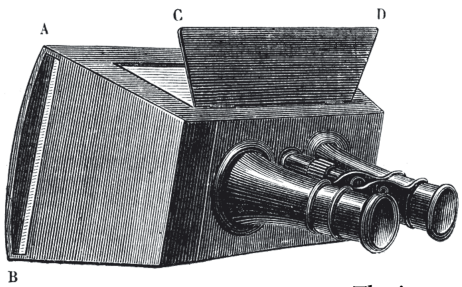
Chapter

15 Vision-Based Control



The task in visual servoing is to control the pose of the robot's end-effector, relative to the target, using visual *features* extracted from the image. As shown in Fig. 15.1 the camera may be carried by the robot or fixed in the world. The configuration of Fig. 15.1a has the camera mounted on the robot's end-effector observing the target, and is referred to as end-point closed-loop or eye-in-hand. The configuration of Fig. 15.1b has the camera at a fixed point in the world observing both the target and the robot's end-effector, and is referred to as end-point open-loop. In the remainder of this book we will discuss only the eye-in-hand configuration.

The image of the target is a function of the relative pose ${}^{C}\xi_{T}$. Features such as coordinates of points, or the parameters of lines of ellipses are extracted from the image and these are also a function of the relative pose ${}^{C}\xi_{T}$.

There are two fundamentally different approaches to visual servo control: Position-Based Visual Servo (PBVS) and Image-Based Visual Servo (IBVS). Position-based visual servoing, shown in Fig. 15.2a, uses observed visual features, a calibrated camera and a known geometric model of the target to determine the pose of the target with respect to the camera. The robot then moves toward that pose and the control is performed in task space which is commonly $SE(3)$. Good algorithms exist for pose estimation but it is computationally expensive and relies critically on the accuracy of the camera calibration and the model of the object's geometry. PBVS is discussed in Sect. 15.1.

A **servo-mechanism**, or servo is an automatic device that uses feedback of error between the desired and actual position of a mechanism to drive the device to the desired position. The word servo is derived from the Latin root *servus* meaning slave and the first usage was by the Frenchman J. J. L. Farcot in 1868 – "Le Servomoteur" – to describe the hydraulic and steam engines used for steering ships.

Error in position is measured by a sensor then amplified to drive a motor that generates a force to move the device to reduce the error. Servo system development was spurred by WW II with the development of electrical servo systems for fire-control applications that used electric motors and electro-mechanical *amplidyne* power amplifiers. Later servo amplifiers used vacuum tubes and more recently solid state power amplifiers (motor drives). Today servomechanisms are ubiquitous and are used to position the read/write heads in optical and magnetic disk drives, the lenses in autofocus cameras, remote control toys, satellite-tracking antennas, automatic machine tools and robot joints.

"Servo" is properly a noun or adjective but has become a verb "to servo". In the context of vision-based control we use the verb "visual servoing".

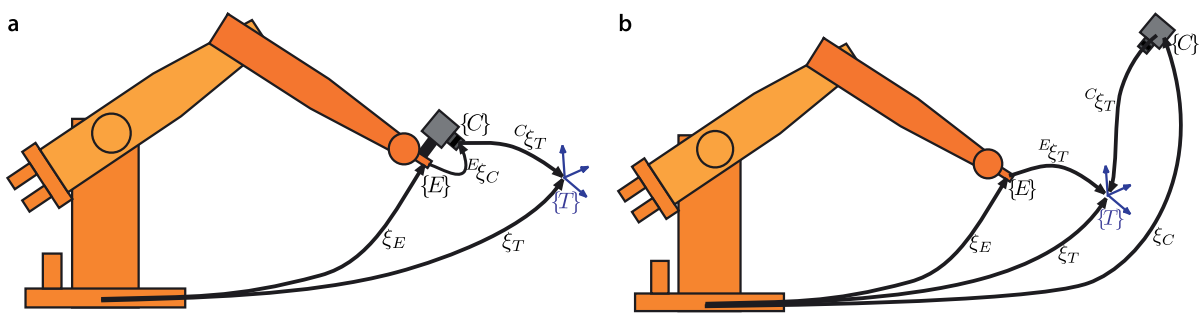


Fig. 15.1. Visual servo configurations and relevant coordinate frames: world, end-effector $\{E\}$, camera $\{C\}$ and target $\{T\}$. **a** End-point closed-loop configuration (eye-in-hand); **b** end-point open-loop configuration

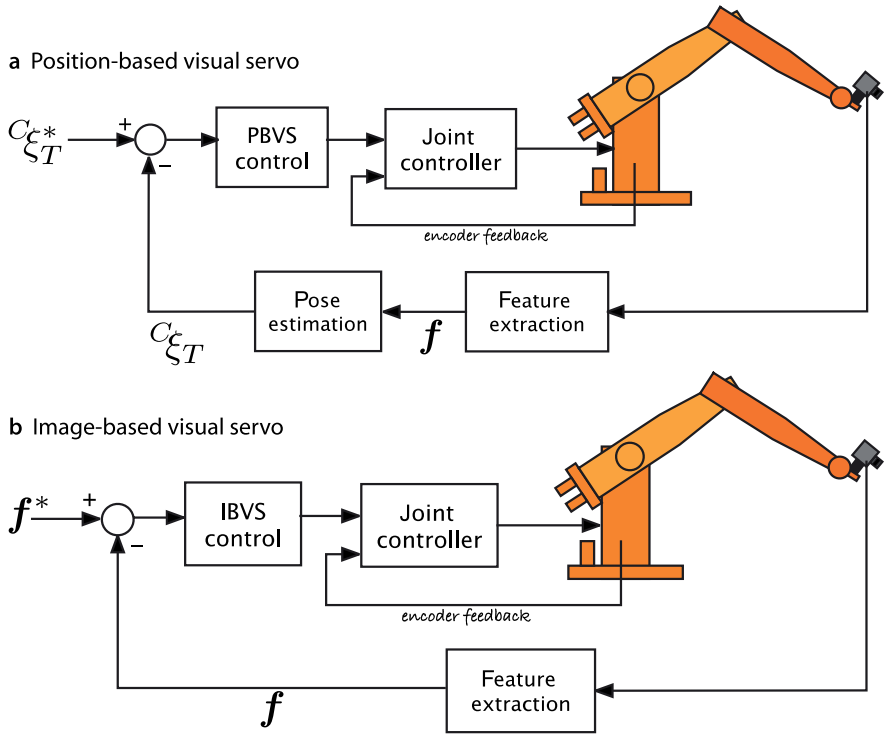


Fig. 15.2.
The two distinct classes of visual servo system

Image-based visual servoing, shown in Fig. 15.2b, omits the pose estimation step, and uses the image features directly. The control is performed in image coordinate space $\mathbb{R}^2$. The desired camera pose with respect to the target is defined *implicitly* by the image feature values at the goal pose. IBVS is a challenging control problem since the image features are a highly non-linear function of camera pose. IBVS is discussed in Sect. 15.2.

15.1 Position-Based Visual Servoing

In a PBVS system the pose of the target with respect to the camera ${}^C\xi_T$ is estimated. The pose estimation problem was discussed in Sect. 11.2.3 and requires knowledge of the target's geometry, the camera's intrinsic parameters and the observed image plane features. The relationships between the poses is shown in Fig. 15.3. We specify the desired relative pose with respect to the target ${}^{C^*}\xi_T$ and wish to determine the motion required to move the camera from its initial pose ξ_C to ξ_C^* which we call ξ_Δ. The actual pose of the target ξ_T is not known. From the pose network we can write

$$\xi_\Delta \oplus {}^{C^*}\xi_T = {}^C\hat{\xi}_T$$

where ${}^C\hat{\xi}_T$ is the estimated pose of the target relative to the camera. We rearrange this as

$$\xi_\Delta = {}^C\xi_T \ominus {}^{C^*}\hat{\xi}_T$$

which is the camera motion required to achieve the desired relative pose. The change in pose might be quite large so we do not attempt to make this movement in one step, rather we move to a point closer to $\{C^*\}$ by

$$\xi_C\langle k+1\rangle = \xi_C\langle k\rangle \oplus \lambda\xi_\Delta\langle k\rangle$$

which is a fraction $\lambda \in (0, 1)$ of the translation and rotation required.

