ViSP 2.6.1: Visual Servoing Platform

Building a visual servoing task

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

October 27, 2011

François Chaumette
Eric Marchand
Filip Novotny
Antony Saunier
Fabien Spindler
Romain Tallonneau

Inria
INVENTORS FOR THE DIGITAL WORLD

Contents

1 Visual features **5**
1.1 Visual features in ViSP . . . 5
1.1.1 Visual features: overview . . . 5
1.1.2 Derived visual features . . . 6
1.1.3 Feature selection . . . 6
1.1.4 From Trackers to visual features . . . 6
1.2 2D visual features . . . 6
1.2.1 Point . . . 6
1.2.2 Line . . . 8
1.2.3 Ellipse . . . 10
1.2.4 Vanishing points . . . 13
1.2.5 Moments . . . 14
1.3 3D visual features . . . 23
1.3.1 Interaction matrix related to a 3-D point . . . 23
1.3.2 Interaction matrix related to a translation . . . 24
1.3.3 Interaction matrix related to $\theta\mathbf{u}$ . . . 27
1.4 Generic feature . . . 28

2 Task and control laws **29**
2.1 Create a task . . . 29
2.2 Control law . . . 30

3 Complete visual servoing experiments **35**
3.1 Building a basic 2D visual servoing task . . . 35
3.2 Building a 2 1/2 D visual servoing task . . . 37

Chapter 1

Visual features

1.1 Visual features in ViSP

1.1.1 Visual features: overview

Let us first examine the visual features from a generic point of view. We will see later how this feature is specialized when considering specific visual features such as points, lines, 3D rotations, moments, etc.

From a formal point of view a visual feature is nothing but:

- a vector $\mathbf{s}$ that defines the value of the visual feature
- an interaction matrix $\mathbf{L_s}$ that describes how $\mathbf{s}$ is modified when the camera is moving with a velocity $\mathbf{v}$:
$$\dot{\mathbf{s}} = \mathbf{L_s}\mathbf{v}$$
- a method to compute the error $\mathbf{s} - \mathbf{s}^*$ where $\mathbf{s}^*$ is the desired value of the visual feature.

The basic visual feature is contained in a class named `vpBasicFeature`*. It is a virtual class that only contains some protected members and pure virtual methods. It cannot be used by the end-user programmer but it provides an interface with the control law computation class* `vpServo` *(see chapter 2).*

A visual feature is mainly built on a vector `s` *that defines the value of the visual information (that is* $\mathbf{s}$*). It also integrates pure virtual methods that return the value of the interaction matrix and the error (defining the error is important since it is not always a simple subtraction). The prototype of the* `vpBasicFeature` *can schematically be defined as follow (to which must be added constructors and destructors, copy operators, etc.):*

```
class vpBasicFeature {
  private :
    vpColVector s ;  // state of the virtual feature
  public :
    double operator[](const int n) ;    // acces to the state vector
    // compute the interaction matrix
    virtual vpMatrix interaction() const = 0;

    // compute the error between two visual features
    virtual vpColVector error(const vpBasicFeature &s_star) = 0 ;
} ;
```

Operators [] are also overloaded to have a direct access to the value of `s`*.*

Let us note that the 3D information that is sometime (often) required to compute the interaction matrix is not really part of the visual feature itself. It will be defined in the derived classes built upon the `vpBasicFeature` *class.*

1.1.2 Derived visual features

As stated, the end-user cannot directly use the basic visual feature defined in the previous paragraph. Therefore some classes are derived from `vpBasicFeature`*. At this level, the goal is to allow the initialization of the state vector and provide the instantiation of the pure virtual methods* `interaction(...)` *and* `error(...)`*. Many other functionalities can be added but are not directly of interest when dealing with the visual servoing purpose.*

ViSP provides the following derived features. We have several 2D features:

- `vpFeaturePoint` *: Point* $\mathbf{s} = (x, y)$,
- `vpFeatureLine` *: Line* $\mathbf{s} = (\rho, \theta)$,
- `vpFeatureEllipse` *: Ellipse* $\mathbf{s} = (x, y, \mu_{20}, \mu_{11}, \mu_{02})$,
- `vpFeatureVanishingPoint` *: Point at infinity* $\mathbf{s} = (x, y)$.

and several 3D features:

- `vpFeaturePoint3D` *: 3D point* $\mathbf{s} = (X, Y, Z)$,
- `vpFeatureThetaU` *: the rotation* ${}^{c^*}\mathbf{R}_c$ *or* ${}^{c}\mathbf{R}_{c^*}$ *that the camera has to realize (expressed with the* $\theta\mathbf{u}$ *parameterization),*
- `vpFeatureTranslation` *: the translation* ${}^{c^*}\mathbf{t}_c$ *or* ${}^{c}\mathbf{t}_{c^*}$ *or* ${}^{c}\mathbf{t}_o$ *that the camera has to realize.*

1.1.3 Feature selection

When building a task (see Section 2.1), if $\dim \mathbf{s} > 1$, it is possible to select only a subset of the visual feature using a selection mechanism. This process will be described in details in section 2.1.

1.1.4 From Trackers to visual features

A visual feature $\mathbf{s}$ is built from a measure $\mathbf{x}(t)$: $\mathbf{s} = \mathbf{s}(\mathbf{x}(t))$. A set of function have been defined to compute visual features from the output of the various trackers existing in ViSP.

These functions are defined through the `vpFeatureBuilder` class. Example of how to build visual feature from trackers will be given in the next paragraph for each feature. Table 1.1 shows the various possibility that have been already implemented in ViSP. Let us note that it is possible to add new feature builder functions and that visual features parameters can be set without relying on these sets of functions.

1.2 2D visual features

1.2.1 Point

The typical mathematical model for a camera is defined by a perspective projection, such that any point M with coordinates $\mathbf{X} = (X, Y, Z)$ is projected onto the image plane in a point m with coordinates $\mathbf{x} = (x, y)$

	trackers							
	vpPoint	vpLine	vpCylinder	vpCircle	vpSphere	vpMeLine	vpDot	vpDot2
vpFeaturePoint	X						X	X
vpFeatureLine		X	X			X		
vpFeatureEllipse				X	X		X	X
vpFeatureVanishingPoint	X	X						
vpFeaturePoint3D	X							

Table 1.1: Relation between visual features and trackers.

with:

$$x = X/Z \ , \ \ y = Y/Z \tag{1.1}$$

By differentiating this equation, we get the variations in the image of the coordinates x and y of m with respect to the speed $\dot{\mathbf{X}}$ of the coordinates of point M:

$$\dot{\mathbf{x}} = \left[\begin{array}{ccc} 1/Z & 0 & -X/Z^2 \\ 0 & 1/Z & -Y/Z^2 \end{array} \right] \dot{\mathbf{X}} \tag{1.2}$$

Whatever configuration is chosen (the eye-in-hand or eye-to-hand configuration, static or mobile point M), the speed $\dot{\mathbf{X}}$ of M according to the kinematic screw $\mathbf{v}$ between the camera and its environment is given by the fundamental kinematics equation:

$$\dot{\mathbf{X}} = -\boldsymbol{\upsilon} - \boldsymbol{\omega} \times \mathbf{X} = -\boldsymbol{\upsilon} + [\mathbf{X}]_\times \, \boldsymbol{\omega} = \left[\begin{array}{cc} -\mathbb{I}_3 & [\mathbf{X}]_\times \end{array} \right] \mathbf{v} \tag{1.3}$$

Equation (1.2) can then be simplifed using Equation (1.1), written in the form:

$$\dot{\mathbf{x}} = \mathbf{L_x}(\mathbf{x}, Z) \ \mathbf{v} \tag{1.4}$$

where:

$$\mathbf{L_x}(\mathbf{x}, Z) = \left[\begin{array}{cccccc} -1/Z & 0 & x/Z & xy & -(1+x^2) & y \\ 0 & -1/Z & y/Z & 1+y^2 & -xy & -x \end{array} \right] \tag{1.5}$$

The point is certainly the most classical visual feature used in visual servoing. It is implemented in ViSP in the `vpFeaturePoint` *class.*

We present here how this class is implemented in ViSP. It could be considered as an example for people who want to add new visual features in the ViSP kernel.

vpFeaturePoint implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defines the value of the visual feature: in the case of the point* $\mathbf{s} = (x, y)$ *;*
- *a set of 3D information that are necessary to compute the interaction matrix: in the case of the point we have only the depth* Z *;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$. *In the case of the point it is defined by equation (1.5) ;*

- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$. *In the case of the point we simply have:*

$$\mathbf{s} - \mathbf{s}^* = \left[\begin{array}{c} x - x^* \\ y - y^* \end{array} \right]$$

A minimal prototype for the `vpFeaturePoint` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeaturePoint.h` *file):*

```
class vpFeaturePoint : public vpBasicFeature {
  protected :
    double Z ;
  public:
    void set_x(const double x) { vpBasicFeature::s[0] = x ; }
    void set_y(const double y) { vpBasicFeature::s[1] = y ; }
    void set_Z(const double Z) { this->Z = Z ; }
    double get_x() const { return  vpBasicFeature::s[0] ; }
    double get_y() const { return  vpBasicFeature::s[1] ; }

  public:
    vpMatrix interaction() ;
    vpColVector error(const vpBasicFeatures& s_star) ;
} ;

vpMatrix
vpFeaturePoint::interaction() {
  vpMatrix Ls ;
  Ls.resize(2,6) ;

  double x = s[0];
  double y = s[1];

  Ls[0][0] = - 1/Z ; Ls[0][1] =  0    ; Ls[0][2] = x/Z ; ... ;
  Ls[1][0] =  0    ; Ls[1][1] = - 1/Z ; Ls[1][2] = y/Z ; ... ;
  return Ls;
}

vpColVector
vpFeaturePoint::error(const vpBasicFeatures& s_star) {
  vpColVector e ;
  e = s  - s_star ;
  return e ;
}
```

1.2.2 Line

Let us recall that we use the (ρ, θ) representation for a line defined by:

$$x \cos\theta + y \sin\theta - \rho = 0 \qquad (1.6)$$

The relation that link the depth Z of a point of the line to its position in the image plane is given by

$$1/Z = Ax + By + C \qquad (1.7)$$

with $A = -A_1/D_1, B = -B_1/D_1$ and $C = -C_1/D_1$ where

$$A_1X + B_1Y + C_1Z + D_1 = 0 \qquad (1.8)$$

is the equation of a plane to which the line belong.

If we differentiate Equation (1.6), which corresponds to the hypothesis that the image of a line remains a straigh line whatever the camera's motion, we get:

$$\dot{\rho} + (x \sin\theta - y \cos\theta)\,\dot{\theta} = \dot{x}\cos\theta + \dot{y}\sin\theta\ ,\ \forall (x,y) \in \mathcal{D} \tag{1.9}$$

Based on Equation (1.6), $\mathbf{x}$ is written according to y if $\cos\theta = 0$ (or y according to $\mathbf{x}$ if that is not the case) and Equation (1.9) can then be rewritten, using (1.4) and (1.7):

$$(\dot{\rho} + \rho \tan\theta\,\dot{\theta}) + y\,(-\dot{\theta}/\cos\theta) = \mathbf{K_1}\ \mathbf{v} + y\ \mathbf{K_2}\ \mathbf{v}\ ,\ \forall y \in \mathbb{R} \tag{1.10}$$

with:

$$\begin{array}{l} \mathbf{K_1} = [\ \lambda_1 \cos\theta \quad \lambda_1 \sin\theta \quad -\lambda_1\rho \quad \sin\theta \quad -\cos\theta - \rho^2/\cos\theta \quad -\rho\tan\theta\ ] \\ \mathbf{K_2} = [\ \lambda_2 \cos\theta \quad \lambda_2 \sin\theta \quad -\lambda_2\rho \quad \rho \quad \rho\tan\theta \quad 1/\cos\theta\ ] \end{array}$$

where $\lambda_1 = -A\rho/\cos\theta - C$ and $\lambda_2 = A\tan\theta - B$.

