Robotic Perception Project
Mater in Robotics
University of Montpellier

Andrea Cherubini

Contents

Chapter 1

Camera calibration

1.1 Objectives

You will calibrate the camera of the e-puck robot using a central projection model and a cardboard cube. You will determine the intrinsic and extrinsic parameters (position with respect to the robot reference frame) of the camera. You will exploit these results for the rest of the project.

1.2 Material

- an e-puck robot with its camera,
- a cardboard cube with squares of 1cm side,
- a linux machine for recording the images acquired by the robot, and processing them with Python to determine the camera intrinsic and extrinsic parameters.

1.3 Calibration with pinhole model

The calibration will be done with the linear camera model, also called pinhole model or central projection model, since it is based on the projection principle illustrated in Fig. 1.1.

1.3.1 Extrinsic model

The extrinsic model transforms the world frame $\mathcal{R}_W$ into the camera frame $\mathcal{R}_C$ by a translation of the world frame towards the center of the camera

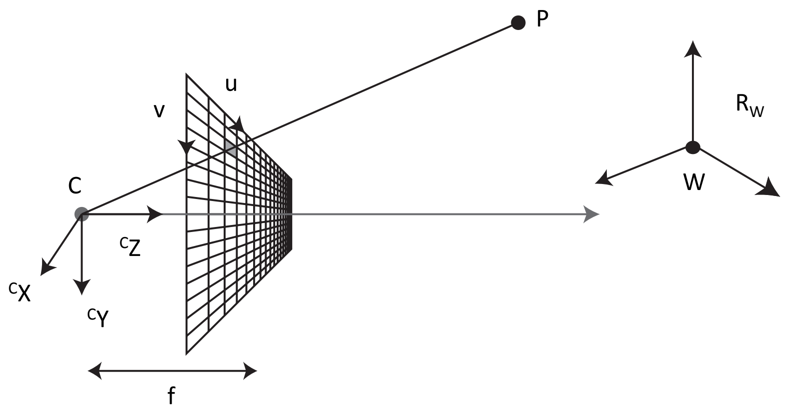


Figure 1.1: Principle of image formation by central projection.

frame, followed by a rotation to align the axes of the two frames (see Fig. 1.1).

Thus, the coordinates $\left[{}^{W}X \; {}^{W}Y \; {}^{W}Z\right]^{\top}$ of a point in the world frame can be expressed in the camera frame $\left[{}^{C}X \; {}^{C}Y \; {}^{C}Z\right]^{\top}$ using the homogeneous matrix formalism :

$$\begin{bmatrix} {}^{C}X \\ {}^{C}Y \\ {}^{C}Z \\ 1 \end{bmatrix} = {}^{C}\mathbf{T}_{W} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix}, \tag{1.1}$$

with: $[p_x \; p_y \; p_z]^{\top}$ coordinates of W in $\mathcal{R}_C$, $[r_{11} \; r_{21} \; r_{31}]^{\top}$ coordinates of ${}^{W}\overrightarrow{X}$ in $\mathcal{R}_C$, $[r_{12} \; r_{22} \; r_{32}]^{\top}$ coordinates of ${}^{W}\overrightarrow{Y}$ in $\mathcal{R}_C$ and $[r_{13} \; r_{23} \; r_{33}]^{\top}$ coordinates of ${}^{W}\overrightarrow{Z}$ in $\mathcal{R}_C$.

1.3.2 Intrinsic model

First, we perform a transformation from the camera reference frame $\mathcal{R}_C$ to the image plance reference frame. The latter is located at the focal length f of $\mathcal{R}_C$. Then, we convert the image plane coordinates to pixel coordinates (u, v), by shifting and scaling (Fig. 1.1).

To do this, we perform a translation that takes into account the center of the image plane in the image frame (u_0, v_0), then a scaling to express the metric coordinates in pixels (u, v). Since it is impossible to dissociate the focal length f from the scaling factors (k_u, k_v), we perform the change of variables: $\alpha_u = k_u f$, $\alpha_v = k_v f$. By also considering $U = uS$ and $V = vS$ (with λ a scalar scaling factor and $S = \lambda {}^{C}Z$), we obtain the pinhole model equation:

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} \alpha_u & 0 & u_0 & 0 \\ 0 & \alpha_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} {}^{C}X \\ {}^{C}Y \\ {}^{C}Z \\ 1 \end{bmatrix}. \tag{1.2}$$

1.3.3 Complete Model

Replacing (1.2) into (1.1), we obtain the complete model:

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} (\alpha_u r_{11} + r_{31} u_0) & (\alpha_u r_{12} + r_{32} u_0) & (\alpha_u r_{13} + r_{33} u_0) & (\alpha_u p_x + p_z u_0) \\ (\alpha_v r_{21} + r_{31} v_0) & (\alpha_v r_{22} + r_{32} v_0) & (\alpha_v r_{23} + r_{33} v_0) & (\alpha_v p_y + p_z v_0) \\ r_{31} & r_{32} & r_{33} & p_z \end{bmatrix} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix} \tag{1.3}$$

which can also be written:

