ViSP: A generic software platform for visual servoing

Éric Marchand, Fabien Spindler, François Chaumette [*]

Abstract

In this paper, we describe a modular software package, named VISP, that allows fast development of visual servoing applications. Visual servoing consists in specifying a task as the regulation of a set of visual features. Various issues have thus to be considered in the design of such applications: among these issues we consider the control of robots motion, visual features modeling, and the tracking of the visual measurements. Our environment features a wide class of control skills, a library for real-time tracking and a simulation toolkit.

1 Motivations

Several software packages or toolboxes written in various languages have been proposed in order to simulate robotic manipulator control: for example the Matlab robotics Toolbox [8] allows simple manipulation of serial-link manipulator and Roboop [14] is a manipulator simulation package (written in C++). On the other hand, only few systems allow a simple specification and the execution of a robotic task on a real system. Some systems, though not always related to vision-based robotics, have been however described in [9].

Visual servoing is a very important research area in robotics. Despite all the research in this field, it seems that there was no software environment that allows fast prototyping of visual servoing tasks. The main reason is that it usually requires specific hardware (the robot and specific framegrabbers). The consequence is that the resulting applications are not portable and can be merely adapted to other environments. Today's software design allows us to propose elementary components that can be combined to build portable high-level applications. Furthermore, the increasing speed of micro-processors allows the development of real-time image processing algorithms on a simple workstation. A visual servoing toolbox for MATLAB / Simulink [3] has been recently proposed but it features only simulation capabilities.

Chaumette, Rives and Espiau [6] proposed to constitute a "*library of canonical vision-based tasks*" for visual servoing that contents the most classical linkages that are used in practice. Toyama and Hager describe in [32] what such a system should be in the case of stereo visual servoing. The presented system (called SERVOMATIC) is specified using the same philosophy as the XVision system [16] and would have been independent from the robot and the tracking algorithms. Following these precedents, VISP (VISP states for "Visual Servoing Platform"), the software environment

[*]Authors are with IRISA - INRIA Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France. E-Mail: `First-name.Name@irisa.fr`. Corresponding author is Éric Marchand (`Eric.Marchand@irisa.fr`).

we present in this paper, features all these capabilities: independence with respect to the hardware, simplicity, extendibility and portability. Moreover, ViSP features a large library of elementary tasks wrt. various visual features that can be combined together, an image processing library that allows the tracking of visual cues at video rate, a simulator, an interface with various classical framegrabbers, etc. The platform is implemented in C++ under Linux.

2 ViSP: Overview and major features

2.1 Control issue

Visual servoing techniques consist of using the data provided by one or several cameras in order to control the motion of a robotic system [13, 19]. A large variety of positioning or target tracking tasks can be implemented by controlling from one to all degrees of freedom of the system. Whatever the sensor configuration, which can vary from one camera mounted on the robot end-effector to several free-standing cameras, a set of visual features $\mathbf{s}$ has to be designed from the visual measurements $\mathbf{x}(t)$ ($\mathbf{s} = \mathbf{s}(\mathbf{x}(t))$), allowing control of the desired degrees of freedom. A control law has to be designed also so that these features $\mathbf{s}$ reach a desired value $\mathbf{s}^*$, defining a correct realization of the task. A desired trajectory $\mathbf{s}^*(t)$ can also be tracked [1, 29]. The control principle is thus to regulate the error vector $\mathbf{s} - \mathbf{s}^*$ to zero.

Control law. To ensure the convergence of $\mathbf{s}$ to its desired value $\mathbf{s}^*$, we need to model or approximate the interaction matrix $\mathbf{L_s}$ that links the time variation of the selected visual features to the relative camera-object kinematics screw $\mathbf{v}$ and which is defined by the classical equation [13]:

$$\dot{\mathbf{s}} = \mathbf{L_s}\mathbf{v} \tag{1}$$

If we want to control the robot using the joint velocities, we have:

$$\dot{\mathbf{s}} = \mathbf{J_s}\dot{\mathbf{q}} + \frac{\partial \mathbf{s}}{\partial t} \tag{2}$$

where $\mathbf{J_s}$ is the features Jacobian and where $\frac{\partial \mathbf{s}}{\partial t}$ represents the variation of $\mathbf{s}$ due to potential motion of the object (for an eye-in-hand system) or due to potential motion of the camera (for an eye-to-hand system). More precisely, if we consider an eye-in-hand system we have:

$$\mathbf{J_s} = \mathbf{L_s}\,{}^c\mathbf{V}_n\,{}^n\mathbf{J}_n(\mathbf{q}) \tag{3}$$

where

- ${}^n\mathbf{J}_n(\mathbf{q})$ is the robot Jacobian expressed in the end-effector frame R_n ;
- ${}^c\mathbf{V}_n$ allows to transform the velocity screw between here the camera frame R_c and the end-effector frame R_n. It is given by :

$$ {}^c\mathbf{V}_n = \begin{bmatrix} {}^c\mathbf{R}_n & [{}^c\mathbf{t}_n]_\times {}^c\mathbf{R}_n \\ \mathbf{0}_3 & {}^c\mathbf{R}_n \end{bmatrix} \tag{4}$$

where ${}^{c}\mathbf{R}_n$ and ${}^{c}\mathbf{t}_n$ are the rotation and translation between frames R_c and R_n and $[\mathbf{t}]_\times$ is the skew matrix related to $\mathbf{t}$. This matrix is constant if the camera is rigidly linked to the end-effector.

Now, if we now consider an eye-to-hand system we have:

$$\mathbf{J_s} = -\mathbf{L_s}{}^{c}\mathbf{V}_{\mathcal{F}}{}^{\mathcal{F}}\mathbf{J}_n(\mathbf{q}) \tag{5}$$

where

- ${}^{\mathcal{F}}\mathbf{J}_n(\mathbf{q})$ is the robot Jacobian expressed in the robot reference frame $R_{\mathcal{F}}$;
- ${}^{c}\mathbf{V}_{\mathcal{F}}$ allows to transform the velocity screw between coordinate frames (here the camera frame R_c and the robot reference frame $R_{\mathcal{F}}$. This matrix is constant if the camera is motionless.

In all cases, a control law that minimizes the error $\mathbf{s} - \mathbf{s}^*$ is then given by:

$$\dot{\mathbf{q}} = -\lambda \widehat{\mathbf{J}_\mathbf{s}^+}(\mathbf{s} - \mathbf{s}^*) - \widehat{\frac{\partial \mathbf{s}}{\partial t}} \tag{6}$$

where λ is the proportional coefficient involved in the exponential convergence of the error and $\widehat{\frac{\partial \mathbf{s}}{\partial t}}$ is an estimation of the object/camera motion. VISP allows to consider both eye-in-hand and eye-to-hand configurations. Finally let us note that a secondary task $\mathbf{e_2}$ can be simply added (as reported in Section 3.2.2) when all the degrees of freedom are not constrained by the visual task. In that case, we have [13]:

$$\dot{\mathbf{q}} = -\lambda \left(\mathbf{W}^+\mathbf{W}\widehat{\mathbf{J}_\mathbf{s}^+}(\mathbf{s} - \mathbf{s}^*) + (\mathbf{I} - \mathbf{W}^+\mathbf{W})\mathbf{e_2}\right) + (\mathbf{I} - \mathbf{W}^+\mathbf{W})\frac{\partial \mathbf{e_2}}{\partial t} \tag{7}$$

where $\mathbf{W}^+$ and $\mathbf{I} - \mathbf{W}^+\mathbf{W}$ are projection operators that guarantee that the camera motion due to the secondary task is compatible with the regulation of $\mathbf{s}$ to $\mathbf{s}^*$.

