ViSP 2.6.2: Visual Servoing Platform

Geometric transformations and objects

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

January 30, 2012

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

INVENTORS FOR THE DIGITAL WORLD

Contents

Chapter 1

Geometric transformations

1.1 3D transformations

1.1.1 Coordinate system

ViSP uses a right-handed coordinate system for 3D data. When considering the camera frame, the $\mathbf{z}$ axis is coming into the screen. All angles are in radians and all distances are in meters. All transformations considered here preserve the distances and the angles between the reference frames.

1.1.2 Rotation representation

Let us first note that the standard representation for rotation in ViSP is the rotation matrix. Nevertheless other representations exist such as three versions of the Euler angles, the $\theta\mathbf{u}$ representation (angle and axe of the rotation) and quaternions.

1.1.2.1 Rotation matrices

Let ${}^a\mathbf{R}_b$ be a rotation matrix from frame $\mathcal{F}_a$ to $\mathcal{F}_b$.

- $\mathbf{R}$ is a 3×3 orthonormal matrix. Hence $\mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I}$. This implies:
  1. $\mathbf{R}^{-1} = \mathbf{R}^T$
  2. $\det(\mathbf{R}) = 1$ (since the chosen coordinate frame is right handed)
  3. the lines $\mathbf{R}_{i\bullet}$ and columns $\mathbf{R}_{\bullet j}$ define an orthogonal basis.

The space of rotation matrices, $SO(3) \subset \mathbb{R}$, is called the special orthogonal group. Under the operation of matrix multiplication, it satisfies the axioms of closure, identity, inverse and associativity. It is defined by:

$$SO(3) = \{\mathbf{R} \in \mathbb{R}^{3\times 3} : \mathbf{R}\mathbf{R}^T = \mathbf{I}, \det(\mathbf{R}) = \mathbf{1}\} \quad (1.1)$$

Rotation matrix can be defined using the `vpRotationMatrix` *class. It inherits from* `vpMatrix` *but most of the methods have been overloaded for efficiency issues or to ensure the specific properties of the rotation matrix (for example the + or − operators are no longer defined).*

Construction. *The default value for a rotation matrix is identity. The rotation can be built either from another rotation matrix or from any other representation (see the next paragraphs):*

```
vpRotationMatrix R(Ri) ;     // where Ri is a vpRotationMatrix
vpRotationMatrix R(rzyx) ;   // where rzyx is a vpRzyxVector
vpRotationMatrix R(rzyz) ;   // where rzyz is a vpRzyzVector
vpRotationMatrix R(rxyz) ;   // where rxyz is a vpRxyzVector
vpRotationMatrix R(thetau) ; // where thetau is a vpThetaUVector
vpRotationMatrix R(q) ;      // where q is a vpQuaternionVector
```

It can also be built directly from three floats $(\theta\mathbf{u}_x, \theta\mathbf{u}_y, \theta\mathbf{u}_z)$ which correspond to the $\theta\mathbf{u}$ representation (see section 1.1.2.2):

```
vpRotationMatrix R(0.2,0.3,0.5) ; // (thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` *can be used:*

```
vpRotationMatrix R ; R.buildFrom(Ri) ;     // where Ri is a vpRotationMatrix
vpRotationMatrix R ; R.buildFrom(rzyx) ;   // where rzyx is a vpRzyxVector
vpRotationMatrix R ; R.buildFrom(rzyz) ;   // where rzyz is a vpRzyzVector
vpRotationMatrix R ; R.buildFrom(rxyz) ;   // where rxyz is a vpRxyzVvector
vpRotationMatrix R ; R.buildFrom(thetau) ; // where thetau is a vpThetaUVector
vpRotationMatrix R;  R.buildFrom(q) ;      // where q is a vpQuaternionVector

vpRotationMatrix R ; R.buildFrom(0.2,0.3,0.5) ; // (thetaux,thetauy,thetauz)
```

Operations. *For example if* `aRb` *and* `bRc` *define the rotation between frame $\mathcal{F}_a$ and $\mathcal{F}_b$ (respectively $\mathcal{F}_b$ and $\mathcal{F}_c$), computing transformation between $\mathcal{F}_a$ and $\mathcal{F}_c$ is implemented by:*

```
vpRotationMatrix aRb(...) ;
vpRotationMatrix bRc(...) ;

vpRotationMatrix aRc ;

aRc = aRb*bRc ;
```

In a similar way the inverse transformation can be obtained using

```
vpRotationMatrix cRa ;

cRa = (aRb*bRc).inverse() ;
cRa = (aRb*bRc).t() ; // (since R^-1 = R^T)
cRa = aRc.t() ;
```

Since a representation of a rotation as a rotation matrix is not minimal it is interesting to have other formulations of a rotation.

1.1.2.2 $\theta\mathbf{u}$ angle-axis representation

$\theta\mathbf{u}$ where $\mathbf{u} = (u_x, u_y, u_z)^\top$ is an unit vector representing the rotation axis and θ is the rotation angle. $\theta\mathbf{u}$ is one of the minimal representation.

It is possible to build the rotation matrix $\mathbf{R}$ from the $\theta\mathbf{u}$ representation using the Rodrigues formula:

$$\mathbf{R} = \mathbf{I}_3 + (1 - \cos\theta)\ \mathbf{u}\mathbf{u}^\top + \sin\theta\ [\mathbf{u}]_\times \tag{1.2}$$

with $\mathbf{I}_3$ the identity matrix of dimension 3×3 and $[\mathbf{u}]_\times$ the skew matrix :

$$[\mathbf{u}]_\times = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \tag{1.3}$$

On the other hand $\theta\mathbf{u}$ is obtained from $\mathbf{R}$ using:

$$\cos\theta = \frac{1}{2}\,(\ \text{trace}(\mathbf{R}) - 1) \tag{1.4}$$

and

$$\sin\theta\ [\mathbf{u}]_\times = \frac{1}{2}(\mathbf{R} - \mathbf{R}^T) \tag{1.5}$$

From 1.4 we get θ :

$$\theta = arccos\left(\frac{\mathbf{R}_{11} + \mathbf{R}_{22} + \mathbf{R}_{33} - 1}{2}\right) \tag{1.6}$$

Fixing $\theta > 0$, if $\sin\theta \neq 0$, $\mathbf{u}$ can be determined from 1.5 :

$$\mathbf{u} = \frac{1}{2\ sin(\theta)} \begin{bmatrix} \mathbf{R}_{32} - \mathbf{R}_{23} \\ \mathbf{R}_{13} - \mathbf{R}_{31} \\ \mathbf{R}_{21} - \mathbf{R}_{12} \end{bmatrix} \tag{1.7}$$

The $\theta\mathbf{u}$ representation is implemented in the `vpThetaUVector` *class. It is nothing but an array of 3 float values. As previously stated, the main rotation representation in ViSP is the rotation matrix. The only operators defined for the* `vpThetaUVector` *are the construction operators:*

Construction. *Default value for the* `vpThetaUVector` *is a null vector. The vector can be build either from another* `vpThetaUVector` *or from any other representation of a rotation:*

```
vpThetaUVector thetau(thetaui) ; // where thetaui is a vpThetaUVector
vpThetaUVector thetau(R) ;       // where R is a vpRotationMatrix
vpThetaUVector thetau(rzyx) ;    // where rzyx is a vpRzyxVector
vpThetaUVector thetau(rzyz) ;    // where rzyz is a vpRzyzVector
vpThetaUVector thetau(rxyz) ;    // where rxyz is a vpRxyzVector
```

Alternatively, the method `buildFrom(...)` *can be used:*

```
vpThetaUVector thetau ; thetau.buildFrom(R) ; // where R is a vpRotationMatrix
vpThetaUVector thetau ; thetau.buildFrom(rzyx) ; // where rzyx is a vpRzyxVector
vpThetaUVector thetau ; thetau.buildFrom(rzyz) ; // where rzyz is a vpRzyzVector
vpThetaUVector thetau ; thetau.buildFrom(rxyz) ; // where rxyz is a vpRxyzVector
```

It can also be built directly from three floats $(\theta\mathbf{u}_x, \theta\mathbf{u}_y, \theta\mathbf{u}_z)$ which correspond to the $\theta\mathbf{u}$ representation:

```
vpThetaUVector thetau(0.2,0.3,0.5) ; // thetaux,thetauy,thetauz
```

Building a rotation matrix `R` *from a $\theta\mathbf{u}$ vector is done through the constructor or the* `buildFrom(...)` *method of the* `vpRotationMatrix` *class.*

```
vpRotationMatrix R(thetau) ; // where thetau is a vpThetaUVector
vpRotationMatrix R ; R.buildFrom(thetau) ; // where thetau is a vpThetaUVector
```

Access. *To access directly to the $\theta\mathbf{u}$ angles, the* `[]` *operator is defined.*

```
vpThetaUVector thetau;
thetau[0] = 0.2 ; // thetaux;
thetau[1] = 0.3 ; // thetauy;
thetau[2] = 0.5 ; // thetauz;
```

1.1.2.3 Euler angles

In the Euler representation, the rotation is contained into 3 rotations around predefined axes (the rotation is then described by 3 angles (φ, θ, ψ)). This is also a minimal representation. ViSP implements three versions of the Euler angles.

Considering each elementary rotation matrix:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \mathbf{R}_z(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.8}$$

The overall rotation is the product of these elementary matrices:

$$\mathbf{R}_{zyx}(\varphi, \theta, \psi) = \mathbf{R}_z(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_x(\psi) \tag{1.9}$$

This representation is not unique. In ViSP, the Euler representations of $\mathbf{R}_{xyz}$and $\mathbf{R}_{zyz}$ are also implemented. In these cases:

$$\mathbf{R}_{xyz}(\varphi, \theta, \psi) = \mathbf{R}_x(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_z(\psi) \tag{1.10}$$

and

$$\mathbf{R}_{zyz}(\varphi, \theta, \psi) = \mathbf{R}_z(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_z(\psi) \tag{1.11}$$

The Euler angles representations are implemented in the `vpRzyxVector` *class for the $\mathbf{R}_{zyx}$ representation, the* `vpRxyzVector` *class for the $\mathbf{R}_{xyz}$ representation, and the* `vpRzyzVector` *class for the $\mathbf{R}_{zyz}$ representation.*

The only operators defined for the `vpRzyxVector`, `vpRxyzVector` *and* `vpRzyzVector` *are the construction operators. Each class is by default a null vector and can be initialised from any representation of a rotation (either directly in the constructor or by using the* `buildFrom(...)` *method).*

vpRzyxVector

```
vpRzyxVector rzyx(r) ; // where r is a vpRzyxVector
vpRzyxVector rzyx(R) ; // where R is a vpRotationMatrix
vpRzyxVector rzyx(thetau) ; // where thetau is a vpThetaUVector
```

```
vpRzyxVector rzyx ; rzyx.buildFrom(R) ; // where R is a vpRotationMatrix
vpRzyxVector rzyx ; rzyx.buildFrom(thetau) ; // where thetau is a vpThetaUVector
```

It can also be built directly from three floats which correspond to the Euler angles around z, y and x:

```
vpRzyxVector rzyx(0.2,0.3,0.5) ; // Euler angles around z, y and x
```

vpRzyzVector

```
vpRzyzVector rzyz(r) ; // where r is a vpRzyzVector
vpRzyzVector rzyz(R) ; // where R is a vpRotationMatrix
vpRzyzVector rzyz(thetau) ; // where thetau is a vpThetaUVector
```

```
vpRzyzVector rzyz ; rzyz.buildFrom(R) ; // where R is a vpRotationMatrix
vpRzyzVector rzyz ; rzyz.buildFrom(thetau) ; // where thetau is a vpThetaUVector
```

It can also be built directly from three floats which correspond to the Euler angles around z, y and z:

```
vpRzyzVector rzyz(0.2,0.3,0.5) ; // Euler angles around z, y and z
```

vpRxyzVector

```
vpRxyzVector rxyz(r) ; // where r is a vpRxyzVector
vpRxyzVector rxyz(R) ; // where R is a vpRotationMatrix
vpRxyzVector rxyz(thetau) ; // where thetau is a vpThetaUVector
```

```
vpRxyzVector rxyz ; rxyz.buildFrom(R) ; // where R is a vpRotationMatrix
vpRxyzVector rxyz ; rxyz.buildFrom(thetau) ; // where thetau is a vpThetaUVector
```

It can also be build directly from three floats which correspond to the Euler angles around x, y and z:

```
vpRxyzVector rxyz(0.2,0.3,0.5) ; // Euler angles around x, y and z
```

Rotation matrix *from an Euler angles vector. The initialisation is done through the constructor or the* `buildFrom(...)` *method of the* `vpRotationMatrix` *class.*

```
vpRotationMatrix R(rzyx) ; // where rzyx is a vpRzyxVector
vpRotationMatrix R(rxyz) ; // where rxyz is a vpRxyzVector
vpRotationMatrix R(rzyz) ; // where rzyz is a vpRzyzVector
// or
vpRotationMatrix R ; R.buildFrom(rzyx) ; // where rzyx is a vpRzyxVector
vpRotationMatrix R ; R.buildFrom(rxyz) ; // where rxyz is a vpRxyzVector
vpRotationMatrix R ; R.buildFrom(rzyz) ; // where rzyz is a vpRzyzVector
```

Access. *Read/write access of the angles is done using the* `[]` *operator:*

```
vpRzyxVector rzyx;
rzyx[0] = 0.2 ; rzyx[1] = 0.3 ; rzyx[2] = 0.5 ; // Euler angles around z, y and x;

vpRzyzVector rzyz;
rzyz[0] = 0.2 ; rzyz[1] = 0.3 ; rzyz[2] = 0.5 ; // Euler angles around z, y and z;

vpRzyxVector rxyz;
rxyz[0] = 0.2 ; rxyz[1] = 0.3 ; rxyz[2] = 0.5 ; // Euler angles around x, y and z;
```

1.1.2.4 Quaternions

Quaternions are another way of representing a rotation. Unlike euler angles, they avoid the gimbal lock issue and unlike rotation matrices, they can be stored only using 4 values. A quaternion is defined by four values: $\mathbf{q} = (x, y, z, w)$. Quaternions form a non-commutative associative algebra over the real numbers. Multiplying two quaternions is equivalent to composing the two rotations they represent. Let us consider quaternion multiplication defined as follows:

$\mathbf{q}_1 = (x_1, y_1, z_1, w_1)$ and $\mathbf{q}_2 = (x_2, y_2, z_2, w_2)$

$$\mathbf{q}_1\mathbf{q}_2 = \begin{bmatrix} w_1 \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} + w_2 \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} \\ w_1 w_2 - \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} \end{bmatrix}$$

A rotation matrix can be easily obtained from a quaternion $\mathbf{q} = (x, y, z, w)$ by the formula:

$$R = \begin{bmatrix} 1 - 2z^2 - 2w^2 & 2yz - 2wx & 2yw + 2zx \\ 2yz + 2wx & 1 - 2y^2 - 2w^2 & 2zw - 2yx \\ 2yw - 2zx & 2zw + 2yz & 1 - 2y^2 - 2z^2 \end{bmatrix} \tag{1.12}$$

The other way around, ViSP provides tools to convert a rotation matrix to $\theta\mathbf{u}$ angle-axis representation (see 1.1.2.2). Let's assume that the rotation is defined by the axis $\mathbf{v} = (x, y, z)$ and the angle θ.

A unit quaternion may be represented as follows: $q = cos(\frac{\theta}{2}) + \frac{\mathbf{v}}{||\mathbf{v}||} sin(\frac{\theta}{2})$. Therefore, the quaternion corresponding to the rotation can be obtained by:

$$\mathbf{q} = \begin{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \frac{sin(\frac{\theta}{2})}{\sqrt{x^2+y^2+z^2}} \\ cos(\frac{\theta}{2}) \end{bmatrix} \tag{1.13}$$

Implementation In ViSP quaternions are implemented as the `vpQuaternionVector` class. Like other rotation representation except vpRotationMatrix, it derives from a vpRotationVector. The quaternion can be built manually or from a rotation matrix. Constructing a quaternion from a rotation matrix first converts it to an angle-axis representation (see 1.1.2.2 for details) and then obtains the quaternion using 1.13.