Immediately we infer that:

$$\left\{ \begin{array}{lcl} \dot{\rho} & = & (\mathbf{K_1} + \rho\sin\theta\ \mathbf{K_2})\ \mathbf{v} \\ \dot{\theta} & = & -\cos\theta\ \mathbf{K_2}\ \mathbf{v} \end{array} \right. \tag{1.11}$$

hence:

$$\begin{array}{l} \mathbf{L}_\rho = [\ \lambda_\rho \cos\theta \quad \lambda_\rho \sin\theta \quad -\lambda_\rho\rho \quad (1+\rho^2)\sin\theta \quad -(1+\rho^2)\cos\theta \quad 0\ ] \\ \mathbf{L}_\theta = [\ \lambda_\theta \cos\theta \quad \lambda_\theta \sin\theta \quad -\lambda_\theta\rho \quad -\rho\cos\theta \quad -\rho\sin\theta \quad -1\ ] \end{array} \tag{1.12}$$

with $\lambda_\rho = -A\rho\cos\theta - B\rho\sin\theta - C$ and $\lambda_\theta = -A\sin\theta + B\cos\theta$.

Cylinders Note that this feature can be use to model the interaction matrix related to the projection of the two limbs of a cylinder. Knowing the equation of a cylinder in the camera frame:

$$(X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - (\alpha X + \beta Y + \gamma Z)^2 - R^2 = 0 \tag{1.13}$$

where R is the radius of the cylinder, α, β and γ are the coordinates of its direction vector and X_0, Y_0 and Z_0 are the coordinates of the nearest point belonging to the cylinder axis from the projection center. In this case, the coefficients of plane (1.8) which the limbs of the cylinder belong are:

$$\left\{ \begin{array}{l} A1 = \gamma\ Y_0 - \beta\ Z_0 \\ B1 = \alpha\ Z_0 - \gamma\ X_0 \\ C1 = \beta\ X_0 - \alpha\ Y_0 \\ D1 = \sqrt{X_0^2 + Y_0^2 + Z_0^2 - R^2} \end{array} \right.$$

The line is implemented in ViSP in the `vpFeatureLine` *class.*
We present here how this class is implemented in ViSP.

vpFeatureLine implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defined the value of the visual feature: in the case of the line* $\mathbf{s} = (\rho, \theta)$ *;*
- *a set of 3D information that are necessary to compute the interaction matrix: in the case of the line we have the 4 coefficients* (A, B, C, D) *of the plane to which the line belong (see Equation (1.8)) ;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$*. In the case of the line it is defined by equation (1.12) ;*

- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$*. In the case of the line we have:*

$$\mathbf{s} - \mathbf{s}^* = \left[\begin{array}{c} \rho - \rho^* \\ \theta - \theta^* \end{array} \right]$$

where the value of $\theta - \theta^*$ *is brought back in the interval* $[-\pi, \pi[$ *.*

A minimal prototype for the `vpFeatureLine` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeatureLine.h` *file):*

```
class vpFeatureLine : public vpBasicFeature {
  private :
    double A,B,C,D ;
  public:
    void setRhoTheta(const double rho,const double theta) {
       vpBasicFeature::s[0] = rho ; vpBasicFeature::s[1] = theta ;
    }
    void setABCD(const double A,const double B,const double C,const double D) {
       this->A = A ; this->B = B ; this->C = C ; this->D = D ;
    }
    double getRho() const { return  vpBasicFeature::s[0] ; }
    double getTheta() const { return  vpBasicFeature::s[1] ; }

  public:
    vpMatrix interaction() ;
    vpColVector error(const vpBasicFeatures& s_star) ;
} ;

vpMatrix
vpFeatureLine::interaction() {
  vpMatrix Ls;
  Ls.resize(2,6);

  double rho = s[0];
  double theta = s[1];

  double co = cos(theta);
  double si = sin(theta);

  double lambda_theta = (A*si - B*co) /D;
  double lambda_rho = (C + rho*A*co + rho*B*si)/D;

  Ls[0][0] = co*lambda_rho;   Ls[0][1] = si*lambda_rho;   Ls[0][2] = -rho*lambda_rho;   ... ;
  Ls[1][0] = co*lambda_theta; Ls[1][1] = si*lambda_theta; Ls[1][2] = -rho*lambda_theta; ... ;
  return Ls;
}

vpColVector
vpFeatureLine::error(const vpBasicFeatures& s_star) {
  vpColVector e ;
  e = s  - s_star ;
  // We brought back the theta error value in the interval [-Pi,Pi[
  while (e[1] < -M_PI) e[1] += 2*M_PI ;
  while (e[1] > M_PI)  e[1] -= 2*M_PI ;
  return e ;
}
```

1.2.3 Ellipse

We use a representation of the ellipse based on the normalized inertial moments $\mathbf{p} = (x_c, y_c, \mu_{20}, \mu_{11}, \mu_{02})$.

The interaction matrix related to the ellipse is given by:

$$
\begin{array}{llll}
L_{x_c}^T = [& -1/Z_c & 0 & x_c/Z_c + a\mu_{20} + b\mu_{11} \\
 & x_c y_c + \mu_{11} & -1 - x_c^2 - \mu_{20} & y_c \quad] \\
L_{y_c}^T = [& 0 & -1/Z_c & y_c/Z_c + a\mu_{11} + b\mu_{02} \\
 & 1 + y_c^2 + \mu_{02} & -x_c y_c - \mu_{11} & -x_c \quad] \\
L_{\mu_{20}}^T = [& -2(a\mu_{20} + b\mu_{11}) & 0 & 2[(1/Z_c + a x_c)\mu_{20} + b x_c \mu_{11}] \\
 & 2(y_c\mu_{20} + x_c\mu_{11}) & -4\mu_{20}x_c & 2\mu_{11} \quad] \\
L_{\mu_{11}}^T = [& -a\mu_{11} - b\mu_{02} & -a\mu_{20} - b\mu_{11} & a y_c \mu_{20} + (3/Z_c - c)\mu_{11} + b x_c \mu_{02} \\
 & 3y_c\mu_{11} + x_c\mu_{02} & -y_c\mu_{20} - 3x_c\mu_{11} & \mu_{02} - \mu_{20} \quad] \\
L_{\mu_{02}}^T = [& 0 & -2(a\mu_{11} + b\mu_{02}) & 2[(1/Z_c + b y_c)\mu_{02} + a y_c \mu_{11}] \\
 & 4y_c\mu_{02} & -2(y_c\mu_{11} + x_c\mu_{02}) & -2\mu_{11} \quad]
\end{array}
\tag{1.14}
$$

where $\frac{1}{Z_c} = ax_c + by_c + c$ is the equation of the plane to which the ellipse belong.

Ellipse as the projection of a circle. A circle may be represented as the intersection of a sphere and a plane:

$$
\left\{ \begin{array}{l} (X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - R^2 = 0 \\ \alpha(X - X_0) + \beta(Y - Y_0) + \gamma(Z - Z_0) = 0 \end{array} \right. \tag{1.15}
$$

The projection of a circle is an ellipse. The interaction matrix related to an ellipse has already been given. Results are therefore given by Equation (1.14) with the particular following values for a, b and c:

$$
\left\{ \begin{array}{l} a = \alpha/(\alpha X_0 + \beta Y_0 + \gamma Z_0) \\ b = \beta/(\alpha X_0 + \beta Y_0 + \gamma Z_0) \\ c = \gamma/(\alpha X_0 + \beta Y_0 + \gamma Z_0) \end{array} \right. \tag{1.16}
$$

Ellipse as the projection of a sphere. A sphere is represented by:

$$
(X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - r^2 = 0 \tag{1.17}
$$

The projection of a sphere is an ellipse. The interaction matrix related to an ellipse has already been given. Results are therefore given by Equation (1.14) with the particular following values for a, b and c:

$$
\left\{ \begin{array}{l} a = X_0/(X_0^2 + Y_0^2 + Z_0^2 - R^2) \\ b = Y_0/(X_0^2 + Y_0^2 + Z_0^2 - R^2) \\ c = Z_0/(X_0^2 + Y_0^2 + Z_0^2 - R^2) \end{array} \right. \tag{1.18}
$$

The ellipse is implemented in ViSP in the `vpFeatureEllipse` *class.*
We here present how this class is implemented in ViSP.

vpFeatureEllipse implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defined the value of the visual feature: in the case of the ellipse* $\mathbf{s} = (x, y, \mu_{20}, \mu_{11}, \mu_{02})$ *;*
- *a set of 3D information that are necessary to compute the interaction matrix: in the case of the ellipse we have the 3 coefficients* (A, B, C) *used in Equation (1.14) ;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$. *In the case of the ellipse it is defined by equation (1.14) ;*
- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$. *In the case of the ellipse we have:*

$$\mathbf{s} - \mathbf{s}^* = \begin{bmatrix} x - x^* \\ y - y^* \\ \mu_{20} - \mu20^* \\ \mu_{11} - \mu11^* \\ \mu_{02} - \mu02^* \end{bmatrix}$$

A minimal prototype for the `vpFeatureEllipse` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeatureEllipse.h` *file):*

```
1  class vpFeatureEllipse : public vpBasicFeature {
2    private :
3      double A,B,C ;
4    public:
5      void set_xy(const double x,const double y) {
6        vpBasicFeature::s[0] = x ; vpBasicFeature::s[1] = y ;
7      }
8      void setMu(const double mu20,const double mu11,const double mu02) {
9        vpBasicFeature::s[2] = mu20 ; vpBasicFeature::s[3] = mu11 ; vpBasicFeature::s[4] = mu02 ;
10     }
11     void setABC(const double A,const double B,const double C,const double D) {
12       this->A = A ; this->B = B ; this->C = C;
13     }
14     double get_x() const { return vpBasicFeature::s[0] ; }
15     double get_y() const { return vpBasicFeature::s[1] ; }
16     double getMu20() const { return vpBasicFeature::s[2] ; }
17     double getMu11() const { return vpBasicFeature::s[3] ; }
18     double getMu02() const { return vpBasicFeature::s[4] ; }
19
20   public:
21     vpMatrix interaction() ;
22     vpColVector error(const vpBasicFeatures& s_star) ;
23 } ;
24
25 vpMatrix
26 vpFeatureLine::interaction() {
27   vpMatrix Ls;
28   Ls.resize(5,6);
29
30   double xc = s[0] ;
31   double yc = s[1] ;
32   double mu20 = s[2] ;
33   double mu11 = s[3] ;
34   double mu02 = s[4] ;
35   double Z = 1/(A*xc + B*yc + C) ;
36
37   Ls[0][0] = -1/Z;   Ls[0][1] = 0;      Ls[0][2] = xc/Z + A*mu20 + B*mu11;   ... ;
```

```
  Ls[1][0] = 0;       Ls[1][1] = -1/Z;   Ls[1][2] = yc/Z + A*mu11 + B*mu02; ... ;
  Ls[2][0] = -2*(A*mu20+B*mu11);  Ls[2][1] = 0 ;  ... ;
  Ls[3][0] = -A*mu11-B*mu02;      Ls[3][1] = -A*mu20-B*mu11; ... ;
  Ls[4][0] = 0;                   Ls[4][1] = -2*(A*mu11+B*mu02);  ... ;
  return Ls;
}

vpColVector
vpFeatureLine::error(const vpBasicFeatures& s_star) {
  vpColVector e ;
  e = s  - s_star ;
  return e ;
}
```

1.2.4 Vanishing points

Assuming a vanishing point which coordinates are given in the image $\mathbf{x} = (x, y)$, the interaction matrix is similar to the point interaction with Z tends toward infinity and is given by [9]:

$$\mathbf{L_x} = \begin{bmatrix} 0 & 0 & 0 & xy & -(1+x^2) & y \\ 0 & 0 & 0 & 1+y^2 & -xy & -x \end{bmatrix} \tag{1.19}$$