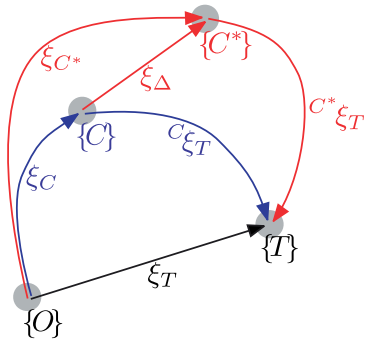


Fig. 15.3. Relative pose network for PBVS example. Frame $\{C\}$ is the current camera pose and frame $\{C^*\}$ is the desired camera pose

Using the Toolbox we start by defining a camera with known parameters

```
>> cam = CentralCamera('default');
```

The target comprises four points that form a square of side length 0.5 m that lies in the xy-plane and centred at (0, 0, 3)

```
>> P = mkgrid( 2, 0.5, 'T', transl(0,0,3) );
```

and we assume that this pose is unknown to the control system. The camera is at some pose `Tc` so the the pixel coordinates of the world points are

```
>> p = cam.plot(P, 'Tcam', Tc)
```

from which the pose of the target with respect to the camera ${}^C\hat{\xi}_T$ is estimated

```
>> Tc_t_est = cam.estpose(P, p);
```

The required motion ξ_Δ is

```
>> Tdelta = TcStar_t * inv(Tc_t_est);
```

and the fractional motion toward the goal is given by

```
>> Tdelta = trinterp(eye(4,4), Tdelta, lambda);
```

giving the new value of the camera pose

```
>> Tc = trnorm(Tc * Tdelta);
```

where we ensure that the transformation remains a proper homogeneous transformation by normalizing it using `trnorm`. At each time step we repeat the process, moving a fraction of the required relative pose until the motion is complete. In this way even if the robot has errors and does not move as requested, or the target moves the motion computed at the next time step will account for that error.

For this example we choose the initial pose of the camera in world coordinates as

```
>> Tc0 = transl(1,1,-3)*trotz(0.6);
```

and the desired pose of the target with respect to the camera is

```
>> TcStar_t = transl(0, 0, 1);
```

which has the target 1 m in front of the camera and fronto-parallel to it. We create an instance of the `PBVS` class

```
>> pbvs = PBVS(cam, 'T0', Tc0, 'Tf', TcStar_t)
Visual servo object: camera=noname
  200 iterations, 0 history
  P= -0.25       -0.25        0.25        0.25
     -0.25        0.25        0.25       -0.25
          0           0           0           0
  Tc0:   t = ( 1 1 -3), R = ( 34.3775deg |  0 0 1)
  Tc*_t: t = ( 0 0 1), R = nil
```

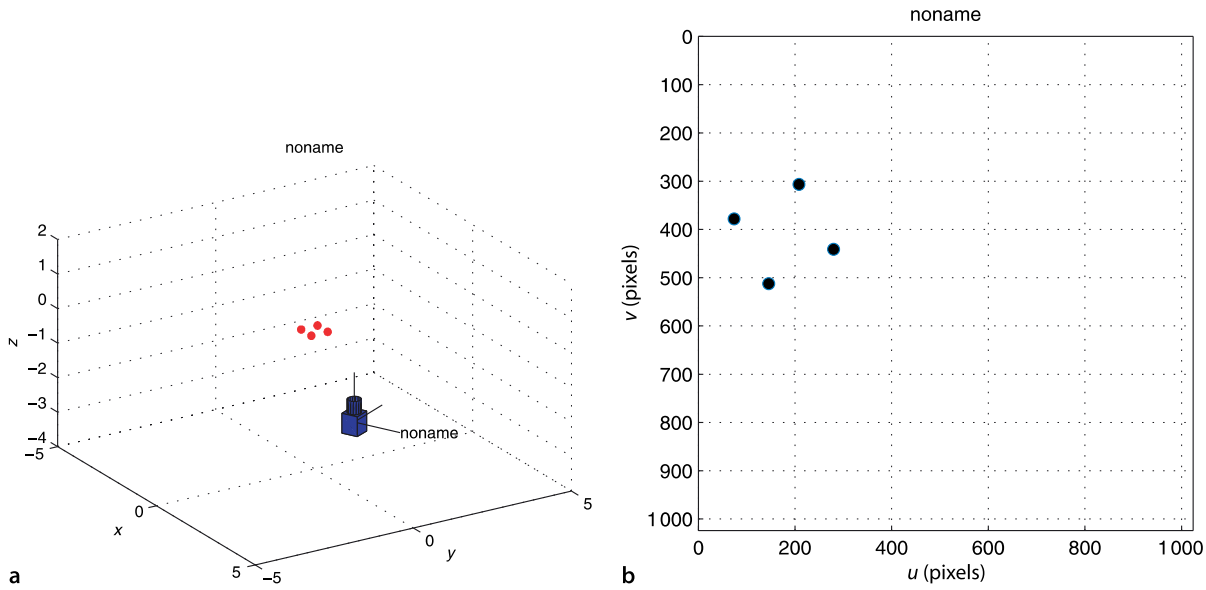


Fig. 15.4. Snapshop from the visual servo simulation. **a** An external view showing camera pose and features; **b** camera view showing current feature positions on the image plane

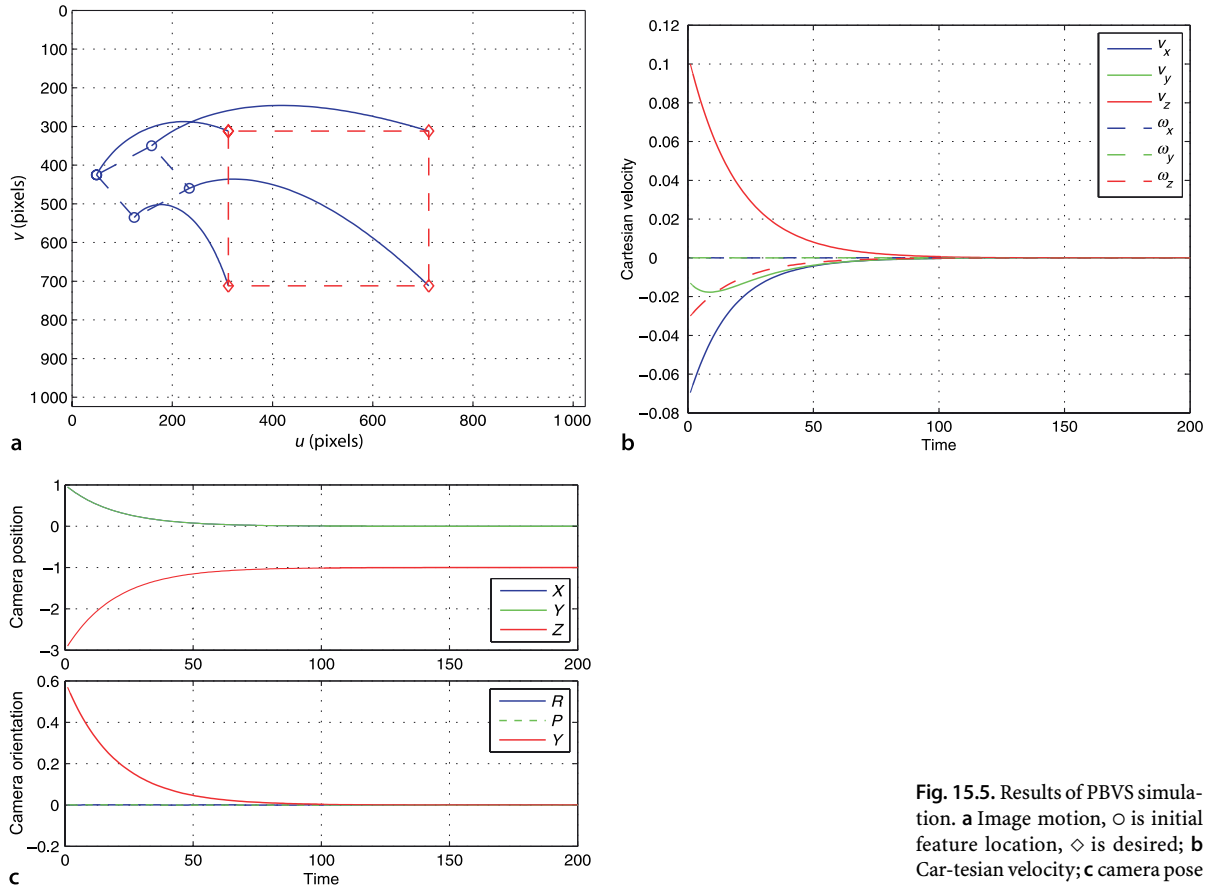


Fig. 15.5. Results of PBVS simulation. **a** Image motion, ○ is initial feature location, ◇ is desired; **b** Car-tesian velocity; **c** camera pose

which is a subclass of the `VisualServo` class and implements the controller outlined above. The object constructor takes a `CentralCamera` object as its argument, and drives this camera to achieve the desired pose relative to the target. Many additional options can be passed to this class constructor. The display methods shows the coordinates of the world points, the initial camera pose, and the desired target relative pose. The simulation is run by

```
>> pbvs.run();
```

which repeatedly calls the `step` method to execute a single time step. The simulation animates both the image plane of the camera and the 3-dimensional visualization of the camera and the world points as shown in Fig. 15.4. The simulation completes after a defined number of iterations or when ξ_Δ falls below some threshold.

The simulation results are stored within the object for later analysis. We can plot the path of the target features in the image, the Cartesian velocity versus time or Cartesian position versus time

```
>> pbvs.plot_p();
>> pbvs.plot_vel();
>> pbvs.plot_camera();
```

which are shown in Fig. 15.5. We see that the feature points have followed a curved path in the image, and that the camera's translation and orientation have converged smoothly on the desired values.