```
vpQuaternionVector q_1(R);              // where R is a vpRotationMatrix
vpQuaternionVector q_2(0., 0., 0., 1.); // where x=0., y=0., z=0., w=1.
```

An empty constructor initializes a null quaternion $\mathbf{q}_0 = (0, 0, 0, 0)$. While quaternion values can be accessed by the `[...]` operator, `vpQuaternionVector` provides helper methods to make access to the quaternion's components more intuitive.

```
std::cout << q[0] << std::endl;  // prints x first component of the quaternion
std::cout << q.x() << std::endl; // same thing
```

Note that unlike the `[...]` operator, `x()`, `y()`, `z()`, `w()` methods are read-only. They return a value but cannot be used to change it.

Conversions Just like it can be constructed from a `vpRotationMatrix`, a `vpQuaternionVector` can be extracted from a `vpHomogeneousMatrix`. This quaternion will describe the rotation information contained in that matrix.

```
vpHomogeneousMatrix M(...);
vpQuaternionVector q;
M.extract(q); // q now describes the rotation contained in M
```

The other way around, a `vpQuaternionVector` can be converted to a `vpRotationMatrix` using 1.12.

```
vpQuaternionVector q;
vpRotationMatrix R(q); // R now describes the same rotation as the quaternion
```

Finally, a `vpHomogeneousMatrix` can be built from both a quaternion and a `vpTranslationVector` the following way:

```
vpHomogeneousMatrix M(vpTranslationVector(X,Y,Z), vpQuaternionVector(x,y,z,w));
```

1.1.3 Translation

Let ${}^{a}\mathbf{t}_b = (t_x, t_y, t_z)$ be a translation from frame $\mathcal{F}_a$ to $\mathcal{F}_b$. In ViSP, a translation is represented by a column vector of dimension 3.

Translation can be defined using the `vpTranslationVector` *class. It inherits from the* `vpColVector` *but most of the members have been overloaded for efficiency issue.*

Construction. *Default value for the translation vector is a null vector. The translation can be built from another translation vector:*

```
vpTranslationVector t(ti) ; // where ti is a vpTranslationVector
```

It can also be built directly from three floats which correspond to the 3 components of the translation along the $\mathbf{x}$,$\mathbf{y}$ *and* $\mathbf{z}$ *axes :*

```
vpTranslationVector t(0.2,0.3,0.5) ; // tx,ty and tz
```

Operations. *Let* `atb` *and* `btc` *define the translation between frame* $\mathcal{F}_a$ *and* $\mathcal{F}_b$ *(respectively* $\mathcal{F}_b$ *and* $\mathcal{F}_c$*), the transformation between* $\mathcal{F}_a$ *and* $\mathcal{F}_c$ *is implemented by:*

```
vpTranslationVector atb(...) ;
vpTranslationVector btc(...) ;

vpTranslationVector atc ;
atc = atb + btc ;
```

In a similar way the inverse transformation can be obtained using:

```
vpTranslationVector cta ;

cta = -(atb + btc) ;
```

To obtain the skew symmetric matrix or the cross product of two translation vectors, the following code can be use:

```
vpTranslationVector t1(...) ;
vpTranslationVector t2(...) ;

vpMatrix S(3,3) ; // initalize a 3x3 matrix
S = t1.skew() ;   // skew symmetric matrix

vpTranslationVector cross ;
cross = vpTranslationVector::cross(t1,t2) ; // cross product t1xt2
```

1.1.4 Pose

1.1.4.1 Pose vector

A pose is a complete representation of a rigid motion in the euclidean space. It is composed of a rotation and a translation and is minimally represented by 6 parameters:

$$^a\mathbf{r}_b = (^a\mathbf{t}_b, \theta\mathbf{u}) \in \mathbb{R}^6, \tag{1.14}$$

where $^a\mathbf{r}_b$ is the pose from $\mathcal{F}_a$ to $\mathcal{F}_b$, with $^a\mathbf{t}_b$ being the translation vector between $\mathcal{F}_a$ and $\mathcal{F}_b$ along the $\mathbf{x}$,$\mathbf{y}$,$\mathbf{z}$ axis and $\theta\mathbf{u}$, the $\theta\mathbf{u}$ representation of the rotation $^a\mathbf{R}_b$ between these frames.

The pose vector is implemented in the `vpPoseVector` *class. It is nothing but the concatenation of a* `vpTranslationVector` *and a* `vpThetaUVector`*. The only operators defined for the* `vpPoseVector` *are the construction operators:*

Construction. *Default value for the pose vector is a null vector. The pose can be built from any representation of a pose:*

```
vpPoseVector r(ri) ; // where ri is a vpPoseVector
vpPoseVector r(t,thetau) ; // where t is a vpTranslationVector
                           // and thetau is a vpThetaUVector
vpPoseVector r(t,R) ;  // where R is a vpRotationMatrix
vpPoseVector r(M) ; // where M is a vpHomogeneousMatrix
```

Alternatively, the operator `buildFrom(...)` *can be used:*

```
vpPoseVector r ; r.buildFrom(t,thetau) ; // where t is a vpTranslationVector
                                         // and thetau is a vpThetaUVector
vpPoseVector r ; r.buildFrom(t,R) ;  // where R is a vpRotationMatrix
vpPoseVector r ; r.buildFrom(M) ; // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to the three components of the translation along the $\mathbf{x}$,$\mathbf{y}$ *and* $\mathbf{z}$ *axes and the three components of the* $\theta\mathbf{u}$ *representation of the rotation:*

```
vpPoseVector r(1.0,1.3,3.5,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Building a homogeneous transformation matrix `M` *(see section 1.1.4.2) from a pose vector is done through the constructor or the* `buildFrom(...)` *operator of the* `vpHomogeneousMatrix` *class.*

```
vpHomogeneousMatrix M(r) ; // where r is a vpPoseVector
vpHomogeneousMatrix M ; M.buildFrom(r) ; // where r is a vpPoseVector
```

Access. *Read/write access of the components is done using the* `[]` *operator:*

```
vpPoseVector r;
r[0] = 1.0 ; r[1] = 1.3 ; r[2] = 3.5 ; // tx, ty, tz
r[3] = 0.2 ; r[4] = 0.3 ; r[5] = 0.5 ; // thetaux, thetauy, thetauz
```

1.1.4.2 Homogeneous transformation matrix

Coordinates relative to a frame $\mathcal{F}_a$ of a 3D point $\mathbf{P}$ relative to another frame $\mathcal{F}_b$ are obtained applying a rotation followed by a translation between the frames :

$$^a\mathbf{P} = {}^a\mathbf{R}_b{}^b\mathbf{P} + {}^a\mathbf{t}_b \tag{1.15}$$

The space representing all these transformations is call the Special Euclidean transformation space :

$$SE(3) = \{m = (\mathbf{R}, \mathbf{t}) : \mathbf{R} \in SO(3), \mathbf{t} \in \mathbb{R}^3\} \tag{1.16}$$

As in most of the robotics literature this transformation (or pose) is represented using a homogeneous transformation matrix given by:

$$^a\mathbf{M}_b = \left[\begin{array}{cc} ^a\mathbf{R}_b & ^a\mathbf{t}_b \\ \mathbf{0}_3 & 1 \end{array}\right] \tag{1.17}$$

where $^a\mathbf{R}_b$ and $^a\mathbf{t}_b$ are, the rotation matrix (see section 1.1.2.1) and translation vector (see section 1.1.3) from frame $\mathcal{F}_a$ to $\mathcal{F}_b$. It is an equivalent representation of the minimal pose vector $\mathbf{r}$ (see section 1.1.4).

The transformation of a homogeneous 3D point between a frame $\mathcal{F}_a$ and a frame $\mathcal{F}_b$ is expressed by:

$$\left[\begin{array}{c} ^a\mathbf{P} \\ 1 \end{array}\right] = {}^a\mathbf{M}_b \left[\begin{array}{c} ^b\mathbf{P} \\ 1 \end{array}\right] \tag{1.18}$$

Homogeneous matrices can be composed:

$$^a\mathbf{M}_c = {}^a\mathbf{M}_b{}^b\mathbf{M}_c, \tag{1.19}$$

and inverted :

$$^a\mathbf{M}_b^{-1} = {}^b\mathbf{M}_a = \left[\begin{array}{cc} ^a\mathbf{R}_b^T & -{}^a\mathbf{R}_b^{T\,a}\mathbf{t}_b \\ \mathbf{0}_3 & 1 \end{array}\right] \tag{1.20}$$

The `vpHomogeneousMatrix` *class implements a homogeneous matrix. It inherits from the* `vpMatrix` *but most of the members have been overloaded for efficiency issue or to ensure the rotation matrix properties (For example the + or − operators are no longer defined).*

Construction. *Default value for the homogeneous matrix is the identity matrix. The homogeneous matrix can be build either from another homogeneous matrix or from another representation of a pose:*

```
vpHomogeneousMatrix M(Mi) ;         // where Mi is a vpHomogeneousMatrix
vpHomogeneousMatrix M(t, R) ;       // where t is a vpTranslationVector and R a vpRotationMatrix
vpHomogeneousMatrix M(t, thetau) ; // where t is a vpTranslationVector and thetau a vpThetaUVector
vpHomogeneousMatrix M(t, q) ;       // where t is a vpTranslationVector and q a vpQuaternionVector
vpHomogeneousMatrix M(r) ;          // where r is a vpPoseVector
```

It can also be built directly from six float which correspond to the pose representation (see section 1.1.4):

```
vpHomogeneousMatrix M(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` *can be used:*

```
vpHomogeneousMatrix M ; M.buildFrom(t, R) ;      // Where t is a vpTranslationVector
                                                 // and R a vpRotationMatrix
vpHomogeneousMatrix M ; M.buildFrom(t, thetau) ; // Where t is a vpTranslationVector
                                                 // and thetau a vpThetaUVector
vpHomogeneousMatrix M ; M.buildFrom(t, q) ;      // Where t is a vpTranslationVector
                                                 // and q a vpQuaternionVector
vpHomogeneousMatrix M ; M.buildFrom(r) ;         // Where r is a vpPoseVector

vpHomogeneousMatrix M ; M.buildFrom(1.2, 2.3, 1.0, 0.2, 0.3, 0.5) ; // (tx,ty,tz,...
                                                                    //  thetaux,thetauy,thetauz)
```

The method `insert(...)` *can be used to insert the rotation and the translation components independently:*

```
vpHomogeneousMatrix M ; M.insert(R) ;      // where R is a vpRotationMatrix
vpHomogeneousMatrix M ; M.insert(thetau) ; // where thetau is a vpThetaUVector
vpHomogeneousMatrix M ; M.insert(t) ;      // and t a vpTranslationVector
```

Extracting the rotation matrix `R` *and the translation vector* `t` *from an homogeneous matrix is done using the* `extract(...)` *method.*

```
vpRotationMatrix R ; M.extract(R) ; // where M is a vpHomogeneousMatrix
vpTranslationVector t ; M.extract(t) ;
```

Operations. *Let* `aMb` *and* `bMc` *define the poses between frame $\mathcal{F}_a$ and $\mathcal{F}_b$ (respectively $\mathcal{F}_b$ and $\mathcal{F}_c$), the pose between $\mathcal{F}_a$ and $\mathcal{F}_c$ is computed by:*

```
vpHomogeneousMatrix aMb(...) ;
vpHomogeneousMatrix bMc(...) ;

vpHomogeneousMatrix aMc ;

aMc = aMb*bMc ;
```

In a similar way the inverse transformation can be obtained using:

```
vpHomogeneousMatrix cMa ;

cMa = (aMb*bMc).inverse() ; // or
cMa = aMc.inverse() ;
```

1.1.5 Twist tranformation matrices

1.1.5.1 Velocity twist matrix

In a frame $\mathcal{F}_a$, the motion of an object is described by a translational velocity $\boldsymbol{v} = (v_x, v_y, v_z)$ and an angular velocity $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$. The resulting vector ${}^a\mathbf{v} = (\boldsymbol{v}, \boldsymbol{\omega})$ is known in the robotics literature as a velocity screw in the frame $\mathcal{F}_a$.

The velocity twist transformation matrix allows us to transform velocity screws among coordinate frames. Knowing the velocity screw in the frame $\mathcal{F}_b$, the corresponding velocity screw in frame $\mathcal{F}_a$ is obtained using:

$$ {}^a\mathbf{v} = {}^a\mathbf{V}_b {}^b\mathbf{v} \tag{1.21}$$

with

$$ {}^a\mathbf{V}_b = \left[\begin{array}{cc} {}^a\mathbf{R}_b & [{}^a\mathbf{t}_b]_\times {}^a\mathbf{R}_b \\ \mathbf{0}_3 & {}^a\mathbf{R}_b \end{array} \right] \tag{1.22}$$

where ${}^a\mathbf{R}_b$ and ${}^a\mathbf{t}_b$ are, respectively, the rotation matrix and translation vector from frame $\mathcal{F}_a$ to $\mathcal{F}_b$. ${}^a\mathbf{V}_b$ is called the velocity twist matrix.

Velocity twist matrices are implemented in the `vpVelocityTwistMatrix` *class. It inherits from the* `vpMatrix` *but most of the members have been overloaded for efficiency issue or to ensure the specific properties (for example the + or − operators are no longer defined).*

Construction. *Default value for the velocity twist matrix is the identity matrix. The velocity twist matrix can be build either from another velocity twist matrix, from a translation and the representation of a rotation, or from an homogeneous matrix:*

```
vpVelocityTwistMatrix V(Vi) ;       // where Vi is a vpVelocityTwistMatrix
vpVelocityTwistMatrix V(t,R) ;      // where t is a vpTranslationVector
                                    // and R a vpRotationMatrix
vpVelocityTwistMatrix V(t,thetau) ; // where thetau is a vpThetaUVector
vpVelocityTwistMatrix V(M) ;        // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to a translation and the $\theta\mathbf{u}$ representation (see section 1.1.2.2):

```
vpVelocityTwistMatrix V(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` *can be used:*

```
vpVelocityTwistMatrix V ; V.buildFrom(t,R) ;      // where R is a vpRotationMatrix
                                                  // and t a vpTranslationVector
vpVelocityTwistMatrix V ; V.buildFrom(t,thetau) ; // where thetau is a vpThetaUVector
vpVelocityTwistMatrix V ; V.buildFrom(M) ;        // where M is a vpHomogeneousMatrix
```

Operations. *Let* `aVb` *and* `bVc` *define the velocity twist matrices between frame $\mathcal{F}_a$ and $\mathcal{F}_b$ (respectively $\mathcal{F}_b$ and $\mathcal{F}_c$), the velocity twist matrix between $\mathcal{F}_a$ and $\mathcal{F}_c$ is computed by:*

```
vpVelocityTwistMatrix aVb(...) ;
vpVelocityTwistMatrix bVc(...) ;

vpVelocityTwistMatrix aVc ;

aVc = aVb*bVc ;
```

There is no specific object in ViSP to represent velocity screws. Conversion of a velocity screw ${}^b\mathbf{v} = (\boldsymbol{v}, \boldsymbol{\omega})$ expressed in frame $\mathcal{F}_b$ to a frame $\mathcal{F}_a$ can be done like:

```
vpVelocityTwistMatrix aVb(...) ;
vpColVector av = aVb*bv ;        // where bv is a vpColVector of dimension 6
                                 // av = (vx,vy,vz,wx,wy,wz)^T
```

1.1.5.2 Force-torque twist matrix

Related to a frame $\mathcal{F}_a$, the force applied to an object is described by a force vector $\mathbf{f} = (\mathbf{f_x}, \mathbf{f_y}, \mathbf{f_z})$ and a torque vector $\tau = (\tau_\mathbf{x}, \tau_\mathbf{y}, \tau_\mathbf{z})$. The resulting vector ${}^a\mathbf{h} = (\mathbf{f}, \tau)$ is known in the robotics literature as a force-torque screw in the frame $\mathcal{F}_a$.