The vanishing point is implemented in ViSP in the `vpFeatureVanishingPoint` *class. We present here how this class is implemented in ViSP.*

vpFeatureVanishingPoint implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defined the value of the visual feature: in the case of the vanishing point* $\mathbf{s} = (x, y)$ *;*
- *a set of 3D information that are necessary to compute the interaction matrix: in the case of the vanishing point we do not need 3D information ;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$*. In the case of the vanishing point it is defined by equation (1.19) ;*
- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$*. In the case of the vanishing point we simply have:*

$$\mathbf{s} - \mathbf{s}^* = \begin{bmatrix} x - x^* \\ y - y^* \end{bmatrix}$$

A minimal prototype for the `vpFeatureVanishingPoint` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeatureVanishingPoint.h` *file):*

```
class vpFeatureVanishingPoint : public vpBasicFeature {
 public:
    void set_x(const double x) { vpBasicFeature::s[0] = x ; }
    void set_y(const double y) { vpBasicFeature::s[1] = y ; }
    double get_x() const { return  vpBasicFeature::s[0] ; }
    double get_y() const { return  vpBasicFeature::s[1] ; }

  public:
    vpMatrix interaction() ;
    vpColVector error(const vpBasicFeatures& s_star) ;
} ;

```

```
13 vpMatrix
14 vpFeatureVanishingPoint::interaction() {
15   vpMatrix Ls ;
16   Ls.resize(2,6) ;
17
18   double x = s[0];
19   double y = s[1];
20
21   Ls[0][0] =  0 ;  Ls[0][1] =  0 ; Ls[0][2] = 0 ; Ls[0][3] = x*y ;   ... ;
22   Ls[1][0] =  0 ;  Ls[1][1] = 0 ;  Ls[1][2] = 0 ; Ls[1][3] = 1+y*y ; ... ;
23   return Ls;
24 }
25
26 vpColVector
27 vpFeatureVanishingPoint::error(const vpBasicFeatures& s_star) {
28   vpColVector e ;
29   e = s  - s_star ;
30   return e ;
31 }
```

1.2.5 Moments

This part covers the usage of moment-based visual features. It is strongly recommended to read **??** first. Most of the features in visual servoing don't have nice decoupling properties. In other words, the interaction matrix associated to them is far from diagonal. Moment features can generate interaction matrices of the following form:

$$\begin{pmatrix} -1 & 0 & 0 & x_{n_{wx}} & x_{n_{wy}} & x_{n_{wz}} \\ 0 & -1 & 0 & y_{n_{wx}} & y_{n_{wy}} & y_{n_{wz}} \\ 0 & 0 & -1 & a_{n_{wx}} & a_{n_{wy}} & 0 \\ 0 & 0 & 0 & c_{i_{wx}} & c_{i_{wy}} & 0 \\ 0 & 0 & 0 & c_{j_{wx}} & c_{j_{wy}} & 0 \\ 0 & 0 & 0 & \alpha_{wx} & \alpha_{wy} & -1 \end{pmatrix}$$

Just like a `vpFeaturePoint` uses the `vpPoint` tracker, the `vpFeatureMoment` makes use of the `vpMoment` primitives. These primitives are stored in a `vpMomentDatabase`, the same way it is described in **??**. Similarily, the `vpFeatureMoment` can access other interaction matrices (also stored in a database) to combine them. Finally, the `vpFeatureMoment` outputs the interaction matrix associated with its feature.

The process is be summed up in Fig. 1.1.

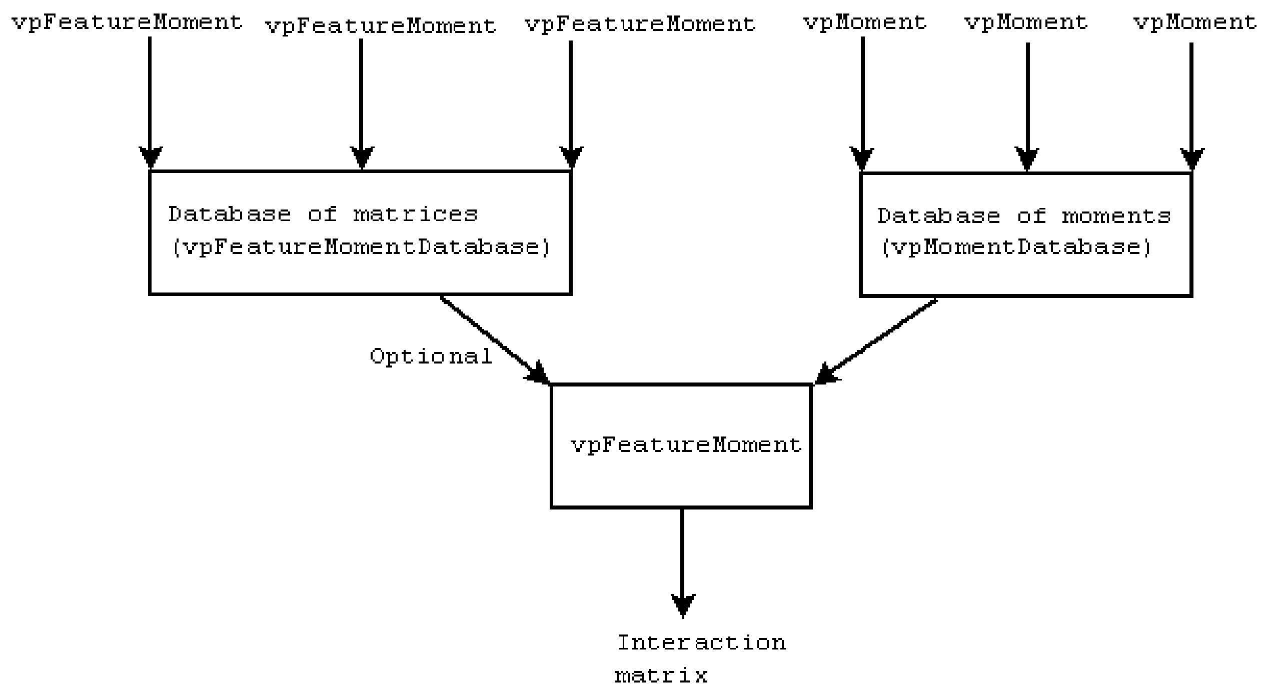

Figure 1.1: General scheme used to compute an interaction matrix from a moment primitives or other features. Several vpFeatureMoments can be used to build a database of interaction matrices. Several moments are also used to build a database of vpMoments. The vpFeatureMoment class uses those databases to compute it's corresponding interaction matrix.

1.2.5.1 Moment features

All moment features share some common behaviours such as being interdependent, their configuration with respect to the object's planar orientation and more. Hence the existence of a parent class called vpFeatureMoment which sums up as follows:

```
class vpFeatureMoment : public vpBasicFeature{
  public:
    vpFeatureMoment(vpMomentDatabase& moments,double A=0.0, double B=0.0, double C=0.0,
                    vpFeatureMomentDatabase* FeatureMoments=NULL)

    //common implementation
    vpBasicFeature* duplicate ()  const;
    vpColVector   error (const vpBasicFeature &s_star, unsigned int select=FEATURE_ALL) const;
    int   getDimension (unsigned int select=FEATURE_ALL) const;
    void  init ();
    vpMatrix  interaction (const unsigned int select=FEATURE_ALL);
    void linkTo(vpFeatureMomentDatabase& featureMoments);


    void update (double A, double B, double C);
    virtual void  compute_interaction ();
    virtual const char* name() = 0;
    virtual const char* momentName() = 0;

};
```

First, the vpFeatureMoment provides a common implementation for most of vpBasicFeature's virtual methods such as duplicate(), error(), getDimension(), etc.. The core methods of the moment features are the following:

- update to specify the object's position with respect to the camera. In fact, this is one parameter the vpMomentObject does not define.
- compute_interaction to compute the interaction matrix after the object's position is updated.
- name to specify the name in order to locate the vpFeatureMoment in the feature database.
- momentName to specify the name in order to locate the vpFeatureMoment in the moment database.

This summary reveals a few unknown themes which we will address in due time.

1.2.5.2 The object's position

In **??**, all the information is taken from the vpMomentObject class. That's why the moments are updated with that class. However, knowing the moment primitives alone is not sufficient to compute the interaction matrix.

In fact, the interaction matrix also depends on the object's position with respect to the camera. This means that the user has to input the parameters of the plane containing the object. The plane parameters requested are (A, B, C) with a plane equation of the following form:

$$\frac{1}{Z} = Ax + By + C \tag{1.20}$$

with:

$$x = \frac{X}{Z}$$

$$y = \frac{Y}{Z}$$

(X, Y, Z) being the coordinates, in meters, of a point.

To properly compute an interaction matrix, we have to use the `vpFeatureMoment::update()` method first. Here is an example of how to compute the interaction matrix associated to the `vpFeatureMomentBasic`:

```
vpMomentDatabase mdb; //database for moment primitives. This will only contain the basic moment.
vpMomentBasic bm; //basic moment (this particular moment is nothing more than a shortcut to
                  //the vpMomentObject)
bm.linkTo(mdb); //add basic moment to moment database

vpFeatureMomentBasic fmb(mdb); //update and compute the vpMoment BEFORE doing any operations
                               //with vpFeatureMoment
bm.update(obj);
bm.compute();

fmb.update(0,0,1); //update the vpFeatureMoment with a plane configuration
std::cout << fmb.interaction(1,1) << std::endl;
```

First, note that `vpMoment` classes and `vpFeatureMoment` classes come in pairs. In fact, the `vpFeatureMoment` needs moment primitives to implement the `vpBasicFeature::get_s()` method. That's why we set `vpMomentBasic` as a dependency of `vpFeatureMomentBasic` in database `mdb` passed to the constructor.

After the `vpFeatureMomentBasic` is constructed, we `update()` and `compute()` the moment primitive `vpMomentBasic`. Then, the object's spatial orientation must be updated and the interaction matrix computed (all in the `fmb.update(0,0,1)` call). $A = 0, B = 0, C = 1$ defines an object positioned parallel to the camera plane at 1 meter depth.

1.2.5.3 Common moment features

Before examining the dependencies, let's take a look at some common interaction matrices. These interaction matrices are programmed into specialized `vpFeatureMoment` classes. Their name is of the form `vpFeatureMoment` *moment-name* and corresponds to the `vpMoment` *moment-name* moment primitive. The mathematical expressions of these interaction matrices may be found in the doxygen documentation and in [12]. Knowing the exact expression of interaction matrices is not critically important. However, it is important to know the order of moments constituing the expression. By knowing this order, the user will be able to initialize the `vpMomentObject` to the right order. A too low object order won't allow the system to compute the interaction matrices correctly. Sometimes, we don't need the whole interaction matrix corresponding to a moment primitive. Therefore, the specialized `vpFeatureMoment` classes provide various selectors to select only the rows we need. These interaction matrices are given by the following classes:

- `vpFeatureMomentBasic` computes the interaction matrix for the `vpMomentBasic` class. A shortcut method `vpFeatureMomentBasic::interaction(i,j)` allows to select $L_{m_{ij}}$ more easily.
- `vpFeatureMomentCentered` computes the interaction matrix for the `vpMomentCentered` class. The same shortcut method `vpFeatureMomentCentered::interaction(i,j)` helps selecting $L_{\mu_{ij}}$.
- `vpFeatureMomentGravityCenterNormalized` computes the interaction matrix for the `vpMomentCenterNormalized` class. It provides `vpFeatureMomentGravityCenterNormalized::selectXn()` and

vpFeatureMomentGravityCenterNormalized::selectYn() selectors. These selectors are used inside the interaction() method: interaction(selectXn() | selectYn()) or simply interaction() for selecting both features, interaction(selectXn()) or interaction(selectYn()) for selecting one of them. The minimum required moment order is 2.