If we consider here, without loss of generality, the case of an eye-in-hand system observing a motionless target we have:

$$\dot{\mathbf{s}} = \mathbf{L_s}\mathbf{v} \tag{8}$$

where $\mathbf{v} = {}^{c}\mathbf{V}_n{}^{n}\mathbf{J}_n(\mathbf{q})\ \dot{\mathbf{q}}$ is the camera velocity. If the low-level robot controller allows $\mathbf{v}$ to be sent as inputs, a simple control law can be obtained:

$$\mathbf{v} = -\lambda \widehat{\mathbf{L}_\mathbf{s}^+}(\mathbf{s} - \mathbf{s}^*) \tag{9}$$

where $\widehat{\mathbf{L_s}}$ is a model or an approximation of the interaction matrix. It can be chosen as [4]:

- $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{r})$ where the interaction matrix is computed at the current position of the visual feature and the current 3D pose (denoted $\mathbf{r}$) if it is available,
- $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{r}^*)$ where the interaction matrix is computed only once at the desired position of $\mathbf{s}$ and $\mathbf{r}$,

- $\widehat{\mathbf{L_s}} = \frac{1}{2}\big(\widehat{\mathbf{L_s}}(\mathbf{s},\mathbf{r}) + \widehat{\mathbf{L_s}}(\mathbf{s}^*,\mathbf{r}^*)\big)$ as reported in [23].

These possibilities have been integrated in VISP but other possibilities (such as a learning process of the interaction matrix [18, 20, 21]) are always possible though they are not currently integrated in the software. Let us point out that, in this case equation 7 can be rewritten by [13]:

$$\mathbf{v} = -\lambda \left(\mathbf{W}^+\mathbf{W}\widehat{\mathbf{L}^+_\mathbf{s}}(\mathbf{s}-\mathbf{s}^*) + (\mathbf{I}-\mathbf{W}^+\mathbf{W})\mathbf{e_2}\right) + (\mathbf{I}-\mathbf{W}^+\mathbf{W})\frac{\partial \mathbf{e_2}}{\partial t}. \tag{10}$$

A library of visual servoing skills. With a vision sensor providing 2D measurements $\mathbf{x}(t)$, potential visual features $\mathbf{s}$ are numerous, since as well 2D data such as coordinates of feature points in the image can be considered, as 3D data provided by a localization algorithm exploiting the extracted 2D measurements. It is also possible to combine 2D and 3D visual features to exploit the advantages of each approach while avoiding their respective drawbacks [24].

A systematic method has been proposed to derive analytically the interaction matrix of a set of visual features defined upon geometric primitives [13, 5, 6]. Any kind of visual features can be considered within the same formalism (coordinates of points, straight line orientation, area or more generally image moments, distance, etc.) Knowing these interaction matrices, the construction of elementary visual servoing tasks is straightforward. As explained in [6] a large library of elementary skills can be proposed. The current version of VISP has the following predefined visual features:

- 2D visual features: 2D points, 2D straight lines, 2D ellipses ;
- 3D visual features: 3D points, 3D straight lines, $\theta\mathbf{u}\mathbf{U}$ where θ and $\mathbf{u}$ are the angle and the axis of the rotation that the camera has to realize. These visual features are useful for 3D [34] or 2 1/2 D visual servoing [24].

Using these elementary visual features, more complex tasks can be considered by stacking the elementary matrices.

For example, if we want to build a 2 1/2 D visual servoing task defined by $\mathbf{s} = (x, y, \log(Z/Z^*), \theta\mathbf{u})$ where (x, y) are the coordinates of a 2D point, Z/Z^* is the ratio between the current and the desired depth of the point, and desired position and where θ and $\mathbf{u}$ are the angle and the axis of the rotation that the camera has to realize, the resulting interaction matrix is given by:

$$\mathbf{L_s} = \begin{bmatrix} \mathbf{L_p} \\ \mathbf{L_z} \\ \mathbf{L_{\theta u}} \end{bmatrix} \tag{11}$$

where $\mathbf{L_p}$, $\mathbf{L_z}$ and $\mathbf{L_{\theta u}}$ have the following forms:

$$\mathbf{L_p} = \begin{bmatrix} -1/Z & 0 & x/Z & xy & -(1+x^2) & y \\ 0 & -1/Z & y/Z & 1+y^2 & -xy & -x \end{bmatrix}, \tag{12}$$

$$\mathbf{L_z} = \begin{bmatrix} 0 & 0 & -1/Z & -y & x & 0 \end{bmatrix}, \tag{13}$$

$$\mathbf{L}_{\theta\mathbf{u}} = \left[\begin{array}{cc} \mathbf{0}_3 & \mathbf{L}_\omega \end{array} \right] \text{ with } \mathbf{L}_\omega = \mathbf{I}_3 - \frac{\theta}{2}\,[\mathbf{u}]_\times + (1 - \frac{\text{sinc}\theta}{\text{sinc}^2\frac{\theta}{2}})\,[\mathbf{u}]_\times^2 \tag{14}$$

All these elementary visual features are available in VISP or can be simply built. The complete code that allows to build this task is given in Section 3.2.3. This way, more feature-based tasks can be simply added to the library.

2.2 Vision-based tracking

Definition of objects-tracking algorithms in image sequences is an important issue for research and applications related to visual servoing and more generally for robot vision. A robust extraction and real-time spatio-temporal tracking of visual measurements $\mathbf{x}(t)$ is one of the keys for success of a visual servoing task. To consider visual servoing within large scale applications, it is now fundamental to consider natural scenes without any fiducial markers and with complex objects in various illumination conditions. From a historical perspective, the use of fiducial markers allowed the validation of theoretical aspects of visual servoing research. If such features are still useful to validate new control laws, it is no longer possible to limit ourselves to such techniques if the final objectives are the transfer of these technologies in the industrial world.

Most of the available tracking techniques can be divided into two main classes: feature-based and model-based tracking (see Figure 1). The former approach focuses on tracking 2D features such as geometric primitives (points, segments, ellipses, ...) or object contours, regions of interest, ... The latter explicitly uses a 3D model of the tracked objects. This second class of methods usually provides a more robust solution (for example, it can cope with partial occlusions of the object). If a 3D model is available, tracking is closely related to the pose estimation and is then suitable for any visual servoing approach. The main advantage of the 3D model-based methods is that the knowledge about the scene allows improvement of robustness and performance by predicting hidden movement of the object and reducing the effects of outlier. Another approach may also be considered when the scene is too complex (due to texture or lack of specific object). This approach is not based on feature extraction and tracking as in the two other cases but on the analysis of the motion in the image sequence. 2D motion computation provides interesting information related to both camera motion and scene structure that can be used within a visual servoing process.