The force-torque twist transformation matrix allows us to transform force-torque screws among coordinate frames. Knowing the force-torque screw in the frame $\mathcal{F}_b$, the corresponding force-torque screw in frame $\mathcal{F}_a$ is obtained using:

$$^a\mathbf{h} = {}^\mathbf{a}\mathbf{F}_\mathbf{b}{}^\mathbf{b}\mathbf{h} \tag{1.23}$$

with

$$^a\mathbf{F}_b = \begin{bmatrix} {}^a\mathbf{R}_b & \mathbf{0}_3 \\ [{}^a\mathbf{t}_b]_\times {}^a\mathbf{R}_b & {}^a\mathbf{R}_b \end{bmatrix} \tag{1.24}$$

where ${}^a\mathbf{R}_b$ and ${}^a\mathbf{t}_b$ are, the rotation matrix and the translation vector from frame $\mathcal{F}_a$ to $\mathcal{F}_b$. ${}^a\mathbf{F}_b$ is called the force-torque twist matrix.

Force-torque twist matrix is implemented in the `vpForceTwistMatrix` *class. It inherits from the* `vpMatrix` *but most of the members have been overloaded for efficiency issue or to ensure the specific properties (for example the + or − operators are no longer defined).*

Construction. *Default value for the force-torque twist matrix is the identity matrix. The force-torque twist matrix can be built either from another force-torque twist matrix, from a translation and a rotation, or from a homogeneous matrix:*

```
vpForceTwistMatrix F(Fi) ;        // where Fi is a vpForceTwistMatrix
vpForceTwistMatrix F(t,R) ;       // where t is a vpTranslationVector
                                  // and R a vpRotationMatrix
vpForceTwistMatrix F(t,thetau) ; // where thetau is a vpThetaUVector
vpForceTwistMatrix F(M) ;         // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to a translation and a rotation in the $\theta\mathbf{u}$ representation (see section 1.1.2.2):

```
vpForceTwistMatrix F(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` *can be used:*

```
vpForceTwistMatrix F ; F.buildFrom(t,R) ;      // where R is a vpRotationMatrix
                                               // and t a vpTranslationVector
vpForceTwistMatrix F ; F.buildFrom(t,thetau) ; // where thetau is a vpThetaUVector
vpForceTwistMatrix F ; F.buildFrom(M) ;        // where M is a vpHomogeneousMatrix
```

Operations. *Let* `aFb` *and* `bFc` *be the force-torque twist matrices between frame $\mathcal{F}_a$ and $\mathcal{F}_b$ (respectively $\mathcal{F}_b$ and $\mathcal{F}_c$), the force-torque twist matrix between $\mathcal{F}_a$ and $\mathcal{F}_c$ is computed by:*

```
vpForceTwistMatrix aFb(...) ;
vpForceTwistMatrix bFc(...) ;

vpForceTwistMatrix aFc ;

aFc = aFb*bFc ;
```

There is no specific object in ViSP to represent force-torque screws. Conversion of a force-torque screw ${}^b\mathbf{h} = (\mathbf{f}, \tau)$ expressed in frame $\mathcal{F}_b$ to a frame $\mathcal{F}_a$ can be done like:

```
vpForceTwistMatrix aFb(...) ;
vpColVector ah = aFb*bh ;         // where bh is a vpColVector of dimension 6
                                  // ah = (fx,fy,fz,taux,tauy,tauz)^T
```

1.2 Vision related transformations

1.2.1 Camera models

1.2.1.1 Perspective camera

The perspective model, or pin-hole model, is the most used camera model in computer vision. The camera is denoted $\mathcal{F}_C = (C, x, y, z)$. The origin C of $\mathcal{F}_C$ is the projection center. The projection plane or (normalized) image plane π is parallel to the (x, y) plane. The intersection of the optical axis (z axis) with the image place is called the principal point and is denoted $\mathbf{x_c}$.

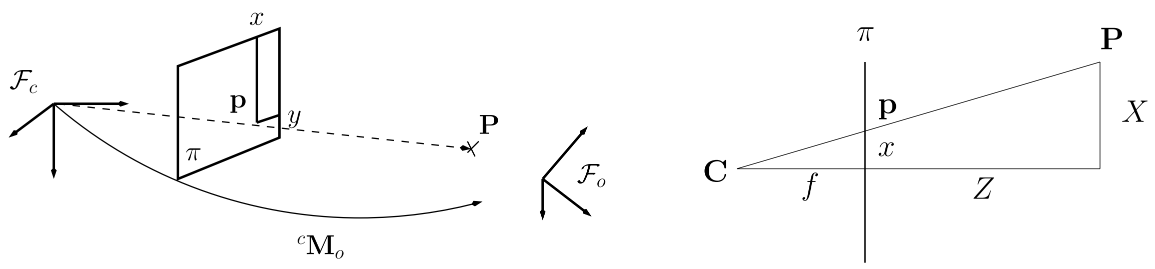


Figure 1.1: Perspective projection model.

Given a 3D point $\mathbf{P} = (X, Y, Z, 1)$ in $\mathcal{F}_C$, the perspective projection of $\mathbf{P}$ in the image plane $\mathbf{p} = (x, y, 1)$ is given by:

$$\mathbf{p} = \mathbf{MP}$$

or

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

It finally gives:

$$x = \frac{X}{Z}, \; y = \frac{Y}{Z} \tag{1.25}$$

Remark: *The equation (1.25) is only valid for points. ViSP allows the projection of various features (point, straight line, circle, sphere, ...).*

Given a 3D point in the object frame $\mathcal{F}_o$ and the camera pose ${}^c\mathbf{M}_o$, it is possible to compute the homogeneous coordinates of the point projected in the image plane:

```
vpHomogeneousMatrix cMo ; cMo[2][3] = 2 ; // the camera pose

vpPoint P ;
// set the point coordinates in the object frame
P.setWorldCoordinates(1,1,0) ;
cout << P.oP << endl ; // output (1,1,0)

// compute the point coordinates in the camera frame
P.changeFrame(cMo) ;
cout << P.cP << endl ; // output (1,1,2)
```

```

// project the 3D point in the 2D image plane
P.project() ;
cout << P.p << endl ; // output (0.5, 0.5)
```

$\mathbf{p} = (x, y, 1)$ coordinates are expressed in meters. To compute the point coordinates in pixel in the image frame $\mathbf{m} = (u, v)$, we have to consider the camera intrinsic parameters. This transformation is given by the following equations :

$$\mathbf{m} = \mathbf{K}\mathbf{p} \text{ with } \mathbf{K} = \begin{bmatrix} p_x & 0 & u_0 \\ 0 & p_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{K}$ is a matrix of intrinsic parameters, $\mathbf{x_c} = (u_0, v_0)$ are the coordinates of the principal point and (p_x, p_y) are the ratio between the focal length and the size of a pixel.

We obtain:

$$\begin{cases} u = p_x\ x + u_0 \\ v = p_y\ y + v_0 \end{cases} \tag{1.26}$$

The previous equation assume a perfect perspective camera which is not always the case. In practice we have to consider more complex (and usually non linear) camera model. In ViSP it is possible to introduce parameters to models the radial distortion. The model becomes:

$$\begin{cases} u = u_0 + p_x\ \ x\ \ \left(1 + k_{ud}\left(x^2 + y^2\right)\right) \\ \\ v = v_0 + p_y\ \ y\ \ \left(1 + k_{ud}\left(x^2 + y^2\right)\right) \end{cases} \tag{1.27}$$

where k_{ud} is a distortion parameter.

If, using this model, the computation of coordinates in pixels from coordinates expressed in meter is straightforward, the inverse transformation is far more complex. To cope with the issue we introduce a second distortion parameter k_{du} ($k_{ud} \neq k_{du}$) which allows to compute the coordinates in meter from the coordinates expressed in pixels:

$$\begin{cases} x = \frac{u-u_0}{p_x}\left(1 + k_{du}\left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right)\right) \\ \\ y = \frac{v-v_0}{p_y}\left(1 + k_{du}\left(\left(\frac{u-u_0}{p_x}\right)^2 + \left(\frac{v-v_0}{p_y}\right)^2\right)\right) \end{cases} \tag{1.28}$$

k_{ud} is used to convert a feature from an undistorted to a distorted representation whereas k_{du} is used to convert a feature from a distorted to an undistorted representation.

Camera parameters implementation *In ViSP, the camera intrinsic parameters are stored in the* `vpCameraParameters` *class.*

The selection of the modelisation is done during the vpCameraParameters initialisation.

For example, to initialise a camera without distorsion:

```
#include <visp/vpCameraParameters.h>

double px = 600;
double py = 600;
double u0 = 320;
double v0 = 240;
```

```

// Create a camera parameter container
vpCameraParameters cam;

// Camera initialization with a perspective projection without distortion model
cam.initPersProjWithoutDistortion(px,py,u0,v0);

// It is also possible to print the current camera parameters
cam.printParameters();
```

The same initialisation (for a model without distortion) can be done by:

```
#include <visp/vpCameraParameters.h>
#include <visp/vpMatrix.h>

// Create a matrix container for the camera parameters.
vpMatrix M(3, 3);

// Fill the matrix with the camera parameters
M[0][0] = 600; // px
M[1][1] = 600; // py
M[0][2] = 320; // u0
M[1][2] = 240; // v0
M[2][2] = 1;

// without distortion model is set.
vpCameraParameters cam;

// Set the parameters
cam.initFromCalibrationMatrix(M);
```

Initialisation of a camera for a model with distortion:

```
#include <visp/vpCameraParameters.h>

double px = 600;
double py = 600;
double u0 = 320;
double v0 = 240;
double kud = -0.19;
double kdu = 0.20;

// Create a camera parameter container
vpCameraParameters cam;

// Camera initialization with a perspective projection with distortion model
cam.initPersProjWithDistortion(px, py, u0, v0, kud, kdu);
```

The code below shows how to know if the current projection model uses distorsion parameters:

```
vpCameraParameters cam;
...
vpCameraParameters::vpCameraParametersProjType projModel;
projModel = cam.get_projModel(); // Get the projection model type
```

Camera parameters parser *ViSP allows to save and load camera parameters in a formated file using a XML parser.*

Example of an XML file "myXmlFile.xml" containing intrinsic camera parameters:

```
<?xml version="1.0"?>
<root>
  <camera>
```

```
    <name>myCamera</name>
    <image_width>640</image_width>
    <image_height>480</image_height>
    <model>
      <type>perspectiveProjWithoutDistortion</type>
      <px>1129.0</px>
      <py>1130.6</py>
      <u0>317.9</u0>
      <v0>229.1</v0>
    </model>
    <model>
      <type>perspectiveProjWithDistortion</type>
      <px>1089.9</px>
      <py>1090.1</py>
      <u0>326.1</u0>
      <v0>230.5</v0>
      <kud>-0.196</kud>
      <kdu>0.204</kdu>
    </model>
  </camera>
</root>
```

The following code shows how to load the camera parameters from a XML file:

```
#include <visp/vpCameraParameters.h>
#include <visp/vpXmlParserCamera.h>

vpCameraParameters cam; // Create a camera parameter container
vpXmlParserCamera p; // Create a XML parser
vpCameraParameters::vpCameraParametersProjType projModel; // Projection model

// We want here to use a perspective projection model without distorsion
projModel = vpCameraParameters::perspectiveProjWithoutDistortion;

// Parse the xml file "myXmlFile.xml" to find the intrinsic camera
// parameters of the camera named "myCamera" for 640x480 images and for
// the specified projection model. Note that the size of the image is optional
// if in the XML file camera parameters are provided only for one image size.
p.parse(cam, "myXmlFile.xml", "myCamera", projModel, 640, 480);

// Print the parameters
std::cout << "Camera Parameters: " << cam << std::endl;

// Save the parameters in a new file "myXmlFileWithNoise.xml"
p.save(cam,"myXmlFileWithNoise.xml", p.getCameraName(), p.getWidth(), p.getHeight());
```

The following code shows how to write the camera parameters in a XML file:

```
// Create a camera parameter container. We want to set these parameters
// for a 320x240 image, and we want to use the perspective projection
// modelisation without distortion.
vpCameraParameters cam;

// Set the pixel ratio (px, py) and the principal point coordinates (u0,v0)
cam.initPersProjWithDistortiont(563.2, 564.1, 162.3, 122.4); // px,py,u0,v0
// Create a XML parser
vpXmlParserCamera p;
// Save the camera parameters in an XML file.
p.save(cam, "myXmlFile.xml", "myNewCamera", 320, 240);
```

Pixel/Meter conversion. *The equations (1.26), (1.27) and (1.28) allows the pixel to meter (respectively meter to pixel) conversion for points. These conversions are implemented in the* `vpPixelMeterConversion` *class (respectively the* `vpMeterPixelConversion` *class).*

To use meter to pixel and pixel to meter conversion, we have first to include the corresponding header files:

```
#include <visp/vpPixelMeterConversion.h>
#include <visp/vpMeterPixelConversion.h>
```

Then we have to initialise a `vpCameraParameters` *as described above with the desired projection model. Here we choose the model without distortion.*

```
#include <visp/vpCameraParameters.h>
vpCameraParameters cam;
cam.initPersProjWithoutDistortion(px,py,u0,v0); // where px,py are the pixel ratio
                                                // and u0,v0 the principal point coordinates
```

We can finally call the desired conversion function, here we want to convert a point:

```
vpPixelMeterConversion::convertPoint(cam,u,v,x,y); // where (u,v) are pixel coordinates
                                                   // to convert in meter coordinates (x,y)

vpMeterPixelConversion::convertPoint(cam,x,y,u,v); // where (x,y) are meter coordinates
                                                   // to convert in pixel coordinates (u,v)
```

ViSP proposes similar conversions for straight lines and moments. But it can be noticed these conversions have no meaning for the projection model with distortion.

Remark: *ViSP implements a calibration tools that allows to compute the camera intrinsic parameters.*

1.2.1.2 Catadioptric sensor

To date this projection model is not considered in ViSP.

1.2.2 Homography

The homography has been introduced in [3]. It is a $\mathbb{R}^2$ to $\mathbb{R}^2$ transformation and is valid for all camera motions.

The main idea is to use the 3D coordinates of a plane $\mathcal{P}$ which is linked to the 3D point $\mathbf{P}$. At first, the point is considered to belong to the plane. In the section 1.2.2.1, the point is considered to belong the plane; while in the section 1.2.2.2 this constraint is removed.

1.2.2.1 Planar structure

We define ${}^a\mathbf{n}$ and d_a the normal to the plane $\mathcal{P}$ and the distance to the projection center $\mathbf{C}a$ as illustrated on figure 1.2

If the point $\mathbf{P}$ belongs to this plane, then its coordinates ${}^a\mathbf{P} = (X_a, Y_a, Z_a)$ in the frame $\mathcal{F}_a$ verify :

$$ {}^a\mathbf{n}^{T\,a}\mathbf{P} = d_a \tag{1.29} $$

With this relation, equation (1.15) can be factorized by ${}^a\mathbf{P}$

$$ {}^b\mathbf{P} = \left({}^b\mathbf{R}_a + \frac{{}^b\mathbf{t}_a\ {}^a\mathbf{n}^T}{d_a} \right)\ {}^a\mathbf{P} \tag{1.30} $$

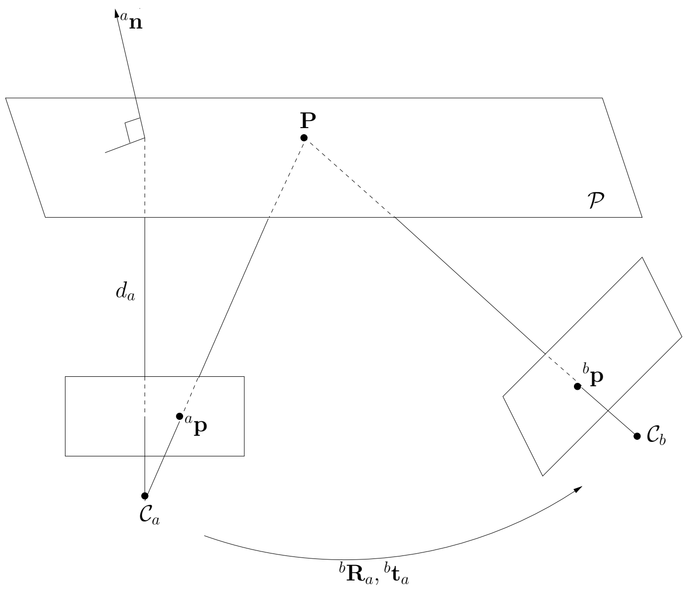


Figure 1.2: Illustration of the motion from one view to another using an homography in the case of a planar structure.

