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).

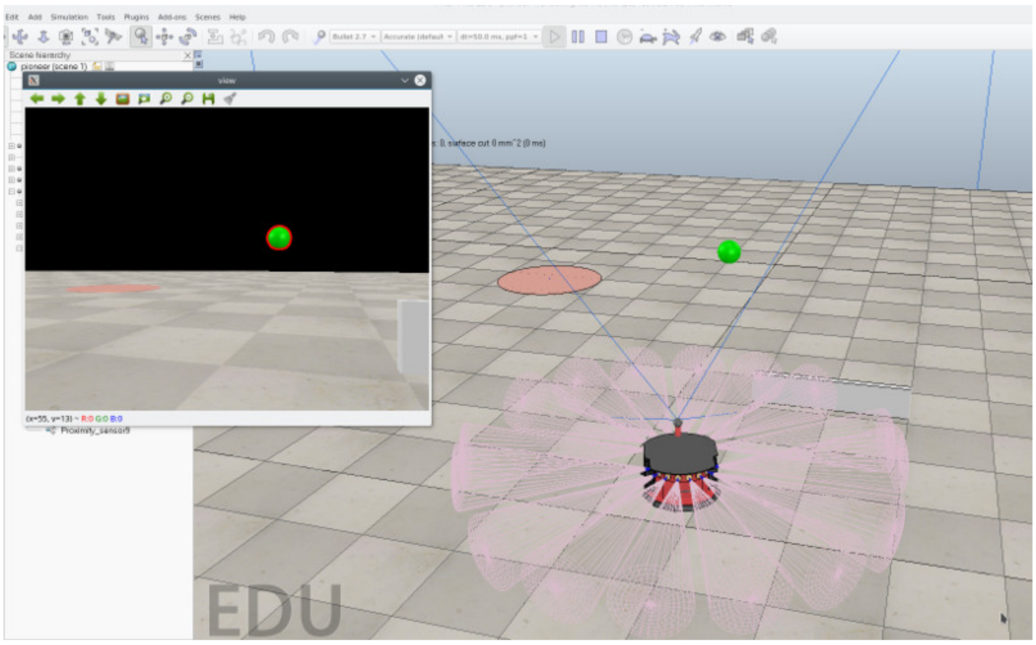

Figure 1: V-REP simulator with the mobile robot, image from the onboard camera and US sensors.

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 control will be performed in C++ and will rely on:

- The ROS framework to handle communication between the simulator and the control program.
- 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

To use this lab (and for some other ROS labs) you need to initialize the ROS environment in order to have access to the ViSP library. To do so, download the following file and execute it:

```
        wget http://www.irccyn.ec-nantes.fr/~kermorga/files/ros_user_setup.sh
        sh ros_user_setup.sh
```

In order to use an IDE like Qt Creator, you should open a new terminal and launch the IDE from the command line.

This lab is available on GitHub as a ROS package called `ecn_sensorbased`. In order to download it, 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
```

1.2 Structure of the ecn_sensorbased ROS package

The package has the classical ROS structure:

```
ecn_sensorbased
├── include
│   └── ecn_sensorbased
│       ├── pioneer_cam.h
│       ├── qp.h
│       └── utils.h
├── launch
│   ├── us_config.yaml
│   └── vrep.launch
├── scenes
│   └── pioneer.ttt
├── scripts
│   └── check_vitals.py
├── src
│   ├── main.cpp
│   └── pioneer_cam.cpp
├── subject
├── package.xml
└── CMakeLists.txt
```

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:

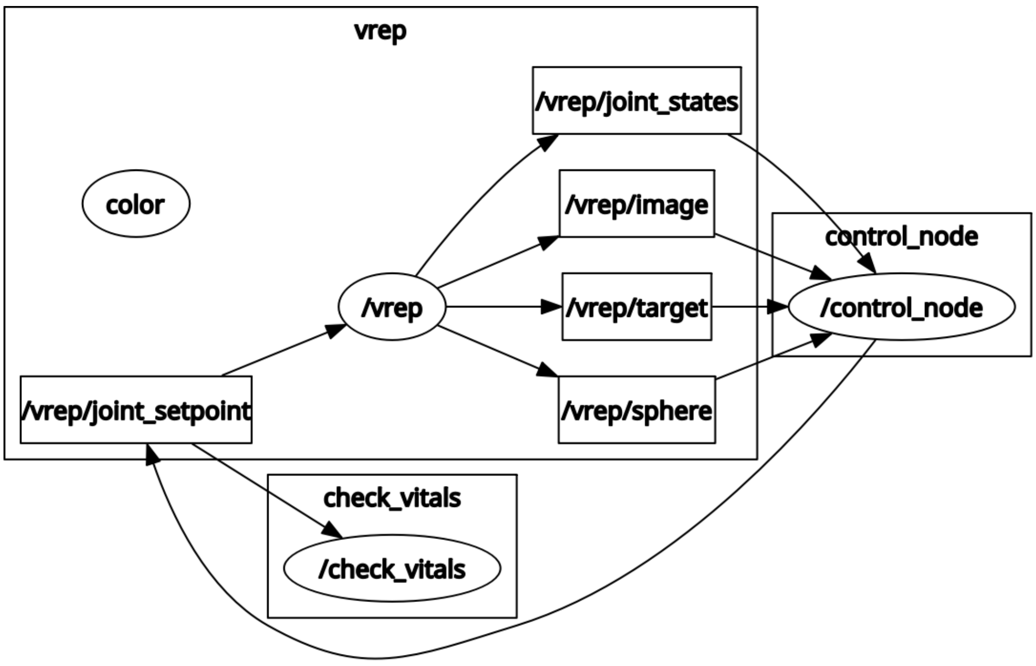

Figure 3: ROS graph showing nodes (ellipses) and communication topics (rectangles)

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.
The `check_vitals` is a script that checks the control node is running and stops the simulation otherwise.

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

A Main classes

A.1 OpenCV classes

These labs use a few OpenCV classes, mainly images and points. The only one that needs to be used is the `cv::Point`, and basically we just have to know that the (u, v) coordinates can be accessed as `P.x` and `P.y`.

A.2 ViSP classes

This library includes many tools for linear algebra, especially for 3D transformations. These classes are mainly used inside the `VVS` class for the minimization algorithm. 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.
- vpHomogeneousMatrix represents a frame transformation matrix M and is initialized from 3 translation and 3 rotation parameters.
- vpPoint: represents a 3D point in the world frame. If the world-to-camera pose is M, we can compute the point projection on the camera plane by:

```
vpPoint P;
vpHomogeneousMatrix M;
P.track(M);
double x = P.get_x();
double y = P.get_y();
```

A.3 Trackers

The CBTracker and GridTracker are very similar and differ only from the object they try to detect. They have only two methods:

- `bool detect(cv::Mat &_im, vector<cv::Point> &_cog)`
  writes the vector of points _cog with the points detected in image _im
- `bool track(cv::Mat &_im, vector<cv::Point> &_cog)`
  updates the vector of points _cog with the points detected in image _im.
  The update tries to be consistent with the current values of the points, ie it will not lead to a 180^o rotation around Z.

Both functions returns True if they managed to detect or track the points. As for all visual trackers, we detect in the first image and track in the next images (except during calibration where there is no reason to track).
The tracker may be lost if the pattern is not here or if the image is blurry, so it can be a good idea to check the return value and re-detect points instead of always tracking them.

A.4 Camera models

The GenericCamera is a virtual class that implements the methods of any camera model:

- `unsigned int nbParam()`
  returns the number of parameters used by this camera model
- `void project(const vpPoint &_P, double &_u, double &_v)`
  takes as an input the 3D point _P that is already expressed in the camera frame, and computes the corresponding pixel coordinates (u, v).
- `void computeJacobianIntrinsic(const vpPoint &_P, vpMatrix &_J)`
  takes as an input the 3D point _P that is already expressed in the camera frame and updates _J so that it corresponds to the intrinsic Jacobian to be used in (??)

- `void computeJacobianExtrinsic(const vpPoint &_P, vpMatrix &_L)`
  takes as an input the 3D point `_P` that is already expressed in the camera frame and updates `_L` so that it corresponds to the extrinsic Jacobian to be used in (??)
- `void updateIntrinsic(const vpColVector &_dxi)`
  takes as input a small change of the intrinsic parameters and updates them according to (??).

Two camera models are considered:

1. `PerspectiveCamera`: Classical perspective camera, with projection equation (??). The class is created but you will have to define the above mentionned methods (project, update Jacobian).
2. `DistortionCamera`: Unified camera model, this class is already entirely written, you may use it to see how to define the methods for the perspective camera (of course the equations will be much more simple).

A.5 The `Pattern` structure

This structure is used to regroup data of a given image: the actual OpenCV image, the corresponding vector of points that have been extracted, and the window name used to display this image.

```
 // structure used to regroup data
struct Pattern
{
    cv::Mat im;                         // OpenCV image
    std::vector<cv::Point> point;    // extracted points
    std::string window;                 // window to display this image
};
```

This structure is only used to pass arguments to the virtual visual servoing class.

A.6 Virtual Visual Servoing class

The `VVS` class implements the minimization algorithm. It is instanciated with:

```
        VVS vvs(cam, d, r, c);
```

where cam is the camera model that is used (PerspectiveCamera or DistortionCamera), d is the inter-point distances of the real landmark, and r and c are the number of rows and columns. The VVS uses these values to initialize the true position of the 3D points.

Only two methods are to be modified:

- `void calibrate(std::vector<Pattern> &_pat)` (already quite written)
  solves the calibration problem from a given set of `Pattern`.

- `void computePose(Pattern &_pat, vpHomogeneousMatrix &_M, const bool &_reset)` solves the pose computation problem from a single `Pattern` (one image with extracted points).
  This method can be highly inspired from the calibrate method, as shown in Section ??.

This class also have useful attributes:

- `std::vector<vpPoint> X_`: a vector of the true 3D position of the points, and is to be used in the reprojection error
- `GenericCamera* cam_`: the camera that we are using or calibrating, it is a pointer to the methods can be called by using an arrow

As a syntax example, here is how we compute the (u,v) pixel coordinates of point k assuming a transformation matrix M (which can be quite useful for reprojection error):

```
double u,v;
X_[k].track(M);                // X_[k] is computed in the camera frame
cam_->project(X_[k], u, v);   // u and v are now the pixel coord. for the current camera model
```