Fiducial markers. Most of papers related to visual servoing consider very basic image processing algorithms. Indeed the basic features considered in the control law are usually 2D points coordinates. Therefore, the corresponding object is usually composed of "white dots on a black background". Such a choice allows using various real-time algorithms (e.g. [33]). The main advantage of this choice is that tracking is very robust and very precise. It is then suitable for all visual servoing control laws (2D but also 2 1/2 D and 3D since the position between camera and target can easily be obtained using pose computation algorithm). From a practical point of view, such algorithms are still useful to validate theoretical aspects of visual servoing research or for educational purposes. Furthermore, in some critical industrial processes such a simple approach ensures the required robustness (see Figure 2).

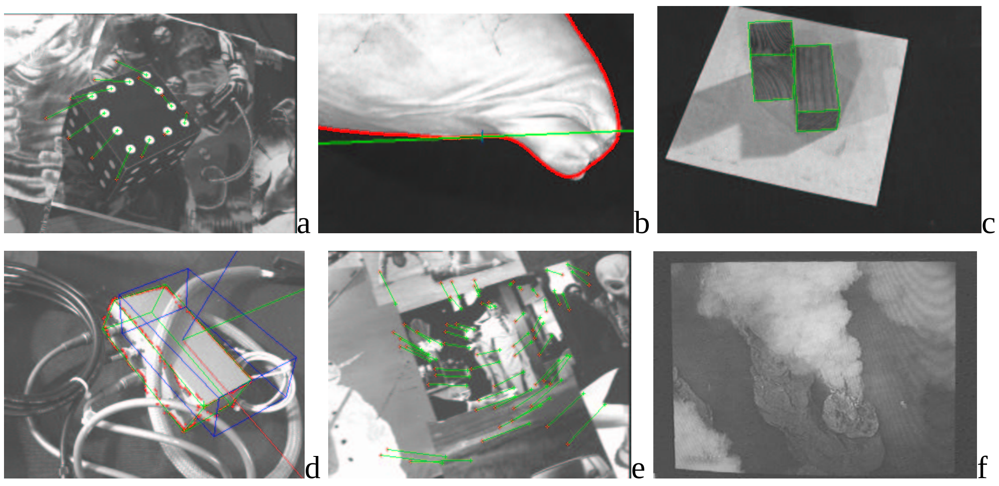

Figure 1: Increasingly difficult examples of feature tracking in visual servoing experiments.

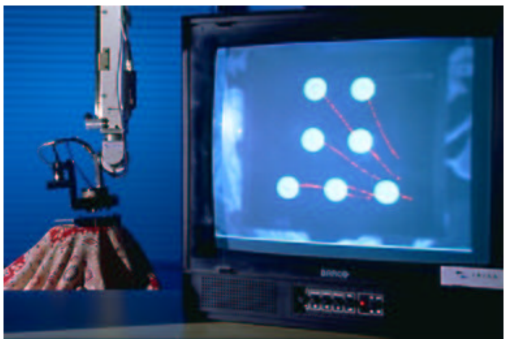

Figure 2: Visual servoing using fiducial markers for a grasping task: image acquired by the camera on the front and eye-in-hand camera on the back.

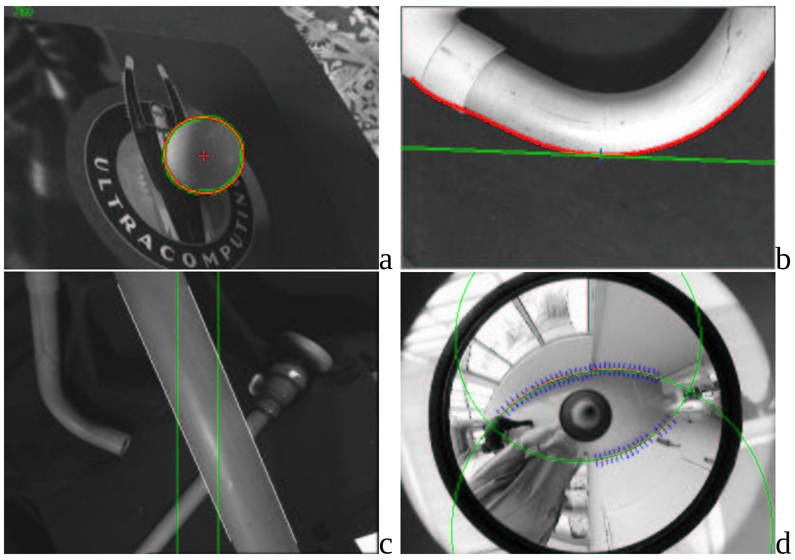

Figure 3: Tracking 2D features using the Moving Edges algorithm within visual servoing experiments: (a) 3D reconstruction of a sphere using active vision, (b) contour following, (c) positioning wrt. a cylinder with joint limits avoidance, (d) ellipses tracking (that correspond to the projection of 3D straight lines in catadioptric images).

2D contour tracking. In order to address the problem of 2D geometric feature tracking, it is necessary to consider at the low level a generic framework that allows local tracking of edge points. From the set of tracked edges, it is then possible to perform a robust estimation of the features parameters using an Iteratively Reweighted Least Squares based on robust M-estimation.

For the first point, few systems allow real-time capabilities on a simple workstation. The XVision system is a nice example of such systems [16]. In our case, we decided to use the Moving Edges (ME) algorithm which is adapted to the tracking of parametric curves [2]. It is a local approach that allows to match moving contours. When dealing with low-level image processing, the contours are sampled at a regular distance. At these sample points, a one dimensional search is performed to the normal of the contour for corresponding edges. An *oriented* gradient mask is used to detect the presence of a similar contour. One of the advantages of this method is that it only searches for edges which are oriented in the same direction as the parent contour. This is therefore implemented with convolution efficiency, and leads to real-time performance.

Figure 3 shows several results of features tracking (line, circle, contours,...) in visual servoing experiments that use the ViSP library. The proposed tracking approach based on the ME algorithm allows a real-time tracking of geometric features in an image sequence. It is robust with respect to partial occlusions and shadows. However, as a local algorithm, its robustness is limited in complex scenes with highly textured environment.

Pose estimation. In particular visual servoing types (most of 3D visual servoing, some 2 1/2 D visual servoing and the 2D visual servoing where the depth information must be recomputed at each iteration), the 3D pose of the camera with respect to the scene is required. This pose estimation process has been widely considered in the computer vision literature. Purely geometric, or numerical and iterative approaches may be considered [11]. Linear approaches use a least-squares method to estimate the pose. Full-scale non-linear optimization techniques (e.g., [22, 7, 12]) consists of minimizing the error between the observation and the forward-projection of the model. In this case, minimization is handled using numerical iterative algorithms such as Newton-Raphson or Levenberg-Marquardt. The main advantage of these approaches are their accuracy. In VISP, various algorithms are available: mainly the iterative approach proposed by Dementhon [11] which is suitable for applications that consider fiducial markers, and a full scale non-linear optimization based on the Virtual Visual Servoing approach [27].