15.2 Image-Based Visual Servoing

IBVS differs fundamentally from PBVS by not estimating the relative pose of the target. The relative pose is implicit in the values of the image features. Figure 15.6 shows two views of a square target. The view from the initial camera pose is shown in red and it is clear that the camera is viewing the target obliquely. The desired view is shown in blue where the camera is further from the target and its optical axis is normal to the plane of the target – a fronto-parallel view.

The control problem can be expressed in terms of image coordinates. The task is to move the feature points indicated by ○-markers to the points indicated by ◇-markers. The points may, but do not have to, follow the straight line paths indicated by the arrows. Moving the feature points in the image *implicitly* changes the pose – we have changed the problem from pose estimation to control of points on the image.

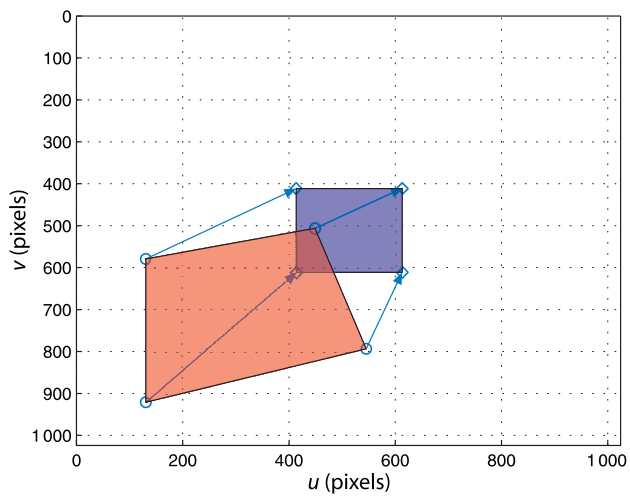


Fig. 15.6. Two views of a square target. The blue shape is the desired view, and the red shape is the initial view

15.2.1 Camera and Image Motion

Consider the default camera

```
>> cam = CentralCamera('default');
```

and a world point at

```
>> P = [1 1 5]';
```

which has image coordinates

```
>> p0 = cam.project( P )
p0 =
   672
   672
```

Now if we displace the camera slightly in the x-direction the pixel coordinates will become

```
>> px = cam.project( P, 'Tcam', transl(0.1,0,0) )
px =
   656
   672
```

Using the camera coordinate conventions of Fig. 11.4, the camera has moved to the right so the image point has moved to the left. The sensitivity of image motion to camera motion is

```
>> ( px - p0 ) / 0.1
ans =
  -160
     0
```

which is an approximation to the derivative $\partial \boldsymbol{p}/\delta_x$. It shows that 1 m of camera motion would lead to -160 pixel of feature motion in the u-direction. We can repeat this for z-axis translation

```
>> ( cam.project( P, 'Tcam', transl(0, 0, 0.1) ) - p0 ) / 0.1
ans =
   32.6531
   32.6531
```

which shows equal motion in the u- and v-directions. For x-axis rotation

```
>> ( cam.project( P, 'Tcam', trotx(0.1) ) - p0 ) / 0.1
ans =
   40.9626
  851.8791
```

the image motion is predominantly in the v-direction. It is clear that camera motion along and about the different degrees of freedom in $SE(3)$ causes quite different motion of image points. Earlier we expressed perspective projection in functional form Eq. 11.10

$$\boldsymbol{p} = \mathcal{P}(\boldsymbol{P}, \boldsymbol{K}, \xi_C)$$

and its derivative with respect to camera pose ξ is

$$\dot{\boldsymbol{p}} = \boldsymbol{J}_p(\boldsymbol{P}, \boldsymbol{K}, \xi_C)\boldsymbol{\nu}$$

where $\boldsymbol{\nu} = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z) \in \mathbb{R}^6$ is the velocity of the camera, the spatial velocity, which we introduced in Sect. 8.1. $\boldsymbol{J}_p$ is a Jacobian-like object, but because we have taken the derivative with respect to a pose $\xi \in SE(3)$ rather than a vector it is technically called an *interaction matrix*. However in the visual servoing world it is more commonly called an *image Jacobian* or a *feature sensitivity matrix*.

Consider a camera moving with a body velocity $\boldsymbol{\nu} = (\boldsymbol{v}, \boldsymbol{\omega})$ in the world frame and observing a world point **P** with camera relative coordinates $\boldsymbol{P} = (X, Y, Z)$. The velocity

of the point relative to the camera frame is

$$\dot{\boldsymbol{P}} = -\boldsymbol{\omega} \times \boldsymbol{P} - \boldsymbol{v} \tag{15.1}$$

which we can write in scalar form as

$$\begin{aligned} \dot{X} &= Y\omega_z - Z\omega_y - v_x \\ \dot{Y} &= Z\omega_x - X\omega_z - v_y \\ \dot{Z} &= X\omega_y - Y\omega_x - v_z \end{aligned} \tag{15.2}$$

The perspective projection Eq. 11.2 for normalized coordinates is

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

and the temporal derivative, using the quotient rule, is

$$\dot{x} = \frac{\dot{X}Z - X\dot{Z}}{Z^2},\ \dot{y} = \frac{\dot{Y}Z - Y\dot{Z}}{Z^2}$$

Substituting Eq. 15.2, $X = xZ$ and $Y = yZ$ we can write this in matrix form

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -\frac{1}{Z} & 0 & \frac{x}{Z} & xy & -(1+x^2) & y \\ 0 & -\frac{1}{Z} & \frac{y}{Z} & 1+y^2 & -xy & -x \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{15.3}$$

which relates camera velocity to feature velocity in normalized image coordinates.

The normalized image-plane coordinates are related to the pixel coordinates by Eq. 11.7

$$u = \frac{f}{\rho_u} x + u_0,\ v = \frac{f}{\rho_v} y + v_0$$

which we rearrange as

$$x = \frac{\rho_u}{f}\overline{u},\ y = \frac{\rho_v}{f}\overline{v} \tag{15.4}$$

where $\overline{u} = u - u_0$ and $\overline{v} = (v - v_0)$ are the pixel coordinates relative to the principal point. The temporal derivative is

$$\dot{x} = \frac{\rho_u}{f}\dot{\overline{u}},\ \dot{y} = \frac{\rho_v}{f}\dot{\overline{v}} \tag{15.5}$$

and substituting Eq. 15.4 and Eq. 15.5 into Eq. 15.3 leads to

$$\begin{pmatrix} \dot{\overline{u}} \\ \dot{\overline{v}} \end{pmatrix} = \underbrace{\begin{pmatrix} -\frac{f}{\rho_u Z} & 0 & \frac{\overline{u}}{Z} & \frac{\rho_u \overline{u}\,\overline{v}}{f} & -\frac{f^2 + \rho_u^2 \overline{u}^2}{\rho_u f} & \overline{v} \\ 0 & -\frac{f}{\rho_v Z} & \frac{\overline{v}}{Z} & \frac{f^2 + \rho_v^2 \overline{v}^2}{\rho_v f} & -\frac{\rho_v \overline{u}\,\overline{v}}{f} & -\overline{u} \end{pmatrix}}_{J_p} \begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{15.6}$$

in terms of pixel coordinates *with respect to the principal point*. We can write this in concise matrix form as

$$\dot{p} = J_p \nu \tag{15.7}$$

where J_p is the 2×6 *image Jacobian* matrix for a point feature.

This is commonly written in terms of u and v rather than $\bar{u}$ and $\bar{v}$ but we use the overbar notation to emphasize that the coordinates are with respect to the principal point, not the image origin which is typically in the top-left corner.

The Toolbox `CentralCamera` class provides the method `visjac_p` to compute the image Jacobian and for the example above it is

```
>> J = cam.visjac_p([672; 672], 5)
J =
  -160     0    32    32  -832   160
     0  -160    32   832   -32  -160
```

where the first argument is the coordinate of the point of interest with respect to the image, and the second argument is the depth of the point. The approximate values computed on page 460 appear as columns one, three and four respectively. Image Jacobians can also be derived for line and circle features and these are discussed in Sect. 15.3.

For a given camera velocity, the velocity of the point is a function of the point's coordinate, its depth and the camera parameters. Each column of the Jacobian indicates the velocity of a feature point with respect to the corresponding component of the velocity vector. The `flowfield` method of the `CentralCamera` class shows the feature velocity for a grid of points on the image plane for a particular camera velocity. For camera translational velocity in the x-direction the flow field is

```
>> cam.flowfield( [1 0 0 0 0 0] );
```

which is shown in Fig. 15.7a. As expected moving the camera to the right causes all the features points to move to the left. The motion of points on the image plane is known as optical flow and can be computed from image sequences as we showed in Sect. 14.8. For translation in the z-direction

```
>> cam.flowfield( [0 0 1 0 0 0] );
```

the points radiate outward from the principal point – the Star Trek warp effect – as shown in Fig. 15.7e. Rotation about the z-axis is

```
>> cam.flowfield( [0 0 0 0 0 1] )
```

causes the points to rotate about the principal point as shown in Fig. 15.7f.