- vpFeatureMomentAreaNormalized computes the interaction matrix for the vpMomentAreaNormalized class. There is no selection needed. The minimum required moment order is 1 for dense objects (image or polygon) but 3 for discrete objects (set of points).
- vpFeatureMomentCInvariant computes the interaction matrix for the vpMomentCInvariant class. There are several selection methods, one for each invariant. They go from selectC1() to selectC10(). Moreover there are selectors selectSx(), selectSy() for symmetric invariants. The minimum required moment order is 6.
- vpFeatureMomentAlpha computes the interaction matrix for the vpMomentAlpha class. There are no selectors. The minimum required moment order is 3.

The orders are summed up again in the following table:

feature class	minimal object order	
	dense object	discrete object
vpMomentBasic	variable	variable
vpMomentCentered	variable	variable
vpMomentGravityCenter	2	3
vpMomentGravityCenterNormalized	2	3
vpMomentAreaNormalized	1	3
vpMomentCInvariant	6	6
vpMomentAlpha	4	4

Table 1.2: Minimal order required to construct a vpMomentObject(order) object used to compute vpFeatureMoment... classes. Values are different if the object is dense (case of a polygon or an image) or discrete (case of a set of points).

In moment based visual servoing as described in [12], only the 4 last ones are directly used.

1.2.5.4 Dependencies and databases

Just like moment primitives, interaction matrices are interdependent and so are vpFeatureMoment classes. They are also dependent on moment primitives. Fig. 1.2 shows the dependencies for the vpFeatureMoment classes.

Just like moment primitives, feature dependencies are handled through a database. The vpFeatureMomentDatabase has a similar behaviour to vpMomentDatabase introduced in section **??**. vpFeatureMoments are added through the vpFeatureMoment::linkTo() method and accessed by name through the vpFeatureMomentDatabase::get() method. Moreover, just like the database of moment primitives has a pre-filled database, there exists a pre-filled database of common vpFeatureMoments. This database is called vpFeatureMomentCommon. It contains all common

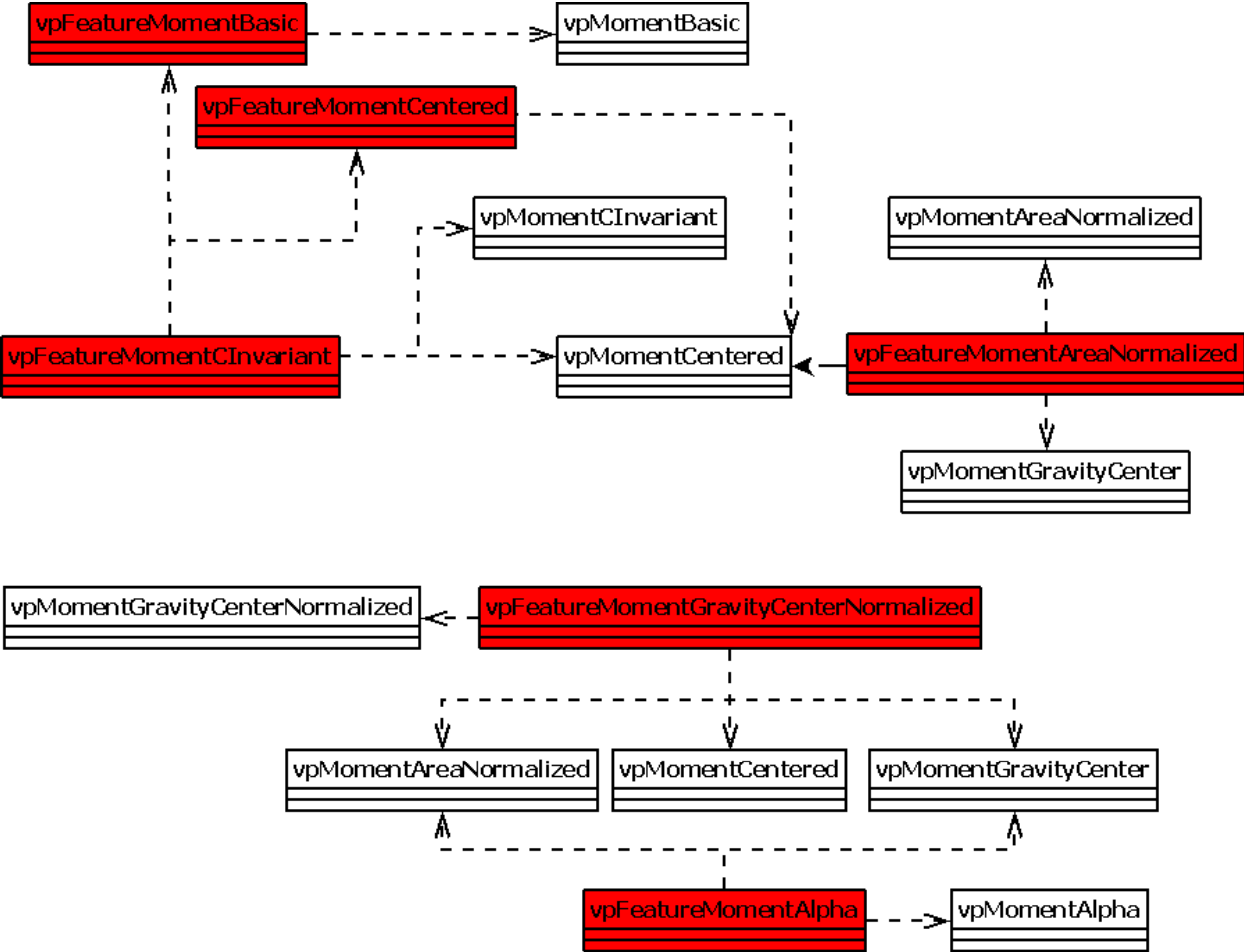

Figure 1.2: Dependencies between features and moments. Dependencies between moment primitives are not shown here but are given in Fig. **??**. vpMomentFeatures are shown in red.

vpFeatureMoments used in visual servoing to compute the interaction matrices described in [12] and their dependencies (showed in red in Fig. 1.1). Just like the vpMomentCommon database, this one provides the vpFeatureMomentCommon::updateAll(A,B,C) method which updates all vpFeatureMoments with the new object orientation and computes the interaction matrices for each feature. And just like vpMomentCommon, the feature database needs additional data to make it work. In this case, the data is less complicated. Here is the vpFeatureMomentCommon constructor:

```
vpFeatureMomentCommon(vpMomentDatabase& moments,double A=0.0,double B=0.0,double C=1.0)
```

Only the mandatory parameter moments, describing the database of moment primitives must be provided. This way, vpFeatureMomentCommon will have access to the moment dependencies.

Moreover, the feature database provides shortcuts to access its elements through the vpFeatureMomentCommon::get() method. These shortcuts are:

- getFeatureAlpha() returning the vpFeatureMomentAlpha.
- getFeatureAn() returning the vpFeatureMomentAreaNormalized.
- getFeatureCInvariant() returning the vpFeatureMomentCInvariant.
- getFeatureGravityNormalized() returning the vpFeatureMomentGravityCenterNormalized.

After this overview of ViSP's moment features, the goal of this tutorial will be presented: a complete visual servoing task.

1.2.5.5 Full visual servoing example

Creating a visual servoing task from scratch uses every notion learned in 1.2.5 and in moment primitives presented in **??**. In this part, we will work with two camera acquisitions: a source object and a destination object. In the end, we want to have the result of visual servoing: the velocity vector reducing the error between the two aquisitions. When repeated, this process will align the initial object with the destination one. The process could be summed up as in Fig. 1.3.

First of all, let's list the useful includes by following the process in the diagram:

```
#include <iostream> //some console output
#include <visp/vpPoint.h> //the basic tracker

#include <vector> //store the polygon
#include <visp/vpMomentObject.h> //transmit the polygon to the object
#include <visp/vpMomentCommon.h> //update the common database with the object
#include <visp/vpFeatureMomentCommon.h> //init the feature database using the information about moment dependenci
#include <visp/vpServo.h> //visual servoing task
```

To compute a velocity using visual servoing, we must at least have a source and a destination object. Let's describe them as polygons stored in a std::vector. In our example, the destination polygon will only be the translated version of the source one:

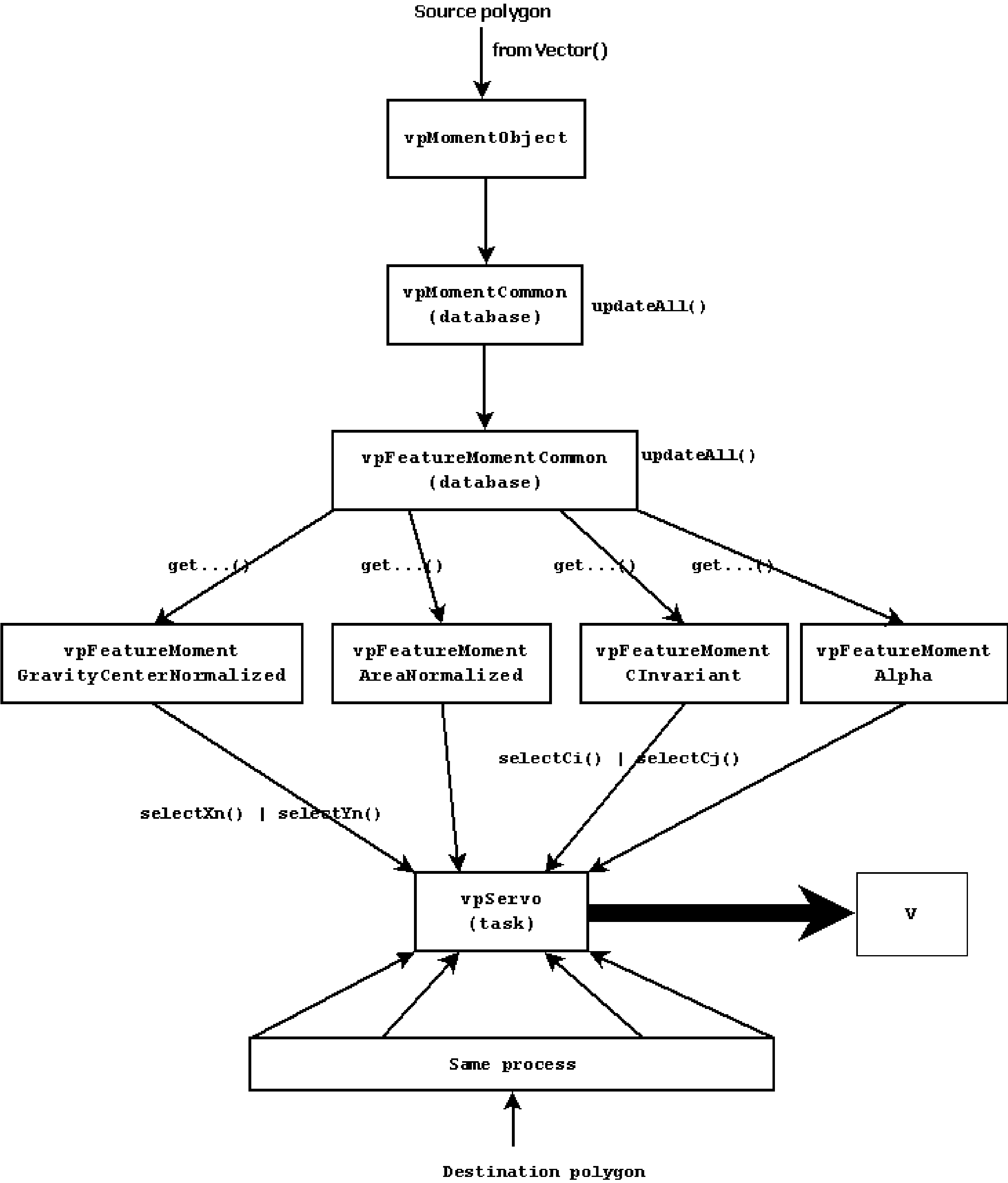

Figure 1.3: Visual servoing process with moments.

```
int main()
{
  // Define source polygon
  vpPoint p;
  std::vector<vpPoint> vec_p_s,vec_p_d; // vectors that contain the vertices of the contour polygon

  p.set_x(-0.2); p.set_y(0.1);   // coordinates in meters in the image plane (vertex 1)
  vec_p_s.push_back(p);
  p.set_x(+0.3); p.set_y(0.1);   // coordinates in meters in the image plane (vertex 2)
  vec_p_s.push_back(p);
  p.set_x(+0.2); p.set_y(-0.1);  // coordinates in meters in the image plane (vertex 3)
  vec_p_s.push_back(p);
  p.set_x(-0.2); p.set_y(-0.15); // coordinates in meters in the image plane (vertex 4)
  vec_p_s.push_back(p);
  p.set_x(-0.2); p.set_y(0.1);   // close the contour (vertex 5 = vertex 1)
  vec_p_s.push_back(p);

  //Define destination polygon. This is the source polygon translated of 0.1 on x-axis
  p.set_x(-0.1); p.set_y(0.1);   // coordinates in meters in the image plane (vertex 1)
  vec_p_d.push_back(p);
  p.set_x(+0.4); p.set_y(0.1);   // coordinates in meters in the image plane (vertex 2)
  vec_p_d.push_back(p);
  p.set_x(+0.3); p.set_y(-0.1);  // coordinates in meters in the image plane (vertex 3)
  vec_p_d.push_back(p);
  p.set_x(-0.1); p.set_y(-0.15); // coordinates in meters in the image plane (vertex 4)
  vec_p_d.push_back(p);
  p.set_x(-0.1); p.set_y(0.1);   // close the contour (vertex 5 = vertex 1)
  vec_p_d.push_back(p);
```

Once the vectors are filled up, objects need to be defined. The main question now is: what order should be used to initialize it? Since the `vpFeatureMomentCommon` contains `vpFeatureMomentCInvariant`, the answer is: at least 6 (see Table 1.2).

```
vpMomentObject src(6); // Create a source moment object with 6 as maximum order
src.setType(vpMomentObject::DENSE_POLYGON); // The object is defined by a countour polygon
src.fromVector(vec_p_s); // Init the dense object with the source polygon

vpMomentObject dst(6); // Create a destination moment object with 6 as maximum order
dst.setType(vpMomentObject::DENSE_POLYGON); // The dense object is defined by a countour polygon
dst.fromVector(vec_p_d); // Init the dense object with the destination polygon
```

Now we need to set up common databases for moment primitives. One for the source object and one for destination. Some reference values for the alpha moment will be needed. Once this is done, we can set up databases for moment features and provide them with the `vpMomentCommon` databases.

```
//init classic moment primitives (for source)
vpMomentCommon mdb_src(vpMomentCommon::getSurface(dst),vpMomentCommon::getMu3(dst),
                       vpMomentCommon::getAlpha(dst),1.);
//Init classic features
vpFeatureMomentCommon fmdb_src(mdb_src);

////init classic moment primitives (for destination)
vpMomentCommon mdb_dst(vpMomentCommon::getSurface(dst),vpMomentCommon::getMu3(dst),
                       vpMomentCommon::getAlpha(dst),1.);
//Init classic features
vpFeatureMomentCommon fmdb_dst(mdb_dst);
```

All databases (moment and feature) are now initialized. We need to update them. First, the moment database is updated with the `vpMomentObject`. Then, the feature database is updated with the object's orientation in space. In our example, the object is only translated along the X axis. Therefore it stays on the same plane and the A,B,C parameters stay the same. The process is done for both source and destination.

```
//update+compute moment primitives from object (for source)
mdb_src.updateAll(src);
//update+compute features (+interaction matrixes) from plane
fmdb_src.updateAll(0.,0.,1.);

//update+compute moment primitives from object (for destination)
mdb_dst.updateAll(dst);
//update+compute features (+interaction matrixes) from plane
fmdb_dst.updateAll(0.,0.,1.);
```

At this point, the visual features are ready for both objects. The next step is to set up the visual servoing task. An eye-in-hand configuration will be used and the interaction matrix for the current object will be computed. The static gain is set to 1.

```
//define visual servoing task
vpServo task;
task.setServo(vpServo::EYEINHAND_CAMERA);
task.setInteractionMatrixType(vpServo::CURRENT);
task.setLambda(1) ;
```

The task is initialized but we still need to add the visual features to it. To do that efficiently, we will use out `get...()` shortcuts. To provide ViSP with the information about which rows to select, the selectors must be used. All rows for the `vpFeatureMomentGravityCenter`, will be used. No selection is needed in this case. Since the object is not symmetric, c_i invariants is a good choice, for example c_4 and c_6 [12].

```
task.addFeature(fmdb_src.getFeatureGravityNormalized(),fmdb_dst.getFeatureGravityNormalized());
task.addFeature(fmdb_src.getFeatureAn(),fmdb_dst.getFeatureAn());
//the object is NOT symmetric
//select C4 and C6
task.addFeature(fmdb_src.getFeatureCInvariant(),fmdb_dst.getFeatureCInvariant(),
               vpFeatureMomentCInvariant::selectC4() | vpFeatureMomentCInvariant::selectC6());
task.addFeature(fmdb_src.getFeatureAlpha(),fmdb_dst.getFeatureAlpha());
```

The visual servoing task is now ready. The last step will display the computed velocity. This last step can be repeted in a loop until the task reaches convergence. A complete example of this is available in ViSP in the file manServoMomentsSimple.cpp.

```
vpColVector v = task.computeControlLaw();
std::cout << v << std::endl;
```

1.3 3D visual features

Assuming that the pose ${}^c\mathbf{M}_o$ between the object frame $\mathcal{F}_c$ and the camera frame $\mathcal{F}_o$ is known, it is also possible to consider 3D visual features.

1.3.1 Interaction matrix related to a 3-D point

Using the fundamental kinematics equation given in (1.3), we immediately get for any point with coordinates $\mathbf{X}$ related to the object:

$$\mathbf{L_X} = \begin{bmatrix} -\mathbb{I}_3 & [\mathbf{X}]_\times \end{bmatrix} \tag{1.21}$$

The point3D is implemented in ViSP in the `vpFeaturePoint3D` *class.*
We present here how this class is implemented in ViSP.

vpFeaturePoint3D implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defines the value of the visual feature: in the case of the point 3D* $\mathbf{s} = (x, y, z)$ *;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$*. In the case of the point 3D it is defined by equation (1.21) ;*
- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$*. In the case of the point 3D we simply have:*

$$\mathbf{s} - \mathbf{s}^* = \begin{bmatrix} x - x^* \\ y - y^* \\ z - z^* \end{bmatrix}$$

A minimal prototype for the `vpFeaturePoint3D` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeaturePoint3D.h` *file):*

```
class vpFeaturePoint3D : public vpBasicFeature {
  public:
    void set_X(const double X) { vpBasicFeature::s[0] = X ; }
    void set_Y(const double Y) { vpBasicFeature::s[1] = Y ; }
    void set_Z(const double Z) { vpBasicFeature::s[2] = Z ; }
    double get_X() const { return  vpBasicFeature::s[0] ; }
    double get_Y() const { return  vpBasicFeature::s[1] ; }
    double get_Z() const { return  vpBasicFeature::s[2] ; }

  public:
    vpMatrix interaction() ;
    vpColVector error(const vpBasicFeatures& s_star) ;
} ;

vpMatrix
vpFeaturePoint3D::interaction() {
  vpMatrix Ls ;
  Ls.resize(3,6) ;

  Ls[0][0] = -1 ; Ls[0][1] =  0 ; Ls[0][2] =  0 ; Ls[0][3] =  0 ; Ls[0][4] = -Z ; Ls[0][5] =  Y ;
  Ls[1][0] =  0 ; Ls[1][1] = -1 ; Ls[1][2] =  0 ; Ls[1][3] =  Z ; Ls[1][4] =  0 ; Ls[1][5] = -X ;
  Ls[2][0] =  0 ; Ls[2][1] =  0 ; Ls[2][2] = -1 ; Ls[2][3] = -Y ; Ls[2][4] =  X ; Ls[2][5] =  0 ;
  return Ls;
}

vpColVector
vpFeaturePoint3D::error(const vpBasicFeatures& s_star) {
  vpColVector e ;
  e = s  - s_star ;
  return e ;
}
```

The points taken into account can be characteristic points of the object [8, 11], or also the origin of R_o (we then have $\mathbf{X} = {}^c\mathbf{t}_o$).

1.3.2 Interaction matrix related to a translation

Thus, with an eye-in-hand camera, if we are interested in the translation $\mathbf{t}$ the camera must achieve, we can consider:

- $\mathbf{t}$ relative to the desired camera frame $\mathcal{F}_{c^*}$. We then have $\mathbf{s} = {}^c\mathbf{t}_{c^*}$ and $\mathbf{s}^* = \mathbf{0}$ [7]. In that case we get:

$$\mathbf{L}_{{}^c\mathbf{t}_{c^*}} = \begin{bmatrix} -\mathbf{I}_3 & [{}^c\mathbf{t}_{c^*}]_\times \end{bmatrix} \quad (1.22)$$

- $\mathbf{t}$ relative to the current camera frame $\mathcal{F}_c$. we then have $\mathbf{s} = {}^{c^*}\mathbf{t}_c$ and $\mathbf{s}^* = \mathbf{0}$. In that case we get:

$$\mathbf{L}_{{}^{c^*}\mathbf{t}_c} = \left[\begin{array}{cc} {}^{c^*}\mathbf{R}_c & \mathbf{0}_3 \end{array} \right] \tag{1.23}$$

- $\mathbf{t}$ relative to the frame $\mathcal{F}_o$ attached to the object [13]. We then have $\mathbf{s} = {}^{c}\mathbf{t}_o$ and $\mathbf{s}^* = {}^{c^*}\mathbf{t}_o$ (see figure 1.4). In that case we get:

$$\mathbf{L}_{{}^{c}\mathbf{t}_o} = \left[\begin{array}{cc} -\mathbf{I}_3 & [{}^{c}\mathbf{t}_o]_\times \end{array} \right] \tag{1.24}$$

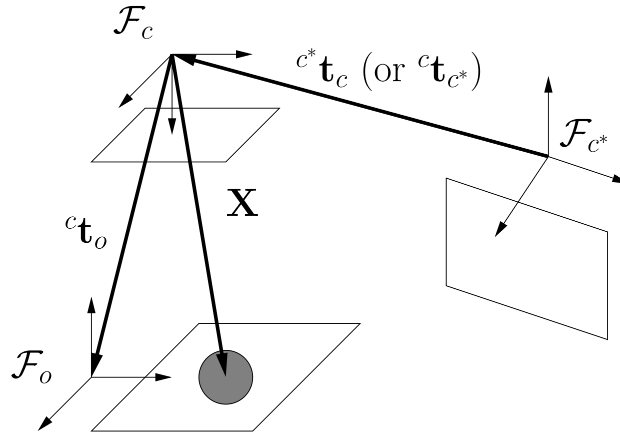

Figure 1.4: Possible 3-D translations considered as visual features in ViSP with a eye-in-hand camera.

