos\theta_r \\ \theta &= \theta_r + \theta_s \end{cases} \tag{12}$$
2. Compute the distance to the nearest wall. In the environment variable, the walls are defined by a list of points available in `envir.walls`. Fig. **??** shows a configuration where the sensor is at (x, y, θ) and is facing a wall defined by (x_1, y_1) and (x_2, y_2). In this case, the distance to the wall is:
$$d = \frac{x_1 y_2 - x_1 y - x_2 y_1 + x_2 y + x y_1 - x y_2}{x_1 \sin\theta - x_2 \sin\theta - y_1 \cos\theta + y_2 \cos\theta} \tag{13}$$
The computed distance is positive if the wall is in front of the sensor, and negative if it is behind (in this case this wall is actually not measured). Also, the denominator may be null if the wall is parallel to the sensor orientation.

Define the `Update` function so that it updates the attribute `s` of the sensor with the distance to the nearest wall.

Q4 At the end of the `Update` method, add the command to append the measurement history:
`s_history_.push_back(s_);`
At the end of the program call the `Plot` method of the range sensor in order to display the measurements.

A Main classes