ViSP 2.6.2: Visual Servoing Platform

Computer vision algorithms

Lagadic project
http://www.irisa.fr/lagadic

July 12, 2012

François Chaumette
Eric Marchand
Fabien Spindler
Romain Tallonneau
Aurélien Yol

Contents

Chapter 1

Camera calibration

1.1 Principle

The basic idea of our approach is to define the pose computation and the calibration problem as the dual problem of 2D visual servoing [5, 9]. In visual servoing, the goal is to move the camera in order to observe an object at a given position in the image. This is achieved by minimizing the error between a desired state of the image features $\mathbf{s}^*$ and the current state $\mathbf{s}$. If the vector of visual features is well chosen, there is only one final position of the camera that allows to achieve this minimization. We now explain why the calibration problem is very similar.

Let us define a virtual camera with intrinsic parameters ξ located at a position such that the object frame is related to the camera frame by the homogeneous 4×4 matrix ${}^c\mathbf{M}_o$. ${}^c\mathbf{M}_o$ defines the pose whose parameters are called extrinsic parameters. Let us consider a 3D point $\mathcal{P}$ which coordinates in the object frame are given by ${}^o\mathbf{P} = ({}^oX, {}^oY, {}^oZ, 1)$. The position of the object point ${}^c\mathbf{P}$ in the camera frame is defined by:

$$ {}^c\mathbf{P} = {}^c\mathbf{M}_o {}^o\mathbf{P} $$

and its projection in the digitized image by:

$$ \mathbf{p} = pr_\xi({}^c\mathbf{P}) = pr_\xi({}^c\mathbf{M}_o {}^o\mathbf{P}) \tag{1.1} $$

where $pr_\xi(.)$ is the projection model according to the intrinsic parameters ξ. The goal of the calibration is to minimize the error between the observed data denoted $\mathbf{s}^*$ (usually the position of a set of features on a calibration grid) and the position of the same features computed by back-projection according to the current extrinsic and intrinsic parameters $\mathbf{s}$ (as defined in 1.1). In order to ensure this minimization we move the virtual camera (initially in ${}^{ci}\mathbf{M}_o$ and modify the intrinsic camera parameters (initially ξ_i) using a visual servoing control law. When the minimization is achieved, the parameters of the virtual camera will be ${}^{cf}\mathbf{M}_o$, that is the real pose, and ξ_f.

We will show in the next paragraphs how to perform this minimization using visual servoing an the interests of considering this approach.

1.2 Visual servoing and calibration

The goal is to minimize the error $\|\mathbf{s} - \mathbf{s}^*\|$. We therefore define the error in the image $\mathbf{e}$ by the simple relation:

$$ \mathbf{e} = \mathbf{s} - \mathbf{s}^* \tag{1.2} $$

The motion of the features in the image is related to the camera velocity $\mathbf{v_c}$ and the time variation of the intrinsic parameters by:

$$\dot{\mathbf{s}} = \frac{\partial \mathbf{s}}{\partial \mathbf{r}} \frac{d\mathbf{r}}{dt} + \frac{\partial \mathbf{s}}{\partial \xi} \frac{d\xi}{dt} \tag{1.3}$$

that can be rewritten as:

$$\dot{\mathbf{s}} = \mathbf{L_s}\mathbf{v} \qquad \text{with} \qquad \mathbf{v} = \begin{bmatrix} \mathbf{v_c} \\ \\ \dot{\xi} \end{bmatrix} \tag{1.4}$$

Matrix $\mathbf{L_s}$ is classically called interaction matrix or image Jacobian in the visual servoing community. It is given:

$$\mathbf{L_s} = \begin{bmatrix} \frac{\partial \mathbf{s}}{\partial \mathbf{r}} & \frac{\partial \mathbf{s}}{\partial \xi} \end{bmatrix} \tag{1.5}$$

If we specify an exponential decoupled decrease of the error $\mathbf{e}$ that is:

$$\dot{\mathbf{e}} = -\lambda \mathbf{e} \tag{1.6}$$

where λ is a proportional coefficient that tunes the decay rate, we finally get:

$$\mathbf{v} = -\lambda {\mathbf{L_s}}^{+} \mathbf{e} \tag{1.7}$$

where $\mathbf{L_s^+}$ is the pseudo inverse of matrix $\mathbf{L_s}$ ($\mathbf{L}^+ = (\mathbf{L}^T\mathbf{L})^{-1}\mathbf{L}^T$ if $\mathbf{L}$ is a full rank matrix).

Comments about the choice of s. Any kind of feature can be considered within this control law as soon as we are able to compute the image Jacobian $\mathbf{L_s}$. In [5], a general framework to compute $\frac{\partial \mathbf{s}}{\partial r}$ is proposed. On the other side $\frac{\partial \mathbf{s}}{\partial \xi}$ is seldom difficult to compute as it will be shown in the next section. This is one of the advantages of this approach with respect to other non-linear calibration approaches. Indeed we are able to perform calibration from a large variety of primitives (points, lines, circles, etc...) within the same framework. Furthermore, considering various kind of primitives within the same calibration step is also possible.

1.3 Multi-images calibration

The intrinsic parameters obtained using one image may be, in practice, very different from the parameters obtained with another image taken from another viewpoint, even if the same lens, the same camera, the same frame grabber and the same calibration grid are used. It is therefore important to consider a multi-image calibration process that integrates within a unique minimization process data from various images.

The underlying idea is to compute a unique set of intrinsic parameters that is correct for all the images (i.e., for all the camera positions). Puget and Skordas considered this idea in [12]. They computed the final intrinsic parameters as the mean value of the parameters computed for each images, the camera positions are then recomputed wrt. to these new intrinsic parameters. Our approach is different. We consider a visual servoing scheme that computes the motion of n virtual cameras and the variation of the l intrinsic parameters that have to be the same for all the images. For n images, we therefore have $6n + l$ unknown variables and $\sum_{i=1}^{n} m_i$ equations (where m_i is the number of features observed in the i^{th} image).

If $\mathbf{s}^i$ is the set of features extracted from i^{th} image, the interaction matrix used in the calibration process is then given by the relation:

$$\begin{bmatrix} \dot{\mathbf{s}}^1 \\ \dot{\mathbf{s}}^2 \\ \vdots \\ \dot{\mathbf{s}}^n \end{bmatrix} = \mathbf{L} \begin{bmatrix} {\mathbf{v_c}}^1 \\ {\mathbf{v_c}}^2 \\ \vdots \\ {\mathbf{v_c}}^n \\ \\ \dot{\xi} \end{bmatrix} \tag{1.8}$$

with

$$\mathbf{L} = \begin{bmatrix} \frac{\partial \mathbf{s}^1}{\partial \mathbf{r}} & 0 & \dots & 0 & \frac{\partial \mathbf{s}^1}{\partial \xi} \\ 0 & \frac{\partial \mathbf{s}^2}{\partial \mathbf{r}} & \dots & 0 & \frac{\partial \mathbf{s}^2}{\partial \xi} \\ \vdots & \vdots & & & \vdots \\ 0 & \dots & 0 & \frac{\partial \mathbf{s}^n}{\partial \mathbf{r}} & \frac{\partial \mathbf{s}^n}{\partial \xi} \end{bmatrix} \tag{1.9}$$

Minimization is handled using the same methodology:

$$\begin{bmatrix} {\mathbf{v_c}}^1 \\ {\mathbf{v_c}}^2 \\ \vdots \\ {\mathbf{v_c}}^n \\ \\ \dot{\xi} \end{bmatrix} = -\lambda \mathbf{L}^+ \begin{bmatrix} \mathbf{s}^1 - \mathbf{s}^{1*} \\ \mathbf{s}^2 - \mathbf{s}^{2*} \\ \vdots \\ \mathbf{s}^n - \mathbf{s}^{n*} \end{bmatrix} \tag{1.10}$$

1.4 Calibration from points

1.4.1 Camera model

In this section, we use the perspective camera model introduced in the *Geometric transformations and objects* manual available from http://www.irisa.fr/lagadic/visp/publication.html. Let us define by $\mathbf{M} = (X\ Y\ Z)^T$ the coordinates of a point in the camera frame. The coordinates of the perspective projection of this point in the image plane is given by $\mathbf{m} = (x\ y)^T$ with:

$$\begin{cases} x = X\ /\ Z \\ \\ y = Y\ /\ Z \end{cases} \tag{1.11}$$

If we denote (u, v) the position of the corresponding pixel in the digitized image, this position is related to the coordinates (x, y) by:

$$\begin{cases} u = u_0 + p_x x + \delta_u \\ \\ v = v_0 + p_y y + \delta_v \end{cases} \tag{1.12}$$

where δ_u and δ_v are geometrical distortions introduced in the camera model. These distortions are due to imperfections in the lenses design and assembly. δ_u and δ_v can be modeled as follow :

$$\begin{cases} \delta_u(x, y) = p_x\ x\ k_{ud}\ \left(x^2 + y^2\right) \\ \\ \delta_v(x, y) = p_y\ y\ k_{ud}\ \left(x^2 + y^2\right) \end{cases} \tag{1.13}$$

or :

$$\begin{cases} \delta_u(u,v) = -(u-u_0)\ k_{du}\ \left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right) \\ \\ \delta_v(u,v) = -(v-v_0)\ k_{du}\ \left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right) \end{cases} \tag{1.14}$$

The six parameters to be estimated are thus $(p_x, p_y, u_0, v_0, k_{ud}, k_{du})$. But in a first time we will only consider the five parameters $\xi_{ud} = (p_x, p_y, u_0, v_0, k_{ud})$.

1.4.2 Deriving the interaction matrix

We have to compute the interaction matrix $\mathbf{L_s}$ that links the motion $\dot{\mathbf{s}} = (\dot{u}, \dot{v})$ of a point $\mathbf{s} = (u, v)$ in the image to $(\mathbf{v_c}, \dot{\xi}_{ud})$. For one point, this Jacobian is given by:

$$\mathbf{L_s} = \left[\begin{array}{cc} \frac{\partial \mathbf{s}}{\partial \mathbf{r}} & \frac{\partial \mathbf{s}}{\partial \xi_{ud}} \end{array}\right] \tag{1.15}$$

where $\frac{\partial \mathbf{s}}{\partial \mathbf{r}}$ is a 2×6 matrix and $\frac{\partial \mathbf{s}}{\partial \xi_{ud}}$ is a 2×5 matrix. Considering a calibration with n points, the full image Jacobian is given by the $2n \times 11$ matrix:

$$\mathbf{L} = (\mathbf{L_{s_1}}, \quad \mathbf{L_{s_2}}, \quad \ldots, \quad \mathbf{L_{s_n}}) \tag{1.16}$$

One of the interest of this approach is that it is possible to consider the background in visual servoing. The image Jacobian $\frac{\partial \mathbf{s}}{\partial \mathbf{r}}$ that relates the motion of a point in the image to the camera motion is quite classical [5, 9] and is given by:

$$\frac{\partial \mathbf{s}}{\partial \mathbf{r}} = \left[\begin{array}{cc} p_x & 0 \\ \\ 0 & p_y \end{array}\right] \left[\begin{array}{cc} 1 + k_{ud}\left(3x^2 + y^2\right) & 2k_{ud}xy \\ \\ 2k_{ud}xy & 1 + k_{ud}\left(x^2 + 3y^2\right) \end{array}\right] \mathbf{L_s} \tag{1.17}$$

with

$$\mathbf{L_s} = \left[\begin{array}{cccccc} -\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & 1+y^2 & -xy & -x \end{array}\right] \tag{1.18}$$

Furthermore, from (1.12), differentiating u and v for ξ_{ud} leads very easily to:

$$\frac{\partial \mathbf{s}}{\partial \xi_{ud}} = \left[\begin{array}{ccccc} x\left(1 + k_{ud}\ \left(x^2 + y^2\right)\right) & 0 & 1 & 0 & x\left(x^2 + y^2\right) \\ \\ 0 & y\left(1 + k_{ud}\ \left(x^2 + y^2\right)\right) & 0 & 1 & y\left(x^2 + y^2\right) \end{array}\right] \tag{1.19}$$

We are now able to write the relation that links the motion of a point in the image to the camera motion and the variation of the intrinsic camera parameters.