The feature translation is implemented in ViSP in the `vpFeatureTranslation` *class. We present here how this class is implemented in ViSP.*

vpFeatureTranslation implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defines the value of the visual feature: in the case of the translation* $\mathbf{s} = (t_x, t_y, t_z)$ *;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$*. In the case of the point 3D it is defined by equation (**??**) ;*
- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$*. In the case of the translation we compute the error only for* $\mathbf{s}^* = 0$*. In that case the error is equal to* $\mathbf{s}$*.*

A minimal prototype for the `vpFeatureTranslation` *class can be given by (complete prototype can be found in the* `src/visual-features/vpFeatureTranslation.h` *file):*

```
1  class vpFeatureTranslation : public vpBasicFeature {
2    public:
3      typedef enum {
4        cdMc, // consider the case where s =c* tc and s* = 0
5        cMcd, // consider the case where s =c tc* and s* = 0
6        cMo   // consider the case where s =c to and s* =c* to
7      } vpFeatureTranslationRepresentationType;
8
9    protected :
10     vpHomogeneousMatrix M ;
```

```
11      vpFeatureTranslationRepresentationType type;
12
13    public:
14      vpFeatureTranslation::vpFeatureTranslation(vpFeatureTranslationRepresentationType type)
15        : vpBasicFeature() {
16        this ->type = type;
17      }
18      void builFrom(const vpHomogeneousMatrix &M) {
19        this->M = M; // set especially the rotation matrix
20        vpBasicFeature::s[0] = M[0][3] ; // set tx
21        vpBasicFeature::s[1] = M[1][3] ; // set ty
22        vpBasicFeature::s[2] = M[2][3] ; // set tz
23      }
24      void set_Tx(const double tx) { vpBasicFeature::s[0] = tx ; }
25      void set_Ty(const double ty) { vpBasicFeature::s[1] = ty ; }
26      void set_Tz(const double tz) { vpBasicFeature::s[2] = tz ; }
27      double get_Tx() const { return  vpBasicFeature::s[0] ; }
28      double get_Ty() const { return  vpBasicFeature::s[1] ; }
29      double get_Tz() const { return  vpBasicFeature::s[2] ; }
30
31      vpMatrix interaction() ;
32      vpColVector error(const vpBasicFeatures& s_star) ;
33  } ;
34
35  vpMatrix
36  vpFeatureTranslation::interaction() {
37    vpMatrix Ls ;
38    Ls.resize(3,6) ;
39
40    if (type == cdMc) {
41      for (int i=0 ; i < 3 ; i++) {
42        for (int j=0 ; j < 3 ; j++) {
43          Ls[i][j] = M[i][j] ;
44          Ls[i][j+3] = 0 ;
45        }
46      }
47    }
48    else if (type == cMcd || type == cMo) {
49      Ls[0][0] = -1; Ls[0][1] =  0; Ls[0][2] =  0; Ls[0][3] = 0; Ls[0][4] = -s[2]; Ls[0][5] = s[1];
50      Ls[1][0] =  0; Ls[1][1] = -1; Ls[1][2] =  0; Ls[1][3] = s[2]; Ls[1][4] = 0; Ls[1][5] = -s[0];
51      Ls[2][0] =  0; Ls[2][1] =  0; Ls[2][2] = -1; Ls[2][3] = -s[1]; Ls[2][4] = s[0]; Ls[2][5] = 0;
52    }
53
54    return Ls;
55  }
56
57  vpColVector
58  vpFeatureTranslation::error(const vpBasicFeatures& s_star) {
59    if(type == cdMc || type == cMcd) {
60      if (s_star.get_s().sumSquare() > 1e-6) {
61        throw(vpFeatureException("s* should be zero !")) ;
62      }
63    }
64    vpColVector e ;
65    e = s  - s_star ;
66    return e ;
67  }
```

1.3.3 Interaction matrix related to $\theta\mathbf{u}$

Remember, first of all, that the $\theta\mathbf{u}$ representation is obtained in a unique manner from the coefficients $r_{ij}(i = 1..3, j = 1..3)$ of a rotation matrix $\mathbf{R}$ using the following equation [4] :

$$\theta\mathbf{u} = \frac{1}{2\ \text{sinc}\theta} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{1.25}$$

where $\theta = \arccos((r_{11}+r_{22}+r_{33}-1)/2))$ and where sincθ is the sinus cardinal such that $\sin\theta = \theta\ \text{sinc}\theta$ and sinc$0 = 1$.

In the case of an eye-in-hand system, it is possible to use for visual servoing the vector $\theta\mathbf{u}$ to represent the rotation ${}^{c^*}\mathbf{R}_c$ between R_{c^*} and R_c. If the matrices ${}^{c^*}\mathbf{R}_{n^*}$ and ${}^{c}\mathbf{R}_n$ are identical, which is usually the case, we can also consider the vector $\theta\mathbf{u}$ associated with the rotation ${}^{n^*}\mathbf{R}_n$. Likewise, with a scene camera, the vector $\theta\mathbf{u}$ can be used to represent either the rotation ${}^{o^*}\mathbf{R}_o$ between the desired frame and the current frame of the object connected to the effector, either the rotation ${}^{n^*}\mathbf{R}_n$ if the matrices ${}^{o^*}\mathbf{R}_{n^*}$ and ${}^{o}\mathbf{R}_n$ are identical (which is also usually the case). In all of the cases mentioned above, the interaction matrix associated with $\theta\mathbf{u}$ is given by [6]:

$$\mathbf{L}_{\theta\mathbf{u}} = \begin{bmatrix} \mathbf{0}_3 & \mathbf{L}_\omega \end{bmatrix} \tag{1.26}$$

with:

$$\mathbf{L}_\omega = \mathbb{I}_3 + \frac{\theta}{2}\ [\mathbf{u}]_\times + (1 - \frac{\text{sinc}\theta}{\text{sinc}^2\frac{\theta}{2}})\ [\mathbf{u}]_\times^2 \tag{1.27}$$

The $\theta\mathbf{u}$ representation is therefore particularly interesting since $\mathbf{L}_\omega$ is singular only for $\theta = 2\pi$. Furthermore, we have:

$$\mathbf{L}_\omega^{-1} = \mathbb{I}_3 + \frac{\theta}{2}\ \text{sinc}^2\frac{\theta}{2}\ [\mathbf{u}]_\times + (1 - \text{sinc}\theta)[\mathbf{u}]_\times^2 \tag{1.28}$$

which guarantees the following, rather convenient, property:

$$\mathbf{L}_\omega^{-1}\ \theta\mathbf{u} = \theta\mathbf{u} \tag{1.29}$$

If you would rather consider the rotations ${}^{c}\mathbf{R}_{c^*}$, ${}^{n}\mathbf{R}_{n^*}$ or ${}^{o}\mathbf{R}_{o^*}$ we immediately infer from (1.26) that:

$$\mathbf{L}_{\theta\mathbf{u}} = \begin{bmatrix} \mathbf{0}_3 & \mathbf{L}_\omega \end{bmatrix} \tag{1.30}$$

with:

$$\mathbf{L}_\omega = -\mathbb{I}_3 + \frac{\theta}{2}\ [\mathbf{u}]_\times - (1 - \frac{\text{sinc}\theta}{\text{sinc}^2\frac{\theta}{2}})\ [\mathbf{u}]_\times^2 \tag{1.31}$$

and we now have:

$$\mathbf{L}_\omega^{-1}\ \theta\mathbf{u} = -\theta\mathbf{u} \tag{1.32}$$

Note that it is not possible to directly take into account the vector $\theta\mathbf{u}$ associated to the rotation ${}^{c}\mathbf{R}_o$ and to base the argument on the difference between $\theta\mathbf{u}$ and $\theta^*\mathbf{u}^*$ (where $\theta^*\mathbf{u}^*$ represents the desired rotation ${}^{c^*}\mathbf{R}_o$). This is because $\theta\mathbf{u} - \theta^*\mathbf{u}^*$ does not represent a rotation in the space SO_3 of rotations [10].

The $\theta\mathbf{u}$ is implemented in ViSP in the `vpFeatureThetaU` *class.*
We present here how this class is implemented in ViSP.

vpFeatureThetaU implementation. *As pointed out in section 1.1.1, a feature is defined by:*

- *a vector* $\mathbf{s}$ *that defines the value of the visual feature. In the case of the* $\theta\mathbf{u}$, $\mathbf{s} = (\theta\mathbf{u}_x, \theta\mathbf{u}_y, \theta\mathbf{u}_z)$ *;*
- *an interaction matrix* $\mathbf{L_s}$ *that describes how* $\mathbf{s}$ *is modified when the camera is moving with a velocity* $\mathbf{v}$. *In the case of the* $\theta\mathbf{u}$ *it is defined by equation (1.26) ;*
- *a method to compute the error* $\mathbf{s} - \mathbf{s}^*$. *In the case of the* $\theta\mathbf{u}$ *we compute the error only for* $\mathbf{s}^* = 0$. *In that case the error is equal to* $\mathbf{s}$.

The prototype for the `vpFeatureThetaU` *class can be found in the* `src/visual-features/vpFeatureThetaU.h` *file.*

1.4 Generic feature

It is obviously impossible to provide a class for each possible visual feature. Therefore we define a `vpGenericFeature` *class for which the user may define the size of* $\mathbf{s}$, *the vector* $\mathbf{s}$ *itself, the interaction* $\mathbf{L_s}$ *and if necessary the way to compute the error.*

If we consider the case of the 2 1/2 visual servoing, as described in section 3.2, the vector $\mathbf{s}$ *used to defined this task is* $\mathbf{s} = (x, y, \log Z, \theta\mathbf{u})$. *If* (x, y) *and* $\theta\mathbf{u}$ *are already defined, nothing has been done dealing with* $\log(Z/Z^*)$. *We can then define a* `vpGenericFeature` *of size 1:*

```
1  vpGenericFeature logZ(1) ;      // log (Z/Z*)
```

It will then be possible to compute and update the interaction matrix at each iteration:

```
1     // compute log (Z/Z*), s_star is then equal to zero
2     logZ.set_s(log(Z/Zd)) ;
3
4     // and the corresponding interaction matrix
5     vpMatrix LlogZ(1,6) ;
6     LlogZ[0][0] = LlogZ[0][1] = LlogZ[0][5] = 0 ;
7     LlogZ[0][2] = -1/Z ;
8     LlogZ[0][3] = -p.get_y() ; // p is here a vpFeaturePoint
9     LlogZ[0][4] =  p.get_x() ;
10    logZ.setInteractionMatrix(LlogZ) ;
11
12    // The error is here equal to s = log(Z/Z*)
```

The new defined `vpGenericFeature` *can then be as any other visual feature. A full example that use this capabiblity is given in section 3.2.*

Chapter 2

Task and control laws

2.1 Create a task

One of the main issue in visual servoing is to set up the task. End-user has then to select a set of visual feature that will define the task. Using these elementary visual features, more complex tasks can be considered by stacking the elementary feature vectors.

The task is specified using the `vpServo` class (that is also used to define and compute the control law). The main methods used to define the task are:

```
int vpServo::addFeature(vpBasicFeature &s, vpBasicFeature &sd) ;
```

where `s` will be regulated to `sd` and

```
int vpServo::addFeature(vpBasicFeature &s) ;
```

where `s` will be regulated to zero.

`vpServo` contains two lists of current and visual features (in fact a list of pointer to `vpBasicFeature` elements) and each call to the `addFeature` method introduce new `vpBasicFeature` in both lists. These lists define the task. Actually the user does not call directly `addFeature` with `vpBasicFeature` but with the derived feature that have been defined. Let's consider again the 2 1/2 D task mentioned in the previous paragraph and illustrated in section 3.2. The task can be set up from the following code:

```
vpServo task ;

vpPoint p, pd ;
vpGenericFeature LogZ(1) ;
vpFeatureThetaU tu ;

<initialize p, pd, logZ, tu >

task.addFeature(p,pd) ;
task.addFeature(logZ) ;
task.addFeature(tu) ;
```

Let us note that in that case `logZ` and `tu` are to be regulated to zero. Dealing with `tu` which describe the rotation ${}^{c^*}\mathbf{R}_c$ that the camera has to achieved, it is even impossible to regulate it to another value than zero. `task.addFeature(tu,tud);` will throw an exception (this is done for the `vpFeatureThetaU` and `vpFeatureTranslation` visual feature).