Rotational motion about the y-axis is

```
>> cam.flowfield( [0 0 0 0 1 0] );
```

is shown in Fig. 15.7b and is very similar to the case of x-axis translation, with some small curvature for points far from the principal point. The reason for this is that the first and fifth column of the image Jacobian above are approximately equal which implies that translation in the x-direction causes almost the same image motion as rotation about the y-axis. You can easily demonstrate this equivalence by watching how the world moves if you translate your head to the right or rotate your head to the right – in both cases the world appears to move to the left. As the focal length increases the element $J[2,5]$ becomes smaller and column five approaches a scalar multiple of column one. We can easily demonstrate this by increasing the focal length to $f = 20$ mm (the default focal length is 8 mm) and the flowfield

```
>> cam.f = 20e-3;
>> cam.flowfield( [0 0 0 0 1 0] );
```

which is shown in Fig. 15.7c is almost identical to that of Fig. 15.7a. Conversely, for small focal lengths (wide-angle cameras) the image motion due to these camera motions will be more dissimilar

```
>> cam.f = 4e-3;
>> cam.flowfield( [0 0 0 0 1 0] );
```

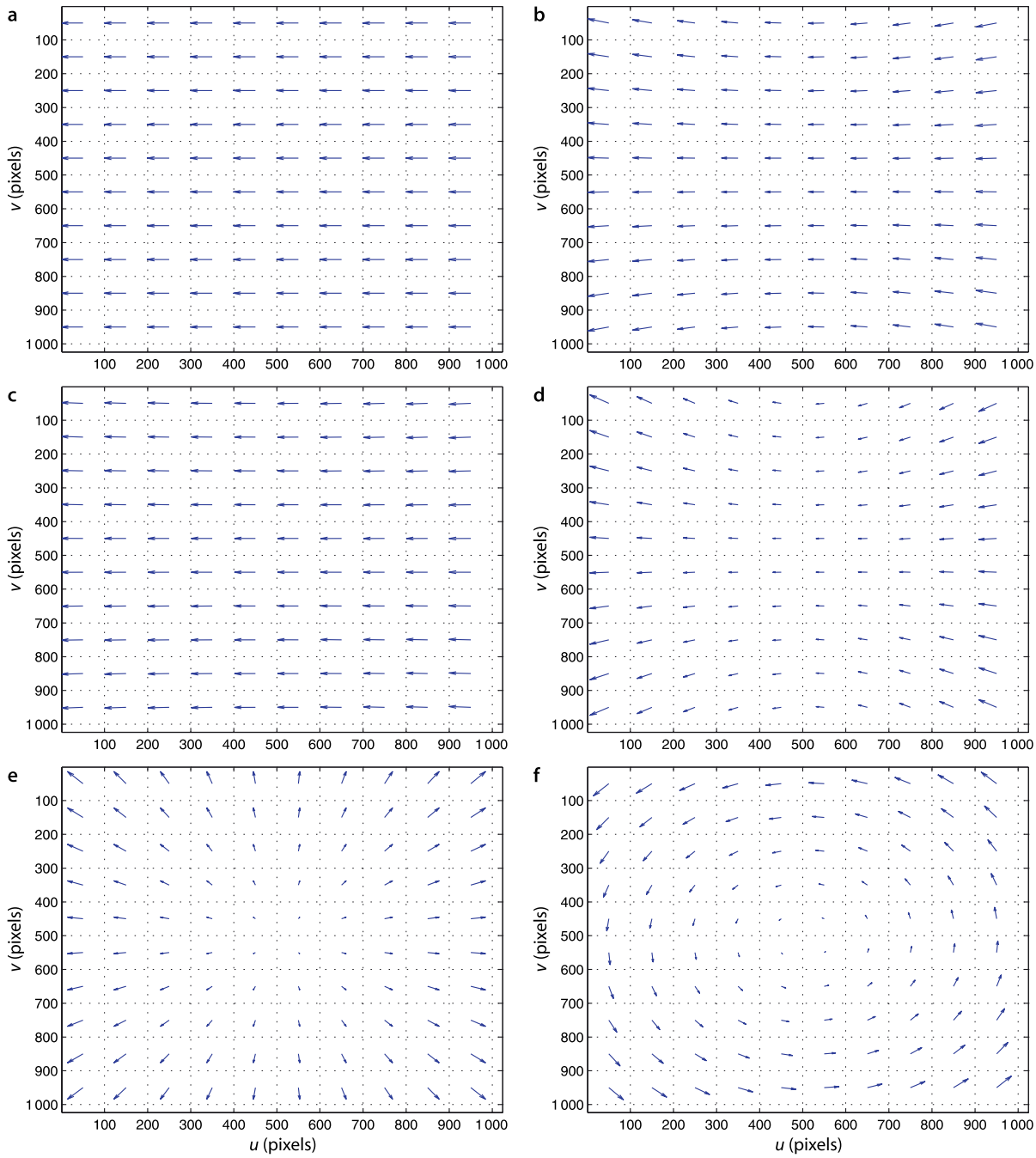


Fig. 15.7. Image plane velocity vectors for canonic camera velocities where all corresponding world points lie in a fronto-parallel plane. **a** x-axis translation; **b** y-axis rotation, $f = 8$ mm; **c** y-axis rotation, $f = 20$ mm; **d** y-axis rotation, $f = 4$ mm; **e** z-axis translation; **f** z-axis rotation

and as shown in Fig. 15.7d the curvature is much more pronounced. The same applies for columns two and four except for a difference of sign – there is an equivalence between translation in the y-direction and rotation about the x-axis.

The Jacobian matrix has some interesting properties. It does not depend at all on the world coordinates X or Y, only on the image plane coordinates (u, v). However the first three columns depend on the point's depth Z and reflects the fact that for a translating camera the image plane velocity is inversely proportional to depth. You can easily demonstrate this to yourself – translate your head sideways and observe that near objects move more in your field of view than distant objects. However, if you rotate your head all objects, near and far, move equally in your field of view.

The matrix has a rank of two,▸ and therefore has a null-space of dimension four. The null-space comprises a set of spatial velocity vectors that individually, or in any linear combination, cause *no motion* in the image. Consider the simple case of a point in front of the camera on the optical axis

The rank cannot be less than 2, even if $Z \to \infty$.

```
>> J = cam.visjac_p([512; 512], 1)
```

The null-space of the Jacobian is

```
>> null(J)
ans =
         0         0   -0.7071         0
         0    0.7071         0         0
    1.0000         0         0         0
         0    0.7071         0         0
         0         0    0.7071         0
         0         0         0    1.0000
```

The first column indicates that motion in the z-direction, along the ray toward the point, results in no motion in the image. Nor does rotation about the z-axis, as indicated by column four. Columns two and three are more complex, combining rotation and translation. Essentially these exploit the image motion ambiguity mentioned above. Since x-axis translation causes the same image motion as y-axis rotation, column three indicates that if one is positive and the other negative the resulting image motion will be zero – that is translating left and rotating to the right.

We can consider the motion of two points by stacking their Jacobians

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \end{pmatrix} = \begin{pmatrix} J_{p_1} \\ J_{p_2} \end{pmatrix} \boldsymbol{\nu}$$

to give a 4×6 matrix which will have a null-space with two columns. One of these camera motions corresponds to rotation around a line joining the two points.

For three points

$$\begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \end{pmatrix} = \begin{pmatrix} J_{p_1} \\ J_{p_2} \\ J_{p_3} \end{pmatrix} \boldsymbol{\nu} \tag{15.8}$$

the matrix will be non-singular so long as the points are not coincident or collinear.

15.2.2 Controlling Feature Motion

So far we have shown how points move in the image plane as a consequence of camera motion. As is often the case it is the inverse problem that is more useful – *what camera motion is needed in order to move the image features at a desired velocity?*

For the case of three points $\{(u_i, v_i), \quad i = 1 \cdots 3\}$ and corresponding velocities $\{(\dot{u}_i, \dot{v}_i)\}$ we can invert Eq. 15.8

$$\boldsymbol{\nu} = \begin{pmatrix} J_{p_1} \\ J_{p_2} \\ J_{p_3} \end{pmatrix}^{-1} \begin{pmatrix} \dot{u}_1 \\ \dot{v}_1 \\ \dot{u}_2 \\ \dot{v}_2 \\ \dot{u}_3 \\ \dot{v}_3 \end{pmatrix} \tag{15.9}$$

and solve for the required camera velocity.

Given feature velocity we can compute the required camera motion, but how do we determine the feature velocity? The simplest strategy is to use a simple linear controller

$$\dot{\boldsymbol{p}}^* = \lambda(\boldsymbol{p}^* - \boldsymbol{p}) \tag{15.10}$$

that *drives* the features toward their desired values $\boldsymbol{p}^*$ on the image plane. Combined with Eq. 15.9 we write

$$\boldsymbol{\nu} = \lambda \begin{pmatrix} \boldsymbol{J}_{p_1} \\ \boldsymbol{J}_{p_2} \\ \boldsymbol{J}_{p_3} \end{pmatrix}^{-1} (\boldsymbol{p}^* - \boldsymbol{p})$$

That's it! This controller will drive the camera so that the feature points move toward the desired position in the image. It is important to note that nowhere have we required the pose of the camera or of the object,◀ everything has been computed in terms of what can be measured on the image plane.