Let us finally note that, if we do not want to consider distortion within the camera model, this equation can be simplified and we replace $\frac{\partial \mathbf{s}}{\partial \xi_{ud}}$ by:

$$\frac{\partial \mathbf{s}}{\partial \xi} = \left[\begin{array}{cccc} x & 0 & 1 & 0 \\ \\ 0 & y & 0 & 1 \end{array}\right] \tag{1.20}$$

with $\xi = (p_x, p_y, u_0, v_0)$

Considering the model with the five parameters $\xi_{du} = (p_x, p_y, u_0, v_0, k_{du})$, the expression of distortion is given by equation 1.14. According these equations, the new interaction matrix is easily obtained from :

$$\frac{\partial \mathbf{s}}{\partial \mathbf{r}} = \begin{bmatrix} p_x & 0 \\ 0 & p_y \end{bmatrix} \mathbf{L_s} \tag{1.21}$$

with

$$\mathbf{L_s} = \begin{bmatrix} -\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & 1+y^2 & -xy & -x \end{bmatrix} \tag{1.22}$$

and :

$$\begin{array}{l} \frac{\partial u}{\partial u_0} = 1 + k_{du}\left(3\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right) \\ \frac{\partial u}{\partial v_0} = 2k_{du}\left(\frac{(u-u_0)(v-v_0)}{{p_y}^2}\right) \\ \frac{\partial u}{\partial p_x} = x + 2k_{du}\left(\frac{u-u_0}{p_x}\right)^3 \\ \frac{\partial u}{\partial p_y} = 2k_{du}\frac{(u-u_0)(v-v0)^2}{{p_y}^3} \\ \frac{\partial u}{\partial k_{du}} = -(u-u_0)\left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right) \\ \\ \frac{\partial v}{\partial u_0} = 2k_{du}\left(\frac{(u-u_0)(v-v_0)}{{p_x}^2}\right) \\ \frac{\partial v}{\partial v_0} = 1 + k_{du}\left(\left(\frac{u-u_0}{p_x}\right)^2 + 3\left(\frac{v-v_0}{p_y}\right)^2\right) \\ \frac{\partial v}{\partial p_x} = 2k_{du}\frac{(u-u_0)^2(v-v0)}{{p_x}^3} \\ \frac{\partial v}{\partial p_y} = y + 2k_{du}\left(\frac{v-v_0}{p_y}\right)^3 \\ \frac{\partial v}{\partial k_{du}} = -(v-v_0)\left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right) \end{array} \tag{1.23}$$

1.5 Camera calibration in ViSP

In ViSP camera calibration is implemented in the `vpCalibration` *class. Each instance of this class contains all the informations that are requested to calibrate one image :*

- *a list of 3D points and their corresponding 2D coordinates in the image,*
- *current camera parameters for the different models that are estimated,*
- *member functions to compute camera calibration,*
- *static member functions to compute multi-view camera calibration.*

Listing 1.1 gives a single-image calibration:

```
1  vpCalibration calibration;
2
3  // Add a set of  3D points (in meters) and their 2D corresponding coordinates in the image (in pixels)
4  // in the calibration structure (X,Y,Z,u,v)
5  calibration.addPoint(0,   0,    0,    vpImagePoint(55.2,   64.3) );
6  calibration.addPoint(0.1, 0,    0,    vpImagePoint(245.1,  72.5) );
7  calibration.addPoint(0.1, 0.1,  0,    vpImagePoint(237.2,  301.6) );
8  calibration.addPoint(0,   0.1,  0.1,  vpImagePoint(34.4,   321.8) );
9  .... (add more points here)
10
11
12 vpCameraParameters cam;     //Camera parameters to estimate
13 vpHomogeneousMatrix cMo;    //resulting pose of the object in the camera frame
14
15 // Compute the calibration with the desired method,
16 // here an initialisation with Lagrange method is done and after
17 // the virtual visual servoing method is used to finalized camera calibration
18 // for the perspective projection model with distortion.
19 calibration.computeCalibration(vpCalibration::CALIB_LAGRANGE_VIRTUAL_VS_DIST,cMo,cam);
20
21 // print camera parameters
22 std::cout << cam << std::endl;
```

Listing 1.1: Intrinsec camera parameters computation example

The resulting intrinsic camera parameters are set in the cam object. The estimated pose of the object in the camera frame is provided in cMo (see Listing line 19).

For a multi-images calibration, you just have to initialize a table of `vpCalibration` *objects (one for each image of the calibration grid) and use the static function :*

```
1  vpCalibration::
2  computeCalibrationMulti(vpCalibrationMethodType method, // Desired method to compute
3                                                          // calibration
4                          unsigned int nbPose,            // Number of views (size of table_cal)
5                          vpCalibration table_cal[],      // Table of filled vpCalibration
6                                                          // structures
7                          vpCameraParameters &cam)        // Camera parameters to estimate
```

In addition to the intrinsic camera parameters, it is also possible to determine the extrinsic camera parameters (i.e the constant transformation from the effector to the camera coordinates (eMc)) by using the different camera poses and end-effector poses that have been computed during the calibration. This can be done by the following static function using Tsai approach [13]:

```
1  vpCalibration::
2  calibrationTsai(unsigned int nbPose,        // Number of poses
3                  vpHomogeneousMatrix cMo[],  // Array of camera poses
4                  vpHomogeneousMatrix rMe[],  // Array of end-effector poses
5                  vpHomogeneousMatrix &eMc)   // Resulting extrinsec parameters
```

1.6 Calibration tools in ViSP

For calibrating the intrinsic camera parameters we provide `calibrate2dGrid` *tool. The source code is available in ViSP* `example/calibration/calibrate2dGrid.cpp`*. This example uses a calibration grid provided in ViSP-images package that can be downloaded from* `http://www.irisa.fr/lagadic/visp/download`*. Once unzipped, the grid is in the folder*

"Calibration".

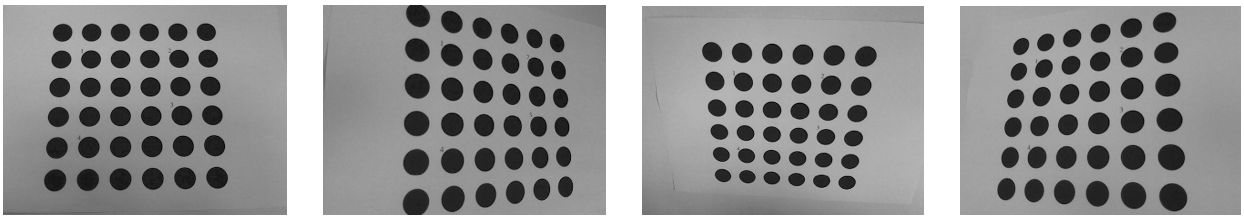

Figure 1.1: Example of grid images that can be used as input to calibrate a camera using `calibrate2dGrid` tool provided in ViSP.

If you use the `calibrate2dGrid` *tool some options might be important:*

- *-p : Path for your image directory.*
- *-f : First image of the sequence.*
- *-n : Number of images.*

Example of usage: *In the next example, images* `I01.pgm` *to* `I04.pgm` *will be used as input to calibrate the camera and provide its intrinsic camera parameters.*

```
calibrate2dGrid -p C:/Temp/I%02d.pgm -f 1 -n 4
```

Note that to compute the extrinsic camera parameters the following example `example/calibration/calibrateTsai.cpp` *is provided.*

Chapter 2

Pose estimation

The pose estimation can be considered as a registration problem that consists of determining the relationship between 3D coordinates of points (or other features: segments, lines, ellipses, ...) and their 2D projections onto the image plane. The position of these 3D features in a world frame have to be known with a good accuracy. Computing pose leads to the estimation of the position and orientation of the camera with respect to a particular world or object frame. In ViSP, it is possible to estimate a pose from points or, since ViSP-2.6.2 from any other visual features.

2.1 Pose estimation from points

ViSP provides several algorithms able to estimate a pose from points. Firstly we will see how this pose estimation is implemented generaly. Then we will see all the approaches (Dementhon, Lagrange, Lowe, virtual visual servoing (VVS) and Ransac) that are available to estimate the pose from points. Note that Dementhon and Lagrange are linear approaches while Lowe and virtual visual servoing are non linear approaches. Non linear approaches require an initial pose that is near the solution to ensure the convergence of the minimisation process. This initial pose could be provided by the linear approaches.

In ViSP, pose estimation from points is implemented in `vpPose` *class. In order to consider point features, we use* `vpPose::addPoint(const vpPoint&)` *method. Note that the point passed as parameter contain either the 2D projection of the point and its 3D coordinates. Note that* `vpPose::clearPose()` *erase the list of points used to compute the pose. Once at least four points are added, the pose is computed using* `vpPose::computePose()`. *Once pose is estimated, it is possible to compute the residual using* `vpPose::computeResidual()`. *Let's see a little example that will illustrate the usage of this class:*

```
1  #include <visp/vpPose.h>
2  #include <iostream>
3  int main()
4  {
5    vpPoint P[4]  ;  // To compute the pose, at least four points are required.
6    vpPose pose ;
7
8    double L = 0.05;
9    // Set the 3D coordinates of the points defining here a 5 by 5 cm planar square plane.
10   P[0].setWorldCoordinates(-L,-L, 0 ) ;
11   P[1].setWorldCoordinates(L,-L, 0 ) ;
12   P[2].setWorldCoordinates(L,L, 0 ) ;
13   P[3].setWorldCoordinates(-L,L, 0 ) ;
14
```

```
15    vpHomogeneousMatrix cMo_ref(0.1,0.2,1,vpMath::rad(10),0,vpMath::rad(10)) ;
16
17    pose.clearPoint(); // Erase the list of points
18    for(int i=0 ; i < 4 ; i++){
19      P[i].project(cMo_ref) ; // 2D projection of the points
20      pose.addPoint(P[i]) ;   // and addition to the list of points used to compute the pose
21    }
22
23    // Pose computation using Dementhon linear approach but other approaches are available
24    vpHomogeneousMatrix cMo ;
25    pose.computePose(vpPose::DEMENTHON, cMo) ;
26    std::cout << "Estimated pose: " << cMo << std::endl ;
27    std::cout << "Residual: " << pose.computeResidual() << std::endl ;
28
29    return 0;
30  }
```

Listing 2.1: Example of pose estimation with `vpPose`. A complete example of this class is available in ViSP (`test/pose/testPose.cpp`)