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \end{bmatrix} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix}. \tag{1.4}$$

Let us now denote:

$$\left\{ \begin{array}{lcl} \mathbf{c_1} & = & \begin{bmatrix} c_{11} & c_{12} & c_{13} \end{bmatrix} \\ \mathbf{c_2} & = & \begin{bmatrix} c_{21} & c_{22} & c_{23} \end{bmatrix} \\ \mathbf{c_3} & = & \begin{bmatrix} c_{31} & c_{32} & c_{33} \end{bmatrix} \end{array} \right. . \tag{1.5}$$

From (1.3), (1.4) and (1.5) we derive the following relationships. For the intrinsic parameters:

$$\left\{ \begin{array}{lcc} \lambda & = & \|\mathbf{c_3}\| \\ u_0 & = & \mathbf{c_1}\mathbf{c_3}^\top / \lambda^2 \\ v_0 & = & \mathbf{c_2}\mathbf{c_3}^\top / \lambda^2 \\ \alpha_u & = & \sqrt{\mathbf{c_1}\mathbf{c_1}^\top / \lambda^2 - u_0^2} \\ \alpha_v & = & \sqrt{\mathbf{c_2}\mathbf{c_2}^\top / \lambda^2 - v_0^2} \end{array} \right. . \tag{1.6}$$

For the extrinsic parameters:

$$\left\{ \begin{array}{lcc} \mathbf{\tilde{r}_1} & = & (\mathbf{c_1} - u_0 * \mathbf{c_3}) \,/\, (\lambda \alpha_u) \\ \mathbf{\tilde{r}_2} & = & (\mathbf{c_2} - v_0 * \mathbf{c_3}) \,/\, (\lambda \alpha_v) \\ \mathbf{r_1} & = & \mathbf{\tilde{r}_1} / \, \|\mathbf{\tilde{r}_1}\| \\ \mathbf{r_2} & = & \mathbf{\tilde{r}_2} / \, \|\mathbf{\tilde{r}_2}\| \\ \mathbf{r_3} & = & \mathbf{c_3} / \lambda \\ p_x & = & (c_{14} - u_0 c_{34}) \,/\, (\lambda * \alpha_u) \\ p_y & = & (c_{24} - v_0 c_{34}) \,/\, (\lambda * \alpha_v) \\ p_z & = & c_{34} / \lambda \end{array} \right. \tag{1.7}$$

Finally, we can compute matrix ${}^C\mathbf{T}_W$ considering that:

$$\left\{\begin{array}{lcl} \mathbf{r_1} & = & \begin{bmatrix} r_{11} & r_{12} & r_{13} \end{bmatrix} \\ \mathbf{r_2} & = & \begin{bmatrix} r_{21} & r_{22} & r_{23} \end{bmatrix} \\ \mathbf{r_3} & = & \begin{bmatrix} r_{31} & r_{32} & r_{33} \end{bmatrix} \end{array}\right. . \tag{1.8}$$

Note that in general the ${}^C\mathbf{R}_W$ matrix thus obtained is not orthonormal.

The objective of the calibration will be to estimate the c_{11}, c_{34} values and to deduce the intrinsic and extrinsic parameters thanks to equations (1.6), (1.7) and (1.8).

1.3.4 Least squares method

The 2D projection of any point P in the image (u, v) can be associated to its 3D position in the world $\left({}^W X, {}^W Y, {}^W Z\right)$ by two independent linear equations[1], where the 12 unknowns are parameters $c_{11} \ldots c_{34}$:

$$\left\{\begin{array}{lcl} (c_{11} - c_{31}u)^W X + (c_{12} - c_{32}u)^W Y + (c_{13} - c_{33}u)^W Z + (c_{14} - c_{34}u) & = & 0 \\ (c_{21} - c_{31}v)^W X + (c_{22} - c_{32}v)^W Y + (c_{23} - c_{33}v)^W Z + (c_{24} - c_{34}v) & = & 0 \end{array}\right. \tag{1.9}$$

By normalizing c_{34} to 1 (or by dividing the two equations by c_{34}), the number of parameters to be determined is reduced to 11. The system can then be expressed in the matrix form

$$\mathbf{Ax} = \mathbf{b}, \tag{1.10}$$

where the vector of 11 unknown parameters is:

$$\mathbf{x} = \left[c_{11}\ c_{12}\ c_{13}\ c_{14}\ c_{21}\ c_{22}\ c_{23}\ c_{24}\ c_{31}\ c_{32}\ c_{33}\right]^\top, \tag{1.11}$$