We do require the depth Z of the point but we will come to that shortly.

For the general case where $N > 3$ points we can stack the Jacobians for all features and solve for camera motion using the pseudo-inverse

$$\boldsymbol{\nu} = \lambda \begin{pmatrix} \boldsymbol{J}_1 \\ \vdots \\ \boldsymbol{J}_N \end{pmatrix}^{+} (\boldsymbol{p}^* - \boldsymbol{p}) \tag{15.11}$$

Note that it is possible to specify a set of feature point velocities which are inconsistent, that is, there is no possible camera motion that will result in the required image motion. In such a case the pseudo-inverse will find a solution that minimizes the norm of the feature velocity error.

For $N \geq 3$ the matrix can be poorly conditioned if the points are nearly co-incident or collinear. In practice this means that some camera motions will cause very small image motions, that is, the motion has low perceptibility. There is strong similarity with the concept of manipulability that we discussed in Sect. 8.1.4 and we take a similar approach in formalizing it. Consider a camera spatial velocity of unit magnitude

$$\boldsymbol{\nu}^T \boldsymbol{\nu} = 1$$

and from Eq. 15.7 we can write the camera velocity in terms of the pseudo-inverse

$$\boldsymbol{\nu} = \boldsymbol{J}_p^+ \dot{\boldsymbol{p}}$$

and substituting yields

$$\begin{aligned} \dot{\boldsymbol{p}}^T \boldsymbol{J}_p^{+^T} \boldsymbol{J}_p^+ \dot{\boldsymbol{p}} &= 1 \\ \dot{\boldsymbol{p}}^T (\boldsymbol{J}_p \boldsymbol{J}_p^T)^{-1} \dot{\boldsymbol{p}} &= 1 \end{aligned}$$

which is the equation of an ellipsoid in the point velocity space. The eigenvectors of $\boldsymbol{J}_p \boldsymbol{J}_p^T$ define the principal axes of the ellipsoid and the singular values of $\boldsymbol{J}_p$ are the radii. The ratio of the maximum to minimum radius is given by the condition number of $\boldsymbol{J}_p$ and indicates the anisotropy of the feature motion. A high value indicates that some of the points have low velocity in response to some camera motions. An alternative to stacking all the point feature Jacobians is to select just three that when stacked result in the best conditioned square matrix which can then be inverted.

Using the Toolbox we start by defining a camera

```
>> cam = CentralCamera('default');
```

The target comprises four points that form a square of side length 0.5 m that lies in the xy-plane and centred at (0, 0, 3)

```
>> P = mkgrid( 2, 0.5, 'T', transl(0,0,3) );
```

and we assume that this pose is unknown to the control system. The desired position of the target features on the image plane are a 400×400 square centred on the principal point

```
>> pStar = bsxfun(@plus, 200*[-1 -1 1 1; -1 1 1 -1], cam.pp');
```

which implicitly has the square target fronto-parallel to the camera.

The camera is at some pose `Tc` so the the pixel coordinates of the world points are

```
>> p = cam.plot(P, 'Tcam', Tc)
```

from which we compute the image plane error

Note that papers based on the task function approach such as Espiau et al. (1992) write this as actual minus demand and write $-\lambda$ in Eq. 15.11 to ensure negative feedback.

```
>> e = pStar - p;
```

and the stacked image Jacobian

```
>> J = visjac_p( ci, p, depth );
```

is a 8×6 matrix in this case since `p` contains four points. The Jacobian does require the point depth which we do not know, so for now we will just choose a constant value. This is an important topic that we will address in Sect. 15.2.3.

The control law determines the required translational and angular velocity of the camera

```
>> v = lambda * pinv(J) * e;
```

where `lambda` is the gain, a positive number, and we take the pseudo-inverse of the non-square Jacobian to implement Eq. 15.11. The resulting velocity is expressed in the camera coordinate frame, and integrating it over a unit time step results in a displacement of the same magnitude. The camera pose is updated by

$$\xi_C\langle k+1\rangle = \xi_C\langle k\rangle \oplus \Delta^{-1}\big(\nu\langle k\rangle\big)$$

where $\Delta^{-1}(\cdot)$ is defined by Eq. 3.12. Using the Toolbox this is implemented as

```
>> Tc = trnorm( Tc * delta2tr(v) );
```

where we ensure that the transformation remains a proper homogeneous transformation by normalizing it using `trnorm`.

For this example we choose the initial pose of the camera in world coordinates as

```
>> Tc0 = transl(1,1,-3)*trotz(0.6);
```

Similar to the PBVS example we create an instance of the `IBVS` class

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar)
```

which is a subclass of the `VisualServo` class and implements the controller outlined above. The option `'T0'` specifies the initial pose of the camera and `'pstar'` specifies the desired image coordinates of the features. The object constructor takes a `CentralCamera` object as its argument, and drives this camera to achieve the desired pose relative to the target. Many additional options can be passed to this class constructor. The display methods shows the coordinates of the world points, the initial absolute pose, the desired image plane feature coordinates. The simulation is run by

```
>> ibvs.run();
```

which repeatedly calls the `step` method to execute a single time step. The simulation animates both the image plane of the camera and the 3-dimensional visualization of the camera and the world points.

The simulation results are stored within the object for later analysis. We can plot the path of the target features on the image plane, the Cartesian velocity versus time or Cartesian position versus time

```
>> ibvs.plot_p();
>> ibvs.plot_vel();
>> ibvs.plot_camera();
>> ibvs.plot_jcond();
```

which are shown in Fig. 15.8. We see that the feature points have followed an almost straight-line path in the image, and the Cartesian position has changed smoothly toward the final value. The condition number of the image Jacobian decreases over the motion indicating that the Jacobian is becoming better conditioned, and this is a consequence of the features moving further apart.

How is $\boldsymbol{p}^*$ determined? The image points can be found by demonstration, by moving the camera to the desired pose and recording the observed image coordinates. Alternatively, if the camera calibration parameters and the target geometry are known the desired image coordinates can be computed for any specified goal pose. Note that this calculation, world point point projection, is computationally cheap and is performed only once before visual servoing commences.

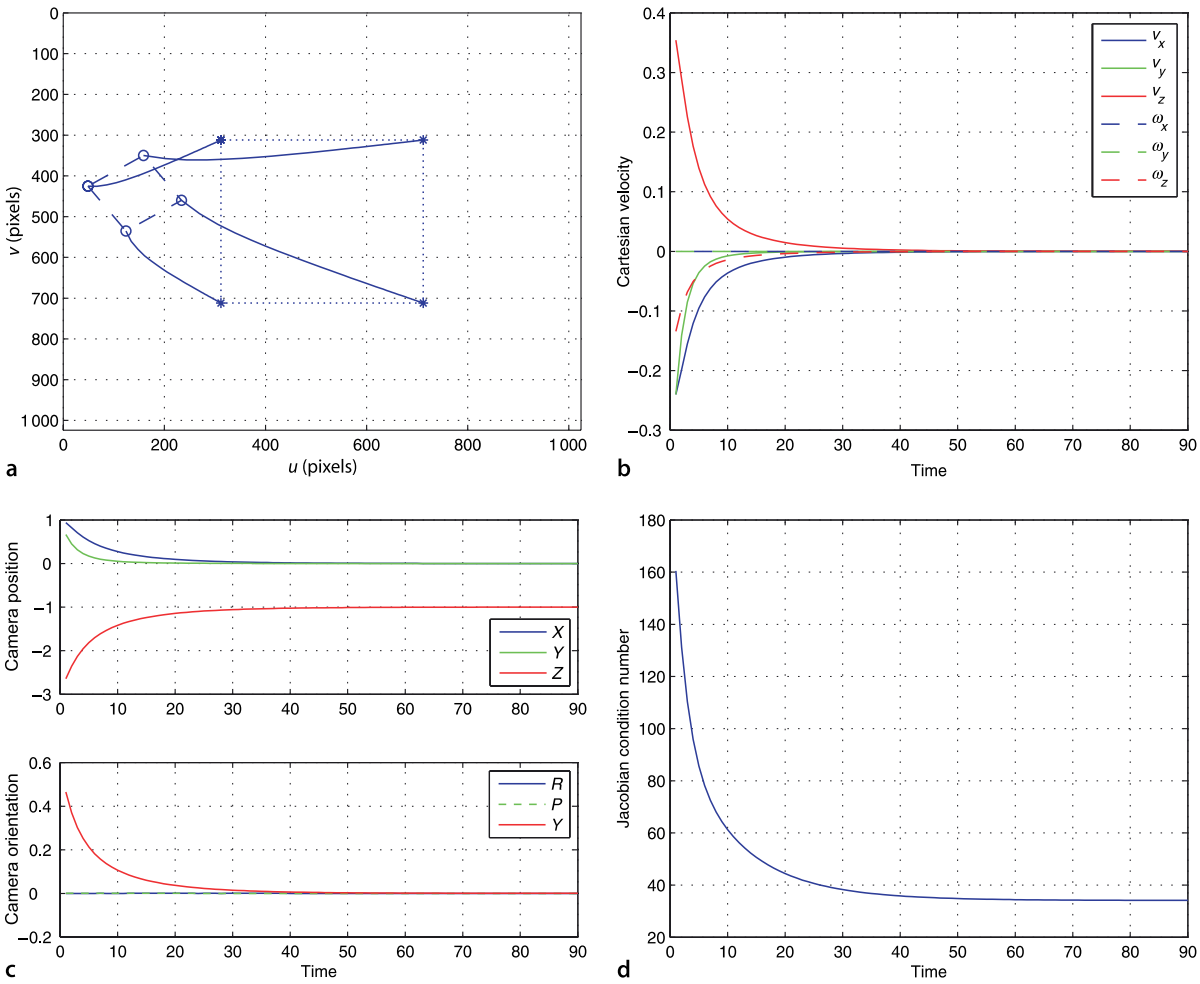


Fig. 15.8. Results of IBVS simulation, created by `IBVS`. **a** Image plane motion, ∗ is desired, ○ is initial, **b** spatial velocity components; **c** camera pose; **d** image Jacobian condition number

The IBVS system can also be expressed in terms of a Simulink® model