The projection equation (1.25) giving $Z_a\,{}^a\mathbf{p} = {}^a\mathbf{P}$, it's now possible to link ${}^a\mathbf{p}$ and ${}^b\mathbf{p}$ coordinates by an homography :

$$\begin{aligned} Z_b\,{}^b\mathbf{p} &= \left({}^b\mathbf{R}_a + \frac{{}^b\mathbf{t}_a\,{}^a\mathbf{n}^\top}{d_a}\right) Z_a\,{}^a\mathbf{p} \\ \frac{Z_b}{Z_a}\,{}^b\mathbf{p} &= \left({}^b\mathbf{R}_a + \frac{{}^b\mathbf{t}_a\,{}^a\mathbf{n}^\top}{d_a}\right) {}^a\mathbf{p} \\ \frac{Z_b}{Z_a}\,{}^b\mathbf{p} &= {}^b\mathbf{H}_a\,{}^a\mathbf{p} \qquad (1.31) \\ & \qquad (1.32) \end{aligned}$$

The homography is the 3×3 matrix :

$$ {}^b\mathbf{H}_a = {}^b\mathbf{R}_a + \frac{{}^b\mathbf{t}_a\,{}^a\mathbf{n}^\top}{d_a} \qquad (1.33)$$

which allows to make a point-to-point correspondence thanks to the introduction of the plane equation. It is an homogeneous transformation defined up to a scale factor. It has eight degrees of freedom.

1.2.2.2 Non planar structure

If the point doesn't belong to $\mathcal{P}$ (figure 1.3), the homography ${}^b\mathbf{H}_a$ related to the plane is not sufficient to describe the relation between ${}^a\mathbf{p}$ and ${}^b\mathbf{p}$. As it will be seen, the relation (1.31) will be improved by a term taking into account the signed distance d between the point and the plane. Noting :

$$d = d_a - {}^a\mathbf{n}^\top (Z_a\,{}^a\mathbf{p}) \qquad (1.34)$$

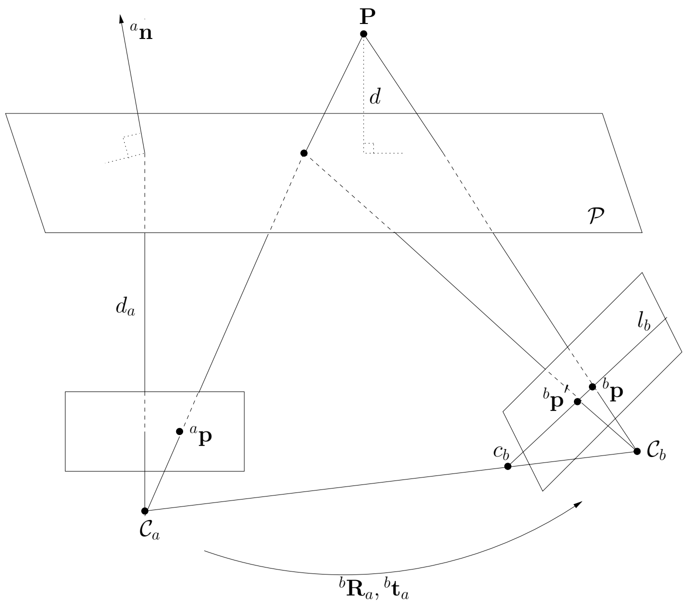


Figure 1.3: Illustration of the motion from a view to another using an homography in the case of a non planar structure.

In that case, the operation realised in equation (1.30) can't be simpified and gives :

$$ {}^b\mathbf{P} = \left({}^b\mathbf{R}_a + \frac{ {}^b\mathbf{t}_a \, {}^a\mathbf{n}^\top}{d_a} \right) {}^a\mathbf{P} + \left(1 - \frac{ {}^a\mathbf{n}^\top}{d_a} \, {}^a\mathbf{P} \right) {}^b\mathbf{t}_a \tag{1.35} $$

Or using ${}^a\mathbf{p}$ and ${}^b\mathbf{p}$ coordinates :

$$ Z_b \, {}^b\mathbf{p} = Z_a \, {}^b\mathbf{H}_a \, {}^a\mathbf{p} + \left(1 - Z_a \, \frac{ {}^a\mathbf{n}^\top}{d_a} \, {}^a\mathbf{p} \right) {}^b\mathbf{t}_a \tag{1.36} $$

Dividing equation members by Z_a, the equivalent relation of (1.31) in a non-planar structure is :

$$ \begin{aligned} \frac{Z_b}{Z_a} \, {}^b\mathbf{p} &= {}^b\mathbf{H}_a \, {}^a\mathbf{p} + \frac{d}{Z_a \, d_a} \, {}^b\mathbf{t}_a \\ \frac{Z_b}{Z_a} \, {}^b\mathbf{p} &= {}^b\mathbf{H}_a \, {}^a\mathbf{p} + \beta \, {}^b\mathbf{t}_a \end{aligned} \tag{1.37} $$

The term $\beta = \frac{d}{Z_a \, d_a}$ is called parallax [1, 5]. This term only depends on the parameters expressed in the camera frame $\mathcal{F}_a$. It measures a relative depth between the point $\mathbf{P}$ and the plane $\mathcal{P}$. Its sign allows to know on which side of the plane is the point. β allows to find the corresponding point of ${}^a\mathbf{p}$ in the image I_b along the epipolar line l_b of the projected point ${}^b\mathbf{p}' = {}^b\mathbf{H}_a \, {}^a\mathbf{p}$ (which can be seen as the corresponding point of ${}^a\mathbf{p}$ considering that $\mathbf{P}$ belong to the plane (figure 1.3)).

Chapter 2

Geometric objects

In this chapter, we present specific trackers that can be used for simulation. For a given camera location, a 3D virtual object is projected in the image plane providing 2D measurements.

2.1 Forward projection of some geometric objects

We have defined various geometric features in ViSP: point, straight line, circle, sphere, cylindre. It allows the simulated forward projection of these object in the image plane given a camera pose. From a formal point of view, these objects are, nothing but:

- a vector $\mathbf{p}$ that defines the 2D position of the object in the image,
- a vector ${}^c\mathbf{P}$ that defines the 3D position of the object in the camera frame $\mathcal{R}_C$,
- a vector ${}^o\mathbf{P}$ that defines the 3D position of the object in the world frame $\mathcal{R}_o$.

In ViSP these geometric features are implemented through a pure virtual class named `vpForwardProjection`*. This class inherits from the* `vpTracker` *class and then of the two vectors* `p` *and* `cP` *introduced in the trackers overview (see part* **??***).*

```
class  vpForwardProjection : public vpTracker
{
  public:
  //! feature coordinates  expressed in world frame
  vpColVector oP ;
  ...
}
```

A classical use of these geometric features is first to define the parameters ${}^o\mathbf{P}$ in the world frame then to express these parameters in the camera frame (i.e.compute ${}^c\mathbf{P}$ knowing the pose ${}^c\mathbf{M}_o$) and finally to project the object into the image plane getting the parameters $\mathbf{p}$.

We thus add some members functions to the class

```
class  vpForwardProjection : public vpTracker
{
  ...
  virtual void setWorldCoordinates(const vpColVector &oP)=0;
  virtual void changeFrame(const vpHomogeneousMatrix &cMo, vpColVector &cP)=0;
  virtual void projection(const vpColVector &cP, vpColVector &p)=0 ;
  ...
}
```

The three former member functions `setWorldCoordinates(...)`, `changeFrame(...)` *and* `projection(...)` *are pure virtual functions meaning that they MUST be redefined in each tracker that inherits from the* `vpForwardProjection` *class. They define the equation that allows to set the vector parameters* `oP` *(feature parameters expressed in the world frame), compute the vector parameters* `cP` *(feature parameters expressed in the camera frame) using the camera location. and compute the projection* `p` *of the feature in the image.*

Warning: the attributes `cP` *and* `p` *are not modified by the functions* `changeFrame(...)` *and* `projection(...)`*. To modify* `cP` *and* `p`*, we define the following functions:*

```
class  vpForwardProjection : public vpTracker
{
  ...
  void changeFrame(const vpHomogeneousMatrix &cMo) ; // compute cP=cMo.oP
  void project() ;                                   // compute p from the projection of cP
  ...
}
```

The behavior of these former functions is similar to previous functions `changeframe(cMo,cP)` *and* `projection(cP,p)`*) although they DO modify the internal attributes* `cP` *and* `p`*.*

To simplify the usage of this class, we also provide two functions:

```
class  vpForwardProjection : public vpTracker
{
  ...
  void project(const vpHomogeneousMatrix &cMo) ;
  void track(const vpHomogeneousMatrix &cMo) ;
  ...
}
```

The `project(cMo)` *and* `track(cMo)` *call the two former functions in sequence providing directly the vector* `p`*.*

ViSP implements also member functions to draw the projected object in the overlay of a displayed image.

```
class  vpForwardProjection : public vpTracker
{
  ...
  virtual void display(vpImage<unsigned char> &I,
                       const vpCameraParameters &cam,
                       const vpColor::vpColorType color=vpColor::green)=0;
  virtual void display(vpImage<unsigned char> &I,
                       const vpHomogeneousMatrix &cMo,
                       const vpCameraParameters &cam,
                       const vpColor::vpColorType color=vpColor::green)=0;
  ...
}
```

The first display member function draws the representation of the projected object in the overlay of image `I` *according to the attributes* `p` *using the camera parameters* `cam`*. The second display member function do the same but according the attributes* `oP` *knowing the pose* `cMo`*. These member functions are pure virtual functions meaning that they MUST be redefined in each tracker that inherits from* `vpForwardProjection` *class.*

Since `vpForwardProjection` *features pure virtual member functions it cannot be use as this. We give in the next sections some examples of how to use it for a various set of geometric features.*

2.2 Point

Given a 3D point ${}^o\mathbf{P} = ({}^oX, {}^oY, {}^oZ, 1)$ in $\mathcal{F}_o$.

Frame transformation. The position of the same point ${}^c\mathbf{P} = ({}^cX, {}^cY, {}^cZ, 1)$ expressed in $\mathcal{F}_c$ is given by:

$$ {}^c\mathbf{P} = {}^c\mathbf{M}_o {}^o\mathbf{P} \tag{2.1} $$

where ${}^c\mathbf{M}_o$ is the homogeneous matrix representing the pose from the camera frame to the world frame.

Projection. The perspective projection $\mathbf{p} = (x, y, 1)$ of ${}^c\mathbf{P}$ in the image plane is then given by:

$$ \mathbf{p} = \mathbf{M}{}^c\mathbf{P} $$

or

$$ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} {}^cX \\ {}^cY \\ {}^cZ \\ 1 \end{bmatrix} $$

It finally gives:

$$ x = \frac{{}^cX}{{}^cZ}, \; y = \frac{{}^cY}{{}^cZ} \tag{2.2} $$

In ViSP, the point tracker is implemented in the `vpPoint` *class.* `oP` *(resp.* `cP`*) is the point coordinates in homogeneous coordinates in the world frame (resp. in the camera frame) whereas* `p` *is its 2D coordinates again in homogeneous coordinates.*

Initialisation. *ViSP features many possibilities to initialize a* `vpPoint`.

```
vpPoint A ;

A.setWorldCoordinates(X,Y,Z) ;  // set A.oP, 3D point coordinates in the world frame
//or
vpColVector oP(3);
oP[0]=X; oP[1]=Y; oP[2]=Z; // X,Y,Z : 3D point coordinates in the worl frame
A.setWorlCoordinates(oP);  // set A.oP
```

Frame transformation and projection. *The behavior of* `setWorldCoordinates(...)`, `changeFrame(...)` *and* `project(...)` *have been described in section 2.1 and implement equations (2.1) and (2.2).*

```
vpPoint A ;
  ... // point initialisation in the world frame
vpHomogeneousMatrix cMo ;

A.changeFrame(cMo) ; // set A.cP, the 3D point coordinates in the camera frame.
A.project() ;        // set A.p, the point 2D homogeneous coordinates in the image plane.

// or
A.project(cMo) ;     // set A.cP and A.p
// or
A.track(cMo) ;       // set A.cP and A.p
```

Operations. *For simplicity issue the* `* operator` *has been overloaded and implements:*

- *the frame transformation operator as in equation (2.1) if a point is "multiplied" by a homogeneous matrix,*

```
vpHomogeneousMatrix cMo ;
vpPoint A,B  ;

B = cMo*A ;                       // B.oP = cMo*A.oP
```

- *a point transfert if a point is "multiplied" by a homography.*

```
vpHomography aHb ;
vpPoint aP,bP  ;

aP = aHb*bP ;                // aP.p = aHb*bP.p
```

Drawing. *The* `display(...)` *member functions described in section 2.1 have been implemented to draw a cross in place of projected point coordinates (see figure 2.1).*

```
vpDisplayGTK display ;                        //a video device interface
vpImage<unsigned char> I(height,width) ;      //initialise an image
display.init(I,posx,posy,"vpPoint display") ; //initialise the display

vpCameraParameters cam ;
vpHomogeneousMatrix cMo ;
vpPoint A ;

A.setWorldCoordinates(X,Y,Z) ; //set A.oP
A.project(cMo) ;               //set A.cP and A.p

vpDisplay::display(I) ;          // display I
A.display(I,cam,vpColor::blue) ; // draw a blue cross over I

vpDisplay::flush(I) ;            // flush the display buffer
```

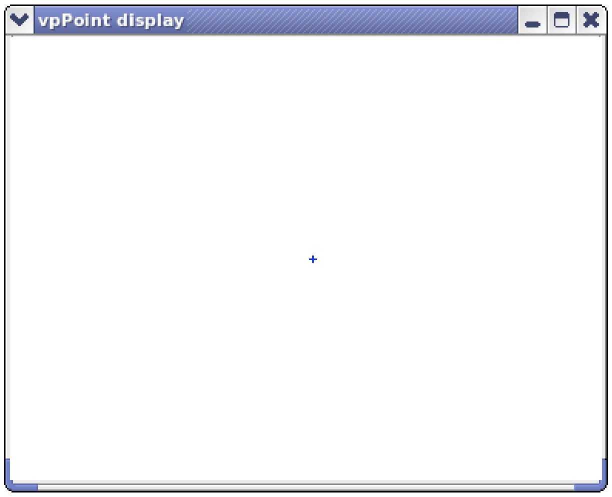


Figure 2.1: Drawing the forward projection of a 3D point in a 2D image.

2.3 Straight line

A 3D straight line in space is represented by the intersection of the two following planes:

$$\left\{ \begin{array}{l} h_1 = A_1X + B_1Y + C_1Z + D_1 = 0 \\ h_2 = A_2X + B_2Y + C_2Z + D_2 = 0 \end{array} \right. \tag{2.3}$$

Frame transformation. Denoting ${}^{\mathbf{o}}\mathbf{N}_i = ({}^oA_i, {}^oB_i, {}^oC_i)$ the normal to the plane $i = 1, 2$ expressed in frame $\mathcal{F}_o$. The plane equation expressed in frame $\mathcal{F}_c$ is given by:

$${}^{\mathbf{c}}\mathbf{N}_i = {}^c\mathbf{M}_o{}^{\mathbf{o}}\mathbf{N}_i{}^cD_i = {}^oD_i - {}^c\mathbf{t}_o{}^o\mathbf{N_i} \tag{2.4}$$

Projection. By using the equations of the perspective projection, we immediately obtain the equation of the 2D straight line noted as $\mathcal{D}$, resulting from the projection in the image of the 3D straight line:

$$ax + by + c = 0 \text{ with } \left\{ \begin{array}{l} a = A_1D_2 - A_2D_1 \\ b = B_1D_2 - B_2D_1 \\ c = C_1D_2 - C_2D_1 \end{array} \right. \tag{2.5}$$

Since parametrization (a, b, c) is not minimal, we will choose the representation (ρ, θ) defined by:

$$x\cos\theta + y\sin\theta - \rho = 0 \tag{2.6}$$

where $\theta = \arctan(b/a)$ and $\rho = -c/\sqrt{a^2 + b^2}$.

