Sensor-based control lab

Labs 3: Multi-sensor control with constraints

1 Content of this lab

The goal is this lab is to control a mobile robot equipped with a pan-tilt camera (see Fig. 1).

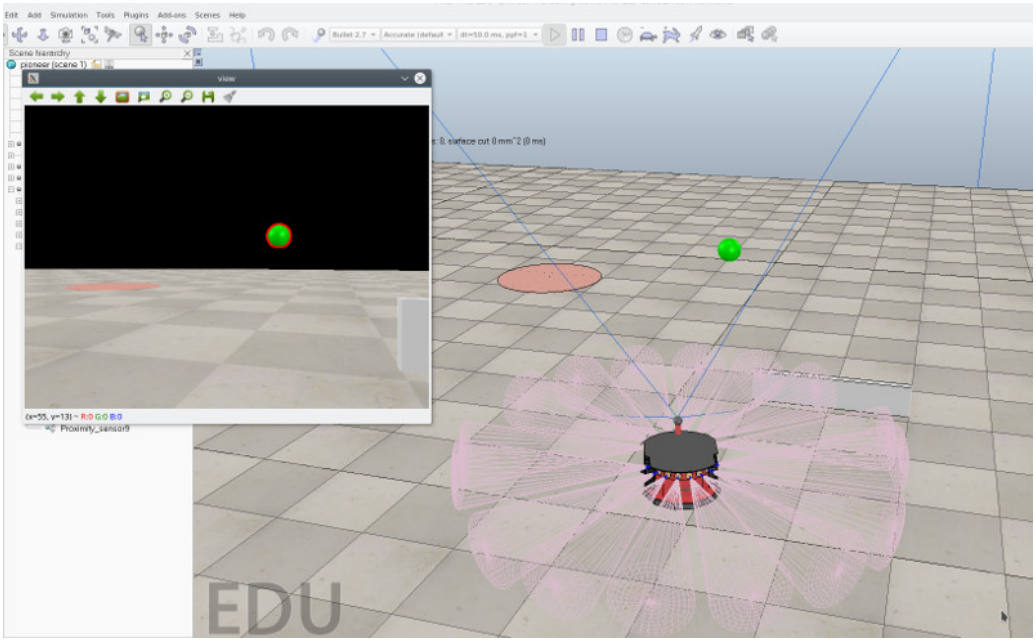

Figure 1: V-REP simulator with the mobile robot, image from the onboard camera and US sensors.

The robot is simulated with V-REP[1]. In the simulation are placed a green sphere and a target. The goal of the robot is to move to the target position while maintaining visibility on the sphere. The robot also have US sensors to detect obstacles but they are not used in this lab. 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]Virtual robot experimentation platform, http://www.coppeliarobotics.com/
[2]Visual Servoing Platform, http://visp.inria.fr

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://www.irccyn.ec-nantes.fr/~kermorga/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_sensorbased`