The matrix $\mathbf{A}$ indexed with respect to the n calibration points is:

$$\mathbf{A} = \begin{bmatrix} {}^W X_1 & {}^W Y_1 & {}^W Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1^W X_1 & -u_1^W Y_1 & -u_1^W Z_1 \\ 0 & 0 & 0 & 0 & {}^W X_1 & {}^W Y_1 & {}^W Z_1 & 1 & -v_1^W X_1 & -v_1^W Y_1 & -v_1^W Z_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ {}^W X_i & {}^W Y_i & {}^W Z_i & 1 & 0 & 0 & 0 & 0 & -u_i^W X_i & -u_i^W Y_i & -u_i^W Z_i \\ 0 & 0 & 0 & 0 & {}^W X_i & {}^W Y_i & {}^W Z_i & 1 & -v_i^W X_i & -v_i^W Y_i & -v_i^W Z_i \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ {}^W X_n & {}^W Y_n & {}^W Z_n & 1 & 0 & 0 & 0 & 0 & -u_n^W X_n & -u_n^W Y_n & -u_n^W Z_n \\ 0 & 0 & 0 & 0 & {}^W X_n & {}^W Y_n & {}^W Z_n & 1 & -v_n^W X_n & -v_n^W Y_n & -v_n^W Z_n \end{bmatrix} \tag{1.12}$$

and

$$\mathbf{b} = \left[u_1\ v_1\ \ldots\ u_i\ v_i\ \ldots\ u_n\ v_n\right]^\top. \tag{1.13}$$

We need a minimum of 6 calibration points to estimate the parameters of the complete model. To avoid system degeneracy, the points should not all

[1]obtained by replacing $S = \lambda^C Z$ in (1.4)

be in the same plane nor on the same visual radius. Having defined properly the points, the system is solved by minimizing a quadratic deviation of the projection distances which leads to pseudo-inverse calculation:

$$\mathbf{x} = \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\mathbf{b} = \mathbf{A}^\dagger\mathbf{b}. \tag{1.14}$$

Once the parameters of the global model have been determined, it is possible to dissociate the parameters of the internal model and those of the external model. For this method to be sufficiently stable, n must be large enough, and the uncertainty on the position of the points must be small enough.

1.4 Experimentation

This work will be done in two steps:

1. acquisition of an image with the e-puck robot,

2. processing the image for calibration under Python.

1.4.1 Acquisition of an image with the e-puck robot

Retrieve (from the internet, using **git clone**) the code necessary to control the e-puck: https://gitlab.com/umrob/epuck-client.

Place the e-puck and the cube as shown in Fig. 1.2.

Follow the instructions from the document **epuck_instructions_students.pdf** in the directory **epuck-client**. For example, if you use the robot with ip address 192.168.1.2:

```
$ ping 192.168.1.2
$ cd epuck-client/build
$ cmake ..
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 0 0
```

This program saves the images acquired by the e-puck in the directory `epuck-client/logs/<dateandtime>`. While the program is running, move the cube with the clamp until you have a correct view of the three planes you will need for calibration (as in Fig. 1.3). At this point, press `ctrl+c` to stop the program. Be careful not to move either the robot or the cube afterwards!

Using the metric paper and a ruler, measure the position ${}^{C}\mathbf{p}_W$ of the cube in the camera frame, as well as the position ${}^{R}\mathbf{p}_W$ of the cube in the robot frame $\mathcal{R}_R$. The robot frame has origin R and axes ${}^{R}X$ and ${}^{R}Y$ on the ground, and ^{R}Z perpendicular to the ground. The point R is in the middle of two wheels of the robot, ^{R}X points to the right and ^{R}Y to the front.

To continue the work on Python, write down these six values (in meters) as well as the path to the best image of the cube (corresponding to the six values), e.g. `epuck-client/logs/<dateandtime>/image0093.png`, .

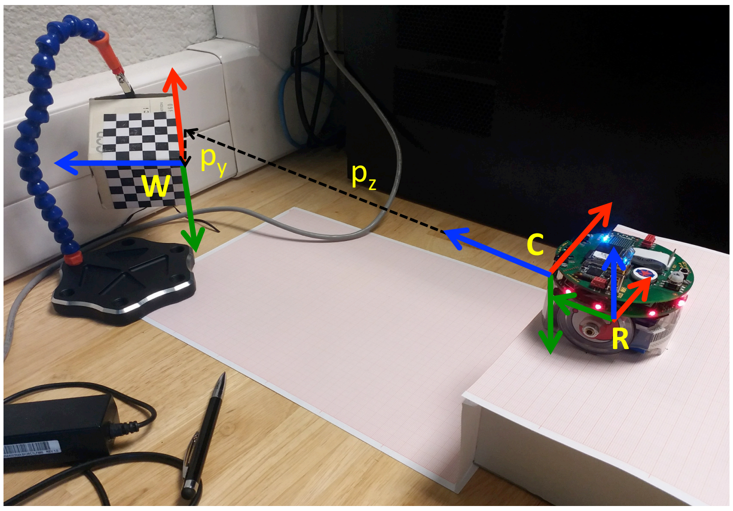


Figure 1.2: Experimental platform for calibration, with camera, robot and world (i.e., cube) frames. In this figure and in the following ones, the reference frames are represented with the RGB color convention (X axis in red, Y in green and Z in blue); p_y and p_z are the translations from the origin of the camera frame to the origin of the world frame (p_x is negligible in the configuration of this figure).