In ViSP, the line tracker is implemented in the `vpLine` *class.* `oP` *(resp.* `cP`*) is the two planes coordinates* `(A1,B1,C1,D1,A2,B2,C2,D2)` *in the world frame (resp. in the camera frame) whereas* `p = (rho,theta)` *is the 2D representation of the projected line.*

Initialisation. *ViSP provides many possibilities to initialize a* `vpLine`.

```
vpLine A ;

A.setWorldCoordinates(A1,B1,C1,D1,A2,B2,C2,D2) ;  // set A.oP
                                                   // A1,B1,C1,D1 : coordinates for plane 1
                                                   // A2,B2,C2,D2 : coordinates for plane 2
// or
vpColVector oP(8) ;
oP[0]=A1; oP[1]=B1; oP[2]=C1; oP[3]=D1; // A1,B1,C1,D1 : coordinates for plane 1
oP[4]=A2; oP[5]=B2; oP[6]=C2; oP[7]=D2; // A2,B2,C2,D2 : coordinates for plane 2
A.setWorldCoordinates(oP) ;             // set A.oP

// or
vpColVector oP1(4), oP2(4) ;                // oP1 and oP2 are coordinates of the two planes
oP1[0]=A1; oP1[1]=B1; oP1[2]=C1; oP1[3]=D1; // A1,B1,C1,D1 : coordinates for plane 1
oP2[0]=A2; oP2[1]=B2; oP2[2]=C2; oP2[3]=D2; // A2,B2,C2,D2 : coordinates for plane 2
A.setWorldCoordinates(oP1,oP2) ;            // set A.oP
```

Frame transformation and projection. *The behavior of* `setWorldCoordinates(...)`, `changeFrame(...)` *and* `project(...)` *have been described in section 2.1 and implement equations (2.4) and (2.6).*

```
vpLine A ;
  ... // line initialisation in the world frame.
vpHomogeneousMatrix cMo ;

A.changeFrame(cMo)  ; // set A.cP
A.project() ;         // set A.p

// or
A.project(cMo) ;      // set A.cP and A.p
// or
A.track(cMo) ;        // set A.cP and A.p
```

Drawing. *The* `display(...)` *member functions described in section 2.1 have been implemented to draw the projected line (see figure 2.2).*

```
vpDisplayGTK display ;                       // a video device interface
vpImage<unsigned char> I(height,width) ;     // initialise an image
display.init(I,posx,posy,"vpLine display") ; // initialise the display

vpCameraParameters cam ;
vpHomogeneousMatrix cMo ;
vpLine A ;

A.setWorldCoordinates(A1,B1,C1,D1,A2,B2,C2,D2) ; // set A.oP : coordinates of the two planes
A.project(cMo) ;                                 // set A.cP and A.p

vpDisplay::display(I) ;            // display I
A.display(I,cam,vpColor::blue) ; // draw a blue line over I

vpDisplay::flush(I) ;              // flush the display buffer
```

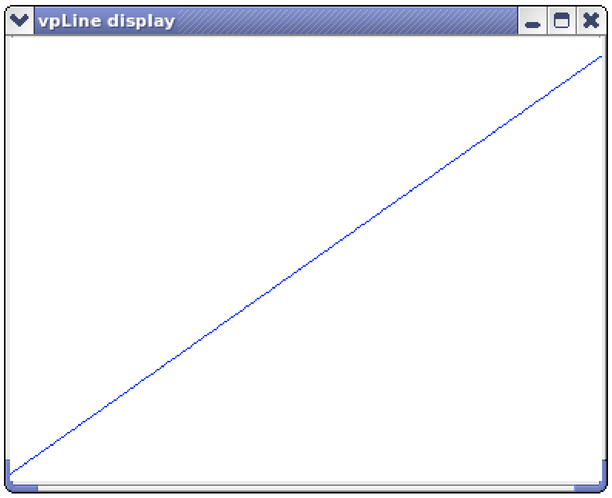


Figure 2.2: Drawing the forward projection of a 3D line in a 2D image.

2.4 Cylinder

A cylinder may be represented by the equation:

$$(X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - (\alpha X + \beta Y + \gamma Z)^2 - R^2 = 0 \tag{2.7}$$

$$\text{with} \left\{ \begin{array}{l} \alpha^2 + \beta^2 + \gamma^2 = 1 \\ \alpha X_0 + \beta Y_0 + \gamma Z_0 = 0 \end{array} \right.$$

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.

Frame transformation. Considering the set of parameters ${}^o\mathbf{P} = ({}^o\alpha, {}^o\beta, {}^o\gamma, {}^oX_0, {}^oY_0, {}^oZ_0, R)$ expressed in the world frame $\mathcal{F}_o$, cylinder coordinates expressed in the camera frame $\mathcal{F}_c$ are given by ${}^c\mathbf{P} = ({}^c\alpha, {}^c\beta, {}^c\gamma, {}^cX_0, {}^cY_0, {}^cZ_0, R)$ where:

$$\left[\begin{array}{c} {}^cX_0 \\ {}^cY_0 \\ {}^cZ_0 \\ 1 \end{array} \right] = {}^c\mathbf{M}_o \left[\begin{array}{c} {}^oX_0 \\ {}^oY_0 \\ {}^oZ_0 \\ 1 \end{array} \right] \quad \text{and} \quad \left[\begin{array}{c} {}^c\alpha \\ {}^c\beta \\ {}^c\gamma \end{array} \right] = {}^c\mathbf{R}_o \left[\begin{array}{c} {}^o\alpha \\ {}^o\beta \\ {}^o\gamma \end{array} \right] \tag{2.8}$$

Projection. The projection of a cylinder on the image plane is (for non-degenerated cases) a set of two straight lines with equation:

$$\left\{ \begin{array}{l} x \ \cos\theta_1 + x \ \sin\theta_1 - \rho_1 = 0 \\ y \ \cos\theta_2 + y \ \sin\theta_2 - \rho_2 = 0 \end{array} \right. \tag{2.9}$$

with:

$$\left\{ \begin{array}{l} \cos\theta_1 = \dfrac{RX_0/D - A}{\sqrt{(RX_0/D - A)^2 + (RY_0/D - B)^2}}, \cos\theta_2 = \dfrac{RX_0/D + A}{\sqrt{(RX_0/D + A)^2 + (RY_0/D + B)^2}} \\[2ex] \sin\theta_1 = \dfrac{RY_0/D - B}{\sqrt{(RX_0/D - A)^2 + (RY_0/D - B)^2}}, \sin\theta_2 = \dfrac{RY_0/D + B}{\sqrt{(RX_0/D + A)^2 + (RY_0/D + B)^2}} \\[2ex] \rho_1 = \dfrac{RZ_0/D - C}{\sqrt{(RX_0/D - A)^2 + (RY_0/D - B)^2}}, \rho_2 = \dfrac{RZ_0/D + C}{\sqrt{(RX_0/D + A)^2 + (RY_0/D + B)^2}} \end{array} \right.$$

$$\text{where} \left\{ \begin{array}{l} A = \gamma \ Y_0 - \beta \ Z_0 \\ B = \alpha \ Z_0 - \gamma \ X_0 \\ C = \beta \ X_0 - \alpha \ Y_0 \\ D = \sqrt{X_0^2 + Y_0^2 + Z_0^2 - R^2} \end{array} \right.$$

In ViSP, the cylinder is implemented in the `vpCylinder` *class.* `oP` *(resp.* `cP`*) correspond to its direction vector, a point of the axis and radius* `(alpha,beta,gamma,X0,Y0,Z0,R)` *expressed in the world frame (resp. in the camera frame) whereas* `p = (rho1, theta1, rho2, theta2)` *is the 2D representation of the two projected lines.*

Initialisation. *ViSP provides many possibilities to initialize a* `vpCylinder`.

```
vpCylinder A ;

A.setWorldCoordinates(alpha,beta,gamma,X0,Y0,Z0,R) ;  // set A.oP
                                                      // alpha,beta,gamma : direction vector
                                                      // X0,Y0,Z0 : point of the axis
                                                      // R : cylinder radius
// or
vpColVector oP(7) ;
oP[0]=alpha; oP[1]=beta; oP[2]=gamma; // alpha,beta,gamma : direction
oP[3]=X0; oP[4]=Y0; oP[5]=Z0;         // X0,Y0,Z0 : point of the axis
oP[6]=R;                              // R : cylinder radius
A.setWorldCoordinates(oP) ;           // set A.oP
```

Frame transformation and projection. *The behavior of* `setWorldCoordinates(...)`, `changeFrame(...)` *and* `project(...)` *have been described in section 2.1 and implement equations (2.8) and (2.9).*

```
vpCylinder A ;
  ... // cylinder initialisation in the world frame
vpHomogeneousMatrix cMo ;

A.changeFrame(cMo)  ; // set A.cP
A.project() ;         // set A.p

// or
A.project(cMo) ;      // set A.cP and A.p
// or
A.track(cMo) ;        // set A.cP and A.p
```

Drawing. *The* `display(...)` *member functions described in section 2.1 have been implemented to draw the set of two straight lines (see figure 2.3).*

```
vpDisplayGTK display ;                            // a video device interface
vpImage<unsigned char> I(height,width) ;          // initialise an image
display.init(I,posx,posy,"vpCylinder display") ; // display initialisation

vpCameraParameters cam ;
vpHomogeneousMatrix cMo ;
vpCylinder A ;

A.setWorldCoordinates(alpha,beta,gamma,X0,Y0,Z0,R) ; // set A.oP
A.project(cMo) ;                                     // set A.cP and A.p

vpDisplay::display(I) ;           // display I
A.display(I,cam,vpColor::blue) ; // draw two blue lines over I

vpDisplay::flush(I) ;             // flush the display buffer
```

2.5 Circle

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

$$\begin{cases} (X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - R^2 = 0 \\ \alpha(X - X_0) + \beta(Y - Y_0) + \gamma(Z - Z_0) = 0 \end{cases} \tag{2.10}$$

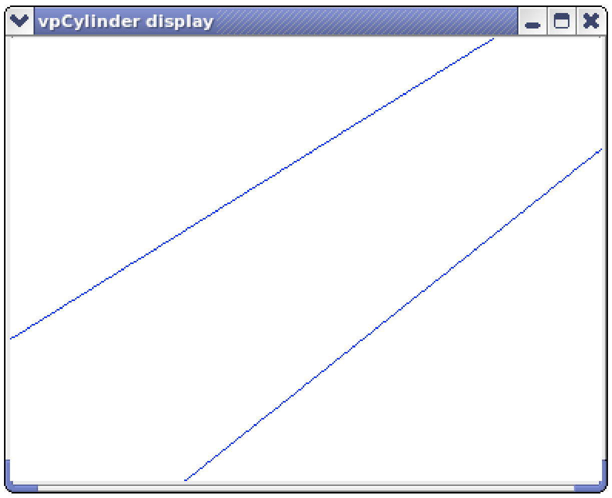


Figure 2.3: Drawing the forward projection of a cylinder in a 2D image.

Frame transformation. Considering the set of parameters ${}^o\mathbf{P} = ({}^o\alpha, {}^o\beta, {}^o\gamma, {}^oX_0, {}^oY_0, {}^oZ_0, R)$ expressed in the world frame $\mathcal{F}_o$, circle coordinates expressed in the camera frame $\mathcal{F}_c$ are given by ${}^c\mathbf{P} = ({}^c\alpha, {}^c\beta, {}^c\gamma, {}^cX_0, {}^cY_0, {}^cZ_0, R)$ where:

$$\begin{bmatrix} {}^cX_0 \\ {}^cY_0 \\ {}^cZ_0 \\ 1 \end{bmatrix} = {}^c\mathbf{M}_o \begin{bmatrix} {}^oX_0 \\ {}^oY_0 \\ {}^oZ_0 \\ 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} {}^c\alpha \\ {}^c\beta \\ {}^c\gamma \end{bmatrix} = {}^c\mathbf{R}_o \begin{bmatrix} {}^o\alpha \\ {}^o\beta \\ {}^o\gamma \end{bmatrix} \tag{2.11}$$

Projection. Its projection on the image plane takes the form of an ellipse or a circle (except in degenerated cases where the projection is a segment), which may be represented as:

$$\frac{(X - x_c + E(Y - y_c))^2}{A^2(1 + E^2)} + \frac{(Y - y_c - E(X - x_c))^2}{B^2(1 + E^2)} - 1 = 0 \tag{2.12}$$

$$\text{with} \left\{ \begin{array}{l} x_c = (K_1K_3 - K_2K_4)/(K_2^2 - K_0K_1) \\ y_c = (K_0K_4 - K_2K_3)/(K_2^2 - K_0K_1) \end{array} \right. \tag{2.13}$$

$$\text{and} \left\{ \begin{array}{l} E = (K_1 - K_0 \pm \sqrt{(K_1 - K_0)^2 + 4K_2^2})/2K_2 \\ A^2 = 2(K_0x_c^2 + 2K_2x_cy_c + K_1y_c^2 - K_5)/(K_0 + K_1 \pm \sqrt{(K_1 - K_0)^2 + 4K_2^2}) \\ B^2 = 2(K_0x_c^2 + 2K_2x_cy_c + K_1y_c^2 - K_5)/(K_0 + K_1 \mp \sqrt{(K_1 - K_0)^2 + 4K_2^2}) \end{array} \right. \tag{2.14}$$

where the coefficients K_i are obtained from the polynomial equation of an ellipse:

$$K_0X^2 + K_1Y^2 + 2K_2XY + 2K_3X + 2K_4Y + K_5 = 0 \tag{2.15}$$

More precisely, we have:

$$\begin{cases} K_0 = a^2(X_0^2 + Y_0^2 + Z_0^2 - R^2) + 1 - 2aX_0 \\ K_1 = b^2(X_0^2 + Y_0^2 + Z_0^2 - R^2) + 1 - 2bY_0 \\ K_2 = ab(X_0^2 + Y_0^2 + Z_0^2 - R^2) - bX_0 - aY_0 \\ K_3 = ac(X_0^2 + Y_0^2 + Z_0^2 - R^2) - cX_0 - aZ_0 \\ K_4 = bc(X_0^2 + Y_0^2 + Z_0^2 - R^2) - cY_0 - bZ_0 \\ K_5 = c^2(X_0^2 + Y_0^2 + Z_0^2 - R^2) + 1 - 2cZ_0 \end{cases} \tag{2.16}$$

with $a = \alpha/(\alpha X_0 + \beta Y_0 + \gamma Z_0)$, $b = \beta/(\alpha X_0 + \beta Y_0 + \gamma Z_0)$ and $c = \gamma/(\alpha X_0 + \beta Y_0 + \gamma Z_0)$.