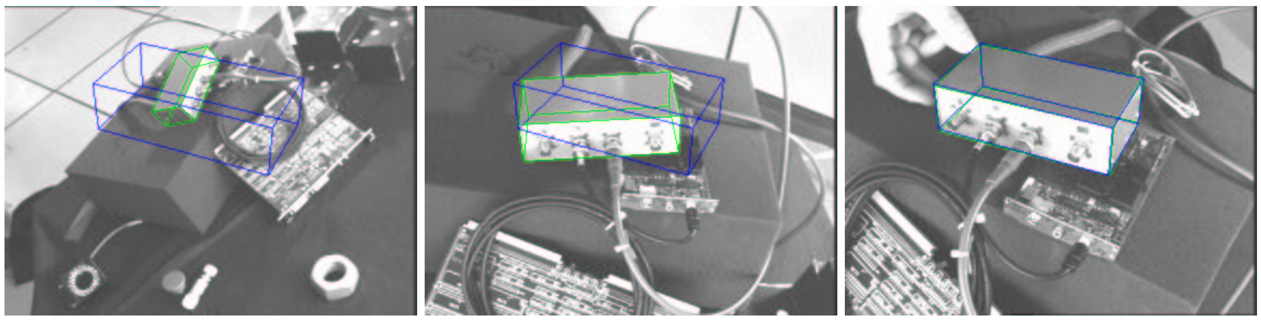

Figure 4: 2D 1/2 visual servoing experiment: in the images the tracked object appears in green and its desired position in blue.

Other tracking capabilities. VISP also features other tracking capabilities that can be considered within visual servoing experiments [28]. It integrates:

- an algorithm to estimate an homography and camera displacement from matched coplanar or not coplanar points [25] ;
- a version of the Hager Belhumeur [15] tracking algorithm that allows the matching of image templates at video rate ;

Though not directly integrated into the software, VISP provides a direct interface with third-party tracking or image processing algorithms. For example, we propose interfaces

- with a point of interests tracker library. We have considered a tracker, built on a differential formulation of a similarity criterion: the well-known Shi-Tomasi-Kanade algorithm [31].
- with a motion estimation algorithm (Motion 2D [30] available in open Source) which has been used for motion-based visual servoing [10]. Such image processing algorithms can be used to handle very complex images, as shown on Figure 1f and for applications dealing with complex object tracking, stabilization of a camera mounted on a submarine robot, etc.
- with a 3D model-based tracking algorithm based on the virtual visual servoing approach [7]. In that case, the image processing is potentially very complex. Indeed, extracting and tracking reliable contour points in real environment is a non trivial issue. In the experiment presented in Figure 4, images were acquired and processed at video rate (50Hz). Tracking is

always performed at below frame rate (usually in less than 10ms). All the images given in Figure 4 depict the current position of the tracked object in green while its desired position appears in blue. The considered object is a video multiplexer. It was placed in a highly cluttered environment. Tracking and positioning tasks were correctly achieved. Multiple temporary and partial occlusions were made by a hand and various work-tools.

Simulation capabilities In order to allow fast prototyping of new control laws, ViSP also provides simulation capabilities. 3D geometric primitives can be forward-projected on the image plane of a virtual camera and a virtual robot can then be controlled. In order to obtain realistic simulation, it is possible to consider virtual robots with specific Jacobian, joint limits, etc. Furthermore noise can be added in measures computation and robot motion and in camera intrinsic parameters. An advantage of this approach wrt. Matlab simulation (such as the Visual Servoing Toolbox for MATLAB / Simulink [3] or the Matlab Robotics toolbox [8]) is that the written code can then be used with only minor modifications on a real robot with real images. Figure 5 shows an example of the simulation output. The display interface is written using the Open Inventor library (a C++ layer of Open GL).

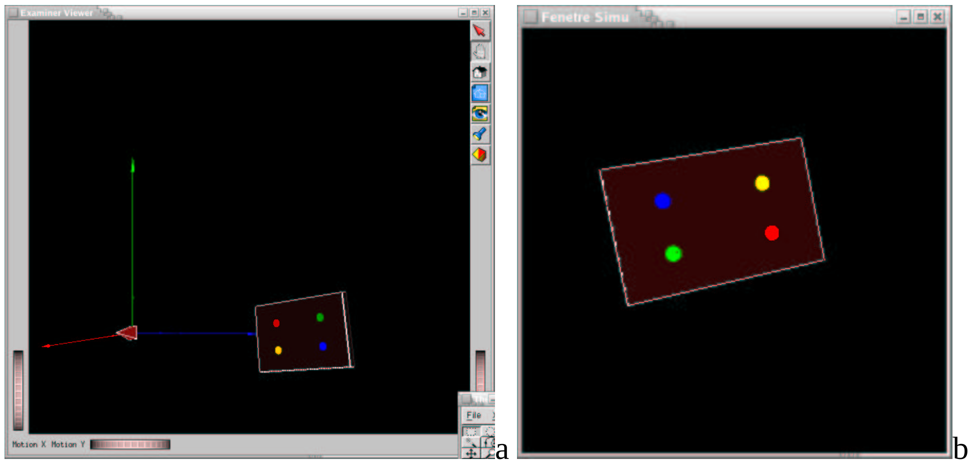

Figure 5: ViSP simulation module built using the Open Inventor library: (a) external view (b) view from the camera.

3 Implementation issues

As already stated, while developing this software, our goal was to allow a portable (independent from the hardware), fast and reliable prototyping of visual servoing applications. We also wanted to provide a package that is suitable for real-time implementation and that allows to perform both simulations and real experiments from the same (or at least very similar) code. Object-oriented programming languages feature these qualities and therefore we choose the C++ language for the implementation of ViSP.

The first part of this section presents the internal architecture of the system and how it has been implemented. Describing the full implementation of the software is out of reach in this paper, therefore, we will focus on the notion of extendibility and portability. The second part describes how to use the available libraries from a end-user point of view. Let us note that all the functionalities described in this section have been implemented and are *fully operational*.

3.1 An overview of the VISP architecture

To fulfill the extendibility and portability requirements, we divided the platform into three different modules: a module for visual features , visual servoing control laws and robot controller, a module for image processing, tracking algorithms and other computer vision algorithms, and a module for visualization dependent library. Figure 6 summarizes the basic software architecture and module dependencies.