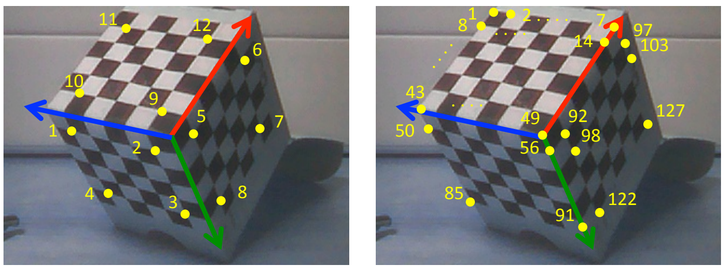


Figure 1.3: Calibration cube, with the world reference frame, and the order of the points. You will have to click on the points indicated, respecting the order. Left: 12 points are used, right: 127 points are used.

1.4.2 Image processing and calibration under Python

You will now continue in Python, to process the image of the cube acquired by the e-puck. You will use 2 scripts: `pixelSelector.py`, and `camCalibration.py`. You will have to modify the latter at the lines with the comment.

1. Start by running the program `pixelSelector.py`. You do not need to modify this program.
2. The program proposes to load an image (function `imread()`). Choose the image acquired by the e-puck (for example, `image0093.png`).
3. Select, by clicking on the image (function `ginput()`), 12 points on the cube, following the order of Fig. 1.3 (left). The program displays these points (red crosses).
4. Since the 3D coordinates of the 12 points are known (and provided in the matrix `xyz`), the program builds a calibration file called `uv_xyz.dat` which contains the data of each of the 12 points in the format:

u (pixels)	v (pixels)	${}^{W}X$ (m)	${}^{W}Y$ (m)	${}^{W}Z$ (m)

5. Launch `camCalibration.py`. The objective of this program (to be completed) is to load the file `uv_xyz.dat` and to use it to estimate the camera intrinsic and extrinsic parameters.
6. The first step is to realize the least square estimation described in Section 1.3.4. Modify the program to build matrices $\mathbf{A}$ and $\mathbf{b}$ from (1.12) and (1.13). Warning: these matrices must have respectively dimension 24×11 and 24×1, which is not the case in the initial version of the program.
7. Once you have defined the right matrices (check by printing them in the console), the program estimates (via function `pinv()`) and displays the 11 parameters $c_{11} \ldots c_{33}$ of the complete model (1.4).
8. Then, given matrix $\mathbf{C}$, estimate the intrinsic and extrinsic parameters of the system and display frames $\mathcal{R}_{\mathcal{C}}$ and $\mathcal{R}_{\mathcal{W}}$. Modify the program to calculate these parameters using equations (1.6) and (1.7).
9. From the translation ${}^{C}\mathbf{p}_{W} = [p_x; p_y; p_z]^{\top}$ between $\mathcal{R}_{\mathcal{C}}$ and $\mathcal{R}_{\mathcal{W}}$ (found after this calibration procedure) and the translation ${}^{R}\mathbf{p}_{W}$ (measured

in Sect. 2.3.1), compute the translation ${}^{C}\mathbf{p}_R$ between $\mathcal{R}_\mathcal{R}$ and $\mathcal{R}_\mathcal{C}$ (coordinates of the camera in the robot frame).

Note these values, which (along with intrinsics u_0, v_0, α_u and α_v) will be necessary for the continuation of the project.

Verifying your results :

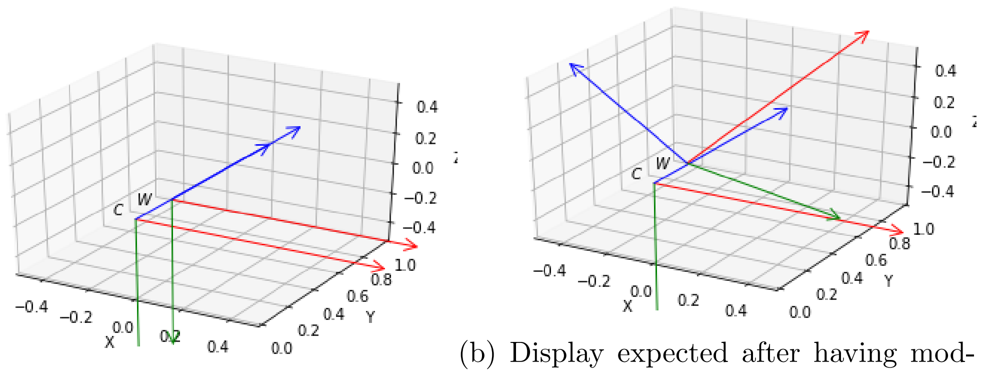
- Intrinsic parameters: since the resolution of the camera is 320×240 pixels, the values of u_0 and v_0 will be of the order of magnitude of a hundred pixels; α_u and α_v also have the order of magnitude of 10^2 pixels.