The natural parametrization (x_c, y_c, E, A, B) is always ambiguous. Furthermore, the orientation E is not defined when the ellipse is a circle ($A = B$). So we use a representation based on the normalized inertial moments $\mathbf{p} = (x_c, y_c, \mu_{20}, \mu_{11}, \mu_{02})$ with:

$$\begin{cases} \mu_{20} = (A^2 + B^2E^2)/(1 + E^2) \\ \mu_{11} = E(A^2 - B^2)/(1 + E^2) \\ \mu_{02} = (A^2E^2 + B^2)/(1 + E^2) \end{cases} \tag{2.17}$$

In ViSP, the circle tracker is implemented in the `vpCircle` *class.* `oP` *(resp.* `cP`*) is the plane coordinates, the center coordinates and the radius of the sphere* `(A,B,C,X0,Y0,Z0,R)` *in the world frame (resp. in the camera frame), whereas* `p = (xc,yx,mu20,mu11,m02)` *is the 2D representation of the projected ellipse based on the inertial moments.*

Initialisation. *ViSP provides many possibilities to initialize a* `vpCircle`*.*

```
vpCircle A ;

A.setWorldCoordinates(A,B,C,X0,Y0,Z0,R) ;  // set A.oP
                                           // A,B,C : coordinates of the plane
                                           // X0,Y0,Z0 : center coordinates
                                           // R : sphere radius
// or
vpColVector oP(7) ;
oP[0]=A; oP[1]=B; oP[2]=C;    // A,B,C : coordinates of the plane
oP[3]=X0; oP[4]=Y0; oP[5]=Z0; // X0,Y0,Z0 : center coordinates of the sphere
oP[6]=R;                      // R : sphere radius
A.setWorldCoordinates(oP) ;   // set A.oP
```

Frame transformation and projection. *The behavior of* `setWorldCoordinates(...)`, `changeFrame(...)` *and* `project(...)` *have been described in section 2.1 and implement equations (2.11), (2.13) and (2.17).*

```
vpCircle A ;
  ... // circle initialisation in the world frame
vpHomogeneousMatrix cMo ;

A.changeFrame(cMo)  ; // set A.cP
A.project() ;         // set A.p

// or
A.project(cMo) ;      // set A.cP and A.p
// or
A.track(cMo) ;        // set A.cP and A.p
```

Drawing. *The* `display(...)` *member functions described in section 2.1 have been implemented to draw an ellipse (see figure 2.4).*

```
vpDisplayGTK display ;                        // a video device interface
vpImage<unsigned char> I(height,width) ;      // initialise an image
display.init(I,posx,posy,"vpCircle display") ; // display initialisation

vpCameraParameters cam ;
vpHomogeneousMatrix cMo ;
vpCircle A ;

A.setWorldCoordinates(A,B,C,X0,Y0,Z0,R) ; // set A.oP
A.project(cMo) ;                          // set A.cP and A.p

vpDisplay::display(I) ;           // display I
A.display(I,cam,vpColor::blue) ; // draw a blue ellipse over I

vpDisplay::flush(I) ;             // flush the display buffer
```

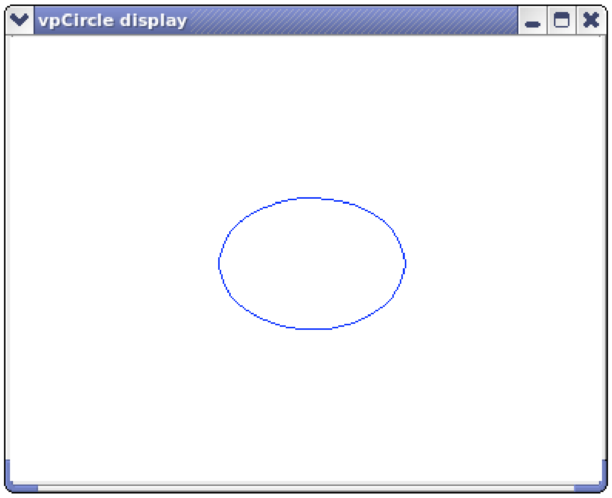


Figure 2.4: Drawing the forward projection of a circle in a 2D image.

2.6 Sphere

A sphere is represented by:

$$(X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 - R^2 = 0 \tag{2.18}$$

Frame transformation. Considering the set of parameters ${}^o\mathbf{P} = ({}^oX_0, {}^oY_0, {}^oZ_0, R)$ expressed in the world frame $\mathcal{F}_o$, cylinder coordinates expressed in the camera frame $\mathcal{F}_c$ are given by ${}^c\mathbf{P} = ({}^cX_0, {}^cY_0, {}^cZ_0, R)$ where:

$$\begin{bmatrix} {}^cX_0 \\ {}^cY_0 \\ {}^cZ_0 \\ 1 \end{bmatrix} = {}^c\mathbf{M}_o \begin{bmatrix} {}^oX_0 \\ {}^oY_0 \\ {}^oZ_0 \\ 1 \end{bmatrix} \tag{2.19}$$

Projection. The image of a sphere is an ellipse (a circle when $X_0 = Y_0 = 0$) with equation:

$$K_0X^2 + K_1Y^2 + 2K_2XY + 2K_3X + 2K_4Y + K_5 = 0 \tag{2.20}$$

$$\text{with} \begin{cases} K_0 = R^2 - Y_0^2 - Z_0^2 \\ K_1 = R^2 - X_0^2 - Z_0^2 \\ K_2 = X_0Y_0 \\ K_3 = X_0Z_0 \\ K_4 = Y_0Z_0 \\ K_5 = R^2 - X_0^2 - Y_0^2 \end{cases}$$

From parameters K_i, the 2D parameters $\mathbf{p} = (x_c, y_c, \mu_{20}, \mu_{11}, \mu_{02})$ are obtained using the equations (2.13) and (2.17).

In ViSP, the sphere tracker is implemented in the `vpSphere` *class.* `oP` *(resp.* `cP`*) is the coordinates of the center of the sphere and its radius in the world frame (resp. in the camera frame)* `(X0,Y0,Z0,R)` *whereas* `p = (xc,yx,mu20,mu11,m02)` *is the 2D representation of the projected ellipse based on the inertial moments.*

Initialisation. *ViSP features many possibilities to initialize a* `vpSphere`*.*

```
vpSphere A ;

A.setWorldCoordinates(X0,Y0,Z0,R) ;  // set A.oP
                                     // X0,Y0,Z0 : center coordinates of the sphere
                                     // R : sphere radius
//or
vpColVector oP(4) ;
oP[0]=X0; oP[1]=Y0; oP[2]=Z0; oP[3]=R; // X0,Y0,Z0,R : center, radius of the sphere
A.setWorldCoordinates(oP) ;            // set A.oP
```

Frame transformation and projection. *The behavior of* `setWorldCoordinates(...)`, `changeFrame(...)` *and* `project(...)` *have been described in section 2.1 and implement equations (2.19), (2.13) and (2.17).*

```
vpSphere A ;
  ... // sphere initialisation in the world frame
vpHomogeneousMatrix cMo ;

A.changeFrame(cMo)  ; // set A.cP
A.project() ;         // set A.p

// or
A.project(cMo) ;      // set A.cP and A.p
// or
A.track(cMo) ;        // set A.cP and A.p
```

Drawing. *The* `display(...)` *member functions described in section 2.1 have been implemented to draw an ellipse (see figure 2.5).*

```
vpDisplayGTK display ;                          // a video device interface
vpImage<unsigned char> I(height,width) ;        // initialise an image
display.init(I,posx,posy,"vpSphere display") ; // display initialisation

vpCameraParameters cam ;
vpHomogeneousMatrix cMo ;
vpSphere A ;

A.setWorldCoordinates(X0,Y0,Z0,R) ; // set A.oP
A.project(cMo) ;                    // set A.cP and A.p

vpDisplay::display(I) ;          // display I
A.display(I,cam,vpColor::blue) ; // draw a blue ellipse over I

vpDisplay::flush(I) ;            // flush the display buffer
```

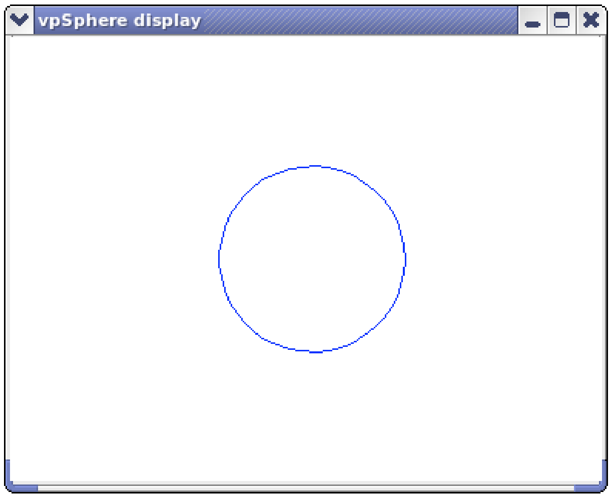


Figure 2.5: Drawing the forward projection of a sphere in a 2D image.

2.7 Image moments

In this part, we address the two-dimensional moment primitives implemented in ViSP [2, 4]. In the strictest sense, ViSP doesn't use moments but smart combinations of moments which are good invariants with respect to scale, translation or rotation. However, for the purposes of notation we will use the word "moment" to describe these moment combinations. When talking about moments in the strict sense, we will use the term "basic moment".

ViSP is able to handle three input formats for moment computation:

- Direct computation from image source (using `vpImage`);
- Computation from a set of points;
- Computation from a polygon defined by its vertices.

2.7.1 The moment object

As said earlier, moment primitives should be computed from the image directly or indirectly, or from other point-like visual features. Since ViSP moment primitives are actually combination of basic moments, we need a `vpMomentObject` class containing m_{ij} moments with:

$$m_{ij} = \sum_{k=1}^{n} x_k^i y_k^j \tag{2.21}$$

All ViSP moments primitives are then obtained by combining data from `vpMomentObject`. This process can be summed up as follows:

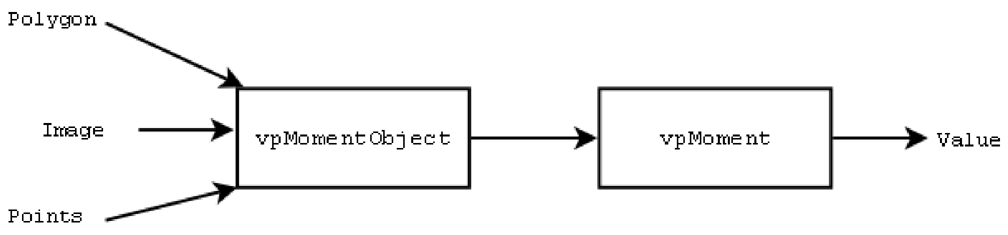


Figure 2.6: General scheme used to compute moment from a polygon, a set of points or from an image.

Besides the type of input data, the `vpMomentObject` allows the user to specify the maximum order of the m_{ij} moments. The order is defined by $i + j$. For example, if a `vpMomentObject` is initialized with order 3 then it will compute $m_{00}, m_{01}, m_{02}, m_{03}, m_{10}, m_{11}, m_{12}, m_{20}, m_{21}, m_{30}$.

Let's explore the different ways to initialize an object. In the following examples we will initialize an object of maximum order 3 using firstly a polygon, secondly a set of points and finaly an image.

1. **Initialization from a polygon**

```
// Define the contour of an object by a 5 clockwise vertices on a plane
vpPoint p;
std::vector<vpPoint> vec_p;    // Vector that contains 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.push_back(p);
p.set_x(+0.3); p.set_y(0.1);   // Coordinates in meters in the image plane (vertex 2)
vec_p.push_back(p);
p.set_x(+0.2); p.set_y(-0.1);  // Coordinates in meters in the image plane (vertex 3)
vec_p.push_back(p);
p.set_x(-0.2); p.set_y(-0.15); // Coordinates in meters in the image plane (vertex 4)
vec_p.push_back(p);
p.set_x(-0.2); p.set_y(0.1);   // Close the contour (vertex 5 = vertex 1)
vec_p.push_back(p);

vpMomentObject obj(3); // Create an image moment object with 3 as maximum order
obj.setType(vpMomentObject::DENSE_POLYGON); // The object is defined by a countour polygon
obj.fromVector(vec_p); // The object now contains all basic moments up to order 3
                       // describing the polygon.
```

This example is quite basic. The data is first stored in a vector of `vpPoints`. It is then computed inside the `vpMomentObject` using `vpMomentObject::fromVector()` method. The format (data as polygon) is specified in the `vpMomentObject::setType()` method. We set it to `vpMomentObject::DENSE POLYGON`.

Note: When using the polygon format, the order of the vertices in the vector does matter. It is important to input them in the clockwise order. You can either input a closed polygon or an open polygon. In the first case you specify all vertices in the clockwise order and the link between the first vertex and the last vertex is then implicit. Or you can explicitly specify it by adding the first vertex twice: first time as the start of the polygon at the beginning on the vertex list and second time as the last vertex of the list.

2. **Initialization from a set of points**

```
// Define 4 discrete points on a plane
vpPoint p;
std::vector<vpPoint> vec_p;    // Vector that contains the 4 points

p.set_x(-0.2); p.set_y(0.1);   // Coordinates in meters in the image plane (point 1)
vec_p.push_back(p);
p.set_x(+0.3); p.set_y(0.1);   // Coordinates in meters in the image plane (point 2)
vec_p.push_back(p);
p.set_x(+0.2); p.set_y(-0.1);  // Coordinates in meters in the image plane (point 3)
vec_p.push_back(p);
p.set_x(-0.2); p.set_y(-0.15); // Coordinates in meters in the image plane (point 4)
vec_p.push_back(p);

vpMomentObject obj(3); // Create a moment object with 3 as maximum order
obj.setType(vpMomentObject::DISCRETE); // The object is constituted by discrete points
obj.fromVector(vec_p); // Initialize the dense object with the points
```

This example is similar except for the `vpMomentObject::DISCRETE` parameter which indicates that we are processing a set of discrete points.

3. **Initialization from an image**

```
vpCameraParameters cam;             // Camera parameters used for pixel to meter conversion
vpImage<unsigned char> I(288, 384); // Image used to define the object
// ... Initialize the image

double threshold = 128; // all pixels strictly above this treshold are taken into account.

vpMomentObject obj(3); // Create an image moment object with 3 as maximum order
obj.setType(vpMomentObject::DENSE_FULL_OBJECT);
obj.fromImage(I, threshold, cam); // Initialize the object from the image
```

In this example we introduce two variables:

- the threshold. All pixels with a luminance below or equal to this threshold are not taken into account.
- the camera parameters. While the values in the discrete and polygon case are expected to be in meters, an image is expressed in pixel coordinates. Therefore, the `vpMomentObject` needs camera parameters in order to convert pixels to meters.

2.7.2 Common moment classes

The `vpMoment` block introduced Fig 2.6 is used to compute the value of a moment from the `vpMomentObject`. Since there are different kinds of moments, typically one for one or two degrees of freedom, this `vpMoment` has some derived classes. These classes are:

- `vpMomentBasic`, just a shortcut for `vpMomentObject`, needed for the database (see 2.7.3). Often referred to as m_{ij} given in 2.21 [4].
- `vpMomentGravityCenter`, the gravity center moment [4] defined by coordinates (x_g, y_g):

$$\begin{cases} x_g = \frac{m_{10}}{m_{00}} \\ \\ y_g = \frac{m_{01}}{m_{00}} \end{cases} \tag{2.22}$$

- `vpMomentCentered`, the centered moments [2] defined by:

$$\mu_{ij} = \sum_{k=1}^{i}\sum_{l=1}^{j}\binom{i}{k}\binom{j}{l}(-x_g)^{i-k}(-y_g)^{j-l}m_{kl} \tag{2.23}$$

- `vpMomentAreaNormalized`, the normalized area (a_n) [4]. This moment primitive makes sense only if we define a reference area and depth respectively called a^* and Z^*. In visual servoing these parameters are set to the destination surface which makes this moment primitive converge to 1. This moment is defined by:

$$a_n = Z^*\sqrt{\frac{a^*}{a}} \tag{2.24}$$

  where a is the current area.
- `vpMomentCInvariant` defined by various combinations of μ_{ij} with $(i, j) \in [0..5]^2$. A complete description of those invariants called $c_i, i \in [1..10]$ is given in [4].
- `vpMomentAlpha`, gives an estimation of the planar orientation of the object [4]. It is defined by:

$$\alpha = \frac{1}{2}arctan\left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}}\right) \tag{2.25}$$

Each of these moments requires a `vpMomentObject` to compute their value on. But different moments require `vpMomentObjects` initialized to different orders. If the object's order is too low, the moment won't be computed. Table 2.1 indicates the minimum required object order for each moment:

All these moments share common characteristics. Let's have a look at the `vpMoment` class from which they are all derived:

```
class vpMoment{
 public:
   std::vector<double>& get();
   void linkTo(vpMomentDatabase& moments);
   void update(vpMomentObject& object);
   virtual void compute()=0;
   virtual const char* name() = 0;
};
```

Listing 2.1: Generic `vpMoment` class from wich all the moment implemented in ViSP derived.