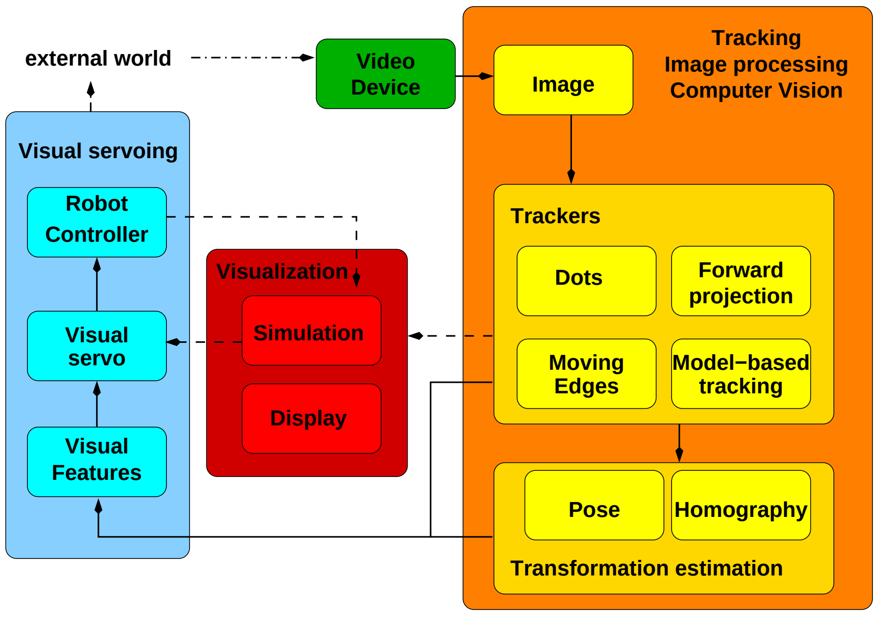

Figure 6: VISP software architecture.

Visual features and control law. Each specific visual feature is derived from a virtual class `vpBaseFeatures`. This class mainly defines few variables (e.g., a vector that describes the parameters s) and a set of virtual members that are feature-dependent (e.g., the way to compute the interaction matrix $\mathbf{L_s}$ or the virtual function that allows the computation of the visual features s from the measurements in the image $\mathbf{x}(t)$). It is important to note that all the relations between the control laws library and the visual features library is done through this class. The virtual functions defined in `vpBaseFeatures` can be directly used by the controller `vpServo` even if they are not yet defined. The consequence is that the controller class never knows the nature of the manipulated features and manipulates only vectors and matrices. Another consequence is that it is not necessary to modify the controller library when adding a new feature. On the other hand, when adding a new feature in the visual features library, the programmer must define the number of its component, the way to compute the interaction matrix, etc. This is done at a lower level (e.g., vpFeaturePoint, vpFeatureLine, ...). A generic feature vpGenericFeature allows the user to simply define a new feature and to test easily its behavior.

The controller (provided in the vpServo class) itself provides therefore a generic interface with the visual features. It computes the control law that minimizes the error $\mathbf{s} - \mathbf{s}^*$ as described in Section 2 according to a list of visual features that defines the task. Various control laws have been implemented as proposed that consider the eye-in-hand or eye-to-hand systems and various formulations of the model of the interaction matrix $\widehat{\mathbf{L}_\mathbf{s}}$. Along with the interaction matrix $\mathbf{L}_\mathbf{s}$, Jacobian $\widehat{\mathbf{J}_\mathbf{s}}$ (that depends of a particular robot, see next paragraph) and its pseudo inverse $\widehat{\mathbf{J}_\mathbf{s}^+}$ and its null space, the corresponding projection operator (W^+ and $\mathbf{I} - \mathbf{W}^+\mathbf{W}$) are also computed if a secondary task has to be added to the main visual task (see equation 7).

Hardware portability. One of the challenges dealing with a visual servoing package is that it has to deal with multiple robotic platforms as well as with various framegrabbers. Obviously the package does not (and cannot) provide an interface with all possible robots and grabbers but we built it in order to facilitate adding new hardware.

A new robot class can be derived from the vpRobot virtual class. Although vpRobot defines the prototypes of each member, it does not provide any interface with a real robot. The new class that has to be implemented for each new robot redefines some pure virtual methods defined in vpRobot such as robot motion orders or Jacobian computation generally specific to a given robot) and inherits all the methods and attributes of vpRobot (i.e., generic control issues). Simulated robots are considered and can be controlled exactly as real robots (specific Jacobian, inverse and forward kinematics, joint limits and singularities can be modeled and simulated).

Similarly, dealing with framegrabbers, a generic vpVideo class has been developed from which a particular framegrabber class can be derived. Some vpVideo pure virtual methods have to be defined within this new class (mainly initialization, acquisition, closing methods). Such an interface with VISP is very simple to add since such methods should already exist on the user's system. In the current version of VISP, some classical framegrabbers are already supported (IEEE 1394, Video4Linux2, Matrox Meteor, IT ICcomp,...). Along with these acquisition capabilities, VISP provides various classes to display images using either the X11 system or higher level libraries such as Open GL or QT.

Image processing and tracking. A template image class vpImage is provided. It has allowed the development of the various trackers described in the previous section. Along with elementary image processing functions, it provides an interface with the images acquisition and display classes. Dealing with the tracking algorithms, a virtual class vpTracker is defined and is then derived according to each particular tracking algorithms. An interface with the visual features class is provided.

Matrices. Furthermore, C++ provides capabilities to handle matrices operation using a "Matlab like" syntax. Various numerical analysis algorithms (e.g., SVD and LU decompositions provided by the GSL library) are widely used throughout.

3.2 VISP from a end-user point of view

Our other claim was that VISP is simple to use. We will therefore describe the software environment, from the end-user point of view, in the light of three simple examples implemented using VISP.

3.2.1 Build a basic 2D visual servoing task

Listing 1 defines a typical initialization process that can be used in most programs using VISP. These lines define the framegrabber (here an IEEE1394 camera), the display system (here X11R6), a robot (the 6 d.o.f Afma gantry robot of IRISA), a camera (with given calibration parameters) and finally creates a task.

Listing 1: Typical code for VISP initialization

```
1   vpImage<unsigned char> I ;
2
3   vpIeee1394 grabber(VIDEO_PORT) ;  // use the 1394 framegraber
4   grabber.open(I) ;        // associate the grabber to the image
5   grabber.acquire(I) ;
6
7   vpDisplayX display(I,‘‘a new X11 window’’) ; // use the X11R6 window system to display the image
8   vpDisplay::display(I) ;                      // associate the display to the image
9
10  vpRobotAfma6 rob ;           // use the Afma6 robot
11  vpCameraParameters cam(u0,v0,px,py) ; // set the value of camera calibration parameters
12
13  vpServo task ;               // create a task
14
15  < code for a specific experiment >
```

Once these initializations have been achieved, the user is ready to define the tracker of the visual cues and the visual servoing task. In this first example we choose the classical positioning task wrt. four points using their x and y coordinates in the image. Dealing with the tracking process, in this example we choose to track fiducial markers (`vpDot`). The features (`vpFeaturePoint`) are created from the tracked markers (`vpDot`) using member functions of the `vpFeatureBuilder` class (in this simple case, it mainly achieves a pixel-to-meter conversion). The desired values of the visual feature $\mathbf{s}^*$ is also defined and a link between the current value (`s[i]`) of the visual feature in the image and the desired value (`sd[i]`) is then created. `sd` is initialized using the `buildFrom` method that allows to set x and y 2D coordinates along with the desired depth Z that is used in the interaction matrix computation. Each call to the `addFeature` method creates a 2×6 interaction matrix which is “stacked” to the current one. At the end of this process a 8×6 matrix and the corresponding error vector is then created. Line 38 specifies that we consider an eye-in-hand configuration with velocity computed in the camera frame. Furthermore, the interaction matrix will be computed at the desired position $\widehat{\mathbf{L}_\mathbf{s}} = \widehat{\mathbf{L}_\mathbf{s}}(\mathbf{s}^*, \mathbf{r}^*)$ (Line 39) and the control law will be computed using the pseudo-inverse of the interaction matrix (other possibilities such as considering the transpose of $\mathbf{L}_\mathbf{s}$ also exist even if we do not recommend at all to use them).