- Extrinsic parameters: for the translation components ${}^{C}\mathbf{p}_W = [p_x\ p_y\ p_z]^\top$ between $\mathcal{R}_\mathcal{C}$ and $\mathcal{R}_\mathcal{W}$ you should find values close to those measured and noted in Sect. 2.3.1. For the rotations, you should see the frames as in the cube figure (see Figures 1.3 and 1.4).

Once you have verified that your result is plausible, proceed as follows:

1. Repeat using all 127 corners between black and white squares. First restart `pixelSelector.py` by setting `manyPoints=true`. Next, click on the 127 points in the order shown in Fig. 1.3(right). The accuracy of the calibration should increase by using more points.

2. To compare the results with 12 or 127 points, re-project the points and draw them in the cube image, using (1.4). You could, for example, draw



(a) Initial display.

(b) Display expected after having modified the function: note that $\mathcal{R}_\mathcal{W}$ is oriented as in Fig. 1.3.

Figure 1.4: Frames $\mathcal{R}_\mathcal{C}$ and $\mathcal{R}_\mathcal{W}$ displayed by script `camCalibration.py`.

them in blue to check that they are near the red points you initially clicked on. You can also compute the average distance between clicked and reprojected points.

Chapter 2

Visual Target Tracking

2.1 Objectives

You will implement a target tracking algorithm so that the e-puck robot can estimate its position (locate itself) relative to a target (the barycenter of a black blob on a white background) seen by its camera. The estimation must be fast enough to be exploited in a closed-loop control law and must work when the robot and/or the target are moving (i.e., if there is relative motion between the two). You will need the calibration parameters obtained in the first chapter of the project, and you will exploit these results in the next chapter.

2.2 Material

- an e-puck robot with its camera,
- a sheet with the printed target,
- a sheet of metric paper allowing to quantify the localization accuracy,
- a linux machine to record the images acquired by the robot and to process them in order to track the target.

2.3 Experimentation

This work will be done in four steps:

1. Record a sequence of images of the target, with the e-puck robot in motion.

2. Develop an algorithm for detecting the barycenter of the target on each image. This algorithm will be developed on the recorded images (thus without needing the robot). This algorithm will have to display the abscissa and ordinate of the barycenter of the target (in pixel coordinates).

3. Project of the barycenter in the robot frame, using the calibration parameters, and from this: localize the robot with respect to the target.

4. Validate the algorithm on the moving robot and on the moving target.

2.3.1 Acquisition of an image sequence with the e-puck

Place the e-puck in front of the target (at 32 cm, position indicated on the metric paper) so that the target is visible in the camera. Follow the instructions in the document **epuck_instructions_students.pdf**. For example, if you use the robot with ip address 192.168.1.2, in a terminator window run:

```
$ ping 192.168.1.2
```

and in another one:

```
$ cd epuck-client/build
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 150 150
```

This program saves the images acquired by the e-puck in the directory `epuck-client/logs/<datetime>`. While the program is running, the robot advances towards the target, with speed 150 on each wheel.

2.3.2 Image processing to detect the target barycenter

The objective of this step is to develop an algorithm to detect the barycenter of the target on each image. This algorithm will be tested on the previously recorded images (thus without needing the robot), and then (section **??**) validated on the moving robot.

The algorithm will have to display on the console the values in pixel coordinates of the barycenter of the target, and draw on the image a circle corresponding to the barycenter. It will also have to compute the area of the target, which will be used to estimate the robot-target distance. You will develop this algorithm in C++ in the function `ProcessImageToGetBarycenter` which is in the file `controlFunctions.cpp` (in `epuck-client/src/epuck`).

There is already a simplified skeleton of the function. To test it without a robot, run – again from the `epuck-client/build` directory:

```
$./apps/imgProcessing-app ../logs/<datetime>/
```

with `<dateandtime>` indicating the sequence you want to work on (the one you recorded in Sec. 2.3.1). Don't forget the `/` at the end of the command! The application displays one by one the images contained in the directory `epuck-client/logs/<dateandtime>`. You will see the following windows.

- The registered color image (window `loggedImg`).
- The same image in black and white (window `greyImg`).
- The same image after processing (window `processedImg`). This image is initially identical to the black and white image because in the current skeleton, no processing is done: the images are just converted to black and white, and by default the central pixel is considered as the barycenter of the target.
- The position of the robot in the world reference frame (window `map`). This frame has: origin on the target, Y parallel to the wall, and X perpendicular and exiting the wall. The position of the red robot is estimated using only the encoders, while the position of the black robot is estimated using only the camera. Initially, the position of the black robot does not change, since in the initial skeleton no processing is done.