```
>> sl_ibvs
```

which is shown in Fig. 15.9. The simulation is run by

```
>> r = sim('sl_ibvs')
```

and the camera pose, image plane feature error and camera velocity are animated. Scope blocks also plot the camera velocity and feature error against time. The initial pose of the camera is set by a parameter of the `pose` block, and the world points are parameters of the `camera` block. The `CentralCamera` object is a parameter to both the `camera` and visual `Jacobian` blocks.

The simulation results are stored in the simulation output object `r`. For example the camera velocity is the second recorded signal[▸]

It is connected to the Output block number 2.

```
>> t = r.find('tout');
>> v = r.find('yout').signals(2).values;
>> about(v)
v [double] : 501x6 (24048 bytes)
```

which has one row for every simulation step, and the columns are the camera spatial velocity components. We can plot camera velocity against time

```
>> plot(t, v)
```

The image plane coordinates are also logged

```
>> p = r.find('yout').signals(1).values;
>> about(p)
p [double] : 2x4x1001 (64064 bytes)
```

which can be plotted by

```
>> plot2(p)
```

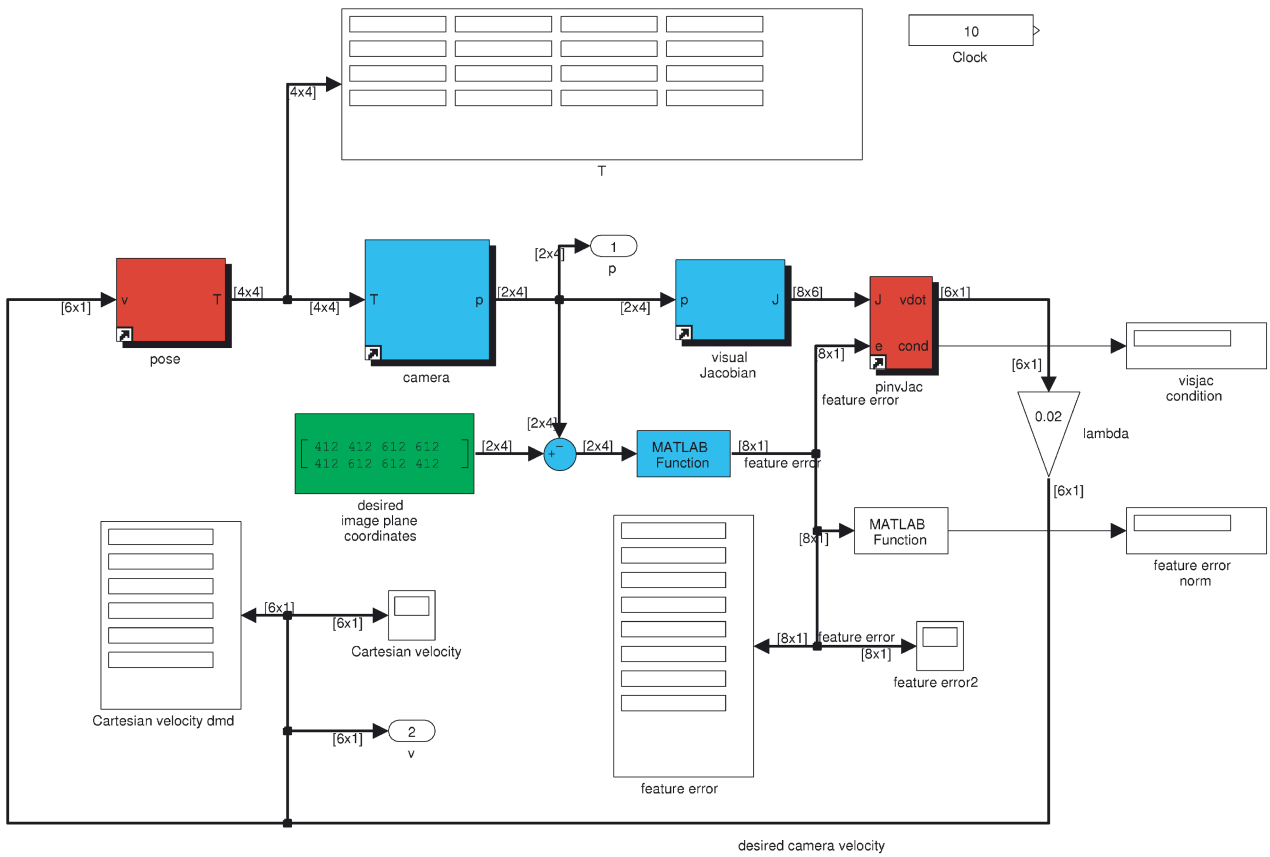


Fig. 15.9. The Simulink® model `sl_ibvs` drives the feature points to the desired positions on the image plane. The initial camera pose is set in the `pose` block and the desired image plane points $\boldsymbol{p}^*$ are set in the green constant block

15.2.3 Depth

Computing the image Jacobian requires knowledge of the camera intrinsics, the principal point and focal length, but in practice it is quite tolerant to errors in these. The Jacobian also requires knowledge of Z_i, the distance to, or the depth of, each point. In the simulations just discussed we have assumed that depth is known – this is easy in simulation but not so in reality. Fortunately, in practice we find that IBVS is remarkably tolerant to errors in Z.

A number of approaches have been proposed to deal with the problem of unknown depth. The simplest is to just assume a constant value for the depth which is quite reasonable if the required camera motion is approximately in a plane parallel to the plane of the object points. To evaluate the performance of different constant estimates of point depth, we can compare the effect of choosing $z = 1$ and $z = 10$ for the example above

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depth', 1)
>> ibvs.run(50)
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depth', 10)
>> ibvs.run(50)
```

and the results are plotted in Fig. 15.10. We see that the image plane paths are no longer straight, because the Jacobian is now a poor approximation of the relationship between the camera motion and image feature motion. We also see that for $Z = 1$ the convergence is much slower than for the $Z = 10$ case. The Jacobian for $Z = 1$ overestimates the optical flow, so the inverse Jacobian underestimates the required camera velocity. Nevertheless, for quite significant errors, the true depth is $Z = 3$, IBVS has converged. For the $Z = 10$ case the displacement at each timestep is large leading to a very jagged path.