Listing 2: An example of task definition: positioning wrt. four points

```
16  int nbpoint =4 ;
17  vpDot dot[nbpoint] ;                 // tracked objects
18  vpFeaturePoint s[nbpoint], sd[nbpoint] ; // current and desired feature (nbpoint points)
19  double xd[nbpoint], yd[nbpoint], Zd[nbpoint] ;
20
```

```
21    < initialize desired visual feature xd[], yd[], and Zd[] >
22
23    for (i=0; i<nbpoint; i++)
24    {
25      dot[i].initTracking(I) ; //  initialize the tracking process
26
27      // build current visual features from the tracked objects
28      vpFeatureBuilder::create(s[i],cam, dot[i])  ;
29
30      // Init here the desired visual features s* along with
31      // the 3D information required to compute the interaction matrix
32      sd[i].buildFrom(xd[i],yd[i],Zd[i]) ;
33
34      task.addFeature(s[i],sd[i]) ;   // add point-to-point image constraint
35                                      // defines the list of visual features, the error vector
36                                      // as well as the interaction matrix
37    }
38    task.setServo(vpServo::EYEINHAND_CAMERA) ;
39    task.setInteractionMatrixType(vpServo::DESIRED, vpServo::PSEUDO_INVERSE) ;
```

It is then straightforward to write the control loop itself. It features the image acquisition (line 43) and the current visual features computation from the result of the dots tracking (line 46 and 47). The task is then automatically updated. Finally the control law given in equation (9) is computed and the result is sent to the robot controller.

Listing 3: Typical code for the visual servoing closed loop

```
40    task.setLambda(0.2) ;  // set the gain λ
41    while(...) {
42      vpColVector v(6) ;     // velocity vector
43      grabber.acquire(I) ;        // acquire a new image
44      for (i=0 ; i < nbpoint ; i++)
45      {
46         dot[i].track(I) ;     // perform the feature tracking and the pixel/meter conversion
47         vpFeatureBuilder::create(s[i],cam, dot[i]);
48      }
49
50      v =  task.computeControlLaw() ;
51
52      // send the computed velocity (expressed in the camera frame) to the robot controller
53      robot.setVelocity(vpRobot::CAMERA_FRAME, v) ;
54    }
```

3.2.2 Introduce more complex image processing and a secondary task

Let us consider now a curve-following task [26]. This problem can be divided in two sub-tasks. The primary task consists of servoing the tangent to the curve (for instance maintaining this tangent horizontal and centered in the image). The positioning skill used in this experiment is therefore a 2D line-to-2D line link and visual features used here are $\mathbf{s} = (\rho, \theta)$ where ρ and θ are the cylindric parameters of the straight line that is the tangent to the curve. A secondary task can be added and it has been specified as a trajectory tracking at a given constant velocity V_x in the X direction of the camera frame (see Figure 7).

Image processing consists here in tracking a spline in the image sequence and in computing the equation of the tangent to the curve from which we can control the camera motion. In our software environment, there is no direct relation between the tangent to a curve (that is, here, the measurement $\mathbf{x}$) and a straight line (that is the visual feature $\mathbf{s}$). As explained in the previous example, VISP usually allows the user to avoid an explicit access to the tracker, but the number

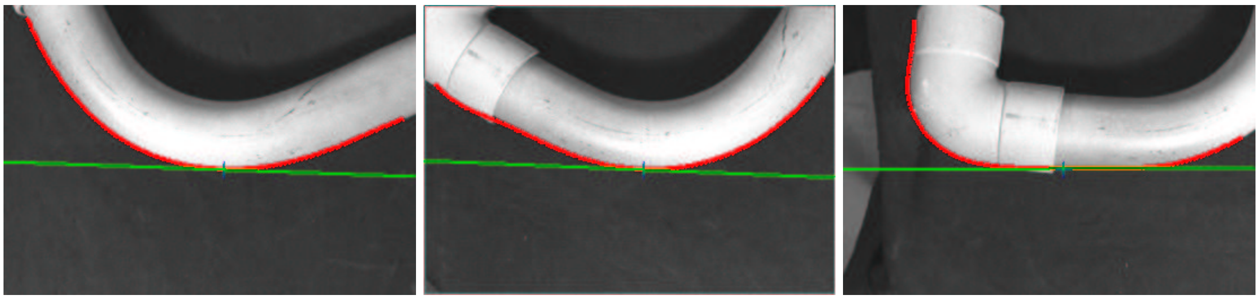

Figure 7: Pipe following task: 3 images acquired during the task with tracked curve (red) and tangent to the curve (green)

of relationships among visual cues and control features is virtually infinite. Therefore a direct access to the trackers is necessary. In this example, the spline tracker is defined in Line 1 whereas the visual features (represented by parameters ρ and θ of a straight line) are associated to this tracker in Line 5. Then in the control loop, the spline is tracked in each frame and the new visual features are computed (Line 14) and introduced in the task. The secondary task is then considered in Line 17 where $g = -V_x$ correspond to $\frac{\partial \mathbf{e_2}}{\partial t}$ in equation 7. For simplicity we have considered that $\mathbf{e_2} = X - X_0 - V_x t$ is always equal to 0. This visual servoing task also features the use of a secondary task. Vector g is combined with the primary task using the projection operator $\mathbf{I} - \mathbf{W}^+\mathbf{W}$.

This example allows us to show the importance of the three libraries: the trackers library, the visual features library, and the controller library and how they interact with each other.

Listing 4: Code for a curve following task: task definition and visual servoing closed loop

```
1  vpSpline S(CUBICSPLINE) ;  // define a tracker (here track a spline using the  ME algorithm)
2  S.initTracking(I) ;
3
4  vpFeatureLine L ;                // define the visual feature: a line
5  L = S.tangent(0,0) ;             // defined by the tangent to the spline
6  vpFeatureLine Ld(0,PI/2) ;       // and that must be seen horizontal and centered in the image
7
8  task.addFeature(L,Ld) ;      // define the visual task
9
10 while(...) {
11   vpColVector v(6), g(6)
12   grabber.acquire(I) ;
13   S.track(I) ;            // Track the spline (visual cue)
14   L = S.tangent(0,0) ;    // compute the tangent
15
16   g[0] = -Vx ;             // define secondary task
17   v =  task.computeControlLaw()+ task.addSecondaryTask(g) ;  // v = -λW⁺WL⁺s(s - s*) + (I - W⁺W)gs
18
19   robot.setVelocity(vpRobot::CAMERA_FRAME, v) ;
20 }
```

3.2.3 Building a 2 1/2 D visual servoing task

We now consider the 2 1/2 D visual servoing task presented in Section 2.1, the visual features s are defined by $\mathbf{s} = (x, y, \log Z/Z*, \theta\mathbf{u})$. To achieve this task one solution is to consider an estimation of the 3D position of the point and the camera pose wrt. to the object. Listing 5 shows how to get the pose ${}^{c}\mathrm{M}_{o}$ from a set of seven points (see Figure 8). The class `vpPose` provides functions that compute the pose from a list of points (list built using the `addPoint` method). Different methods can be considered to compute the pose. In this example, it is first initialized using the Dementhon-Davis [11] approach and improved using a non-linear minimization method.