To move from one image to the next: select with the mouse the window named `<processedImg>`, then press any key on the keyboard to move to the next images. The function also displays the calculation time (in milliseconds) needed to process the image.

You will have to modify the function `ProcessImageToGetBarycenter`, at the lines with the comment `//TODOM2`. To do this, adapt the simple code snippet developed in C++ for OpenCV3 here: https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/. To adapt this code, take into account that in `ProcessImageToGetBarycenter`:

- `greyImg` is your source image, in black and white, which is the *input* of your algorithm.
- `processedImg` is a copy of `greyImg` on which you can do your processing and draw landmarks for debugging (typically, the circle corresponding to the barycenter)
- `targetBarycenter` must contain the barycenter of the target. This will be one of the two *outputs* of your algorithm.

- `areaPix` must contain the area (in pixels) of the target – equivalent to the m_{00} moment. This will be the second *output* of your algorithm.

Note: by changing the last argument you pass to the OpenCV function `threshold`, you will be able to reverse the image...

Each time you modify `ProcessImageToGetBarycenter`, don't forget to compile, and then restart the application:

```
$make
$./apps/imgProcessing-app ../logs/<dateandtime>/
```

Once the barycenter is well detected on each image of the sequence with reasonable computation time, you can go to the next step.

2.3.3 Location of the robot with respect to the target

We note $\left({}^{C}X_B, {}^{C}Y_B, {}^{C}Z_B\right)$ the coordinates of the barycenter B in the camera frame (orange in Fig. 2.1). If the target is perfectly parallel to the image plane, the following relation allows to compute the target depth (in meters) in the camera frame:

$$ {}^{C}Z_B = \sqrt{\frac{A_m \alpha_u \alpha_v}{A_p}} \tag{2.1} $$

With A_m the target area in meters, A_p the target area in pixels, α_u and α_v the intrinsic parameters of the camera that you found in Chapter 1. Even if the parallelism hypothesis is not perfectly verified, we can use (2.1) to estimate ${}^{C}Z_B$. Then, from equation (1.2) we have:

$$ \begin{cases} u_B S = u_B \, \lambda {}^{C}Z_B = \lambda \left(\alpha_u \, {}^{C}X_B + u_0 \, {}^{C}Z_B\right) \\ v_B S = v_B \, \lambda {}^{C}Z_B = \lambda \left(\alpha_v \, {}^{C}Y_B + v_0 \, {}^{C}Z_B\right). \end{cases} \tag{2.2} $$

from which we can derive:

$$ \begin{cases} {}^{C}X_B = \frac{(u_B - u_0) \, {}^{C}Z_B}{\alpha_u} \\ {}^{C}Y_B = \frac{(v_B - v_0) \, {}^{C}Z_B}{\alpha_v}, \end{cases} \tag{2.3} $$

with u_0 and v_0 the intrinsic parameters of the camera that you found in Chapter 1.

We can then find the coordinates of the barycenter in the robot reference frame (see Fig. 1.2 and also the green reference frame in Fig. 2.1):

$$ \begin{cases} {}^{R}X_B = {}^{C}X_B + {}^{R}p_{x,C} \\ {}^{R}Y_B = {}^{C}Z_B + {}^{R}p_{y,C}. \end{cases} \tag{2.4} $$