Note that the `vpPose::computePose()` *takes two parameters (see Listing 2.1, line 25). The first is the method that has to be used to compute the pose and the second is the homogeneous matrix corresponding to the resulting estimated pose of the object in the camera frame. In this example Dementhon approach is used. But as described in the next sections, it is also possible to use Lagrange, Lowe, virtual visual servoing or Ransac approaches. Note that for non linear approaches (VVS and Ransac),* `cMo` *needs to be initialised.*

2.1.1 Dementhon approach

Let us consider N 3D points $\mathcal{P}_i$ whose coordinates in the object frame are given by ${}^o\mathbf{P}_i = ({}^oX_i, {}^oY_i, {}^oZ_i, 1)$. The coordinates of these points in the camera frame obtained by perspective projection of these points in the image plane are given by $\mathbf{p}_i = (x_i, y_i, 1)$ with

$$x_i = \frac{{}^cX_i}{{}^cZ_i} \tag{2.1}$$

$$y_i = \frac{{}^cY_i}{{}^cZ_i} \tag{2.2}$$

Knowing the position of the camera in the object frame ${}^c\mathbf{M}_o$, we have:

$$x_i = \frac{r_{11}{}^oX_i + r_{12}{}^oY_i + r_{13}{}^oZ_i + t_x}{r_{31}{}^oX_i + r_{32}{}^oY_i + r_{33}{}^oZ_i + t_z} \tag{2.3}$$

$$y_i = \frac{r_{21}{}^oX_i + r_{22}{}^oY_i + r_{23}{}^oZ_i + t_y}{r_{31}{}^oX_i + r_{32}{}^oY_i + r_{33}{}^oZ_i + t_z} \tag{2.4}$$

Let us pose:

$$\mathbf{I}^\top = \frac{1}{t_z}\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \end{pmatrix}$$

$$\mathbf{J}^\top = \frac{1}{t_z}\begin{pmatrix} r_{21} & r_{22} & r_{23} & t_y \end{pmatrix}$$

$$\varepsilon_\mathbf{i} = \frac{1}{t_z}\left(r^o_{31}X_i + r^o_{32}Y_i + r^o_{33}Z_i\right)$$

The projection equations (2.3, 2.4) can be rewritten as:

$$\underbrace{x_i\,(\varepsilon_i+1)}_{\mathbf{B_1}} = \underbrace{({}^oX_i \;\; {}^oY_i \;\; {}^oZ_i \;\; 1)}_{\mathbf{A}}\,\mathbf{I} \tag{2.5}$$

$$\underbrace{y_i\,(\varepsilon_i+1)}_{\mathbf{B_1}} = \underbrace{({}^oX_i \;\; {}^oY_i \;\; {}^oZ_i \;\; 1)}_{\mathbf{A}}\,\mathbf{J} \tag{2.6}$$

If ε_i is known, this corresponds to two linear systems with N equations and 4 unknowns. Only 4 non coplanar points are necessary to solve such system.
However, if ε_i is not known, we have to go through a linear iterative solution. Let us denotes :

$$\mathbf{I}^* = \frac{1}{t_z}(r_{11} \quad r_{12} \;\; r_{13}) \quad \text{and} \quad \mathbf{J}^* = \frac{1}{t_z}(r_{31} \quad r_{32} \;\; r_{33})$$

Hereafter we present the algorithm used to estimate the pose parameters [4].

Algorithm 1 Dementhon linear iterative resolution

1: Initialization $\varepsilon_i = 0$
2: Solve the linear systems

$$\mathbf{AI} = \mathbf{B_1} \text{ and } \mathbf{AJ} = \mathbf{B_2}$$

3: From the definition of $\mathbf{I}$ and $\mathbf{J}$ we have

$$\mathbf{i} = (r_{11} \;\; r_{12} \;\; r_{13}) \quad \text{and} \quad \mathbf{j} = (r_{21} \;\; r_{22} \;\; r_{23})$$

Since **i** and **j** are unit vectors ($\mathbf{i} = \frac{\mathbf{I}^*}{\|\mathbf{I}^*\|}$ and $\mathbf{j} = \frac{\mathbf{J}^*}{\|\mathbf{J}^*\|}$), we have:

$$t_z = \frac{\mathbf{i}}{\mathbf{I}^*} = \frac{1}{\| \mathbf{I}^* \|} \text{ where } t_z = \frac{2}{(\| \mathbf{I}^* \| + \| \mathbf{J}^* \|)}$$

4: We can now compute $\mathbf{k} = \mathbf{i} \times \mathbf{j} = (r_{31} \; r_{32} \; r_{33})^\top$ and with 2.5 and 2.6 we can also simply compute t_x and t_y.
5: Set $\varepsilon_i = (r_{31}{}^oX_i + r_{32}{}^oY_i + r_{33}{}^oZ_i)/t_z$ and iterate to 2.

In ViSP to estimate a pose from points by the linear Dementhon approach that doesn't require an initial pose use:

```
pose.computePose(vpPose::DEMENTHON, cMo) ; // cMo doesn't need to be initialised
```

2.1.2 Lagrange approach

In this section we will use the projection equations from the Dementhon approach (2.1, 2.2, 2.3, 2.4). With a simple development of these equations, it leads to:

$$r_{31}{}^oX_ix_i + r_{32}{}^oY_ix_i + r_{33}{}^oZ_ix_i + x_it_x - r_{11}{}^oX_i - r_{12}{}^oY_i - r_{13}{}^oZ_i - t_x = 0$$
$$r_{31}{}^oX_iy_i + r_{32}{}^oY_iy_i + r_{33}{}^oZ_iy_i + y_it_y - r_{21}{}^oX_i - r_{22}{}^oY_i - r_{23}{}^oZ_i - t_y = 0$$

We obtain an homogeneous system with 12 unknown parameters: $\mathbf{AI} = 0$ where $\mathbf{A}$ depends on the data extracted from the image and $\mathbf{I}$ is function of the parameters to be estimated. Each point of the object gives two equations

$$\mathbf{A}_i\mathbf{I} = \begin{bmatrix} 0 & 0 \end{bmatrix}^\top$$

$$\mathbf{A}_i = \begin{pmatrix} -{}^oX_i & -{}^oY_i & -{}^oZ_i & 0 & 0 & 0 & x_i{}^oX_i & x_i{}^oY_i & x_i{}^oZ_i & -1 & 0 & x_i \\ 0 & 0 & 0 & -{}^oX_i & -{}^oY_i & -{}^oZ_i & y_i{}^oX_i & y_i{}^oY_i & y_i{}^oZ_i & 0 & -1 & y_i \end{pmatrix}$$

So, the system that has to be solved is $\mathbf{AI} = 0$ where $\mathbf{I}$ is a non nul vector. Lagrange approach consists in solving the system under the constraint that $[r_{31}\ r_{32}\ r_{33}]^\top$ is a unitary vector, and by considering that r_{ij} is a rotation matrix.

Let's now decompose the homogeneous sytem:

$$\mathbf{AX}_1 + \mathbf{BX}_2 = 0$$

with:
$\mathbf{X}_1 = (r_{31}\ r_{32}\ r_{33})^\top$ and $||\mathbf{X}_1|| = 1$
$\mathbf{X}_2 = (r_{11}\ r_{12}\ r_{13}\ r_{21}\ r_{22}\ r_{23}\ t_x\ t_y\ t_z)^\top$
$\mathbf{A}_i = \begin{pmatrix} x_i{}^oX_i & x_i{}^oY_i & x_i{}^oZ_i \\ y_i{}^oX_i & y_i{}^oY_i & y_i{}^oZ_i \end{pmatrix}$
$\mathbf{B}_i = \begin{pmatrix} -{}^oX_i & -{}^oY_i & -{}^oZ_i & 0 & 0 & 0 & -1 & 0 & x_i \\ 0 & 0 & 0 & -{}^oX_i & -{}^oY_i & -{}^oZ_i & 0 & -1 & y_i \end{pmatrix}$

As a direct solution ($\mathbf{I} = 0$) is impossible, we will here consider a minimization with Lagrangian, and we will minimize the following criterion:

$$C = ||\mathbf{A} \cdot \mathbf{X}_1 + \mathbf{B} \cdot \mathbf{X}_2||^2 + \lambda(1 - ||\mathbf{X}_1||^2)$$

Then by nullifying the partial derivatives:

$$\begin{aligned} \frac{1}{2}\frac{\partial C}{\partial X_1} &= \mathbf{A}^\top\mathbf{A} \cdot \mathbf{X}_1 + \mathbf{A}^\top C \cdot \mathbf{X}_2 - \lambda\mathbf{X}_1 = 0 \\ \frac{1}{2}\frac{\partial C}{\partial X_2} &= \mathbf{B}^\top\mathbf{A} \cdot \mathbf{X}_1 + \mathbf{B}^\top C \cdot \mathbf{X}_2 = 0 \end{aligned}$$

We obtain :

$$\begin{aligned} \mathbf{X}_2 &= -(\mathbf{B}^\top\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{A} \cdot \mathbf{X}_1 \\ \mathbf{E} \cdot \mathbf{X}_1 &= \lambda\mathbf{X}_1 \text{ with } \mathbf{E} = \mathbf{A}^\top\mathbf{A} - \mathbf{A}^\top\mathbf{B}(\mathbf{B}^\top\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{A} \end{aligned}$$

Note that if $\mathbf{X}_1$ is an unitary eigenvector of $\mathbf{E}$ and corresponding to the eigenvalue λ we have $C = \lambda$. So, $\mathbf{X}_1$ is the unitary eigenvector corresponding to the minimal eigenvalue of $\mathbf{E}$. Thus, we obtain $\mathbf{X}_1$ and $\mathbf{X}_2$, and to get $\mathbf{I}$, we just have to note that $I_{12} = Z_0 > 0$.

In ViSP to estimate a pose from points by the linear Lagrange approach that doesn't require an initial pose use:

```
1 pose.computePose(vpPose::LAGRANGE, cMo) ; // cMo doesn't need to be initialised
```

2.1.3 Lowe approach

This approach corresponds to a non linear minimisation as the one described in Section 2.1.4. The difference with the virtual visual servoing approach is in the Jacobian used for the minimisation.

In ViSP to estimate a pose from points by the non linear Lowe approach that requires an initial pose use:

```
pose.computePose(vpPose::LAGRANGE, cMo) ; // cMo doesn't need to be initialised
pose.computePose(vpPose::LOWE, cMo) ;     // cMo is initialised by Lagrange
```

2.1.4 Virtual visual servoing approach

The fundamental principle of the proposed approach is to define the pose computation problem as the dual problem of 2D visual servoing [5] [9]. In visual servoing, the goal is to move a camera in order to observe an object at a given position in the image. This is achieved by minimizing the error between a desired state of the image features $\mathbf{s}^*$ and the current state $\mathbf{s}$. If the vector of visual features is well chosen, there is only one final position of the camera that allows this minimization to be achieved. An explanation will now be given as to why the pose computation problem is very similar.