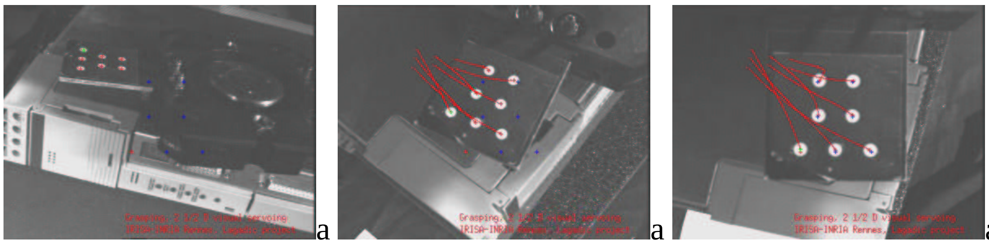
a a a

Figure 8: 2 1/2 D visual servoing task as implemented in section 3.2.3

Listing 5: An example of opse estimation

```
int nbPoint = 7 ;
vpDot dot[nbPoint] ; // current visual feature associated with a tracker
                     // (here track 7 white dots)

// init the 3D coordinates (X,Y,Z) of the points in object frame
vpPoint point[nbPoint] ;
point[0].setWorldCoordinates(-L,-L,0) ;  point[1].setWorldCoordinates(L,-L,0) ;
point[2].setWorldCoordinates(L,L,0) ;    point[3].setWorldCoordinates(-L,L,0) ;
point[4].setWorldCoordinates(2*L,3*L,0); point[5].setWorldCoordinates(0,3*L,0) ;
point[6].setWorldCoordinates(-2*L,3*L,0) ;


vpPose pose ;
for (i=0; i<nbPoint; i++)
{
   dot[i].initTracking(I, cam) ;   // initialize the tracking process
   dot[i].track(I) ;               // get the 2D position of the point in meter
   vpPixelMeterConversion::convertPoint(cam,  dot[i], point[i])  ;
   // at this point the 2D coordinates (x,y) and 3D coordinates (X,Y,Z)
   // are available
   pose.addPoint(point[i]) ;         // consider this point in the pose computation algorithm
}

vpHomogeneousMatrix cMo ;
pose.computePose(vpPose::DEMENTHON, cMo) ;
pose.computePose(vpPose::NON_LINEAR, cMo) ;

cout << ''Pose'' << cMo << endl ;
```

Listing 6 shows how to build the 2 1/2 D visual servoing control law. In a first time, we initialized the current and desired value of the visual features. The basic features related to a point and to $\theta\mathbf{u}$ are available in the visual feature library (`vpFeaturePoint` and `vpFeatureThetaU`). However there is no predefined basic feature for $\log(Z/Z^*)$. In such case it is possible to use a generic `vpGenericFeature` feature. The user has then to compute, at each iteration, the state vector, the interaction matrix, and the error vector. The task is then built by “stacking” the different visual features using the `addFeature` method. Let us note that dealing with `logZ` and `tu` these desired values are zero thus, we do not have to specify them (this is implicitly done). Line 68 specifies that the interaction matrix will be computed at the current position $\widehat{\mathbf{L}_{\mathbf{s}}} = \widehat{\mathbf{L}_{\mathbf{s}}}(\mathbf{s}, \mathbf{r})$. In the closed loop itself, we find the point tracking that provides the measurement necessary to compute the pose and the position of the 2D point (Line 83). The pose is updated from these measurements using a non-linear minimization method and the displacement $\theta\mathbf{u}$ along with the depth Z of the point are updated (let us note that operator * has been overloaded to allows simple frame transformation: ${}^c\mathbf{P} = {}^c\mathrm{M}_o{}^o\mathbf{P}$). The interaction matrix related to $\log Z/Z^*$ has to be computed according to equation (13) (see Line 97). The global task interaction is then updated and the control law computed.

Listing 6: An example of a 2 1/2 D visual servoing task

```
55
56        // define the feature ---------------------------------------------
57        vpFeaturePoint p, pd) ;            // 2D reference point
58        vpGenericFeature logZ(1) ;         // log (Z/Z*)
59        vpFeatureThetaU tu ;               // ThetaU
60
61        // build the task (stack the features) -----------------------------
62        task.addFeature(p,pd) ;
63        task.addFeature(logZ) ;
64        task.addFeature(tu) ;   // $\mathbf{s} = (x, y, \log Z/Z^*, \theta\mathbf{u})^\top$
65
66        //-----------------------------------------------------------------
67        // interaction matrix computed at the current position
68        task.setInteractionMatrixType(vpServo::CURRENT) ;
69
70        // compute Z* and p*
71        vpColVector cP = cdMo*point[0] ;
72        double Zd ;   Zd = cP[2] ;
73        pd.set_x(cP[0]/Zd) ;   pd.set_y(cP[1]/Zd) ;
74
75        while(1) { //-----------------------------------------------------
76          g.acquire(I) ;
77          vpDisplay::display(I) ;
78
79          // compute the pose using  a non linear minimisation method
80          pose.clearPoint() ;
81          for (i=0 ; i < nbPoint ; i++) {
82            dot[i].track(I) ;
83            vpPixelMeterConversion::convertPoint(cam, dot[i], point[i])  ;
84            pose.addPoint(point[i]) ;
85          }
86          pose.computePose(vpPose::NON_LINEAR, cMo) ;
87
88          // compute the current Z
89          cP = cMo * point[0] ;   Z = cP[2] ;
90          p.buildFrom(point[0].get_x(), point[0].get_y(), Z) ;
91
92          // compute log (Z/Z*) and the corresponding interaction matrix
93          logZ.set_s(log(Z/Zd)) ;
94          vpMatrix LlogZ(1,6) ;
95          LlogZ[0][0] = LlogZ[0][1] = LlogZ[0][5] = 0 ;
96          LlogZ[0][2] = -1/Z ;  LlogZ[0][3] = -p.get_y() ;  LlogZ[0][4] =  p.get_x() ;
97          logZ.setInteractionMatrix(LlogZ) ;
98
99          cdMc = cdMo*cMo.inverse() ; // Compute the displacement to achieve
100         tu.buildFrom(cdMc) ;
101
102         v = task.computeControlLaw() ;
103         robot.setVelocity(vpRobot::CAMERA_FRAME, v) ;
104       }
```

In this example, to compute the depth of the reference point and the rotation that the camera has to achieve we have considered a pose estimation process. Let us note that this can also be achieved by the estimation of the homography between the current and the desired image (as explained in [24]). ViSP features also the capability to estimate such homography using various algorithms [17, 24] and to extract from this homography the camera displacement and some useful values such as Z/Z^*.