Most importantly, each moment has a `vpMoment::get()` method returning one or more values in a vector. The reason for this is that a moment can have a lot of values (`vpMomentCentered` provides μ_{ij} with $i+j \leq order$) or just one (`vpMomentAlpha` provides α) or two (`vpMomentCInvariant` provides x_g and y_g). In all cases `vpMoment::get()` is the common way to access the moment's values. For example:

moment class	minimal object order	
	dense object	discrete object
`vpMomentBasic`	variable	variable
`vpMomentCentered`	variable	variable
`vpMomentGravityCenter`	1	1
`vpMomentGravityCenterNormalized`	1	2
`vpMomentAreaNormalized`	0	2
`vpMomentCInvariant`	5	5
`vpMomentAlpha`	3	3

Table 2.1: Minimal order required to construct a vpMomentObject(order) object used to compute `vpMoment...` classes. Values are different if the object is dense (case of a polygon or an image) or discrete (case of a set of points).

```
double xg = gravityMoment.get()[0];
double mu_ij = centeredMoment.get()[j*(order-1)+i];
```

Often however, the classic moments provide shortcuts such as:

```
double xg = gravityMoment.getXg();
double mu_ij = centeredMoment.get(i,j);
```

However the `vpMoment::get()` method does not compute anything. Therefore the user has to compute the moment's values first. But before computing anything, it is necessary to initialize the moment with a `vpMomentObject` using `vpMoment::update()`. Then, `vpMoment::compute()` computes all the moments. The process could be summed up in the following way:

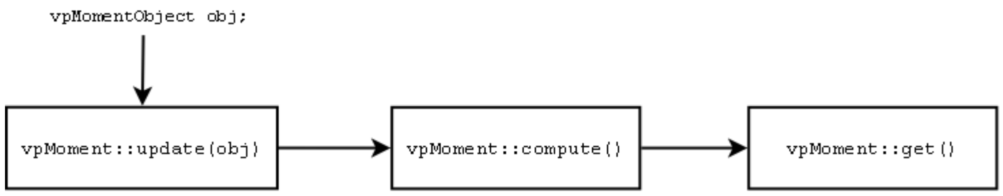


Figure 2.7: Moment initialization and computation.

The following code shows how to compute the gravity center:

```
std::vector<vpPoint> vec_p; // Vector that contains the vertices of the contour polygon
// [... init vec_p ...]
vpMomentObject obj(1);      // Create an image moment object with 1 as
                            // maximum order (because only m00,m01,m10
                            // are needed to compute the gravity center primitive.
obj.setType(vpMomentObject::DENSE_POLYGON); // The object is defined by a countour polygon
obj.fromVector(vec_p);      // Initialize the dense object with the polygon

vpMomentGravityCenter g;    // Declaration of gravity center
g.update(obj);              // Specify the object
g.compute();                // Compute the moment

std::cout << "xg=" << g.getXg() << std::endl; // Access to xg
std::cout << "yg=" << g.getYg() << std::endl; // Access to yg
```

Note that luckily, the `vpMomentGravityCenter` does not have any non-trivial dependencies to other moments. Other cases are a little more complicated.

The `linkTo()` method introduced in Listing 2.1 will be discussed later, in databases subsection (see 2.7.3). It is just important to remember that the `vpMoment::name()` method also introduced in Listing 2.1 serves as a key to identify the moment in the database.

It is becoming obvious from the above formulas that moments are not calculated from the basic moment directly but depend on each other. These dependencies could be summed up by the UML diagram presented Fig. 2.8:

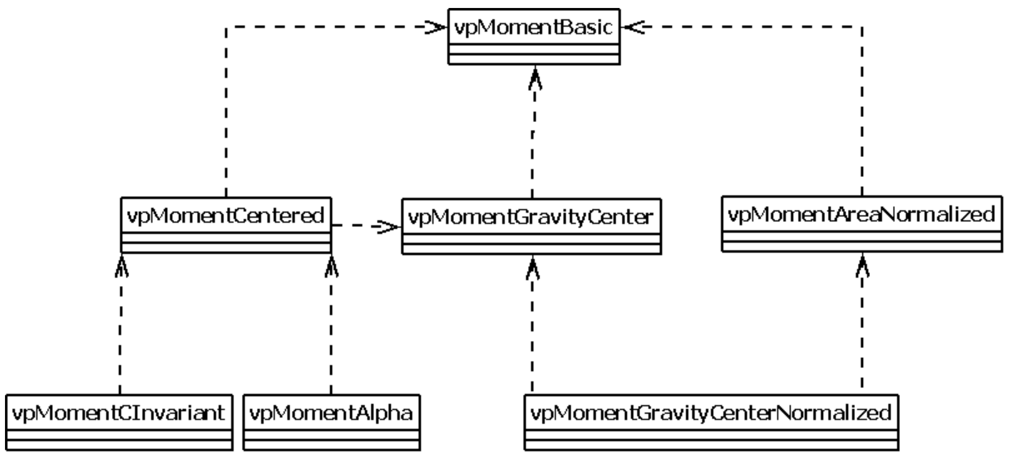


Figure 2.8: UML diagram that gives dependencies between moments implemented in ViSP.

Obviously, ViSP moments are heavily interdependent. Therefore, the library provides tools to handle dependencies between them by using a moment database.

2.7.3 Moment database

Instead of having a static implementation of the dependencies, ViSP provides a dynamic moment database. A moment database is a database where each type of moment is stored. Each of the moments stored in the database can access all other moments in the database. The dynamic moment identifier of the moment is located inside the `vpMoment` class and is available using `vpMoment::name()` method. Therefore the user must guarantee all moment dependencies are insite the database before computing a moment.

The simple moment database. In the following example, we compute the `vpMomentAlpha` moment primitive. According to the dependency diagram presented Fig. 2.8, we need to compute `vpMomentAlpha`, `vpMomentCentered`, `vpMomentGravityCenter` and `vpMomentBasic` that must be computed first. In practice the `vpMomentBasic` is optional, it is handeled by `vpMomentObject`. However, it is necessary if you are planning to use `vpFeatureMoment`.

```
vpPoint p;
std::vector<vpPoint> vec_p; // Vector that contains the vertices of the contour polygon

p.set_x(1); p.set_y(1);      // Coordinates in meters in the image plane (vertex 1)
vec_p.push_back(p);
p.set_x(2); p.set_y(2);      // Coordinates in meters in the image plane (vertex 2)
```

```
vec_p.push_back(p);
p.set_x(-3); p.set_y(0);     // Coordinates in meters in the image plane (vertex 3)
vec_p.push_back(p);
p.set_x(-3); p.set_y(-1);    // Coordinates in meters in the image plane (vertex 4)
vec_p.push_back(p);

vpMomentObject obj(3);       // Object of order 3
obj.setType(vpMomentObject::DENSE_POLYGON); // Object is the inner part of a polygon
obj.fromVector(vec_p);       // Initialize the dense object with the polygon

vpMomentDatabase db;         // Moment database
vpMomentGravityCenter g;     // Declaration of gravity center
vpMomentCentered mc;         // Centered moments
vpMomentAlpha alpha;         // Alpha moment

g.linkTo(db);                // Add gravity center to database
mc.linkTo(db);               // Add centered moments depending on gravity center
alpha.linkTo(db);            // Add alpha depending on centered moments

db.updateAll(obj);           // All of the moments must be updated, not just alpha

g.compute();                 // Compute the gravity center moment
mc.compute();                // Compute centered moments AFTER gravity center
alpha.compute();             // Compute alpha moment AFTER centered moments.
```

Listing 2.2: Example that shows how to compute `vpMomentAlpha` (see 2.25) by handling moment dependencies introduced in Fig. 2.8.

The beginning of this example initializes the object from a polygon up to order 3. This has already been covered. Afterwards, we declare a `vpMomentDatabase` to hold the moments. Then, all the moments need to be linked to the database. To do so, we use `vpMomentObject::linkTo(db)`. Once all the moments are linked, we need to make them aware of the object they will be processing. This is done by calling `db.updateAll(obj)`. You can also call `moment.update(obj)` for each moment but this simplifies it. Once the moments are updated with the object and are linked to the database, we need to compute the moments. This is done by using the `vpMoment::compute()` method. In this last step, the order of computation does matter. The moments must be computed according to the dependency diagram: computing `vpMomentGravityCenter` enables to compute `vpMomentCentered` which in turn enables to compute `vpMomentAlpha` (see Fig. 2.8).

The common moment database. In most cases ViSP's moments are used for visual servoing. And in most visual servoing cases, the following set of moment primitives is used:

- `vpMomentGravityCenterNormalized` for planar translations;
- `vpMomentAreaNormalized` for in-depth translations;
- `vpMomentCInvariant` for non-planar rotations;
- `vpMomentAlpha` for planar rotations.

All dependencies considered, this implies to systematically declare, link and compute seven moments. In order to avoid repetitive code, ViSP provides a shortcut: `vpMomentCommon`. This class is a moment database which already contains all the classic moments. Moreover, it contains information about in which order to compute the moments. Therefore the user does not have to worry about dependency at all. He needs only to call `vpMomentCommon::updateAll(object)`. The following example demonstrates the computation and retrieval of a `vpMomentCInvariant` moment:

```cpp
#include <visp/vpMomentObject.h>
#include <visp/vpPoint.h>
#include <visp/vpMomentCInvariant.h>
#include <visp/vpMomentCommon.h>
#include <iostream>

int main()
{
  // Define two discrete points
  vpPoint p;
  std::vector<vpPoint> vec_p; // Vector that contains the vertices of the contour polygon

  p.set_x(1); p.set_y(1);      // Coordinates in meters in the image plane (vertex 1)
  vec_p.push_back(p);
  p.set_x(2); p.set_y(2);      // Coordinates in meters in the image plane (vertex 2)
  vec_p.push_back(p);
  p.set_x(-3); p.set_y(0);     // Coordinates in meters in the image plane (vertex 3)
  vec_p.push_back(p);
  p.set_x(-3); p.set_y(1);     // Coordinates in meters in the image plane (vertex 4)
  vec_p.push_back(p);

  vpMomentObject obj(5);       // Object initialized up to order 5 to handle
            // all computations required by vpMomentCInvariant
  obj.setType(vpMomentObject::DENSE_POLYGON); // object is the inner part of a polygon
  obj.fromVector(vec_p);       // Initialize the discrete object with two points

  // Initialization with default values
  vpMomentCommon db(vpMomentCommon::getSurface(obj),
                    vpMomentCommon::getMu3(obj),
                    vpMomentCommon::getAlpha(obj),
                    1.);
  bool success;

  db.updateAll(obj);           // Update AND compute all moments

  //get C-invariant
  vpMomentCInvariant& C = static_cast<vpMomentCInvariant&>(db.get("vpMomentCInvariant",success));
  if (success)
    std::cout << C.get(0) << std:: endl;
  else
    std::cout << "vpMomentCInvariant not found." << std::endl;

  return 0;
}
```

There are now fewer lines of code needed to compute moments. Note that the `db.updateAll(obj)` call not only updates the moments with the object but also computes them. Let's take a closer look at some important lines in the example code:

- The database declaration: `vpMomentCommon db( vpMomentCommon::getSurface(obj), vpMomentCommon::getMu3(obj), vpMomentCommon::getAlpha(obj), 1.)` The common moment database uses different types of moments, some of which may need additional parameters. This is the case of the `vpMomentAreaNormalized` (requiering Z^* and a_n) and the `vpMomentAlpha` moment which will be discussed later. These parameters are (in this order) a reference area, a reference set $\mu_3 = \mu_{ij}$ with $i + j = 3$, a reference alpha value and a reference depth. All of these are computed by static helpers from the reference object. In classical visual servoing, this reference object will be the destination object.

- The object order: The `vpMomentCInvariant` uses combination of 5th order centered moments. Therefore, the `vpMomentObject` must be computed up to order 5 before computing this moment. Since the `vpMomentCInvariant` is a part of the classical moments, we have to declare `vpMomentObject obj(5)` so the computation isn't blocked by a too low order.

- The retrieval of the moment: Contrary to the previous example, we didn't declare and compute the `vpMomentCInvariant` (or any other moment for that matter) manually. Therefore we don't have explicit access to it, it only exists in the database. Obviously, there is a method to retrieve a moment from a database: `vpMomentDatabase::get()`. The method takes the class name as a text string as its parameter and a bool indicating whether or not the moment exists in the database. Therefore, to retrieve a `vpMomentCInvariant` from a database we do:

```
 vpMomentCInvariant& C = static_cast<vpMomentCInvariant&>(db.get("vpMomentCInvariant",
                                                                  success));
```

2.7.4 Shedding more light on `vpMomentAlpha`

In the previous subsection, we saw that the `vpMomentAlpha` class requires additionnal parameters.

This moment primitive is used to estimate the planar orientation, in radians, of the object. Without any additionnal information the orientation is only estimated with a precision of π. In other words, the two following object presented Fig. 2.9 will have the same `vpMomentAlpha` value:

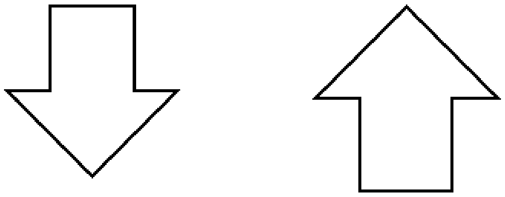


Figure 2.9: Example of two different objects that have the same α moment value since `vpMomentAlpha` is estimated with a precision of π.

The consequence of this in visual servoing is the robot converging to the symmetric of the destination object instead of converging to the object like it is supposed to.

To remove this ambiguity, ViSP enables us to define a reference alpha. Let's have a look at the `vpMomentAlpha` class:

```
class vpMomentAlpha : public vpMoment {
  public:
    vpMomentAlpha();
    vpMomentAlpha(std::vector<double>& ref,double alphaRef);

    void compute();
    double get();
    const char* name() {return "vpMomentAlpha";}
};
```

Besides the usual methods and shortcuts, there are two constructors:

- Reference constructor: `vpMomentAlpha::vpMomentAlpha()` handles the moment when it is computed from a reference instance of an object. A reference instance of an object is simply the same object rotated to 0 rad. All rotations will be computed with respect to that object. In most visual servoing cases, the reference object will be the destination object. However, to create a reference object descriptor, we need more than just the reference alpha. We also need the third-order centered moments stored in a vector. A shortcut to get them from an object is accessible via `vpMomentCommon::getMu3(object)`. The reference object descriptor is the $(\alpha_{ref}, \mu_{3ref})$ pair.

- Usual constructor: `vpMomentAlpha::vpMomentAlpha(std::vector<double> ref,double alphaRef)` enables the class to compute the alpha moment with full precision. It needs the μ_{3ref} set as first parameter and the α_{ref} as second parameter. The reference alpha is nothing more than `alpha.get()` where `alpha` is a `vpMomentAlpha` computed on a reference object.

