Task-based control lab

Lab 2: Robot arm control

1 Content of this lab

The goal is this lab is to control one arm of the Baxter robot, first in simulation then on the real robot.
The robot is simulated with CoppeliaSim[1]. In the simulation is placed a green sphere. The goal is to move the arm so that the sphere is centered and at a given distance from the camera. The robot control will be performed in C++ and will rely on:

- The ROS framework to handle communication between the simulator and the control program (actually hidden inside the robot class).
- The ViSP library[2] to manipulate vectors and matrices and to perform linear algebra.

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

1.1 Environment setup

ROS environment should already be set up in the computers. In order to use an IDE like Qt Creator, you should open a terminal and launch the IDE from the command line.
A tutorial on how to use Qt Creator (or any other IDE able to load CMake files) can be found on my webpage: `http://pagesperso.ls2n.fr/~kermorgant-o/coding%20tools.html#config`.

This lab is available on GitHub as a ROS package called `ecn_sensorbased`. On the P-ro computers, it can be downloaded and compiled using `rosdl ecn_baxter_vs`

If you want to do it manually, you should first go in the `ros/src` directory. The package can then be downloaded through git:

```
1 git clone https://github.com/oKermorgant/ecn_baxter_vs.git
```

Remember to call `catkin build` after downloading the packages in your ROS workspace, and before trying to load it in the IDE.

1.2 Structure of the `ecn_baxter_vs` package

The only file to be modified is `main.cpp`, which is the main file for the C++ program.
The simulation can be launched with: `roslaunch ecn_baxter_vs sim.launch`.
When both the simulation and the control program run, the ROS graph looks like this:

[1] http://www.coppeliarobotics.com/
[2] Visual Servoing Platform, http://visp.inria.fr

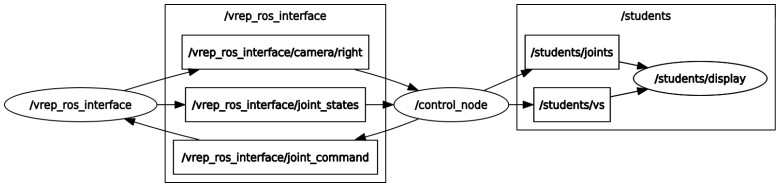

Figure 1: ROS graph showing nodes (ellipses) and communication topics (rectangles)

We can see that the simulator sends the joint states and the camera image to the control node. In return, the control node sends the joint setpoints to the simulator. It also sends data to be displayed by the `display` node.

1.3 The Baxter robot (arm)

The considered robot is one arm of Baxter. A camera is placed at inside the end-effector of the robot. The robot is controlled through joint velocity $\dot{\mathbf{q}}$. In the initial state of the control node, the error and interaction matrix are not computed, inducing a null joint velocity setpoint.

2 Expected work

2.1 Basic visual servoing

The first goal is to actually compute a classical visual servoing. The features are defined as $\mathbf{s} = (x, y, a)$ where (x, y) is the position of the center of gravity of the sphere and a is its area in the image.
The current features are accessed through `arm.x(), arm.y(), arm.area()` while the desired ones are set to 0 for (x, y) and accessed through `arm.area_d()` for the area. The error vector `e` has to be defined from the current and desired features.
Similarly, the interaction matrix of $\mathbf{s}$ should be defined in `L`. The feature Jacobian, that expresses the time-variation of $\mathbf{s}$ with regards to $\dot{\mathbf{q}}$ should then be computed from `L` and `arm.cameraJacobian()`.
The desired joint velocities can then be computed with $\dot{\mathbf{q}} = -\lambda \mathbf{J}^{+}\mathbf{e}$. Of course this control law does not take into account the joint limits.
The gain λ should be equal to `arm.lambda()`, in order to be tuned online from the Baxter image window slider.

2.2 Avoiding useless rotations

The 3 current features allow any rotation around the Z-axis of the camera as it does not change the object in the image. In order to stabilize the system we add a constraint that is having the X-axis of the camera orthogonal to the Z-axis of the base frame.
The rotation matrix ${}^{b}\mathbf{R}_{c}$ can be obtained with:

```
1 const auto bRc = arm.cameraPose().getRotationMatrix();
```

Knowing that ${}^{b}\dot{\mathbf{R}}_{c} = {}^{b}\mathbf{R}_{c} \left[\boldsymbol{\omega}_{c}\right]_{\times}$, express the interaction matrix of this new feature and add it to the control.

2.3 Visual servoing with joint limits

We will use the augmented Jacobian framework[3]. In this framework, the constraints are modeled as additional sensor features. The desired value simply corresponds to the avoidance direction, and can usually be defined by using the center of the valid interval: $\mathbf{q}^* = \frac{1}{2}(\mathbf{q}^+ + \mathbf{q}^-)$. A safe interval $(\mathbf{q}^{s-}, \mathbf{q}^{s+})$ has to be defined with a margin $\rho \in]0., 0.5[$ with:

$$\begin{cases} \mathbf{q}^{s+} = & \mathbf{q}^+ - \rho(\mathbf{q}^+ - \mathbf{q}^-) \\ \mathbf{q}^{s-} = & \mathbf{q}^- + \rho(\mathbf{q}^+ - \mathbf{q}^-) \end{cases}$$

ρ can be tuned online through the slider, its value is available with `arm.rho()`.

