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).

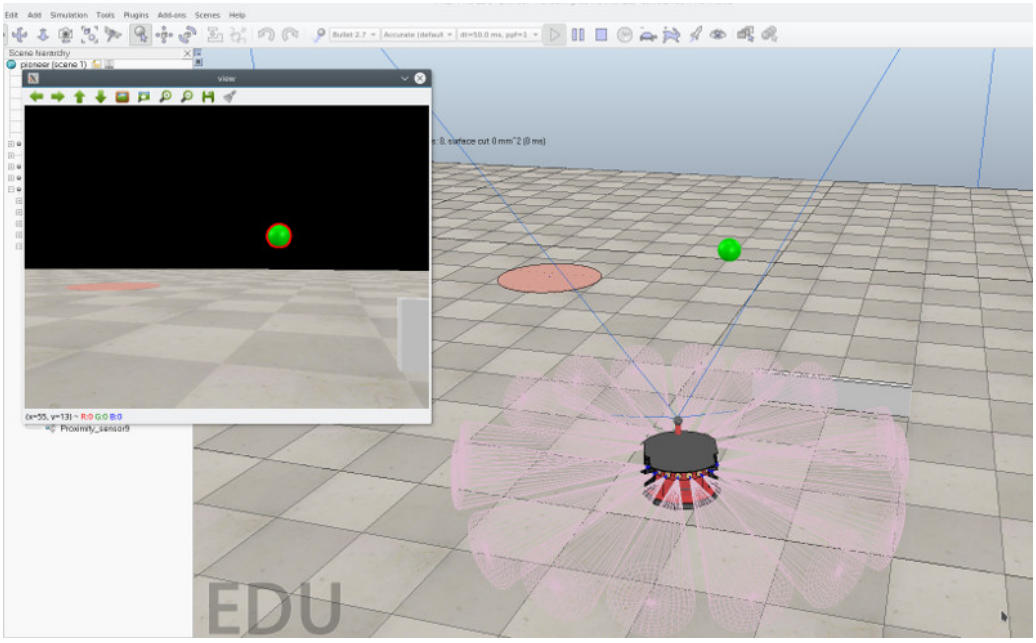

Figure 1: V-REP simulator with the mobile robot, image from the onboard camera and US sensors.

The robot is simulated with V-REP[1]. 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
├── test
├── 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.

1.3 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

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 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.

1.4 Calibration procedure

Calibration will begin by saving several images acquired from the camera (program `record`) The calibration program will then have to read them and extract the interest points (already written in `calibration.cpp`). 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:

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

The update of $\boldsymbol{\xi}$ and $\mathbf{M}_i$ is then done with:

$$\boldsymbol{\xi}^{\text{new}} = \boldsymbol{\xi} + \mathrm{d}\boldsymbol{\xi} \tag{6}$$

$$\mathbf{M}_i^{\text{new}} = \exp(-\mathbf{v}_i) \times \mathbf{M}_i \tag{7}$$

where $\exp(-\mathbf{v}_i)$ is the exponential map.

Matrices $\mathbf{J}_i$ and $\mathbf{L}_i$ are of course not constant and depend on the several considered points. The formulation of these matrices are given in Section B for a perspective camera.

1.5 Pose computation procedure

In pose computation we assume the camera is already calibrated and we are dealing with 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:

$$\mathbf{M} = \arg\min \|\mathbf{s}(\mathbf{M}) - \mathbf{s}^*\|^2 \tag{8}$$

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

$$\mathrm{d}\mathbf{s} = \mathbf{L}\mathbf{v} \tag{9}$$

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.

1.6 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.

2 Calibration implementation

You will first calibrate the camera you are working on. The first step is to save several images of the landmark (point grid) on the hard drive.