Feature selection. It can be of interest to select a subset of the visual feature defined in a derived basic feature (for example, we may want to consider only the x coordinates of a point or the θu_z component of $\theta\mathbf{u}$). The provided solution is a modified version of `addFeature`:

```
int vpServo::addFeature(vpBasicFeature &s, vpBasicFeature &sd, int select) ;
```

that can be used as follow. Let us consider another solution for a 2 1/2 D visual servoing task. Let $\mathbf{s} = (\mathbf{t}, x, y, \theta u_z)$ be the vector of visual feature where $\mathbf{t}$ is the translation that the camera has to realize. Mainly wrt. to the previous case, only the third component of $\theta\mathbf{u}$ has to be selected.

```
vpServo task ;

vpTranslationVector t ;
vpPoint p, pd ;
vpThetaU tu ;

<initialiaze p, pd, t, tu >

task.addFeature(t) ;
task.addFeature(p,pd) ;
task.addFeature(tu, vpThetaU::selectThetaUZ) ;
```

Note that more than one sub-features can be selected using logical operator. For example, if we want to select the $\theta\mathbf{u}_x$ and $\theta\mathbf{u}_z$ of the $\theta\mathbf{u}$ feature we wrote:

```
task.addFeature(tu, vpThetaU::selectThetaUX() | vpThetaU::selectThetaUZ()) ;
```

Here the bitwise logical operator `|` act as an union operator (this is not the classical logical operator `||`) .

2.2 Control law

The control law is also handled by the `vpServo` class. A few members and methods are defined in order to set up a specific control law. Let us first recall here the visual servoing control. The notations used hereby will be also used in the implementation of the `vpServo` class.

From the previous definition, a general control is then computed using the following equations:

$$\mathbf{e} = \mathbf{W}_\mathbf{q}^+ \mathbf{e_1} + (\mathbf{I} - \mathbf{W}_\mathbf{q}^+ \mathbf{W}_\mathbf{q})\mathbf{e_2} \tag{2.1}$$

with

$$\mathbf{e_1} = \mathbf{W}_\mathbf{q}\left(\varepsilon \widehat{\mathbf{L}_\mathbf{s}} {}^c\mathbf{V}_a {}^a\mathbf{J}_e\right)^+ (\mathbf{s} - \mathbf{s}^*) \tag{2.2}$$

where $\mathbf{W}_\mathbf{q}$ is computed from $\mathbf{J_1} = \mathbf{L_s} {}^c\mathbf{V}_a {}^a\mathbf{J}_e$ and where $\varepsilon = 1$ in the eye-in-hand case and -1 in the eye-to-hand case.
Finally we get:

$$\dot{\mathbf{q}} = -\lambda \mathbf{e} \tag{2.3}$$

Note that from an implementation point of view, at the control law level, ViSP only manipulates matrices and vectors. It has no knowledge of the underlying visual feature.

Twist transformations and Jacobians. It has then to be noted that, in this set of equations, whereas some informations are defined by the task (e.g., $\mathbf{L_s}$, $\mathbf{s}$ or $\mathbf{s}^*$) or are computed automatically (e.g., $\mathbf{W_q}$), some matrices have to be defined by the programmer. These matrices are: ${}^c\mathbf{V}_a$ and ${}^a\mathbf{J}_e$.

It is clear that only the end-user knows in which frame his robot jacobian is expressed, the value of the camera-effector transformation (in the eye-in-hand case) or with respect to which frame the pose is computed in the eye-to-hand case. Therefore we have defined some methods (in the `vpServo` class) that allows to set these matrices. When computing the control law, the program will test if the matrices are indeed initialized and updated (if required). Table 2.1 summarizes the matrices twist transformations and jacobians that have to be initialized (I) or updated at each iteration of the control loop (U) depending of the chosen method. Note that, in the eye-to-hand case, when the camera is mobile, ${}^c\mathbf{V}_\mathcal{F}$ must also be updated.

		${}^c\mathbf{V}_e$	${}^c\mathbf{V}_\mathcal{F}$	${}^\mathcal{F}\mathbf{V}_e$	${}^e\mathbf{J}_e$	${}^\mathcal{F}\mathbf{J}_e$
eye-in-hand	$\dot{\mathbf{s}} = \mathbf{L_s}\ \mathbf{v}$					
	$\dot{\mathbf{s}} = \mathbf{L_s}{}^c\mathbf{V}_e\ {}^e\mathbf{J}_e\ \dot{\mathbf{q}}$	I			U	
eye-to-hand	$\dot{\mathbf{s}} = -\mathbf{L_s}\ \mathbf{v}$					
	$\dot{\mathbf{s}} = -\mathbf{L_s}{}^c\mathbf{V}_e\ {}^e\mathbf{J}_e\ \dot{\mathbf{q}}$	U			U	
	$\dot{\mathbf{s}} = -\mathbf{L_s}{}^c\mathbf{V}_\mathcal{F}\ {}^\mathcal{F}\mathbf{V}_e{}^e\mathbf{J}_e\dot{\mathbf{q}}$		I (or U)	U	U	
	$\dot{\mathbf{s}} = -\mathbf{L_s}{}^c\mathbf{V}_\mathcal{F}\ {}^\mathcal{F}\mathbf{J}_e\ \dot{\mathbf{q}}$		I (or U)			U

Table 2.1: Twist transformations and jacobians that have to be initialized (I) or updated in the control loop (U) depending of the chosen method. ${}^c\mathbf{V}_\mathcal{F}$ relates to the twist transformation matrix from camera frame $\mathcal{F}_c$ to the robot reference frame $\mathcal{F}_\mathcal{F}$.

The method defined to initialize or update these matrices are for the jacobians:

```
vpServo::set_eJe(vpMatrix &eJe)
vpServo::set_fJe(vpMatrix &fJe)
```

and for the twist transformation matrices:

```
vpServo::set_cVe(vpVelocityTwistMatrix &cVe)
vpServo::set_cVf(vpVelocityTwistMatrix &cVf)
vpServo::set_fVe(vpVelocityTwistMatrix &fVe)
```

Since twist transformation matrices are computed from pose we also provide these more convenient interfaces:

```
vpServo::set_cVe(vpHomogeneousMatrix &cMe)
vpServo::set_cVf(vpHomogeneousMatrix &cMf)
vpServo::set_fVe(vpHomogeneousMatrix &fMe)
```

Choice of the control law. The choice of the control law is automatically done with respect to the initialized matrices. When more than one choice is possible (it should not be) a warning is printed on `stderr`. An error is sent if, when computing the control law, one of these matrices is not initialized (that is when no choice is possible). However it is better to select explicitely the type of desired control law using the `vpServo::setServo` api.

```
vpServo::setServo(vpServo::vpServoType servoType)
```

with

```
class vpServo
{
...
public:
  typedef enum
```

```
  {
    NONE,
    EYEINHAND_CAMERA,        // consider the case where ṡ = Ls v
    EYEINHAND_L_cVe_eJe,     // consider the case where ṡ = Ls cVe eJe q̇
    EYETOHAND_L_cVe_eJe,     // consider the case where ṡ = −Ls cVe eJe q̇
    EYETOHAND_L_cVf_fVe_eJe, // consider the case where ṡ = −Ls cVF FVe eJe q̇
    EYETOHAND_L_cVf_fJe      // consider the case where ṡ = −Ls cVF FJe q̇
  } vpServoType;
...
};
```

Listing 2.1: Control law selector.

A flag is positioned when the matrix is initialized or updated. It is modified when the articular velocity is sent to robot controller. If the twist or jacobian matrices are not updated (they should be updated) prior to a new computation of the control law, a warning is printed on `stderr`.

An error is sent when the size of the jacobian is not compatible with the number of degrees of freedom available on the robot (a good solution to avoid this error is to read the jacobian from the `vpRobot` class...)

Example. The following listing is a piece of code showing how an eye-to-hand task can be implemented. The chosen control law is given by $\dot{\mathbf{s}} = -\mathbf{L_s}{}^c\mathbf{V}_{\mathcal{F}}\ {}^{\mathcal{F}}\mathbf{J}_e\dot{\mathbf{q}}$. Therefore the camera location wrt. the robot reference frame ${}^c\mathbf{M}_{\mathcal{F}}$ has to be known (but has not to be updated if the camera is motionless). An other requirement is that the jacobian of the robot expressed in the robot reference frame has also to be known `(robot.get_fJe(fJe))`. If such jacobian is not available then `vpRobot::get_fJe(vpMatrix &fJe)` should return an error (this is very important to consider such case when you write the `vpMyRobot` class).

```
vpMyRobot robot ;
vpServo task ;

...

task.setServo(vpServo::EYETOHAND_L_cVf_fJe) ;
// cMf is the position of the camera wrt. robot reference frame
task.set_cVf(cMf) ;
task.setLambda(0.2) ; // value of the constant control gain λ

while(...) {
  vpColVector dotq(6) ;      // velocity vector

  ...     // everything useful to update the visual feature

  vpMatrix fJe  ;
  robot.get_fJe(fJe) ;
  task.set_fJe(fJe) ;

  dotq = task.computeControlLaw() ;

  robot.setVelocity(dotq, vpRobot::ARTICULAR_FRAME) ;
}
```

Listing 2.2: Typical code for the visual servoing closed loop.

Interaction matrix. In (2.2) $\widehat{\mathbf{L_s}}$ is a model or an approximation of the interaction matrix. ViSP allows to consider various possibilities [1]:

- $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{p})$ where the interaction matrix is computed at the current position of the visual feature and the current 3D pose (denoted $\mathbf{p}$),
- $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{p}^*)$ where the interaction matrix is computed only once at the desired position of $\mathbf{s}$ and $\mathbf{p}$,
- $\widehat{\mathbf{L_s}} = \frac{1}{2}\big(\widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{p}) + \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{p}^*)\big)$ [5] .

The choice is done using the following method:

```
void vpServo::setInteractionMatrixType(vpServo::vpInteractionMatrixType type) ;
```

with:

```
typedef enum
{
  CURRENT,      // interaction matrix computed at current position $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{p})$
  DESIRED,      // interaction matrix computed at desired position $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{p}^*)$
  MEAN,         // interaction matrix computed as $\widehat{\mathbf{L_s}} = \frac{1}{2}(\widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{p}) + \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{p}^*))$
  USER_DEFINED // user defined interaction matrix
} vpInteractionMatrixType;
```

Listing 2.3: Interaction matrices selector.

`vpServo::setInteractionMatrixType` only set up a flag that will be used by the `vpServo::computeControlLaw()` or `vpServo::computeInteractionMatrix()` method. It is of the programmer responsibility to ensure that $\mathbf{s}$ or $\mathbf{s}^*$ are properly initialized prior to the interaction matrix computation (this includes 2D and, if necessary, 3D informations). The following listing shows how to use this function in the code.

```
vpServo task ;
task.setInteractionMatrixType(vpServo::CURRENT)

while(...) {
  ...
  dotq = task.computeControlLaw() ;
  ...
}
```

Another control law is possible using the transpose of the interaction matrix and no longer its pseudo inverse (though it is not recommended at all, behavior is really bad!)

$$\mathbf{e_1} = \mathbf{W_q}\big(\varepsilon\widehat{\mathbf{L_s}}\,{}^c\mathbf{V}_a\,{}^a\mathbf{J}_e\big)^\top(\mathbf{s} - \mathbf{s}^*) \tag{2.4}$$

this capability can be activated using

```
task.setInteractionMatrixType(vpServo::DESIRED,  vpServo::TRANSPOSE) ;
```

or

```
task.setInteractionMatrixType(vpServo::CURRENT,  vpServo::TRANSPOSE) ;
```

Chapter 3

Complete visual servoing experiments

We will describe the software environment from the end-user point of view, in the light of two simple examples implemented using ViSP.