To illustrate the principle, consider the case of an object with various 3D features $\mathbf{P}$ (for instance, ${}^o\mathbf{P}$ are the 3D coordinates of object points in the object frame). A virtual camera is defined whose position and orientation in the object frame is defined by $\mathbf{r}$. The approach consists in estimating the real pose by minimizing the error Δ between the set of observed data $\mathbf{s}^*$ (usually the position of a set of features in the image, but here when considering the specific case of pose estimation from points, $\mathbf{s}^* = \mathbf{p}$) and the position $\mathbf{s}$ of the same features computed by forward-projection according to the current pose:

$$\Delta = \sum_{i=1}^{N} \left(s_i(\mathbf{r}) - s_i^*\right)^2 = \sum_{i=1}^{N} \left(pr_\xi(\mathbf{r}, {}^o\mathbf{P}_i) - s_i^*\right)^2, \tag{2.7}$$

where $pr_\xi(\mathbf{r}, {}^o\mathbf{P})$ is the projection model according to the intrinsic parameters ξ and camera pose $\mathbf{r}$ and where N is the number of considered features. It is supposed here that intrinsic parameters ξ are available but it is possible, using the same approach, to also estimate these parameters.

In this formulation of the problem, a virtual camera (initially at $\mathbf{r}_i$) is moved using a visual servoing control law in order to minimize this error Δ. At convergence, the virtual camera reaches the pose $\mathbf{r_d}$ which minimizes this error. $\mathbf{r_d}$ is the real camera pose we are looking for. An important assumption is to consider that $\mathbf{s}^*$ is computed (from the image) with sufficient precision.

As in classical visual servoing, we define a task function $\mathbf{e}$ to be achieved by the relation:

$$\mathbf{e} = (\mathbf{s}(\mathbf{r}) - \mathbf{s}^*) \tag{2.8}$$

The derivative of equation (2.8) would be given by:

$$\dot{\mathbf{e}} = \frac{\partial \mathbf{e}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{r}} \frac{d\mathbf{r}}{dt} = \mathbf{L}\mathbf{v} \tag{2.9}$$

Matrix $\mathbf{L}$ is classically called the interaction matrix or image Jacobian in the visual servoing community. It links the motion of the features in the image to the camera velocity $\mathbf{v}$. If we specify an exponentially decoupled decrease of the error $\mathbf{e}$:

$$\dot{\mathbf{e}} = -\lambda \mathbf{e} \tag{2.10}$$

where λ is a proportional coefficient that tunes the decay rate, the following control law is obtained:

$$\mathbf{v} = -\lambda \widehat{\mathbf{L}_\mathbf{s}}^{+} \mathbf{e} \tag{2.11}$$

where $\widehat{\mathbf{L}}^{+}$ is the pseudo inverse[1] of $\widehat{\mathbf{L}_\mathbf{s}}$ and where $\widehat{\mathbf{L}_\mathbf{s}}$ is a model of $\mathbf{L}$.

Finally, Rodrigues' formula is then used to map the velocity vector $\mathbf{v}$ to its corresponding instantaneous displacement allowing the pose to be updated. To apply the update to the displacement between the object and camera, the exponential map (see for example [10]) is applied using homogeneous matrices resulting in:

$${}^{c}\mathbf{M}_{o}^{k+1} = {}^{c}\mathbf{M}_{o}^{k} \exp(\mathbf{v}) \tag{2.12}$$

where k denotes the number of iteration of the minimization process.

In ViSP to estimate a pose from points by the non linear virual visual servoing approach that requires an initial pose use:

```
pose.computePose(vpPose::LAGRANGE, cMo) ;   // cMo doesn't need to be initialised
pose.computePose(vpPose::VIRTUAL_VS, cMo) ; // cMo is initialised by Lagrange
```

The same sequencing of pose estimation can also be done using:

```
pose.computePose(vpPose::LAGRANGE_VIRTUAL_VS, cMo) ; // cMo is initialised by Lagrange
```

It is also possible to initialise the pose by Dementhon before virtual visual servoing by using:

```
pose.computePose(vpPose::DEMENTHON_VIRTUAL_VS, cMo) ; // cMo is initialised by Dementhon
```

With virtual visual servoing pose estimation it is also possible to enable the computation of the pose parameters covariance matrix. The covariance matrix could then give an estimation feedback:

```
pose.setCovarianceComputation(true) ;
pose.computePose(vpPose::DEMENTHON_VIRTUAL_VS, cMo) ; // cMo is initialised by Dementhon
std::cout << "Covariance matrix: " << pose.getCovarianceMatrix() << std::endl ;
```

2.1.5 Ransac approach

The Ransac procedure is opposite to that of conventional techniques. Rather than using as much of data as possible to obtain an initial solution and then attempting to eliminate the invalid data points, Ransac uses a small initial data set and enlarges this set with consistent data when possible [6] [7].
If we consider the pose estimation, Ransac approach would be to select 4 points (as this is the minimum number of points necessary to compute a pose), compute the pose from them and check the number of points (in the entire data set) that are projecting close enough from their initial projection to be considered as inliers. Afterwards, it would compute the final pose from all the inliers.

The following Algorithm 2 details the Ransac approach for pose computation in ViSP. In this algorithm we suppose that the points in entry already have their 2D projection and 3D world coordinates matched. We also suppose that the minimum reprojection error used as a threshold and the minimum number of inliers to reach a consensus are known.

[1] In our case since the number of rows is greater than the number of columns the pseudo inverse of a matrix $\mathbf{A}$ is defined by: $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$ where $\mathbf{A}^{\top}$ is the transpose of $\mathbf{A}$.

Algorithm 2 Ransac iterative resolution used to pose estimation

1: Initialisation
2: **while** not enough inliers **AND** max iteration not reached **do**
3: Pick randomly 4 points.
4: Compute the pose $\mathbf{r}$ from the random points with Dementhon approach (see Section 2.1.1)
5: Compute the residual r from the obtained pose
6: **if** $r < threshold$ **then**
7: **for** each point ${}^{o}\mathbf{P}_i$ **do**
8: Compute the error between the projection of the point using the estimated pose $\mathbf{r}$ and the 2D coordinates of the point:
$$error = ||pr_{\xi}(\mathbf{r}, {}^{o}\mathbf{P}_i) - \mathbf{p}_i||$$
9: **if** error < threshold **then**
10: The points is an inlier.
11: **end if**
12: **end for**
13: **end if**
14: **end while**
15: Recompute the pose using all the inliers with a virtual visual servoing approach by initialising with Lagrange (see Section 2.1.4)

In ViSP pose estimation using Ransac approach is achieved by:

```
pose.computePose(vpPose::RANSAC, cMo) ; // cMo doesn't need an initialisation
```

2.1.6 Another point matching tool

This point matching tool doesn't match points from their descriptors as in 3.1 but uses Ransac pose estimation approach presented in Section 2.1.5 to determine the inliers that correspond to matched 2D and 3D coordinates of points. To this end, the approach takes 2D point projections and 3D point coordinates as entry and create every possible pair of 2D and 3D points to determine which are the inliers that match together. Moreover, the pose of the inliers is given as output.

```
/*!
  Match a vector p2D of  2D point (x,y)  and  a vector p3D of 3D points
  (X,Y,Z) using the Ransac algorithm.

  At least numberOfInlierToReachAConsensus of true correspondance are required
  to validate the pose

  The inliers are given in a vector of vpPoint listInliers.

  The pose is returned in cMo.

  \param p2D : Vector of 2d points (x and y attributes are used).
  \param p3D : Vector of 3d points (oX, oY and oZ attributes are used).
  \param numberOfInlierToReachAConsensus : The minimum number of inlier to have
  to consider a trial as correct.
  \param threshold : The maximum error allowed between the 2d points and the
  reprojection of its associated 3d points by the current pose (in meter).
  \param ninliers : Number of inliers found for the best solution.
  \param listInliers : Vector of points (2d and 3d) that are inliers for the best solution.
```

```
  \param cMo : The computed pose (best solution).
  \param maxNbTrials : Maximum number of trials before considering a solution
  fitting the required \e numberOfInlierToReachAConsensus and \e threshold
  cannot be found.
*/
static void findMatch(std::vector<vpPoint> &p2D,
                      std::vector<vpPoint> &p3D,
                      const int &numberOfInlierToReachAConsensus,
                      const double &threshold,
                      unsigned int &ninliers,
                      std::vector<vpPoint> &listInliers,
                      vpHomogeneousMatrix &cMo,
                      const int &maxNbTrials = 10000);
```

Listing 2.2: Point matching through Ransac

The static function `vpPose::findMatch()` *return the resulting matched points in* `listInliers` *vector. Note that for a perfect matching,* `threshold` *should be equal to 0 and* `numberOfInlierToReachAConsensus` *to the minimum number of 2D points or 3D points (these sizes can be differents). A complete example showing the usage of this function is available in ViSP (*`test/pose/testFindMatch.cpp`*).*

2.2 Pose estimation from any visual features

Contrary to the pose estimation from points introduced in 2.1, this section shows how to compute the pose from any visual features including not only points, but also segments, lines, vanishing points, 3D points, ellipses, and even your own features. The algorithm used to estimate the pose is based on a virtual visual servoing approach described in 2.1.4. It unfortunately hardly depends on the initialisation. However, this time we won't be able to use Dementhon (see 2.1.1) or Lagrange (see 2.1.2) approaches for the initialisation (as the features we might use for the pose computation won't be just points).

2.2.1 Basic usage

In ViSP pose estimation from any visual features is implemented in `vpPoseFeatures` *class. The features used as input could be added with* `addFeaturePoint()`, `addFeaturePoint3D()`, `addFeatureVanishingPoint()`, `addFeatureSegment()`, `addFeatureEllipse()` *and* `addFeatureLine()`. *Once at least six features are added, the* `computePose()` *function does the pose estimation. Listing 2.3 gives an example of* `vpPoseFeatures` *usage:*

```
#include <visp/vpConfig.h>
#include <visp/vpPoseFeatures.h>
#include <iostream>

int main()
{
  vpHomogeneousMatrix cMo(0., 0., 1., vpMath::rad(0), vpMath::rad(0), vpMath::rad(60));

  vpPoseFeatures pose;

  // 2D Point Feature
  vpPoint ptFP;
  ptFP.setWorldCoordinates(0.0,-0.5,0.0);

  // 3D point Feature
  vpPoint ptFP3D;
  ptFP3D.setWorldCoordinates(0.0,0.0,-1.5);
```

```
18
19    // Segment Feature
20    vpPoint pt1FS, pt2FS;
21    pt1FS.setWorldCoordinates(-0.5,-0.5/2.0,0.0);
22    pt2FS.setWorldCoordinates(0.5,0.5/2.0,0.0);
23
24    // Line Feature
25    vpLine line;
26    line.setWorldCoordinates(0.0,1.0,0.0,.0, 0.0,0.0,1.0,0.0);
27    // Horizontale line resulting from the intersection of planes y=0 and z=0
28
29    // Ellipse Feature
30    vpCircle circle;
31    circle.setWorldCoordinates(0.0, 0.0, 1.0 , 0.0, 0.0, 0.0, 0.25);
32    // Intersection of plane z=0 and a sphere with radius R=0.25m
33
34    // Vanishing Point Feature
35    vpLine l1;
36    l1.setWorldCoordinates(0.0,1.0,0.2,0.0,
37                           1.0,0.0,0.0,-0.25);
38    vpLine l2;
39    l2.setWorldCoordinates(0.0,1.0,0.2,0.0,
40                             -1.0,0.0,0.0,-0.25);
41
42    ptFP.project(cMo);
43    ptFP3D.project(cMo);
44    pt1FS.project(cMo);
45    pt2FS.project(cMo);
46    line.project(cMo);
47    circle.project(cMo);
48    l1.project(cMo);
49    l2.project(cMo);
50
51    pose.addFeaturePoint(ptFP);
52    pose.addFeaturePoint3D(ptFP3D);
53    pose.addFeatureSegment(pt1FS,pt2FS); // The feature segment is built from 2 points
54    pose.addFeatureLine(line);
55    pose.addFeatureEllipse(circle);
56    pose.addFeatureVanishingPoint(l1,l2); // The feature vanishing point is built from 2 lines
57
58    // Constant gain used in the vitual visual servoing
59    // v = -lambda L+ (s-s*)
60    pose.setLambda(0.6);
61
62    // Initial value of the pose to estimate
63    vpHomogeneousMatrix cMo2(0.4, 0.3, 1.5, vpMath::rad(0), vpMath::rad(0), vpMath::rad(0));
64    pose.computePose(cMo2);
65
66    std::cout << "Resulting cMo : " << std::endl;
67    std::cout << vpPoseVector(cMo2).t() << std::endl;
68
69   return 0;
70  }
```

Listing 2.3: Example of pose estimation with `vpPoseFeatures`

2.2.2 Advanced usage with C++11

The thing is that we wanted to go a little bit further... Actually the basic usage shows how to compute the pose from ViSP visual features. But thanks to C++11, you will see that it is also possible to use your own features in our pose computation class.