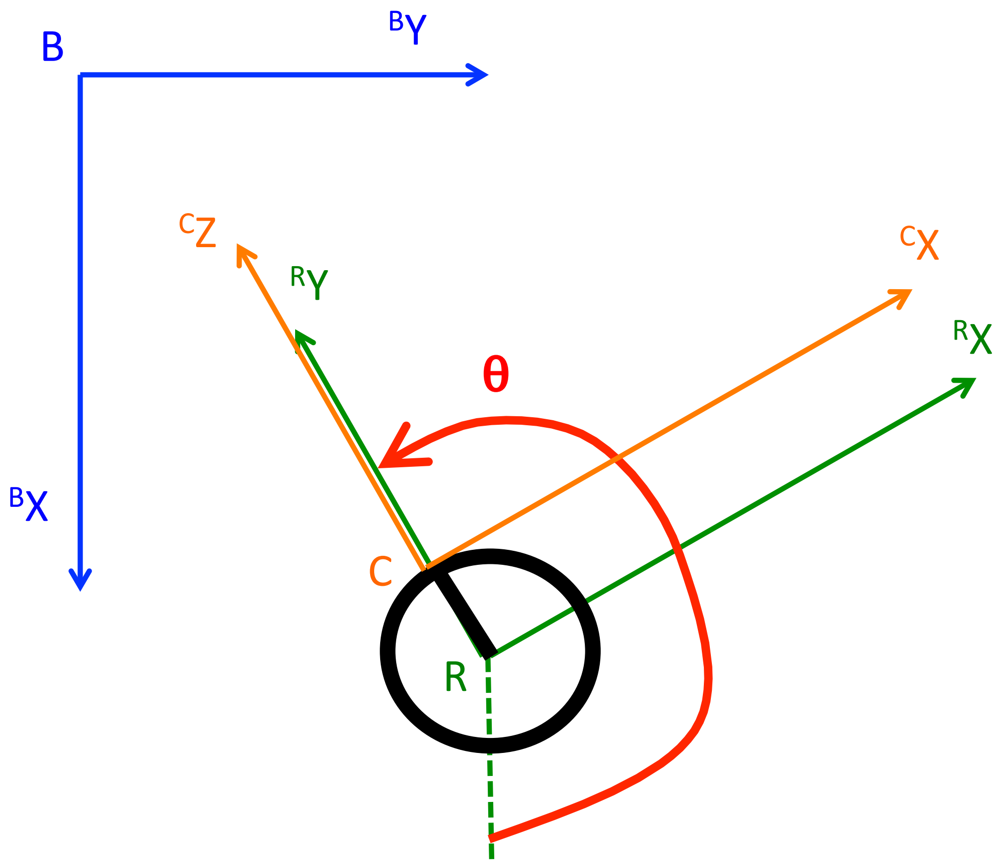


Figure 2.1: Robot (green), camera (orange) and barycenter (blue) reference frames.

With the orientation θ of the robot, estimated via the encoders, we can then compute the robot coordinates in the barycenter frame (see Fig. 2.1):

$$\begin{pmatrix} {}^{B}X_R \\ {}^{B}Y_R \end{pmatrix} = \begin{pmatrix} \sin\theta & \cos\theta \\ -\cos\theta & \sin\theta \end{pmatrix} \begin{pmatrix} -{}^{R}X_B \\ -{}^{R}Y_B \end{pmatrix}. \tag{2.5}$$

The idea is to use these equations to complete `GetCurrPoseFromVision` so that it calculates the robot pose. You must modify this function at the lines with the comment `//TODOM2`. In this function:

- `baryctr` is the barycenter of the target, calculated at the previous step. It is the first *input* of the function. Its fields `baryctr.x` and `baryctr.y` correspond respectively to u_B and v_B.
- `theta` is the orientation θ of the robot (see Fig. 2.1) estimated by the encoders. It is the second *input* of the function.

- `areaPix` is the surface of the target, computed in the previous step, and noted A_p. It is the third *entry* of the function.
- `log` indicates the folder where the function should save the robot pose.
- `curRobPose` is $\left({}^{B}X_R, {}^{B}Y_R, \theta\right)$,i.e., the pose of the robot relative to the target. This will be the only *output* of the function.

Consider putting displays in the terminal (`printf` or `cout`) to check the different steps of the function.

2.3.4 Validation of the algorithm on the robot

Place the target in front of the e-puck so that it is visible for the robot camera. If you change the initial position of the robot, remember to change it also in the `main` (variable `initPose`).

Again, if you use the robot with ip address 192.168.1.2, in a terminator window run:

```
$ ping 192.168.1.2
```

and in another one:

```
$ cd epuck-client/build
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 150 150
```

The behavior is similar but this time the image processing methods which you implemented in function `ProcessImageToGetBarycenter` run while the robot is moving.

Perform several tests: with different wheel speeds (e.g., 50,60 to rotate the robot) and by moving the target.

Chapter 3

Visual servoing

3.1 Objectives

The objective of this chapter is to realize a visual servoing of the e-puck, so that it centers in its image the target detected in Chapter 2. To this end, you will control the linear velocity v and angular velocity ω of the robot, according to the position of the target's barycenter in the image (u_B, v_B). The goal will be to center (u_B, v_B) in the image. An example is shown in this video: https://www.youtube.com/watch?v=bhL78Nu5kKY

3.2 Material

- an e-puck robot with its camera,
- a sheet with the printed target,
- a linux machine to check the control law on the recorded images, before implementing it on the real robot.

3.3 Experimentation

3.3.1 Visual servoing based on the image and on the robot localization

In the first step, you will realize a hybrid image/position visual servoing. The servoing in the image will consist in using the angular velocity ω in order to center the abscissa of the barycenter in the image (position u_B at the center). For the position control, you will use the localization results of Chapter 2, along with the linear velocity v, in order to regulate the depth of the target ${}^{C}Z_B$ to a desired value, denoted ${}^{C}Z_B^*$.

Question 1 Write on a sheet of paper a control law for v and one for ω to accomplish the two tasks shown above.

Question 2 Implement this control law on the e-puck. To do this, you must modify the `ControlRobotWithVisualServoing` function. This function takes as input the image coordinates of the barycenter of the target (u_B, v_B) (`baryc` in the code) and must compute as output the velocities of the robot: v and ω (noted `vel` and `omega` in the code).

Modify function `ControlRobotWithVisualServoing` to realize the control law which you found in **Question 1**.