The following example demonstrates how to differenciate two symmetric objects using a reference alpha:

```
#include <visp/vpPoint.h>
#include <visp/vpMomentObject.h>
#include <visp/vpMomentGravityCenter.h>
#include <visp/vpMomentDatabase.h>
#include <visp/vpMomentCentered.h>
#include <visp/vpMomentAlpha.h>
#include <iostream>
#include <vector>
#include <algorithm>

//generic function for printing
void print (double i) { std::cout << i << "\t";}

int main()
{
  vpPoint p;
  std::vector<vpPoint> vec_p; // Vector that contains the vertices of the contour polygon

  p.set_x(1); p.set_y(1);      // Coordinates in meters in the image plane (vertex 1)
  vec_p.push_back(p);
  p.set_x(2); p.set_y(2);      // Coordinates in meters in the image plane (vertex 2)
  vec_p.push_back(p);
  p.set_x(-3); p.set_y(0);     // Coordinates in meters in the image plane (vertex 3)
  vec_p.push_back(p);
  p.set_x(-3); p.set_y(-1);    // Coordinates in meters in the image plane (vertex 4)
  vec_p.push_back(p);

  //////////////////////////////REFERENCE VALUES////////////////////////////////
  vpMomentObject objRef(3);    // Reference object. Must be of order 3 because we will need
            // the 3rd order centered moments
  objRef.setType(vpMomentObject::DENSE_POLYGON); // Object is the inner part of a polygon
  objRef.fromVector(vec_p);    // Initialize the dense object with the polygon

  vpMomentDatabase dbRef;      // Reference database
  vpMomentGravityCenter gRef; // Declaration of gravity center
  vpMomentCentered mcRef;      // Centered moments
  vpMomentAlpha alphaRef;      // Declare alpha as reference

  gRef.linkTo(dbRef);          // Add gravity center to database
  mcRef.linkTo(dbRef);         // Add centered moments
  alphaRef.linkTo(dbRef);      // Add alpha depending on centered moments

  dbRef.updateAll(objRef);     // All of the moments must be updated, not just alpha

  gRef.compute();              // Compute the reference gravity center moment
  mcRef.compute();             // Compute the reference centered moments AFTER gravity center
  alphaRef.compute();          // Compute the reference alpha moment AFTER centered moments.

  // The order of values in the vector must be as follows: mu30 mu21 mu12 mu03
  std::vector<double> mu3ref(4);
  mu3ref[0] = mcRef.get(3,0);
  mu3ref[1] = mcRef.get(2,1);
  mu3ref[2] = mcRef.get(1,2);
  mu3ref[3] = mcRef.get(0,3);

  std::cout << "--- Reference object ---" << std::endl;
  std::cout << "alphaRef=" << alphaRef << std::endl << "mu3="; // print reference alpha
  std::for_each (mu3ref.begin(), mu3ref.end(),print);
  std::cout << std::endl;

  ////////////CURRENT VALUES (same object rotated 180deg - must be
  ////////////entered in reverse order)////////////////
  vec_p.clear();

  p.set_x(-3); p.set_y(1); // Coordinates in meters in the image plane (vertex 4)
  vec_p.push_back(p);
  p.set_x(-3); p.set_y(0); // Coordinates in meters in the image plane (vertex 3)
  vec_p.push_back(p);
  p.set_x(2); p.set_y(-2); // Coordinates in meters in the image plane (vertex 2)
  vec_p.push_back(p);
  p.set_x(1); p.set_y(-1); // Coordinates in meters in the image plane (vertex 1)
  vec_p.push_back(p);

  vpMomentObject obj(3);    // Second object. Order 3 is also required

  obj.setType(vpMomentObject::DENSE_POLYGON); // Object is the inner part of a polygon
  obj.fromVector(vec_p);    // Initialization of the dense object with the polygon

  vpMomentDatabase db;      // Database construction
  vpMomentGravityCenter g; // Declaration of gravity center
  vpMomentCentered mc;      // mc containts centered moments
  vpMomentAlpha alpha(mu3ref,alphaRef.get()); // Declare alpha as relative to a reference

  g.linkTo(db);             // Add gravity center to database
  mc.linkTo(db);            // Add centered moments
  alpha.linkTo(db);         // Add alpha depending on centered moments

  db.updateAll(obj);        // All of the moments must be updated

  g.compute();              // Compute the gravity center moment
  mc.compute();             // Compute centered moments AFTER gravity center
  alpha.compute();          // Compute alpha AFTER centered moments.

  std::cout << "--- current object ---" << std::endl;
  std::cout << "alpha=" << alpha.get() << std::endl;

  return 0;
}
```

Note that in `vpMomentCommon`, the class already handles the alpha disambiguation by asking the right parameters in its constructor.

2.7.5 Creating custom moments

All the previous parts demonstrate how to use the `vpMoment` classes implemented in ViSP. However these are only pre-implemented classic moment primitives. The user should be able to implement his own moment classes. In this subsection we'll show how to implement new moments compatible with the ViSP moments framework. Let's take a more detailed look at the `vpMoment` class:

```
class vpMoment{
  protected:
    std::vector<double> values;
    inline vpMomentDatabase& getMoments(){ return *moments; }
  public:
    inline vpMomentObject& getObject(){ return *object;}
    vpMoment();
    std::vector<double>& get(){ return values;}
    void linkTo(vpMomentDatabase& moments);
    void update(vpMomentObject& object);
    virtual void compute()=0;
    virtual const char* name() = 0;
};
```

Basically the two methods to implement are `vpMoment::name()` which is the moment's class name and `vpMoment::compute()` which describes how to compute the moment from the `vpMomentObject`.

We will now implement a new moment called `vpMomentI1` defined by [4]:

$$I_1 = -\mu_{20}\mu_{02} + {\mu_{11}}^2 \tag{2.26}$$

Therefore, our moment depends on `vpMomentCentered` and consequently on `vpMomentgravityCenter`. Here is the `vpMomentI1` on the UML dependency diagram:

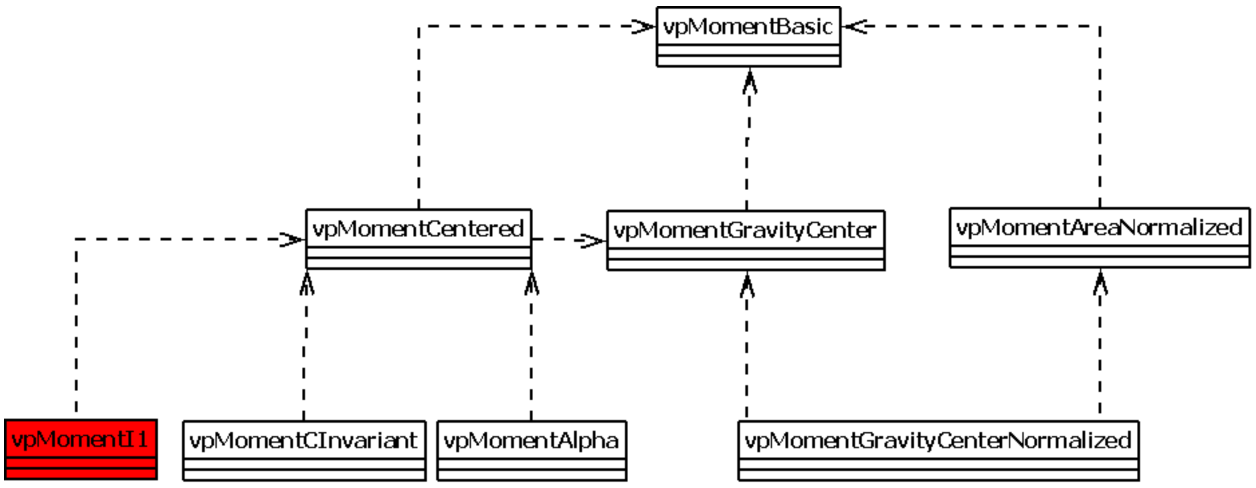


Figure 2.10: UML diagram that shows how to introduce a new moment given in 2.26 by considering moment dependencies.

Class definition. Here is the class outline of the new moment:

```
class vpMomentI1: public vpMoment {
  public:
    vpMomentI1();
    void compute();
    const char* name(){return "vpMomentI1";}
};
```

The `name()` enables the `vpMomentI1` to be accessible from a moment database. We'll now implement the default constructor and the moment computation.

Moment constructor. The `vpMoment` class has a protected attribute called `std::vector<double> values` which will store the moment values. In our case, there is only one value. Still, we need to set the dimension of the `value` vector accordingly:

```
vpMomentI1::vpMomentI1(){
  values.resize(1); //only one moment value
}
```

Moment computation. In this method we will take advantage of all the precalculated moments of ViSP in order to write the fewest possible code. In the previous subsections we noted that all moments linked to a database have access to a shared moment database. Moreover, all moments are updated with a `vpMomentObject`, this means they have access to it. Knowing this, let's try to implement formula 2.26:

```
void vpMomentI1::compute(){
  bool found_moment_centered; //true if vpMomentCentered is found

  vpMomentCentered& momentCentered = (
    static_cast<vpMomentCentered&>(getMoments().get("vpMomentCentered",found_moment_centered)));

    if(!found_moment_centered)
      throw vpException(vpException::notInitialized,"vpMomentCentered not found in database");

    values[0] = - momentCentered.get(2,0) * momentCentered.get(0,2)
                + momentCentered.get(1,1) * momentCentered.get(1,1);
}
```

In this method we do a few things:

- We access the database. The protected function `getMoments()` returns a `vpMomentDatabase` containing all shared moments. Then we use `vpMomentDatabase::get()` to look for a `name()` string, here `"vpMomentCentered"`.
- We check the availability of the moment. This is not done automatically and is very important because it saves a lot of debugging time.
- We combine the `vpMomentCentered` moments to compute our value. The `value[0]` is guaranteed to exist by the constructor.

That's it. A new moment is implemented. Let's see how to use this moment.

Using `vpMomentI1`. Our new moment behaves exactly as the classical moments already in of ViSP. Here is the code:

```
vpPoint p;
std::vector<vpPoint> vec_p; // Vector that contains the vertices of the contour polygon

//[...]
vpMomentObject obj(3);      // Object of order 3
obj.setType(vpMomentObject::DENSE_POLYGON); // Object is the inner part of a polygon
obj.fromVector(vec_p);      // Initialize the dense object with the polygon

vpMomentDatabase db;        // Moment database
vpMomentGravityCenter g;    // Declaration of gravity center moment
vpMomentCentered mc;        // Declaration of centered moments
vpMomentI1 i1;              // Our new moment

g.linkTo(db);               // Add gravity center to database
```

```
mc.linkTo(db);              // Add centered moments depending on gravity center
i1.linkTo(db);              // Add alpha depending on centered moments

db.updateAll(obj);          // All of the moments must be updated, not just alpha

g.compute();                // Compute the moment
mc.compute();               // Compute centered moments AFTER gravity center
i1.compute();               // Compute vpMomentI1 AFTER centered moments.

double value = i1.get()[0]; // Access to I1 value
```

To make the `vpMomentI1` more intuitive for external use, we could add a custom `getI1()` method returning `get()[0]`.

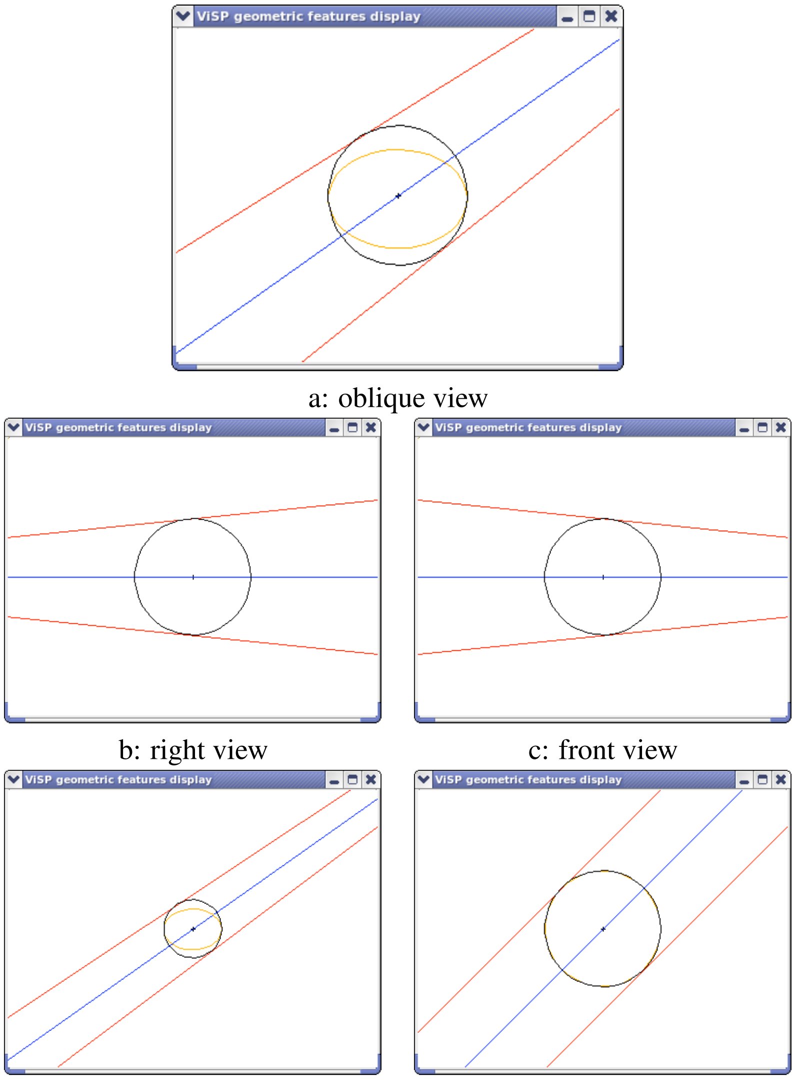
2.8 Example

In this section we present an example for building and drawing geometric features.

The following code is an example of drawing a 3D scene based on the previous geometric primitives and rendering it from a desired point of view. this code can also be found in the ViSP source tree `ViSP/examples/manual/geometric-features/manGeometricFeatures.cpp`

```
// For 2D image
#include <visp/vpImage.h>

// Video device interface
#include <visp/vpDisplay.h>
#include <visp/vpDisplayGTK.h>

// For frame transformation and projection
#include <visp/vpHomogeneousMatrix.h>
#include <visp/vpCameraParameters.h>

// Needed geometric features
#include <visp/vpPoint.h>
#include <visp/vpLine.h>
#include <visp/vpCylinder.h>
#include <visp/vpCircle.h>
#include <visp/vpSphere.h>

int main()
{
  unsigned int height = 288;
  unsigned int width = 384;
  vpImage<unsigned char> I(height,width); //image creation
  I = 255; // set I as a white image

  // create a display window
  vpDisplayGTK display;
  // initialize a display attached to image I
  display.init(I,100,100,"ViSP geometric features display");
  // camera parameters to digitalize the image plane
  vpCameraParameters cam(600,600,width/2,height/2); // (px,py,u0,v0)

  // pose of the camera with reference to the scene
  vpTranslationVector t(0,0,1); // tz = 1m
  vpRxyzVector rxyz(-M_PI/4,0,0); // rx = -45deg
  vpRotationMatrix R(rxyz);
  vpHomogeneousMatrix cMo(t,R);

39   // scene building, geometric features definition in the world frame
40   vpPoint point;
41   point.setWorldCoordinates(0,0,0); // X0=0,Y0=0,Z0=0
42   vpLine line;
43   line.setWorldCoordinates(1,1,0,0,0,0,1,0); // planes:(X+Y=0)&(Z=0)
44   vpCylinder cylinder;
45   cylinder.setWorldCoordinates(1,-1,0,0,0,0,0.1); // alpha=1,beta=-1,gamma=0,
46                                                   // X0=0,Y0=0,Z0=0,R=0.1
47   vpCircle circle;
48   circle.setWorldCoordinates(0,0,1,0,0,0,0.1); // plane:(Z=0),X0=0,Y0=0,Z=0,R=0.1
49   vpSphere sphere;
50   sphere.setWorldCoordinates(0,0,0,0.1); // X0=0,Y0=0,Z0=0,R=0.1
51 
52   // change frame to be the camera frame and project features in the image plane
53   point.project(cMo);
54   line.project(cMo);
55   cylinder.project(cMo);
56   circle.project(cMo);
57   sphere.project(cMo);
58 
59   // display the scene
60   vpDisplay::display(I); // display image I
61   // draw the projections of the 3D geometric features in the image plane
62   point.display(I,cam,vpColor::black);   // draw a black cross over I
63   line.display(I,cam,vpColor::blue);     // draw a blue line over I
64   cylinder.display(I,cam,vpColor::red);  // draw two red lines over I
65   circle.display(I,cam,vpColor::orange); // draw an orange ellipse over I
66   sphere.display(I,cam,vpColor::black);  // draw a black ellipse over I
67 
68   vpDisplay::flush(I);    // flush the display buffer
69   vpDisplay::getClick(I); // wait for a click in the display to exit
70   return 0;
71 }
```

In the previous code, we choose to represent the scene from a specific view point determined by the lines 34 and 35. The displayed image with these values is the figure 2.11.a . But it is also possible to obtain for example a front view (see figure 2.11.c), a right view (see figure 2.11.b), a top view (see figure 2.11.e) or a further view (see figure 2.11.d) changing respectively these lines to `rxyz(-M_PI/2,0,0)`, `rxyz(-M_PI/2,0,M_PI/2)`, `rxyz(0,0,0)` *and* `t(0,0,2)`.


ViSP geometric features display
ViSP geometric features display
ViSP geometric features display
ViSP geometric features display
ViSP geometric features display

a: oblique view

b: right view

c: front view

d: further view

e: top view

Figure 2.11: Drawing the projection of a 3D scene in a 2D image from different view points.

Bibliography

[1] B. Boufama and R. Mohr. Epipole and fundamental matrix estimation using virtual parallax. In *IEEE Int. Conf. on Computer Vision*, pages 1030–1036, 1995.

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

[3] O. Faugeras and F. Lustman. Motion and structure from motion in a piecewise planar environment. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 2(3):485–508, 1988.

[4] 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.

[5] B. Triggs. Plane + parallax, tensors and factorization. In *European Conference on Computer Vision, ECCV'00*, pages 522–538, Dublin, Eire, May 2000.