What is C++11? It is a new norm of the C++ language. It introduces new fonctionnality like the variadic templates, and a new version of the standard library where you can find classes like `std::tuple`. Unfortunately, we won't develop here about new fonctionnalities of C++11, but just see how it can be benefic for our pose estimation class. We will see now how to use the `vpPoseFeatures` class to compute the pose from non-ViSP visual features.

Let's first define one classe and one static function for the example:

```
class vp_createPointClass{
public:
  int value;

  vp_createPointClass() : value(0){}

  int vp_createPoint(vpFeaturePoint &fp,const vpPoint &v){
    value += 1;
    vpFeatureBuilder::create(fp,v);
    return value;
  }
};

void vp_createLine(vpFeatureLine &fl,const vpLine &v){
  vpFeatureBuilder::create(fl,v);
}
```

Listing 2.4: Class and function that will be used in the pose estimation

`vp_createPoint()` and `vp_createLine()` represent the functions that will be used to create the feature point `fp` and line `fl`.

Note that here, to have a working example, we use `vpClasses` but it could have been your own classes as parameters of these functions. The only constraint is that the first parameter of your `create()` function has to extend `vpBasicFeature` and the others have to extend `vpForwardProjection`.

Now let's see how it can be used in our `vpPoseFeature` class:

```
int main()
{
  vpHomogeneousMatrix cMo(0., 0., 1., vpMath::rad(0), vpMath::rad(0), vpMath::rad(60));

  vpPoseFeatures pose;
  vpPoint pts[3];

  //2D Point Feature
  pts[0].setWorldCoordinates(0.0,-0.25,0);

  //Segment Feature
  pts[1].setWorldCoordinates(0.0,0.25,0);
  pts[2].setWorldCoordinates(-0.25,0.25,0);

  //Line Feature
  vpLine line;
  line.setWorldCoordinates(0.0,1.0,0.0,.0,0.0,0.0,1.0,0.0);

  pts[0].project(cMo);
  pts[1].project(cMo);
  pts[2].project(cMo);
  line.project(cMo);

  vpFeaturePoint fp;
```

```
25    vpFeatureLine fl;
26    vpFeatureSegment fs;
27
28    void ( *ptr )(vpFeatureSegment& , vpPoint& , vpPoint&)
29      = &vpFeatureBuilder::create;
30
31    int (vp_createPointClass::*ptrClass)(vpFeaturePoint&, const vpPoint&)
32      = &vp_createPointClass::vp_createPoint;
33
34    vp_createPointClass cpClass;
35
36    pose.addSpecificFeature(&cpClass, ptrClass, fp, pts[0]);
37    pose.addSpecificFeature(&vp_createLine, fl, line);
38    pose.addSpecificFeature(ptr, fs, pts[], pts[2]);
39
40    pose.setLambda(0.6);
41
42    vpHomogeneousMatrix cMo2(0.4, 0.3, 1.5, vpMath::rad(0), vpMath::rad(0), vpMath::rad(0));
43    pose.computePose(cMo2);
44
45    std::cout << "Resulting cMo : " << std::endl;
46    std::cout << vpPoseVector(cMo2).t() << std::endl;
47
48    return 0;
49  }
```

Listing 2.5: Pose estimation with specific features

As for `vpPoseFeatures::addFeaturePoint()` or `vpPoseFeatures::addFeatureLine()`, `vpPoseFeatures::addSpecificFeature()` allows to add a visual feature to the list of features that will be used to compute the pose.
It takes, as parameters, a pointer toward the function that will be used to create the feature (equivalent to `vpFeatureBuilder::create()` functions). Then the next parameter has to be a `vpBasicFeature` and the lattest has to be a `vpForwardProjection` (Type has, of course, to match with your `create()` function parameters).

A complete example of this class is available in ViSP (`test/pose/testPoseFeatures.cpp`).

2.3 3D model-based tracker using robust virtual visual servoing

The 3D model-based tracker aims to track 3D objects in the scene by calculating the pose between the camera and the objects [2] [3]. Here, non-linear pose estimation is formulated by means of a virtual visual servoing approach. In this context, the derivation of point-to-curves interaction matrices are used for different 3D geometrical primitives including straight lines and cylinders. A local moving edges tracker is used in order to provide real-time tracking of points normal to the object contours. Robustness is obtained by integrating a M-estimator into the visual control law via an iteratively re-weighted least squares implementation.

2.3.1 Overview

As described in Section 2.1.4, the principle of the proposed approach is to define the pose computation problem as the dual problem of 2D visual servoing by using a virtual visual servoing approach.

In visual servoing, the control law that performs the minimization of Δ (see 2.7) could be handled using a least squares approach [5] [9]. However, when outliers are present in the measures, a robust estimation is required. M-estimators can be considered as a more general form of maximum likelihood estimators [8].

They are more general because they permit the use of different minimization functions not necessarily corresponding to normally distributed data. Many functions have been proposed in the literature which allow uncertain measures to be less likely considered and in some cases completely rejected. In other words, the objective function is modified to reduce the sensitivity to outliers. The robust optimization problem is then given by:

$$\Delta_{\mathcal{R}} = \sum_{i=1}^{N} \rho\Big(s_i(\mathbf{r}) - s_i^*\Big), \tag{2.13}$$

where $\rho(u)$ is a robust function [8] that grows sub-quadratically and is monotonically nondecreasing with increasing $|u|$. Iteratively Re-weighted Least Squares (IRLS) is a common method of applying the M-estimator. It converts the M-estimation problem into an equivalent weighted least-squares problem.

To embed robust minimization into visual servoing, a modification of classical control laws is required to allow outlier rejection.

2.3.1.1 Robust Control Law

The objective of the control scheme is to minimize the objective function given in equation (2.13). This new objective is incorporated into the control law in the form of a weight which is given to specify a confidence in each feature location. Thus, the error to be regulated to zero is defined as:

$$\mathbf{e} = \mathbf{D}(\mathbf{s}(\mathbf{r}) - \mathbf{s}^*), \tag{2.14}$$

where $\mathbf{D}$ is a diagonal weighting matrix given by

$$\mathbf{D} = \text{diag}(w_1, \ldots, w_k)$$

Each w_i reflects the confidence in the i-th feature. The computation of weights w_i is described in [8]. If $\mathbf{D}$ was constant, the derivative of equation (2.14) would be given by:

$$\dot{\mathbf{e}} = \frac{\partial \mathbf{e}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{r}} \frac{d\mathbf{r}}{dt} = \mathbf{DLv}, \tag{2.15}$$

where $\mathbf{v}$ is the camera velocity screw and $\mathbf{L}$ is called the interaction matrix related to $\mathbf{s}$. This matrix depends on the value of the image features $\mathbf{s}$ and their corresponding depth Z in the scene (which is available here). If an exponential decrease of the error $\mathbf{e}$ is specified:

$$\dot{\mathbf{e}} = -\lambda \mathbf{e}, \tag{2.16}$$

where λ is a positive scalar, the following control law is obtained

$$\mathbf{v} = -\lambda (\widehat{\mathbf{DL}_\mathbf{s}})^{+} \mathbf{D}\big(\mathbf{s}(\mathbf{r}) - \mathbf{s}^*\big), \tag{2.17}$$

where $(\widehat{\mathbf{DL}_\mathbf{s}})^{+}$ is the pseudo inverse[2] of $\widehat{\mathbf{DL}_\mathbf{s}}$ and where $\widehat{\mathbf{DL}_\mathbf{s}}$ is a model of $\mathbf{DL}$.

As for virtual visual servoing described in Section 2.1.4, Rodrigues' formula is then used to map the velocity vector $\mathbf{v}$ to its corresponding instantaneous displacement allowing the pose to be updated.

[2] In our case since the number of row is greater that the number of columns the pseudo inverse of a matrix $\mathbf{A}$ is defined by: $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$ where $\mathbf{A}^{\top}$ is the transpose of $\mathbf{A}$.

2.3.1.2 Visual features for model-based tracking

In the proposed control law 2.17, the visual features $\mathbf{s}$ that are used are composed of a set of distances between local point features obtained from a fast image processing step and the contours of a more global 3D model. In this case, the desired value $\mathbf{s}^*$ is zero. The assumption is made that the contours of the object in the image can be described as piecewise linear segments. All distances are then treated according to their corresponding segments.

The derivation of the interaction matrix that links the variation of the distance between a fixed point and a moving straight line to the virtual camera motion is now given. In Figure 2.1, $\mathbf{p}$ is the tracked point and $\mathbf{l}(\mathbf{r})$ is the current line feature position.

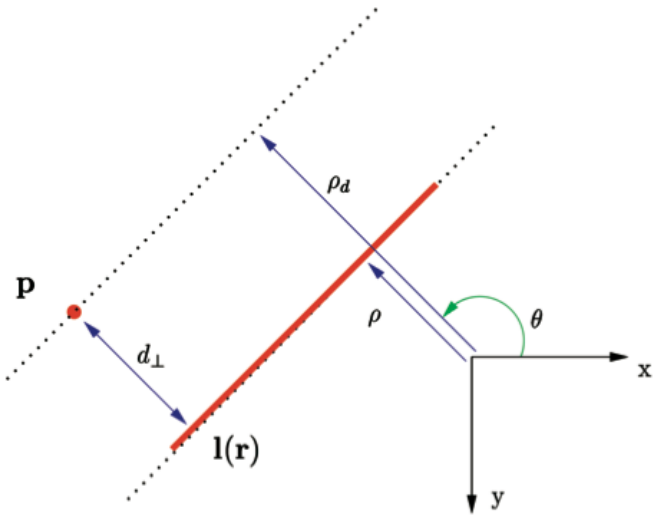

Figure 2.1: Distance of a point to a straight line

The position of the straight line is given by its polar coordinates representation,

$$x\cos\theta + y\sin\theta = \rho, \forall(x,y) \in \mathbf{l}(\mathbf{r}), \tag{2.18}$$