To update ${}^{C}Z_B$, it will be necessary to also pass as parameter of the function `areaPix` (and therefore to also modify `ControlFunctions.hpp` and `EPuckV2Driver.cpp`).

Once you have prepared the function, you can test it on saved images (to avoid damaging the robot). To do this, add the call to the function in `epuck-client/apps/imgProcessing-app/main.cpp`, after the calculation of `baryc`:

```
float v, w;
struct timeval startTime;
ControlRobotWithVisualServoing(baryc, areaPix, v, w);
```

To check if the values of v and ω seem reasonable in terms of sign and order of magnitude, display them in the terminal (`printf` or `cout`). Then test on the recorded images:

```
$ cd epuck-client/build
$ make
$./apps/imgProcessing-app ../logs/<datetime>/
```

Finally, if v and ω seem reasonable, test on the real robot:

```
$./apps/epuck-app visServo 192.168.1.2
```

3.3.2 Visual servoing based only on the image

If the camera has not been calibrated or the actual area of the target is unknown, the previous method cannot be applied. However, we can still perform a visual servoing based only on the image, by setting (u_B, v_B) to a set point (u_B^*, v_B^*).

You will apply the following control law (see: lecture slides on **visual servoing**):

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = -\lambda \mathbf{L}^{\dagger} \left(\begin{bmatrix} {}^{I}X_B \\ {}^{I}Y_B \end{bmatrix} - \begin{bmatrix} {}^{I}X_B^* \\ {}^{I}Y_B^* \end{bmatrix} \right) \tag{3.1}$$

with :

- v and ω the linear and angular velocities of the e-puck robot,
- λ a positive scalar gain,
- $\left({}^{I}X_B, {}^{I}Y_B\right)$ the normalized current coordinates of the target barycenter in the image frame, i.e.

$$\begin{cases} {}^{I}X_B = \frac{u_B - u_0}{\alpha_u} \\ {}^{I}Y_B = \frac{v_B - v_0}{\alpha_v} \end{cases} \tag{3.2}$$

- $\left({}^{I}X_B^*, {}^{I}Y_B^*\right)$ the desired normalized coordinates of the target's barycenter in the image frame,
- $\mathbf{L}$ the interaction matrix relating $\left({}^{I}X_B, {}^{I}Y_B\right)$ and the camera velocity in free space (which has dimension 6):

$$\mathbf{L} = \begin{bmatrix} -\frac{1}{{}^{C}Z_B} & 0 & -\frac{{}^{I}X_B}{{}^{C}Z_B} & {}^{I}X_B\,{}^{I}Y_B & -1 - {}^{I}X_B^2 & {}^{I}Y_B \\ 0 & -\frac{1}{{}^{C}Z_B} & \frac{{}^{I}Y_B}{{}^{C}Z_B} & 1 + {}^{I}Y_B^2 & -{}^{I}X_B\,{}^{I}Y_B & -{}^{I}X_B \end{bmatrix}. \tag{3.3}$$

The matrix $\mathbf{L}$ verifies the following condition:

$$\begin{bmatrix} {}^{I}\dot{X}_B \\ {}^{I}\dot{Y}_B \end{bmatrix} = \mathbf{L} \begin{bmatrix} v_{X,C} \\ v_{Y,C} \\ v_{Z,C} \\ \omega_{X,C} \\ \omega_{Y,C} \\ \omega_{Z,C} \end{bmatrix}, \tag{3.4}$$

with $[v_{X,C} \ldots \omega_{Z,C}]$ speed of the camera in free space.

Thus, control law (3.1) guarantees asymptotic convergence from $\left({}^{I}X_B, {}^{I}Y_B\right)$ to $\left({}^{I}X_B^*, {}^{I}Y_B^*\right)$. The convergence rate is determined by λ.

On the e-puck robot, the camera cannot move freely in space. It is constrained and has only 2 degrees of freedom.

Question 3 Re-write matrix $\mathbf{L}$ to take into account only these 2 degrees of freedom.

Question 4 In this part of the project, we consider the origin of the camera frame and the robot frame to be the same. Under this assumption, write the velocities v and ω of the robot according to the velocities of its camera.

Rewrite the control law (3.1) by taking into account your answers to **Question 3** and **4**. For the desired position, use: for the abscissa the center of the image, for the ordinate a point low enough in the image [1].

Question 6 Implement controller (3.1) on the e-puck. To do this, you will proceed as before, modifying `ControlRobotWithVisualServoing` and testing first on the recorded images, then on the real robot. In the program, you can give a constant (and realistic) value to ${}^{C}Z_B$. It is advisable to display (for example with a blue circle) the desired barycenter (u_B^*, v_B^*). To do this, add the display to the function `processImageToGetBarycenter` which is in `ControlFunctions.cpp`.

[1]Choosing an ordinate close to the image center will give numerical problems when inverting $\mathbf{L}$.