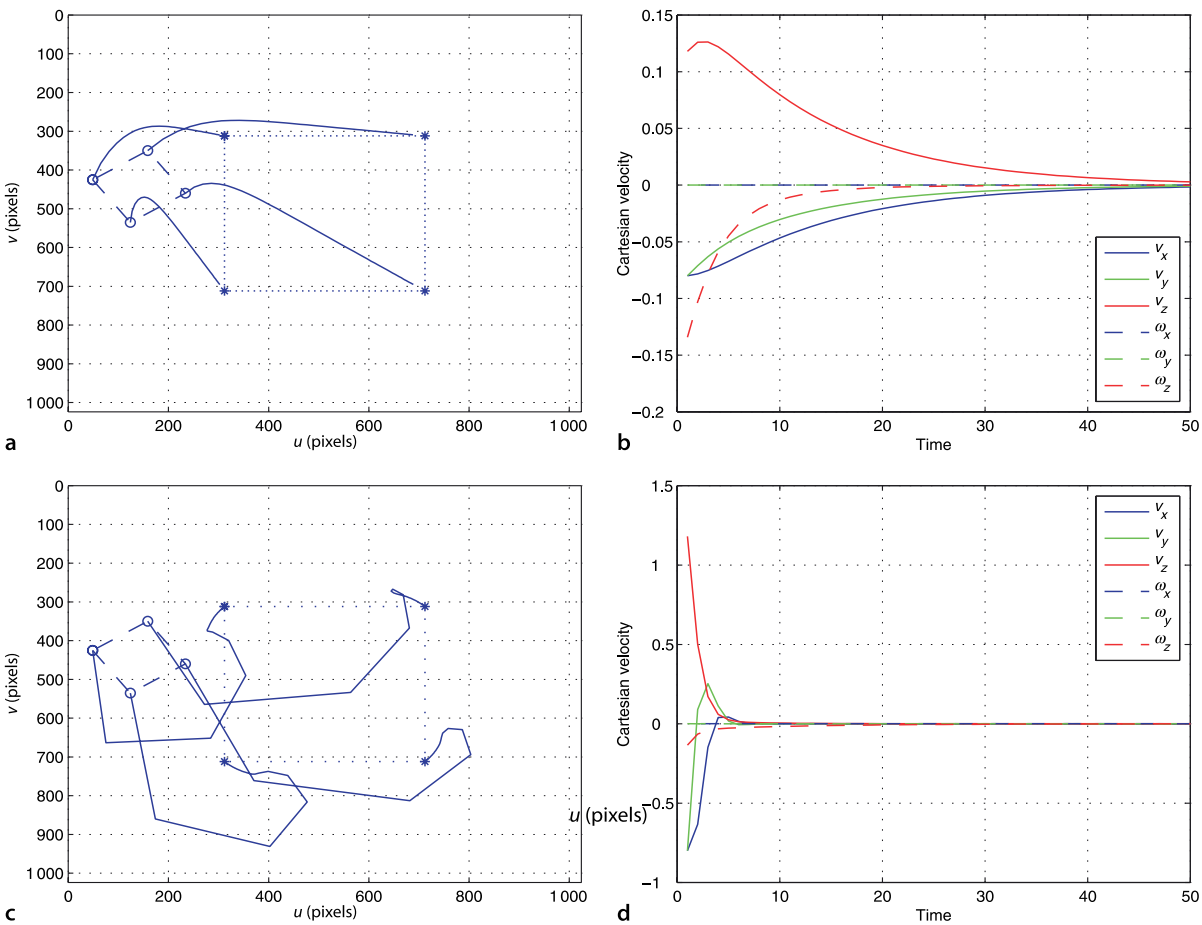


Fig. 15.10. Results of IBVS with different constant estimates of point depth: **a, b** Image and camera motion for $Z = 1$; **c, d** Image and camera motion for $Z = 10$

A second approach is to use standard computer vision techniques to estimate the value for Z. If the camera intrinsic parameters were known we could use sparse stereo techniques from consecutive camera positions to estimate the depth of each feature point.

A third approach is to estimate the value of Z online using measurements of robot and image motion. We can create a simple depth estimator by rearranging Eq. 15.6 into estimation form

$$\begin{aligned}\begin{pmatrix}\dot{u}\\ \dot{v}\end{pmatrix} &= \left(\begin{array}{ccc|ccc} -\frac{f}{\rho_u Z} & 0 & \frac{\bar{u}}{Z} & \frac{\rho_u \bar{u}\bar{v}}{f} & -\frac{f^2+\rho_u^2\bar{u}^2}{\rho_u f} & \bar{v} \\ 0 & -\frac{f}{\rho_v Z} & \frac{\bar{v}}{Z} & \frac{f^2+\rho_v^2\bar{v}^2}{\rho_v f} & -\frac{\rho_v \bar{u}\bar{v}}{f} & -\bar{u} \end{array}\right)\begin{pmatrix}\boldsymbol{v}\\ \omega\end{pmatrix} \\ &= \left(\tfrac{1}{Z}\boldsymbol{J}_t \mid \boldsymbol{J}_\omega\right)\begin{pmatrix}\boldsymbol{v}\\ \omega\end{pmatrix} \\ &= \frac{1}{Z}\boldsymbol{J}_t\boldsymbol{v} + \boldsymbol{J}_\omega\omega\end{aligned}$$

which we rearrange as

$$(\boldsymbol{J}_t\boldsymbol{v})\frac{1}{Z} = \begin{pmatrix}\dot{u}\\ \dot{v}\end{pmatrix} - \boldsymbol{J}_\omega\omega \tag{15.12}$$

The right-hand side is the observed optical flow from which the expected optical flow due to rotation of the camera is subtracted – a process referred to as derotating optical flow. The remaining optical flow, after subtraction, is only due to translation. Writing Eq. 15.12 in compact form

$$\mathbf{A}\boldsymbol{\theta} = \mathbf{b} \tag{15.13}$$

we have a simple linear equation with one unknown parameter $\theta = 1/Z$ which can be solved using least-squares.

In our example we can enable this by

```
>> ibvs = IBVS(cam, 'T0', Tc0, 'pstar', pStar, 'depthest')
>> ibvs.run()
>> ibvs.plot_z()
>> ibvs.plot_p()
```

and the result is shown in Fig. 15.11. Figure 15.11b shows the estimated and true point depth versus time. The estimate depth was initially zero, a poor choice, but it has risen rapidly and then tracked the actual target depth and then tracked it accurately as the controller converges. Figure 15.11a shows the feature motion, and we see that the features initially move in the wrong direction because of the error in depth.

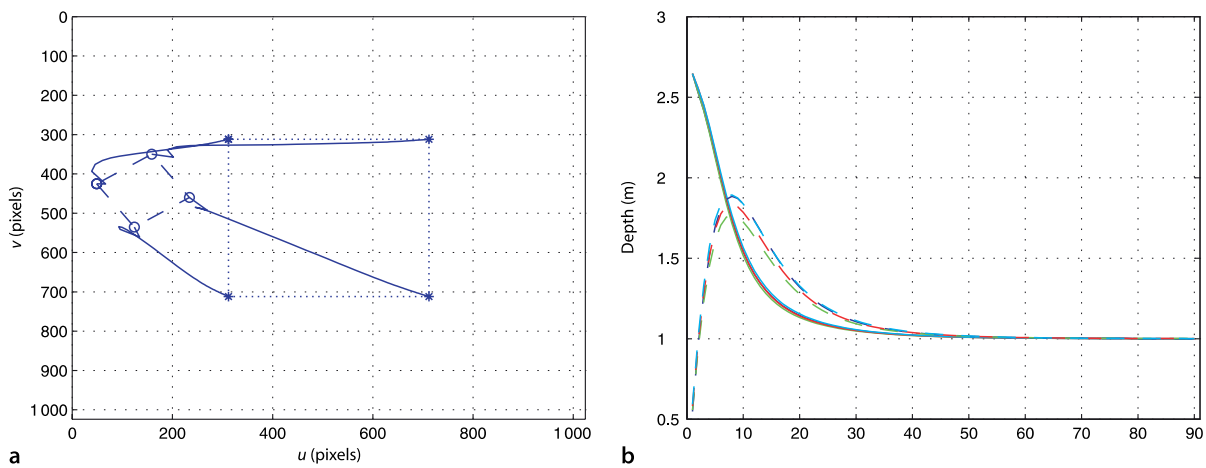


Fig. 15.11. IBVS with online depth estimator. **a** Feature paths; **b** comparison of estimated (dashed) and true depth (solid) for all four points

15.2.4 Performance Issues

The control law for PBVS is defined in terms of the 3-dimensional workspace so there is no mechanism by which the motion of the image features is directly regulated. For the PBVS example shown in Fig. 15.5 the feature points followed a curved path on the image plane, and therefore it is possible that they could leave the camera's field of view. For a different initial camera pose

```
>> pbvs.T0 = transl(-2.1, 0, -3)*trotz(5*pi/4);
>> pbvs.run()
```

the result is shown in Fig. 15.12a and we see that two of the points move outside the image which would cause the PBVS control to fail. By contrast the IBVS control for the same initial pose

```
>> ibvs = IBVS(cam, 'T0', pbvs.T0, 'pstar', pStar, 'lambda',
0.002, 'niter', Inf, 'eterm', 0.5)
>> ibvs.run()
>> ibvs.plot_p();
```

gives the feature trajectories shown in Fig. 15.12b.

Conversely for image-based visual servo control there is no direct control over the Cartesian motion of the camera. This can sometimes result in surprising motion, particularly when the target is rotated about the z-axis

```
>> ibvs = IBVS(cam, 'T0', transl(0,0, -1)*trotz(1), 'pstar',
pStar);
>> ibvs.run()
>> ibvs.plot_camera
```

which is shown in Fig. 15.13(top). We see that the camera has performed an unnecessary translation along the z-axis – away from the target and back again. This phenomenon is termed camera retreat. The resulting motion is not time optimal and can require large and possibly unachievable camera motion. An extreme example arises for a pure rotation about the optical axis by π rad

```
>> ibvs = IBVS(cam, 'T0',  transl(0,0, -1)*trotz(pi), ...
    'pstar', pStar, 'niter', 10);
>> ibvs.run()
>> ibvs.plot_camera
```

which is shown in Fig. 15.13 (bottom). The feature points are, as usual, moving in a straight line toward their desired values, but for this problem the paths all pass through the origin which is a singularity and where IBVS will fail. The only way the target points can be at the origin in the image is if the camera is at negative infinity, and that is where it is headed!