3.1 Building a basic 2D visual servoing task

Listing 3.1 defines a typical initialization process that can be used in most programs using ViSP. These lines define the framegrabber (here for a firewire camera), the display system (here X11R6), a robot (the 6 d.o.f Afma gantry robot of INRIA), a camera (with given calibration parameters) and finally create a task.

```
vpImage<unsigned char> I ;
// use the 1394 framegraber coming with libdc1394-2.x
vp1394TwoGrabber grabber ;
// associate the grabber to the image
grabber.open(I) ;
grabber.acquire(I) ;

// use the X11R6 window system to display the image
// associate the display to the image
vpDisplayX display(I,"a new X11 window") ;
// display the image
vpDisplay::display(I) ;

// use the Afma6 robot
vpRobotAfma6 robot ;
// define camera calibration parameters

// define some camera parameters
vpCameraParameters cam(u0,v0,px,py) ;
// create a new task

vpServo task ;
< code for a specific experiment >
```

Listing 3.1: Typical code for ViSP initialization.

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. Dealing with the tracking process, in this example we choose to track fiducial markers (`vpDot2`) . The features (`vpFeaturePoint`) are created from the tracked markers (`vpDot2`) using member functions of the `vpFeatureBuilder` class (in this simple case, it mainly achieves a pixel-to-meter conversion). The desired positions of the visual feature $\mathbf{s}^*$ is also defined and a link between the current position (`s[i]`) of the

visual feature in the image and the desired position (`sd[i]`) is then created. 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 19 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 $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}^*, \mathbf{r}^*)$ (line 20) and the control law will computed using the pseudo-inverse of the interaction matrix (other possibilities such as considering the transpose of $\mathbf{L_s}$ also exist).

```
1   vpDot2 dot[4] ;                // tracked objects
2   vpFeaturePoint s[4], sd[4] ; // current and desired feature (4 points)
3   for (i=0; i<4; i++)
4   {
5     dot[i].initTracking(I) ; //  initialize the tracking process
6
7     // build current visual features from the tracked objects
8     vpFeatureBuilder::create(s[i],cam, dot[i])  ;
9
10    // Init here the desired visual features s* along with
11    // the 3D information required to compute the interaction matrix
12    sd[i].buildFrom(x[i],y[i],Z[i]) ;
13
14    // add point-to-point image constraint
15    // defines the list of visual features, the error vector
16    // as well as the interaction matrix
17    task.addFeature(s[i],sd[i]) ;
18  }
19  task.setServo(vpServo::EYEINHAND_CAMERA) ;
20  task.setInteractionMatrixType(vpServo::DESIRED, vpServo::PSEUDO_INVERSE);
```

Listing 3.2: An example of task definition: positioning wrt. four points.

It is then straightforward to write the loop itself. It features the image acquisition (line 4), the image processing to extract the new position of the dots (line 8) and the value of the visual feature `s` is recomputed (line 10). The task is automatically updated and the control law $\mathbf{v} = -\lambda \mathbf{L}^+_{\mathbf{s}|\mathbf{s}=\mathbf{s}^*}(\mathbf{s} - \mathbf{s}^*)$ is computed (line 13). Finally the result is sent to the robot controller.

```
1   task.setLambda(0.2) ;  // set the gain λ
2   while(...) {
3     vpColVector v(6) ;      // velocity vector
4     grabber.acquire(I) ;    // acquire a new image
5     for (i=0 ; i < 4 ; i++)
6     {
7       // perform the feature tracking
8       dot[i].track(I) ;
9       // and the pixel/meter conversion
10      vpFeatureBuilder::create(s[i],cam, dot[i]);
11    }
12
13    v =  task.computeControlLaw() ;  // v = −λL_s^+(s − s*)
14
15    // send the computed velocity (expressed in the camera frame)
16    // to the robot controller
17    robot.setVelocity(vpRobot::CAMERA_FRAME, v) ;
18  }
```

Listing 3.3: Typical code for the visual servoing closed loop.

A complete example named `servoAfma6FourPoints2DCamVelocityInteractionDesired.cpp` can be found in ViSP source tree.

3.2 Building a 2 1/2 D visual servoing task

If we now consider the case of a 2 1/2 D visual servoing task, the visual features $\mathbf{s}$ are defined by $\mathbf{s} = (x, y, \log Z, \theta\mathbf{u})$ (see Figure 3.1) . Therefore, to achieve this task we need to know the 3D position of the point and the camera pose wrt. to the scene. Listing 3.4 shows how to get the pose ${}^{c}\mathbf{M}_{o}$ from a set of five points. The class `vpPose` provides function 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 [2] approach and improved using a non-linear minimization method using virtual visual servoing approach.

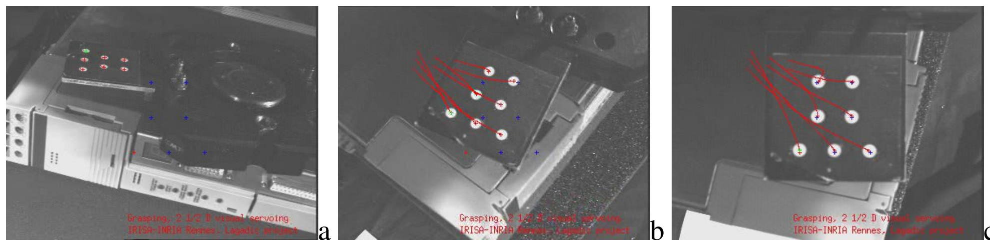

Figure 3.1: 2 1/2 D visual servoing task as implemented in section 3.2

```
vpDot2 dot[7] ; // Create 7 dot trackers
vpPoint point[7] ; // Create 7 point visual features

// init the 3D coordinates (X,Y,Z) of the points,
// our 5 points are planar, that's why the Z coordinate is null.
double L = 0.06;
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<7; i++)
{
  // initialize the tracking process
  dot[i].initTracking(I, cam) ;
  // get the 2D position of the point in meter
  dot[i].track(I) ;

  // pixel-to-meter conversion
  vpPixelMeterConversion::convertPoint(cam,  dot[i], point[i])  ;
  // here the 2D coordinates (x,y) and 3D coordinates (X,Y,Z) of the point are available

  // consider this point in the pose computation algorithm
  pose.addPoint(point[i]) ;
}
vpHomogeneousMatrix cMo ;
pose.computePose(vpPose::DEMENTHON, cMo) ;
pose.computePose(vpPose::VIRTUAL_VS, cMo) ;

cout << "Pose" << cMo << endl ;
```

Listing 3.4: An example of task definition: positioning wrt. seven points.

Listing 3.5 shows how to build a 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 $\log Z$. In such case it is possible to use a generic `vpGenericFeature` feature. The user has then to define, 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 when dealing with `logZ` and `tu` the desired position is not defined. Since these features must be regulated to zero we do not have to specify their desired values (this is implicitly done). Line 23 specifies that the interaction matrix will be computed at the current position $\widehat{\mathbf{L_s}} = \widehat{\mathbf{L_s}}(\mathbf{s}, \mathbf{p})$. 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. 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: ${}^c\mathbf{p} = {}^c\mathbf{M}_o{}^o\mathbf{p}$). The interaction matrix related to $\log Z$ has to be computed according to equation (8). The global task interaction is then updated and the control law computed.

```
1   // define the features
2   vpFeaturePoint p(point[0]), pd(point[0]) ;  // 2D reference point
3   vpGenericFeature logZ(1) ;                  // log (Z/Z*)
4   vpFeatureThetaU tu ;                        // ThetaU
5
6   // build the features:
7   // - 1st and 2nd feature: 2D point and  log(Z/Z*)
8   double Zd,Z ;
9   vpColVector cP ; // point coordinates in camera frame
10  cP = cMo*point[0] ;     Z = cP[2] ;  //     current Z and p
11  p.buildFrom(point[0].get_x(), point[0].get_y(), Z) ;
12
13  cP = cdMo*point[0] ;   Zd = cP[2] ; // desired Zd and pd
14  pd.buildFrom(point[0].get_x(), point[0].get_y(), Zd) ;
15  logZ.set_s(log(Z/Zd)) ;
16
17  // - 3rd feature ThetaU
18  // compute the rotation that the camera has to achieve
19  vpHomogeneousMatrix cdMc ;
20  cdMc = cdMo*cMo.inverse() ;     tu.buildFrom(cdMc) ;
21
22  // interaction matrix computed at the current position
23  task.setInteractionMatrixType(vpServo::CURRENT) ;
24
25  // build the task (stack the features)
26  task.addFeature(p,pd) ;
27  task.addFeature(logZ) ;
28  task.addFeature(tu) ;   // s = (x, y, log Z, θu)^T
29
30  // control loop
31  while(1) {
32    g.acquire(I) ;
33    vpDisplay::display(I) ;
34
35    // compute the pose using  a non linear minimisation method
36    pose.clearPoint() ;
37    for (i=0 ; i < 7 ; i++) {
38      dot[i].track(I) ;
39      vpPixelMeterConversion::convertPoint(cam, dot[i], point[i])  ;
40      pose.addPoint(point[i]) ;
41    }
42    pose.computePose(vpPose::LOWE, cMo) ;
43
44    // compute the current Z
45    cP = cMo * point[0] ;   Z = cP[2] ;
```

```
46      p.buildFrom(point[0].get_x(), point[0].get_y(), Z) ;
47
48      // compute log (Z/Z*) and the corresponding interaction matrix
49      logZ.set_s(log(Z/Zd)) ;
50      vpMatrix LlogZ(1,6) ;
51      LlogZ[0][0] = LlogZ[0][1] = LlogZ[0][5] = 0 ;
52      LlogZ[0][2] = -1/Z ;
53      LlogZ[0][3] = -p.get_y() ;
54      LlogZ[0][4] =  p.get_x() ;
55      logZ.setInteractionMatrix(LlogZ) ;
56
57      cdMc = cdMo*cMo.inverse() ; // Compute the displacement to achieve
58      tu.buildFrom(cdMc) ;
59
60      v = task.computeControlLaw() ;
61      robot.setVelocity(vpRobot::CAMERA_FRAME, v) ;
62
63      vpDisplay::flush(I) ;
64    }
```

Listing 3.5: An example of task definition: positioning wrt. seven points.

In this example, to compute the rotation that the camera has to achieved we have considered a pose estimation process. Let us note that this can also be achieved by the estimation of an homography between the current and desired image (as explained in [6]). ViSP has also the capability to estimate a homography using various algoritms [3, 6] and to extract from this homography the camera displacement.

Bibliography

[1] 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.

[2] 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.

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

[4] W. Khalil and E. Dombre. *Modeling, identification and control of robots*. Hermes chPenton Sciences, London, 2002.

[5] 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.

[6] 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.

[7] P. Martinet, N. Daucher, J. Gallice, and M. Dhome. Robot control using monocular pose estimation. In *Workshop on new trends in image-based robot servoing, IROS'97*, pages 1–12, Grenoble, September 1997.

[8] P. Martinet, J. Gallice, and D. Khadraoui. Robot control using 3D visual features. In *Word Automation Congress, WAC'96*, volume 3, pages 497–502, Montpellier, May 1996.

[9] P. Rives and J.R. Azinheira. Linear structures following by an airship using vanishing point and horizon line in a visual servoing scheme. In *IEEE Int. Conf. on Robotics and Automation, ICRA'04*, volume 1, pages 255–260, New Orleans, USA, May 2004.

[10] C. Samson, M. Le Borgne, and B. Espiau. *Robot Control: the Task Function Approach*. Clarendon Press, Oxford, United Kingdom, 1991.

[11] F. Schramm, G. Morel, A. Micaelli, and A. Lottin. Extended 2d visual servoing. In *IEEE Int. Conf. on Robotics and Automation, ICRA'04*, pages 267–273, New Orleans, May 2004.

[12] O. Tahri and F. Chaumette. Point-based and region-based image moments for visual servoing of planar objects. *IEEE Trans. on Robotics*, 21(6):1116–1127, December 2005.

[13] 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.