The distance between point $\mathbf{p}$ and line $\mathbf{l}(\mathbf{r})$ can be characterized by the distance $d_\perp$ perpendicular to the line. In other words the distance parallel to the segment does not hold any useful information unless a correspondence exists between a point on the line and $\mathbf{p}$ (which is not the case). Thus the distance feature from a line is given by:

$$d_l = d_\perp(\mathbf{p}, \mathbf{l}(\mathbf{r})) = \rho(\mathbf{l}(\mathbf{r})) - \rho_d, \tag{2.19}$$

where

$$\rho_d = x_d\cos\theta + y_d\sin\theta, \tag{2.20}$$

with x_d and y_d being the coordinates of the tracked point. Thus,

$$\dot{d}_l = \dot{\rho} - \dot{\rho_d} = \dot{\rho} + \alpha\dot{\theta}, \tag{2.21}$$

where $\alpha = x_d\sin\theta - y_d\cos\theta$. Deduction from (2.21) gives $\mathbf{L}_{d_l} = \mathbf{L}_\rho + \alpha\mathbf{L}_\theta$. The interaction matrix related to d_l can be thus derived from the interaction matrix related to a straight line given by (see [5] for its

complete derivation):

$$\begin{array}{l} \mathbf{L}_\theta = [\, \lambda_\theta \cos\theta \;\; \lambda_\theta \sin\theta \;\; -\lambda_\theta \rho \;\; \rho\cos\theta \;\; -\rho\sin\theta \;\; -1 \,] \\ \mathbf{L}_\rho = [\, \lambda_\rho \cos\theta \;\; \lambda_\rho \sin\theta \;\; -\lambda_\rho \rho \;\; (1+\rho^2)\sin\theta \;\; -(1+\rho^2)\cos\theta \;\; 0 \,] \end{array} \tag{2.22}$$

where $\lambda_\theta = (A_2 \sin\theta - B_2 \cos\theta)/D_2$, $\lambda_\rho = (A_2 \rho \cos\theta + B_2 \rho \sin\theta + C_2)/D_2$, and $A_2X + B_2Y + C_2Z + D_2 = 0$ is the equation of a 3D plane which the line belongs to.

From (2.21) and (2.22) the following interaction matrix is obtained:

$$\mathbf{L}_{d_l} = \begin{bmatrix} \lambda_{d_l} \cos\theta \\ \lambda_{d_l} \sin\theta \\ -\lambda_{d_l} \rho \\ (1+\rho^2)\sin\theta - \alpha\rho\cos\theta \\ -(1+\rho^2)\cos\theta - \alpha\rho\sin\theta \\ -\alpha \end{bmatrix}^\top , \tag{2.23}$$

where $\lambda_{d_l} = \lambda_\rho + \alpha\lambda_\theta$.

Let it be noted that the case of a distance between a point and the projection of a cylinder is very similar to this case. Indeed, if the considered 3D object is a cylinder, its projection in the image can be represented by two straight lines (in all non degenerated cases) and parameters $A_2/D_2, B_2/D_2$, and C_2/D_2 can have the same value for both lines. More precisely, we have:

$$\begin{cases} A_2/D_2 = -X_o/(X_o^2 + Y_o^2 + Z_o^2 - R^2) \\ B_2/D_2 = -Y_o/(X_o^2 + Y_o^2 + Z_o^2 - R^2) \\ C_2/D_2 = -Z_o/(X_o^2 + Y_o^2 + Z_o^2 - R^2) \end{cases}$$

where R is the radius of the cylinder and where X_o, Y_o and Z_o are the coordinates of the point of the axis of the cylinder the nearest of the camera optical center.

2.3.1.3 Visual features low-level tracking

When dealing with image processing, the normal displacements are evaluated along the projection of the object model contours using the spatio-temporal Moving Edges algorithm (ME) [1]. One of the advantages of the ME method is that it does not require any prior edge extraction. Only point coordinates and image intensities are manipulated. The ME algorithm can be implemented with convolution efficiency, and leads to real-time computation [1, 11]. The process consists in searching for the correspondent p^{t+1} in image I^{t+1} of each point p^t. A 1D search interval $\{Q_j, j \in [-J, J]\}$ is determined in the direction δ of the normal to the contour (see Figure 2.2a and Figure 2.2b). For each point p^t and for each position Q_j lying in the direction δ, a criterion ζ_j is computed. This criterion is nothing but the convolution values computed at Q_j using a pre-determined mask M_δ function of the orientation of the contour.

The new position p^{t+1} is given by:

$$Q^{j^*} = \arg \max_{j \in [-J,J]} \zeta_j \text{ with } \zeta_j = \mid I^t_{\nu(p^t)} * M_\delta + I^{t+1}_{\nu(Q_j)} * M_\delta \mid$$

where $\nu(.)$ is the neighbourhood of the considered pixel.

This low level search produces a list of k points which are used to calculate distances from corresponding projected contours.

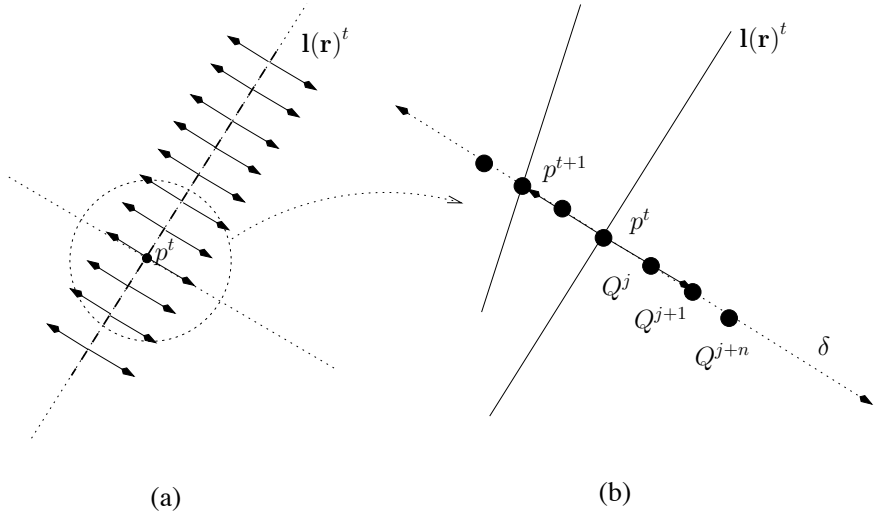

Figure 2.2: Determining points position in the next image using the ME algorithm: (a) calculating the normal at sample points, (b) sampling along the normal.

2.3.2 Tracker usage

ViSP 3D model-based tracker is implemented in `vpMbEdgeTracker` *class. It inherits from the generic* `vpMbTracker` *class that has mainly virtual methods.*

Construction. *The tracker can be built using the default constructor:*

```
1 vpMbEdgeTracker tracker;
```

Settings initialisation. *The tracker needs to be initialized with some settings. These settings could be read from an xml configuration file loaded by* `loadConfigFile()`*. An example of such an xml file is provided in Listing 2.11.*

```
1 tracker.loadConfigFile("object.xml");
```

Below, we give a description of the tracker parameters that should be initialised in the xml file ("object.xml"):

- `ecm:mask:size`*: Corresponds to the size of the convolution masks M_δ introduced in section 2.3.1.3 and used to detect an edge.*
- `ecm:mask:nb_mask`*: Number of mask used to determine the object contour. This number determines the precision of the normal of the edge for every sample. If the precision is 2 degrees, then there are 360/2 = 180 masks.*
- `ecm:range:tracking`*: Seek range in pixels on both sides of the reference pixel to search a contour.*
- `ecm:contrast:edge_threshold`*: Set the likelihood threshold used to determined if the moving edge is valid or not. This threshold depends on the mask size.*
- `ecm:contrast:mu1`*: Set the minimum image contrast $1 - \mu_1$ allowed to detect a contour.*

- `ecm:contrast:mu2`*: Set the maximum image contrast* $1 + \mu_2$ *allowed to detect a contour.*
- `sample:step`*: Set the minimum distance in pixels between two discretized points.*
- `sample:nb_sample`*: Set the number of points to track. The complexity of the algorithm is linear to the number of points to track.*
- `camera:width`*: Image width. This parameters is not used.*
- `camera:Height`*: Image height. This parameters is not used.*
- `camera:u0`*: Intrinsic camera parameter that corresponds to the horizontal coordinate in pixel of the principal point.*
- `camera:v0`*: Intrinsic camera parameter that corresponds to the vertical coordinate in pixel of the principal point.*
- `camera:px`*: Intrinsic camera parameter that corresponds to the ratio between the focal length and the size of a pixel along the x axis.*
- `camera:py`*: Intrinsic camera parameter that corresponds to the ratio between the focal length and the size of a pixel along the y axis. If the pixels are square,* `px=py`*.*

Note that the tracker is not able to consider the intrinsic camera distorsion parameters. If the distorsion has to be considered, the images need first to be undistorted before the tracking is applied.

Instead of loading an xml file, it is also possible to initialise or modify the tracker settings using `setCameraParameters()` *and* `setMovingEdge()` *methods:*

```
vpCameraParameters cam(px, py, u0, v0);  // camera:px, py, u0, v0
tracker.setCameraParameters(cam); // Initialise the intrinsic camera parameters without distorsion
vpMe me;
me.setMaskSize(5);          // ecm:mask:size
me.setMaskNumber(180);      // ecm:mask:nb_mask
me.setRange(7);             // ecm:range:tracking
me.setThreshold(500);       // ecm:contrast:edge_threshold
me.setMu1(0.5);             // ecm:contrast:mu1
me.setMu2(0.5);             // ecm:contrast:mu2
me.setSampleStep(4);        // sample:step
me.setNbTotalSample(250);   // sample:nb_sample
tracker.setMovingEdge(me);  // Initialise the moving edges
```

Model initialisation. *Then, the 3D model of the object to track has to be loaded. As explained in section 2.3.3 the model of the object could be in* `vrml` *or* `cao` *format.*

```
tracker.loadModel("object.wrl");
tracker.loadModel("object.cao");
```

Pose initialisation. *The tracker needs also an initial pose of the object in the camera frame. Three different ways are allowed to set this initial pose. Firstly using the blocking* `initClick()` *method that requires a human click interaction. In that case, the 3D coordinates in the object frame of at least four points are required. The user has than to click in the image on the 2D corresponding points. Given the 2D and 3D*

coordinates of these points, a pose computation is performed to initialise the tracker. This step requires intrinsic camera parameters that are read from the xml config file. The coordinates of the 3D points could be provided in a file with `.init` *extension. The structure of this file is given Listing 2.7 and an example is given Listing 2.14.*

```
tracker.initClick(I, "object.init");
```

It is also possible to provide the 3D coordinates of the points using:

```
vpPoint p3d;
std::vector vector_p3d;
p3d.setWorldCoordinates(Xi, Yi, Zi);
vector_p3d.push_back(p3d);
...
tracker.initClick(I, vector_p3d); // The vector should contain at least 4 points
```

The second way to set the initial pose is to use `initFromPoints()`*. This method doesn't require an user interaction. Here the 2D and 3D coordinates of the points used to compute the initial pose are given. These coordinates could be provided in an* `.init` *file. The structure of this file is given Listing 2.8.*

```
tracker.initFromPoints(I, "object.init");
```

It is also possible to provide the 2D and 3D coordinates of the points as vectors using:

```
vpPoint p3d;
std::vector vector_p3d;
p3d.setWorldCoordinates(Xi, Yi, Zi); // 3D coordinates of the points in the object frame
vector_p3d.push_back(p3d);
...
vpImagePoint p2d;
std::vector vector_p2d;
p2d.set_uv(ui, vi); // 2D pixels coordinates in the image
vector_p2d.push_back(p2d);
...
tracker.initFromPoints(I, vector_p2d, vector_p3d); // The vectors should contain at least 4 points
```

The third way to set the initial pose is to use `initFromPose()`*. This method doesn't require an used interaction. Here the pose of the object in the camera frame is given. These pose implemented in* `vpPoseVector` *format where the rotation uses the in* $\theta\mathbf{u}$ *representation could be provided in a* `.pos` *text file. The structure of this file is given Listing 2.9.*

```
tracker.initFromPose(I, "object.pos");
```

It is also possible to provide the pose of the object in the camera frame using:

```
vpPoseVector cPo;
// initialize cPo
vpHomogeneousMatrix cMo(cPo);
tracker.initFromPose(I, cMo);
```

Advanced initialisation. *It is possible to turn on the computation of the estimated pose parameters covariance matrix using:*

```
tracker.setCovarianceComputation(true);
```

Tracking. *The tracking of the object in an image is performed using* `track()` *method. The resulting pose is available using* `getPose()`. *It is also possible to get the covariance matrix of the pose parameters using* `getCovarianceMatrix()`. *This matrix could be used to have a feedback of the tracking quality.*

```
tracker.track(I);
vpHomogeneousMatrix cMo = tracker.getPose();
vpMatrix covariance = tracker.getCovarianceMatrix();
```

2.3.3 3D model

The 3D model of the object to track can be represented in two different formats: `vrml` or `cao` which is specific to ViSP. When loading the model, with the `loadModel(...)` method, the extension of the file (either *.wrl* or *.cao*) is used to detect the format.