The classical weighting function is depicted in Fig. 2. It is available in the code, as detailed in Appendix A.2.

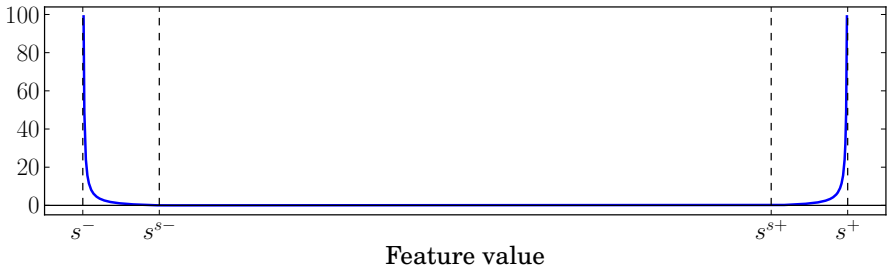

Figure 2: Typical weighting function from $(s^-, s^{s-}, s^{s+}, s^+)$

The control then yields:

$$\dot{\mathbf{q}} = \arg\min \|\mathbf{H}\tilde{\mathbf{J}}\dot{\mathbf{q}} + \lambda\mathbf{H}\tilde{\mathbf{e}}\|^2$$

$$\text{where: } \mathbf{H} = \begin{bmatrix} \mathbf{I}_3 & 0 \\ 0 & \text{Diag}(h_i) \end{bmatrix}, \tilde{\mathbf{J}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{I}_n \end{bmatrix} \text{ and } \tilde{\mathbf{e}} = \begin{bmatrix} \mathbf{e} \\ \mathbf{q} - \mathbf{q}^* \end{bmatrix}$$

The solution to the minimization can be explicitly written as $\dot{\mathbf{q}} = -\lambda(\mathbf{H}\tilde{\mathbf{J}})^+\mathbf{H}\tilde{\mathbf{e}}$.

2.4 Visual servoing and joint control with joint limits

If we use the above formulation then the robot avoids its joint limits but the control law may be discontinuous. Indeed, $(\mathbf{H}\tilde{\mathbf{J}})$ is a 10×7 matrix whose rank may vary from 4 (all joints in their safe zone) to 7 (at least 3 joints close to their limits). A solution is to set arbitrary values for 3 joints.

A good choice is to use the default value (output of `arm.init()`) as desired value for joints $3 \ldots 7$ and keep their weight equal to 1. The joints $1 \ldots 4$ will be used for the visual servoing while avoiding their limits.

[3] O. Kermorgant, F. Chaumette. Dealing with constraints in sensor-based robot control. In IEEE Trans. on Robotics, Feb 2014

2.5 Use on the real robot

If the simulation is working, the only thing to modify to run on the real robot is the `Baxter arm` declaration. You should indicate whether you want to control the left or right arm. The visual detector is designed to detect green objects on the right camera and red objects on the left one. An additional image processing window will open in order to tune the Hue and Saturation thresholds, that may depend on the light conditions.

In order to be connected to Baxter, you have to be in the `ros` workspace and first run `source baxter.sh`.

A Main classes and tools

A.1 ViSP classes

This library includes many tools for linear algebra, especially for 3D transformations. The documentation is found here: `http://visp-doc.inria.fr/doxygen/visp-daily/classes.html`.
The main classes from ViSP (at least in this lab) are:

- `vpFeaturePoint` (variable `p`) represents a Cartesian 2D point as feature. It can be updated with `p.set_xyZ(x,y, 1);`. The main interest is that the corresponding 2×6 interaction matrix can be retrieved by `p.interaction();`
- `vpMatrix` represents a classical matrix, can then be transposed, inversed (or pseudo-inversed), multiplied with a vector, etc.
- `vpColVector` is a column vector with classical mathematical properties.

These class can be declared and used as follows:

```
1  vpMatrix M(2,6);  // matrix of dim. 2x6
2  M[0][0] = M[1][1] = 1; // element-wise assignment
3  vpColVector v(6); // column vector of dim. 6
4  v[0] = 1;  // element-wise assignment
5  M*v;   // matrix-vector multiplication
```

A.2 Utility functions

In addition to the given classes and solvers, several utility function are defined:

- `void ecn::putAt(M, Ms, r, c)`: Writes matrix `Ms` inside matrix `M`, starting from given row and column.
- `void ecn::putAt(V, Vs, r)`: Writes column vector `Vs` inside vector `V`, starting from given row. These two functions do nothing but print an error if the submatrix or vector does not fit in the main one.

- `double ecn::weight(s, s_act, s_max)`: Returns the weighting function defined during the lectures:

$$h = \begin{cases} 0 & \text{if } s < s_{\text{act}} \\ \dfrac{s - s_{\text{act}}}{s_{\text{max}} - s} & \text{otherwise} \end{cases}$$

This formulation corresponds to upper bounds (ie. $s_{\text{act}} < s_{\text{max}}$). For lower bounds (ie. $s_{\text{act}} > s_{\text{max}}$) it simply returns $h(-s, -s_{\text{act}}, -s_{\text{max}})$.