4 Conclusion

ViSP is a *fully functional* modular architecture that allows fast development of visual servoing applications. The platform takes the form of a library which can be divided in three main modules: two are dedicated to control issues (one for control processes and one for canonical vision-based tasks that contains the most classical linkages) and one dedicated to real-time tracking. Let us finally note that ViSP also features a virtual six dof robot that allows to simulate visual servoing experiments.

ViSP is developed within the INRIA Lagadic project and its kernel, available under the Linux system, is distributed using an Open Source license (QPL License). Other modules such as the 3D model-based tracker are subject to other licences. The ViSP website is located at `http://www.irisa.fr/lagadic/visp`. The examples given in the paper and many others can be found in the software distribution.

Acknowledgment

The authors want to acknowledge the contribution of the members of the Lagadic group: Nicolas Mansard, Anthony Remazeilles, Andrew Comport and Muriel Pressigout as well as the former PHD students (Ezio Malis, Youcef Mezouar,...) who contribute to the creation of this software.

References

[1] F. Berry, P. Martinet, and J. Gallice. Turning around an unknown object using visual servoing. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'2000*, volume 1, pages 257–262, Takamatsu, Japan, November 2002.

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

[3] E. Cervera. Visual servoing toolbox for matlab / simulink. `http://vstoolbox.sourceforge.net/`, 2003.

[4] F. Chaumette. Potential problems of stability and convergence in image-based and position-based visual servoing. In D.J. Kriegman, G. Hager, and A.S. Morse, editors, *The confluence of vision and control*, Lecture Notes in control and information sciences, No 237, pages 67–78. Springer, June 1997.

[5] F. Chaumette. Image moments: a general and useful set of features for visual servoing. *IEEE Trans. on Robotics*, 20(4):713–723, August 2004.

[6] F. Chaumette, P. Rives, and B. Espiau. Classification and realization of the different vision-based tasks. In K. Hashimoto, editor, *Visual Servoing*, volume 7, pages 199–228. World Scientific Series in Robotics and Automated Systems, Singapour, 1993.

[7] 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.

[8] P. Corke. A robotics toolbox for matlab. *IEEE Robotics & Automation Magazine*, 3(1):24–32, September 1996.

[9] E. Coste-Manière and B. Espiau (Eds). Special issue on integrated architecture for robot control and programming. *Int. Journal of Robotics Research*, 17(4), April 1998.

[10] A. Crétual and F. Chaumette. Visual servoing based on image motion. *Int. Journal of Robotics Research*, 20(11):857–877, November 2001.

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

[12] T. Drummond and R. Cipolla. Real-time visual tracking of complex structures. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24(7):932–946, July 2002.

[13] 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.

[14] R. Gourdeau. Object-oriented programming for robotic manipulator simulation. *IEEE Robotics & Automation Magazine*, 4(3):21–29, September 1997.

[15] G. Hager and P. Belhumeur. Efficient region tracking with parametric models of geometry and illumination. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20(10):1025–1039, October 1998.

[16] G. Hager and K. Toyama. The XVision system: A general-purpose substrate for portable real-time vision applications. *Computer Vision and Image Understanding*, 69(1):23–37, January 1998.

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

[18] K. Hosoda and M. Asada. Versatile visual servoing without knowledge of true jacobian. In *IEEE/RSJ Int. Conf on Intelligent Robots and Systems, IROS'94*, pages 186–193, Munich, Germany, August 1994.

[19] 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.

[20] M. Jägersand, O. Fuentes, and R. Nelson. Experimental evaluation of uncalibrated visual servoing. In *IEEE Int. Conf. on Robotics and Automation, ICRA'97*, volume 3, pages 2874–2880, Albuquerque, NM, April 1997.

[21] J.-T. Lapresté, F. Jurie, M. Dhome, and F. Chaumette. An efficient method to compute the inverse jacobian matrix in visual servoing. In *IEEE Int. Conf. on Robotics and Automation, ICRA'04*, New Orleans, April 2004.

[22] D.G. Lowe. Fitting parameterized three-dimensional models to images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13(5):441–450, May 1991.

[23] E. Malis. Improving vision-based control using efficient second-order minimization techniques. In *IEEE Int. Conf. on Robotics and Automation, ICRA'04*, volume 2, pages 1843–1848, New Orleans, April 2004.

[24] E. Malis, F. Chaumette, and S. Boudet. 2 1/2 D visual servoing. *IEEE Trans. on Robotics and Automation*, 15(2):238–250, April 1999.

[25] E. Malis, F. Chaumette, and S. Boudet. 2 1/2 D visual servoing with respect to unknown objects through a new estimation scheme of camera displacement. *Int. Journal of Computer Vision*, 37(1):79–97, June 2000.

[26] E. Marchand. Visp: A software environment for eye-in-hand visual servoing. In *IEEE Int. Conf. on Robotics and Automation, ICRA'99*, volume 4, pages 3224–3229, Detroit, Michigan, Mai 1999.

[27] E. Marchand and F. Chaumette. Virtual visual servoing: a framework for real-time augmented reality. In *EUROGRAPHICS'02 Conf. Proceeding*, volume 21(3) of *Computer Graphics Forum*, pages 289–298, Saarebrücken, Germany, September 2002.

[28] E. Marchand and F. Chaumette. Feature tracking for visual servoing purposes. *Robotics and Autonomous Systems*, 52(1):53–70, jun 2005. special issue on "Advances in Robot Vision", D. Kragic, H. Christensen (Eds.).

[29] Y. Mezouar and F. Chaumette. Path planning for robust image-based control. *IEEE Trans. on Robotics and Automation*, 18(4):534–549, August 2002.

[30] J.-M. Odobez and P. Bouthemy. Robust multiresolution estimation of parametric motion models. *Journal of Visual Communication and Image Representation*, 6(4):348–365, December 1995.

[31] J. Shi and C. Tomasi. Good features to track. In *IEEE Int. Conf. on Computer Vision and Pattern Recognition, CVPR'94*, pages 593–600, Seattle, Washington, June 1994.

[32] K. Toyama, G. Hager, and J. Wang. Servomatic: A modular system for robust positioning using stereo visual servoing. In *Int. Conf. on Robotics and Automation*, pages 2636–2643, Minneapolis, April 1996.

[33] M. Vincze and C. Weiman. On optimizing tracking performance for visual servoing. In *IEEE Int. Conf. on Robotics and Automation, ICRA'97*, volume 3, pages 2856–2861, Albuquerque, NM, April 1997.

[34] W. Wilson, C. Hulls, and G. Bell. Relative end-effector control using cartesian position-based visual servoing. *IEEE Trans. on Robotics and Automation*, 12(5):684–696, October 1996.