The design of the 3D model need to respect the two following restrictions:

- Firstly, it is important to note that the faces of the modelled object have to be oriented so that the normal of the face goes out of the object (see Fig. 2.3 and Listing 2.12). This assumption is used by the tracker to determine if a face is visible. Note that a line is always visible.
- Secondly, it is crutial that the faces of the model are not systematically modelled by triangles (see Fig. 2.3.a). The reason is that all the lines that appear in the model must match object edges.

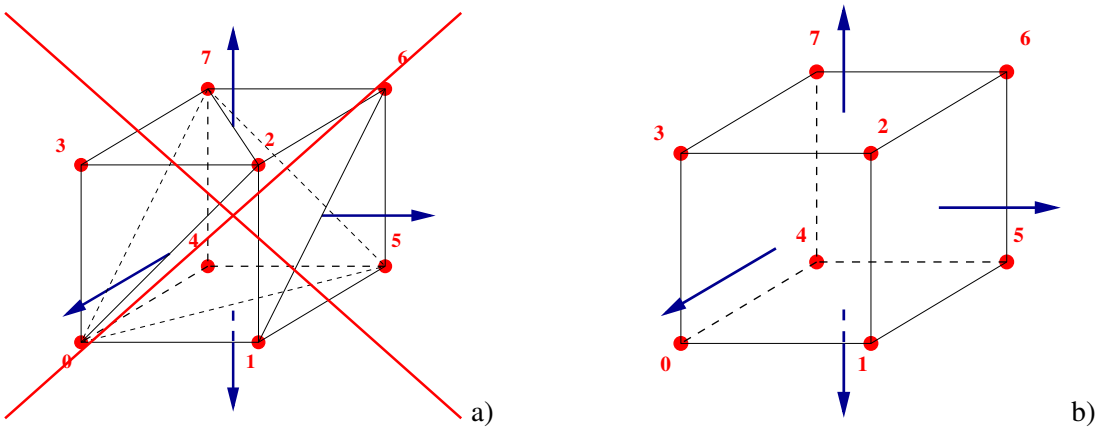

Figure 2.3: The 3D model of the object to track must be designed such as the normal of the faces are going out of the faces and such as the lines of the model match the object edges. In a) the faces of the cube are modelled by two triangles. It will be not possible to track this object since lines between points 1-6, 2-7, 0-7, ...don't match cube edges. The model presented in b) would be possible to track. The faces are modelled as squares.

2.3.3.1 Model in vrml format

The `vrml` (Virtual Reality Modeling Language) format is a standard for representing 3D objects. The parser implemented in ViSP is based on the Coin3D library (`http://www.coin3d.org/`). The format can be generated from a 3D modeller (like, for example, Blender). This simplifies the creation of a complex model. But in that case, as explained previously be sure that all the lines of the model match object edges and that

the normal of the faces are going outside the object (see Fig. 2.3). Listing 2.12 corresponds to the `vrml` model of the cube introduced Fig. 2.3. Note that ViSP data set provides other examples of `vrml` files in `ViSP-images/mbt` folder.

2.3.3.2 Model in cao format

This is a text based format developped for ViSP that doesn't require any third party library. To illustrate the model structure, we provide hereafter in Listing 2.6 the `cao` model of the objects presented Fig 2.4. The `cao` format allows to define a face with a set of lines (see lines 14 and 15 Listing 2.6) or a set of points (see lines 16 to 20). A cylinder is defined by two points on its revolution axis and by its radius (see line 21 and 22). All the coordinates must be represented in the same global object frame. This means that if two similar structures, rigidely linked, are part of the object, it is not possible to have one structure to represent the first one and a transformation to represent the second one.

```
1  V1 //standard flag (for an optional extension).
2  7 // Number of points describing the object
3  -2 -2 0  // Five 3D coordinates of the points on the pyramid in the object frame (in m.)
4   0 -2 0
5   0  0 0
6  -2  0 0
7  -1 -1 2
8   3 -2 0  // Two 3D coordinates of the points on cylinder axis in the object frame (in m.)
9   3  2 0
10 3 // Number of lines to track
11 1 2 //  \
12 2 4 //  | Index of the points representing the extremities of the lines
13 4 1 // /
14 1 // Number of faces to track defined by a set of lines previously described
15 3 0 1 2  // Face described as follow : nbLine IndexLine1 indexLine2 ... indexLineN
16 4 // Number of faces to track defined by a set of points previously described
17 4 0 3 2 1 // Face described as follow : nbPoint IndexPoint1 IndexPoint2 ... IndexPointN
18 3 0 1 4
19 3 0 4 3
20 3 2 3 4
21 1 // Number of cylinder
22 5 6 0.5 // Index of the limits points on the cylinder axis (used to know the 'height'
23         // of the cylinder) and radius of the cyclinder (in m.)
```

Listing 2.6: Description of a 3D model defined in `cao` format and corresponding to the objects presented Fig 2.4. For the sake of the simplicity the comments are represented with standard C style. However, only the lines beginning with the symbol # are indeed comments.

Moreover, listing 2.13 corresponds to the `vrml` model of the cube introduced Fig. 2.3. Note that ViSP data set provides other examples of `cao` files in `ViSP-images/mbt` folder.

2.3.4 Pose initialisation

Depending on the method `initClick()`, `initFromPoints()` or `initFromPose()` used to set the tracker initial pose (see 2.3.2) different files my be used.

```
1  4 // Number of 3D points in the file (minimum is four)
2  0.01 0.01 0.01    //  \
3  ...               //  | 3D coordinates in meters in the object frame
4  0.01 -0.01 -0.01  //  /
```

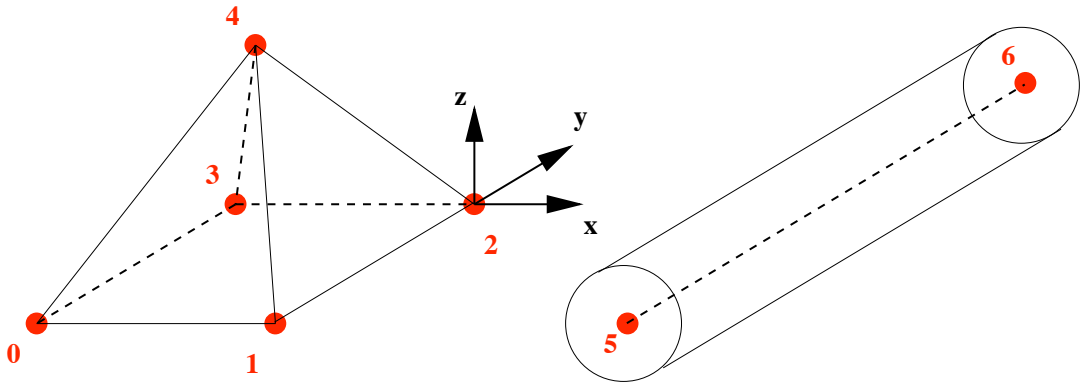

Figure 2.4: Index of the points that are used to describe the `cao` model presented Listing 2.13).

Listing 2.7: Structure of a `.init` text file used by `initClick()`. In this file the coordinates of at least four 3D points are defined.

```
4 // Number of 3D points in the file (minimum is four)
0.01 0.01 0.01    //  \
...               //  | 3D coordinates in meters in the object frame
0.01 -0.01 -0.01  //  /
4 // Number of image points in the file (has to be the same as the number of 3D points)
100 200    //  \
...        //  | 2D coordinates in pixel in the image
50 10      //  /
```

Listing 2.8: Structure of a `.init` text file used by `initFromPoints()`. In this file the coordinates of at least four 3D points and their corresponding pixel coordinates are defined.

```
// The six value of the pose vector
0.0000    // t_x in meters       \
0.0000    // t_y in meters       |
1.0000    // t_z in meters       | Example of value for the pose vector where Z = 1 meter
0.0000    // thetaU_x in radians |
0.0000    // thetaU_y in radians |
0.0000    // thetaU_z in radians /
```

Listing 2.9: Example of a `.pos` text file used by `initFromPose()`. It contains the coordinates of a pose vector corresponding to the pose of the object in the camera frame.

2.3.5 Example

A complete example is available in `example/tracking/mbtTracking.cpp`. The following minimalist example presented in Listing 2.10 shows the main functions used to track a cube defined Fig 2.5:

```
#include <visp/vpMbEdgeTracker.h>
#include <visp/vpImage.h>
#include <visp/vpHomogeneousMatrix.h>
#include <visp/vpCameraParameters.h>

int main()
```

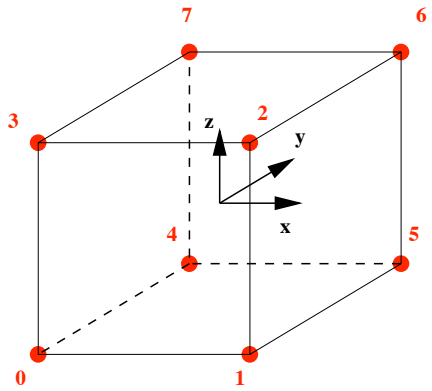

Figure 2.5: Index of the points that are used to describe the cube either in `vrml` format (see Listing 2.12) or in `cao` format (see Listing 2.13).

```
7  {
8    vpMbEdgeTracker tracker;   // Create a model-based tracker.
9    vpImage<unsigned char> I;
10   vpHomogeneousMatrix cMo;   // Pose computed using the tracker.
11   vpCameraParameters cam;
12   vpMatrix covarianceMatrix; // Covariance matrix associated to the pose estimation
13   // Acquire an image
14
15   // Load the configuration of the tracker
16   tracker.loadConfigFile("cube.xml");
17   // Get the camera parameters used by the tracker (from the configuration file "cube.xml").
18   tracker.getCameraParameters(cam);
19   // Load the 3d model, to read .wrl model the 3d party library coin is required, if coin is
20   // not installed .cao file can be used.
21   tracker.loadModel("cube.wrl");
22   // Tell that the covariance matrix has to be computed
23   tracker.setCovarianceComputation(true);
24   // Initialise manually the pose by clicking on the image points associated to the 3d points
25   // containned in the cube.init file.
26   tracker.initClick(I, "cube.init");
27
28   while(true){
29     // Acquire a new image
30     tracker.track(I);                                  // track the object on this image
31     // Get the resulting covariance matrix from the current tracking
32     covarianceMatrix = tracker.getCovarianceMatrix();  // get the covariance matrix
33     cMo = tracker.getPose();                           // get the pose
34     tracker.display(I, cMo, cam, vpColor::darkRed, 1); // display the model at the computed pose.
35   }
36
37   // Cleanup memory allocated by xml library used to parse the xml config file
38   // in vpMbEdgeTracker::loadConfigFile()
39   vpXmlParser::cleanup();
40
41   return 0;
42 }
```

Listing 2.10: Example that shows how to use the model-based tracker class in order to track a cube. To initialise the tracker line 26, the user has to click in the image on the pixels corresponding to the points 0, 3, 2 and 6 defined Fig 2.5 with 3D coordinates defined in Listing 2.14.