The files that need to be modified are:

- `calibration.cpp`: main file, defines the image path, initial guess for the camera parameters and grid pattern (size, dimension)
- `perspective_camera.h`: defines the camera model, you need to write how to project a point depending on the current camera parameters, and how to compute the Jacobian of these coordinates
- `vvs.cpp`: the `calibrate` method of this class has to be written according to the algorithm from Section 1.4.
  The overall architecture is already written, you have to write the content of the minimization loop:
  - project a point from current estimation
  - compute the Jacobians and write them into the global Jacobian
  - update the parameter estimation

The calibration with virtual visual servoing is a highly non-linear, overconstrained algorithm. Indeed it tries to solve $2m \times n$ equations (2 coordinates for m points in n images) with only $n_{\boldsymbol{\xi}} + 6 \times n$ unknown (the number of intrinsic parameters and 6 pose parameters for n images). Hence, the initial guess is of great importance not to fall in a local minimum. Here are some initial guess tips:

- Camera parameters: we can assume a perspective camera with the middle pixel being on the optical axis, and a pixel dimension proportional to the image resolution. A good initial guess could thus be written as:

```
// camera model with default parameters (in calibration.cpp)
const double pxy = 0.5*(im[0].rows+im[0].cols);
PerspectiveCamera cam(pxy, pxy, 0.5*im[0].cols, 0.5*im[0].rows);
```

- Initial pose: as it uses Cartesian coordinates, the VVS is very sensible to big error on the rotation around Z. However as we know the points have been detected, the first row can be used to get a coarse estimation of this rotation. We also know that the object lies in front of the camera, which means a positive value for t_z. So a vector of n homogeneous matrices could be initialized as:

```
M_.resize(n_im);
// wild guess for initial pose, then will use the last found at initial value
for(unsigned int n=0;n<n_im;++n)
    M_[n].buildFrom(0,0.,0.5,
    0,0,atan2(_cog[n][5].y-_cog[n][0].y, _cog[n][5].x-_cog[n][0].x));
```

  where we assume the camera is at 0.5 m from the object.

The calibration can be first carried with the point grid pattern, using the `GridTracker` class. You can then check that it also works on the chessboard, using the `CBTracker`. The detail of input/output for all classes and methods are found in Appendix A.

3 Pose computation implementation

The source file for the pose computation does not exist, so you can copy and modify the `calibration.cpp` to a new file `posecomputation.cpp`[3]. This time we do not want to load images from the hard drive but directly read them from the camera. This is possible with an infinite while loop and by using the `cv::VideoCapture` class.

For the pose computation, you can already initialize the camera with the parameters that have been found in the calibration. The files that need to be modified are then:

- `posecomputation.cpp`: main file (has to be created), reads images from the camera, defines the grid pattern (size, dimension) and call the pose computation from VVS
- `perspective_camera.h`: defines the camera model, it should be already done for the calibration
- `vvs.cpp`: the `computePose` method of this class has to be written according to the algorithm from Section 1.5.

This time the initial guess is the same as above for the first image, but can be the previously found homogeneous matrix as the camera will not move too much between two images. This will allow the pose computation converge faster.

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:

[3] do not forget adding your new file to the CMakeLists.txt

```
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 (4)
- `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 (4)
- `void updateIntrinsic(const vpColVector &_dxi)`
  takes as input a small change of the intrinsic parameters and updates them according to (6).

Two camera models are considered:

1. `PerspectiveCamera`: Classical perspective camera, with projection equation (3). 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 1.5.

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
```

B Jacobians for a perspective camera model

In this section we recall the intrinsic and extrinsic Jacobians for a classical camera model, with parameters $\boldsymbol{\xi} = (p_x, p_y, u_0, v_0)$. The projection model is:

$$\begin{bmatrix} {}^c\mathbf{X} \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{M}_i \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \tag{10}$$

$$\begin{cases} u &= p_x.X/Z + u_0 = p_x.x + u_0 \\ v &= p_y.Y/Z + v_0 = p_v.y + v_0 \end{cases} \tag{11}$$

The Jacobian of (u, v) with regards to the intrinsic parameters is quite easy to compute:

$$\mathbf{J} = \begin{bmatrix} x & 0 & 1 & 0 \\ 0 & y & 0 & 1 \end{bmatrix} \tag{12}$$

The Jacobian with regards to the extrinsic parameters (also called the interaction matrix) yields:

$$\mathbf{L} = \begin{bmatrix} -p_x/Z & 0 & p_x.x/Z & p_x.xy & -p_x(1+x^2) & p_x.y \\ 0 & -p_y/Z & p_y.y/Z & p_y(1+y^2) & -p_y.xy & -p_y.x \end{bmatrix} \tag{13}$$

These two matrices should be defined in the `PerspectiveCamera::computeJacobianIntrinsic` and `PerspectiveCamera::computeJacobianExtrinsic` methods, and depend of course on the considered point.