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

```
git clone https://github.com/oKermorgant/ecn_sensorbased.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_sensorbased` ROS package

The package has the classical ROS structure:

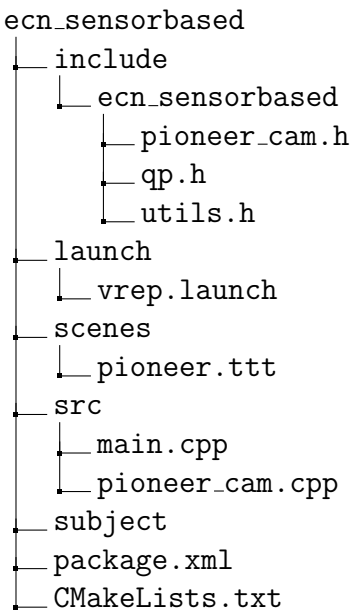

Figure 2: Files used by the package

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_sensorbased vrep.launch`. It will be run and stopped automatically from the main control.
When both the simulation and the control program run, the ROS graph looks like this:
We can see that the simulator sends the joint states, camera image, target and sphere positions to the control node. In return, the control node sends the joint setpoints to the simulator. They are velocity setpoints for the wheels and the pan-tilt camera joints.

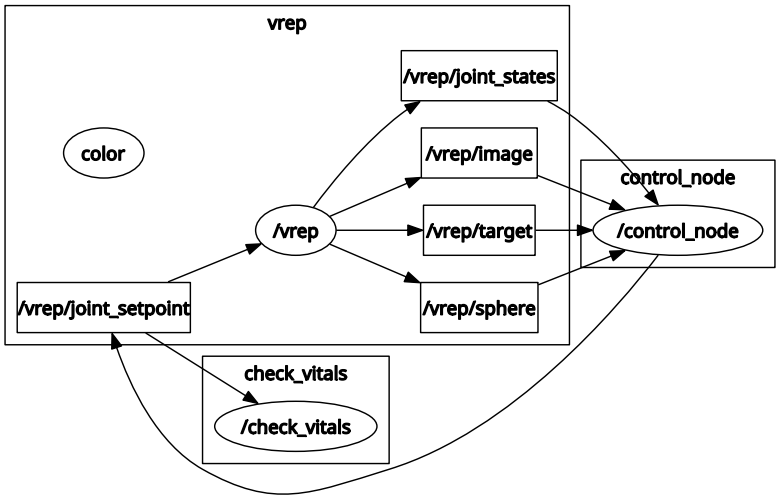

Figure 3: ROS graph showing nodes (ellipses) and communication topics (rectangles)

1.3 The PioneerCam robot

The simulated robot is a Pioneer P3dx. A pan-tilt camera is placed at the front of the robot. The robot is controlled through joint velocity with $\dot{\mathbf{q}} = (v, \omega, \dot{q}_p, \dot{q}_t)$, where (v, ω) are the linear and angular velocity of the mobile base considered as a unicycle, and (q_p, q_t) are the pan and tilt joint angles. In the initial state of the control node, the visibility is not taken into account and the robot follows a very simple control law:

$$\begin{cases} v^* &= \lambda_v(x - d) \\ \omega^* &= \lambda_\omega \text{atan2}(y, x) \end{cases} \tag{1}$$

where (x, y) is the coordinate of the target in the robot frame, d is the final distance to the target (set to 0.1 m), and $(\lambda_v, \lambda_\omega)$ are the control gains.

You should verify that this control runs fine, even if it raises several problems:

- The robot wheels have limited velocities, which is not taken into account with the current control law
- The sphere visibility is not ensured

2 Expected work

The goal is to modify the control law in order to take into account the maximum velocity of the wheels, and the visibility constraint. The objective is to minimize $(v - v^*)^2 + (\omega - \omega^*)^2$ where (v^*, ω^*) is the initial control law (1). However, this minimization should be done under the following constraints:

1. Wheel velocity limits: $|\omega_l| < \omega_{\max}$ and $|\omega_r| < \omega_{\max}$ where (ω_l, ω_r) are the left and right wheel velocities. They are linked to the Cartesian velocities through the kinematic model:

$$\begin{cases} v &= \frac{r}{2}(\omega_l + \omega_r) \\ \omega &= \frac{r}{2b}(\omega_r - \omega_l) \end{cases} \tag{2}$$

where r is the wheel radius, b is the distance between the wheels.

2. The visibility constraint can be taken into account from the current position of the sphere and its Jacobian (see Appendix A.2 for how to get them).

3. Finally, we would like to have a trajectory that follows the one from (1). Do to so, a simple constraint is to write that we must have $v/\omega = v^*/\omega^*$. With this constraint, the robot will follow the desired trajectory radius even if it is not at the desired velocity.

From all theses constraints, we have to express how to build the canonical QP matrices and vectors in order to get the control as the solution to the problem (see Appendix A.3).

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:

- `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:

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

A.2 The PioneerCam class

The PioneerCam class hides all the ROS communication aspects in order to have a higher-level access. As this is not a lab on robot modeling, all Jacobians are already available - even if it could be a nice exercice to compute the camera Jacobian.
The main methods of the `PioneerCam` class can be seen in the `pioneer.h` file:

- Getters for robot model or measurements:
  - `double robot.radius()`, `robot.base()`, `robot.wmax()` : kinematic model.
  - `void robot.getImagePoint(vpColVector &)`: get the measured center of gravity of the sphere in the image, in normalized coordinates.
  - `vpColVector robot.getCamLimits()`: get the upper bounds for (x, y). Lower bounds are the opposite.

  - `vpMatrix robot.getCamJacobian()`: get the Jacobian between the camera velocity screw in its own frame and the robot command $\dot{\mathbf{q}}$. This Jacobian is to be multiplied by the interaction matrix of a point to get the actual Jacobian of the image coordinates with regards to the robot command.

- `robot.setVelocity(vpColVector)`: send a $\dot{\mathbf{q}}$ velocity. Wheel velocity limits will be applied before actually sending it to the simulator.

A.3 QP solvers

Three functions are related to QP solvers, assuming the following formulation:

$$\begin{aligned} \mathbf{x} = \quad & \arg\min \quad \|\mathbf{Q}\mathbf{x} - \mathbf{r}\|^2 \\ & \text{s.t.} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \text{s.t.} \quad \mathbf{C}\mathbf{x} \leq \mathbf{d} \end{aligned} \tag{3}$$

- `qp.solveQP ( Q, r, A, b, C, d, x)` for a classical QP with equality and inequality constraints
- `qp.solveQPe ( Q, r, A, b, x)` with only equality constraints
- `qp.solveQPi ( Q, r, C, d, x)` with only inequality constraints

A.4 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:

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

  This function is designed for upper bounds (ie. assumes that $s_{\text{act}} < s_{\max}$), but can be easily used for lower bounds by sending the opposite values formulation $(-s, -s_{\text{act}}, -s_{\max})$.

- `double ecn::weightBothSigns(s, s_act, s_max)`: Returns the weighting function considering both lower and upper bounds, assuming opposite bounds (which is true for the wheel velocities and the visibility).