This example uses three external files:

- `cube.xml`: for the tracker settings (see line 16 in Listing 2.10). The content of this file is given Listing 2.11;
- `cube.wrl`: for the 3D model (see line 21 in Listing 2.10). The content of this file is given Listing 2.12;
- `cube.init`: for the manual initialisation of the tracker (see line 26 in Listing 2.10). The content of this file is given Listing 2.14. This initialisation is optional if the initial pose is known.

Tracker settings file: The settings file (`cube.xml`) looks like:

```
<?xml version="1.0"?>
<conf>
  <ecm>
    <mask>
      <size>5</size>
      <nb_mask>180</nb_mask>
    </mask>
    <range>
      <tracking>7</tracking>
    </range>
    <contrast>
      <edge_threshold>5000</edge_threshold>
      <mu1>0.5</mu1>
      <mu2>0.5</mu2>
    </contrast>
  </ecm>
  <sample>
    <step>4</step>
    <nb_sample>250</nb_sample>
  </sample>
  <camera>
    <width>640</width>
    <height>480</height>
    <u0>338.7036994</u0>
    <v0>234.5083345</v0>
    <px>547.7367575</px>
    <py>542.0744058</py>
  </camera>
</conf>
```

Listing 2.11: Example of an xml configuration file that contains all the tracker initial settings

This file contains all the parameters for the moving edge presented in Section 2.3.2 as well as the intrinsic parameters of the camera. The xml parser included in ViSP depends on the libxml2 library `http://xmlsoft.org`.

3D model of a cube in vrml format: Listing 2.12 gives a minimalist example of `cube.wrl` corresponding to a $0.2 \times 0.2 \times 0.2$ cube presented Fig 2.5.

```
#VRML V2.0 utf8

DEF fst_0 Group {
children [

# Object "cube"
Shape {

geometry DEF cube IndexedFaceSet {

```

```
11 coord Coordinate {
12 point [
13 -0.1 -0.1 -0.1,
14  0.1 -0.1 -0.1,
15  0.1 -0.1  0.1,
16 -0.1 -0.1  0.1,
17 -0.1  0.1 -0.1,
18  0.1  0.1 -0.1,
19  0.1  0.1  0.1,
20 -0.1  0.1  0.1 ]
21 }
22
23 coordIndex [
24  0,4,5,1,-1,
25  1,5,6,2,-1,
26  6,7,3,2,-1,
27  3,7,4,0,-1,
28  0,1,2,3,-1,
29  7,6,5,4,-1]}
30 }
31
32 ]}
```

Listing 2.12: 3D model of the cube defined in `vrml` format and presented Fig 2.5. Lines 13 to 20 precise the eight vertices coordinates of the cube, respectively for point 0 to 7. Then lines 24 to 29 define the six oriented faces, so that the normal of the face goes outside the cube. More precisely, line 24 corresponds to the closed face that relies points 0, 4, 5 and 1.

This file can usually be generated from a 3D modeller.

3D model of a cube in cao format: Note that it is also possible to define the same cube in `cao` format. The corresponding `cube.cao` file is given Listing 2.13. To use this file in the example presented Listing 2.10, line 21 has to be replaced by:

```
1   tracker.loadModel("cube.cao");
```

```
1  V1
2  8
3  -0.1 -0.1 -0.1,
4   0.1 -0.1 -0.1,
5   0.1 -0.1  0.1,
6  -0.1 -0.1  0.1,
7  -0.1  0.1 -0.1,
8   0.1  0.1 -0.1,
9   0.1  0.1  0.1,
10 -0.1  0.1  0.1
11 0
12 0
13 6
14 4 0 4 5 1
15 4 1 5 6 2
16 4 6 7 3 2
17 4 3 7 4 0
18 4 0 1 2 3
19 4 7 6 5 4
20 0
```

Listing 2.13: 3D model of the cube defined in `cao` format and presented Fig 2.5. Lines 11, 12 and20 indicate respectively that there are no lines, no faces defined with polygons and no cylinder to track. Then line 13 indicates that there are 6 faces defines with points indexes. Each one is defined with 4 points.

Pose initialisation file: The `cube.init` is a text file containing the coordinates of several 3D points (at least 4), in the object frame. The function `initClick(...)` will wait for the user to click on the 2D image points corresponding to these 3D points. The initial pose is computed from this set of 2D/3D correspondences. Listing 2.14 gives an example of an initialisation file.

```
4 // Number of points in the file (minimum is four)
-0.1 -0.1 -0.1   //  \
-0.1 -0.1  0.1   //  | coordinates in the object basis
 0.1 -0.1  0.1   //  |
 0.1  0.1  0.1   // /
```

Listing 2.14: Example of a `cube.init` file used by `initClick()` method to compute the initial pose. The content of this file defines the coordinates of four 3D point of a cube. They correspond to the points 0, 3, 2 and 6 defined Fig 2.5.

Chapter 3

2D motion estimation

This chapter dedicated to homography estimation is under construction...

3.1 Example of use: the `vpPlanarObjectDetector` class

Any algorithm able to detect and match points of interest can be used to compute a relation between a pattern in a reference frame and the same pattern in a current frame.

The class `vpPlanarObjectDetector` *compute the homography between a reference pattern and the current pattern. This class uses the* `vpFernsClassifier` *for the detection and the matching.*

The reference pattern is learned in a reference image. Then, for each new image, the pattern is detected and the homography is computed using the pair of matching points and a robust method (Ransac in this case [6] [7]).

This class is very similar to `vpFernClassifier`*. A minimal prototype for this class is given below:*

```
class vpPlanarObjectDetector
{
  // The fern classifier used to detect the template
  vpFernClassifier fern;
  // The homography computed between the reference and the current template.
  vpHomography homography;
  // The image points in the current image (inliers).
  std::vector< vpImagePoint > currentImagePoints;
  // The image points in the reference image (correponding to currentImagePoints).
  std::vector< vpImagePoint > refImagePoints;
  // Minimal number of inliers to consider the homography correct.
  unsigned int minNbMatching;

  vpPlanarObjectDetector ();
  vpPlanarObjectDetector (const std::string &dataFile, const std::string &objectName);

  unsigned int buildReference (const vpImage< unsigned char > &I);
  unsigned int buildReference (const vpImage< unsigned char > &I, vpImagePoint &iP,
                               unsigned int height, unsigned int width);
  unsigned int buildReference (const vpImage< unsigned char > &I, const vpRect rectangle);

  bool matchPoint (const vpImage< unsigned char > &I);
  bool matchPoint (const vpImage< unsigned char > &I, vpImagePoint &iP,
                   const unsigned int height, const unsigned int width);
  bool matchPoint (const vpImage< unsigned char > &I, const vpRect rectangle);

  void recordDetector (const std::string &objectName, const std::string &dataFile);
```

```
28   void load (const std::string &dataFilename, const std::string &objName);
29
30   void display (vpImage< unsigned char > &I, bool displayKpts=false);
31   void display (vpImage< unsigned char > &Iref, vpImage< unsigned char > &Icurrent,
32                 bool displayKpts=false);
33
34   std::vector< vpImagePoint > getDetectedCorners () const;
35   void getHomography (vpHomography &_H) const;
36
37   void getReferencePoint (const unsigned int _i, vpImagePoint &_imPoint);
38   void getMatchedPoints (const unsigned int _index, vpImagePoint &_referencePoint,
39                          vpImagePoint &_currentPoint);
40
41   void setMinNbPointValidation (const unsigned int _min);
42   unsigned int getMinNbPointValidation () const ;
43 };
```

To consider the homography correct, this class requires, by default, a minimum of 10 *pairs of inliers (pairs of points coherent with the final homography). It is possible to modify this value using the method* `setMinNbPointValidation()`. *The homography, as well as the pair of points used to compute it can be obtained using the methods:* `getHomography()`, `getReferencePoint()` *and* `getMatchedPoints()`. *A complete example of this class is available in ViSP (*`example/key-point/planarObjectDetector.cpp`*).*

Bibliography

[1] P. Bouthemy. A maximum likelihood framework for determining moving edges. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(5):499–511, May 1989.

[2] A.I. Comport, E. Marchand, and F. Chaumette. Robust model-based tracking for robot vision. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'04*, volume 1, pages 692–697, Sendai, Japan, September 2004.

[3] A.I. Comport, E. Marchand, M. Pressigout, and F. Chaumette. Real-time markerless tracking for augmented reality: the virtual visual servoing framework. *IEEE Trans. on Visualization and Computer Graphics*, 12(4):615–628, July 2006.

[4] D. Dementhon and L. Davis. Model-based object pose in 25 lines of codes. *Int. J. of Computer Vision*, 15(1-2):123–141, 1995.

[5] B. Espiau, F. Chaumette, and P. Rives. A new approach to visual servoing in robotics. *IEEE Trans. on Robotics and Automation*, 8(3):313–326, June 1992.

[6] N. Fischler and R.C. Bolles. Random sample consensus: A paradigm for model fitting with application to image analysis and automated cartography. *Communication of the ACM*, 24(6):381–395, June 1981.

[7] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2001.

[8] P.-J. Huber. *Robust Statistics*. Wiler, New York, 1981.

[9] S. Hutchinson, G. Hager, and P. Corke. A tutorial on visual servo control. *IEEE Trans. on Robotics and Automation*, 12(5):651–670, October 1996.

[10] Y. Ma, S. Soatto, J. Košecká, and S. Sastry. *An invitation to 3-D vision*. Springer, 2004.

[11] E. Marchand, P. Bouthemy, F. Chaumette, and V. Moreau. Robust real-time visual tracking using a 2D-3D model-based approach. In *IEEE Int. Conf. on Computer Vision, ICCV'99*, volume 1, pages 262–268, Kerkira, Greece, September 1999.

[12] P. Puget and T. Skordas. An optimal solution for mobile camera calibration. In *European Conf. on Computer Vision, ECCV'90*, volume 427 of *Lecture Notes in Computer Science*, pages 187–198, Antibes, France, April 1990.

[13] R.Y. Tsai and R. Lenz. A new technique for autonomous and efficient 3D robotics hand-eye calibration. Technical Report RC12212, IBM T.J. Watson Research Center, Yorktown Heights, New York, October 1987.