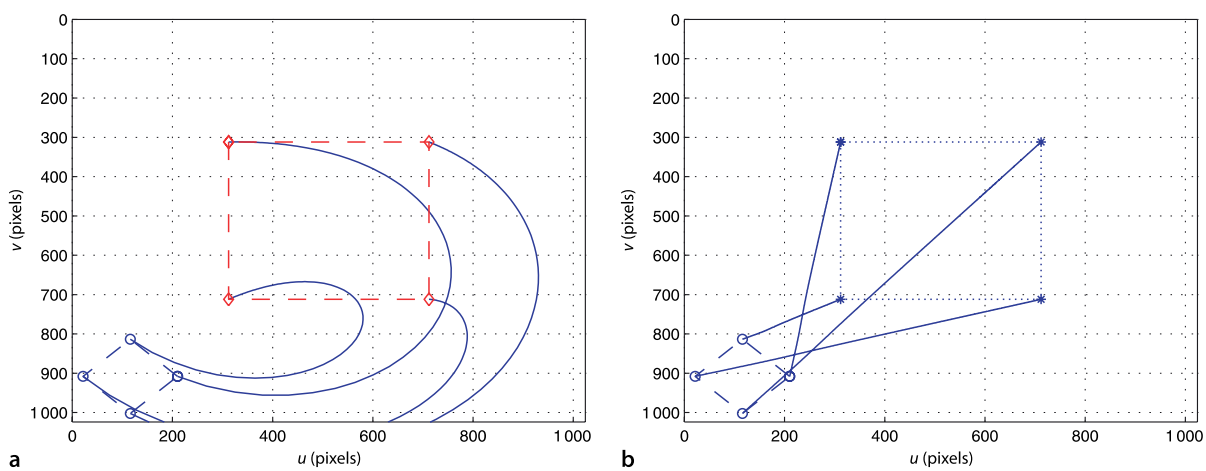


Fig. 15.12. Image plane feature paths for **a** PBVS and **b** IBVS

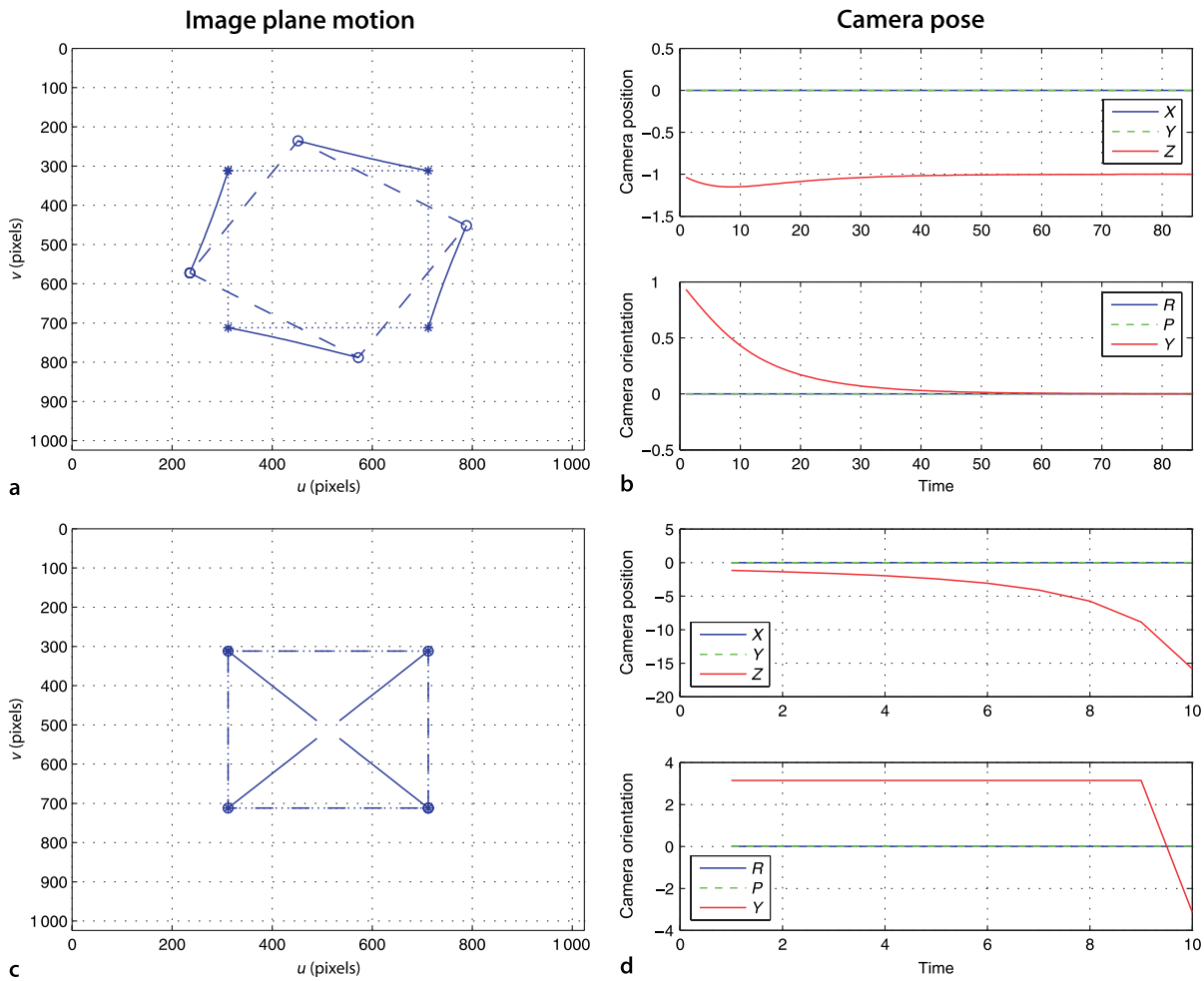


Fig. 15.13. IBVS for pure target rotation about the optical axis. **a, b** for rotation of 1 rad; **c, d** for rotation of π rad

A final consideration is that the image Jacobian is a linearization of a highly nonlinear system. If the motion at each time step is large then the linearization is not valid and the features will follow curved rather than linear paths in the image, as we saw in Fig. 15.10. This can occur if the desired feature positions are a long way from the initial positions and/or the gain λ is too high. One solution is to limit the maximum norm of the commanded velocity

$$\boldsymbol{\nu} = \begin{cases} \boldsymbol{\nu}_{\max} \frac{\boldsymbol{\nu}}{|\boldsymbol{\nu}|} & |\boldsymbol{\nu}| > \boldsymbol{\nu}_{\max} \\ \boldsymbol{\nu} & |\boldsymbol{\nu}| \leq \boldsymbol{\nu}_{\max} \end{cases}$$

The feature paths do not have to be straight lines and nor do the features have to move with asymptotic velocity – we have used these only for simplicity. Using the trajectory planning methods of Chap. 3 the features could be made to follow any arbitrary trajectory in the image and to have an arbitrary speed versus time profile.

In summary, IBVS is a remarkably robust approach to vision-based control. We have seen that it is quite tolerant to errors in the depth of points. We have also shown that it can produce less than optimal Cartesian paths for the case of large rotations about the optical axis. We will discuss remedies to these problems in the next chapter.

15.3 Using Other Image Features

So far we have considered only point features. In a real system we would use the feature extraction techniques discussed in Chap. 13 and the points would be the centroids of distinct regions, or Harris or SURF corner features. The points would then be used for pose estimation in a PBVS scheme, or directly in an IBVS scheme. For both PBVS or IBVS we need to solve the correspondence problem, that is, for each observed feature we must determine which desired image plane coordinate it corresponds to. IBVS can also be formulated to work with other image features such as lines, as found by the Hough transform, or the shape of an ellipse.

15.3.1 Line Features

For a line the Jacobian is written in terms of the (ρ, θ) parameterization that we used for the Hough transform in Sect. 13.2

$$\begin{pmatrix} \dot{\theta} \\ \dot{\rho} \end{pmatrix} = \boldsymbol{J}_l \boldsymbol{\nu}$$

and the Jacobian is

$$\boldsymbol{J}_l = \begin{pmatrix} \lambda_\theta \sin\theta & \lambda_\theta \cos\theta & -\rho\lambda_\theta & -\rho\sin\theta & -\rho\cos\theta & -1 \\ \lambda_\rho \sin\theta & \lambda_\rho \cos\theta & -\lambda_\rho\rho & -\cos\theta(1+\rho^2) & \sin\theta(1+\rho^2) & 0 \end{pmatrix}$$

where $\lambda_\theta = (a\cos\theta - b\sin\theta)/d$ and $\lambda_\rho = -(a\rho\sin\theta + b\rho\cos\theta + c)/d$. The Jacobian describes how the line parameters change as a function of camera velocity. Just as the point feature Jacobian required some partial 3-dimensional knowledge, the point depth Z, the line feature Jacobian requires the equation of the plane $aX + bY + cZ + d = 0$ that contains the line. Since each line is the intersection of two planes and therefore lies in two planes we choose the plane for which $d \neq 0$. Like a point feature, a line provides two rows of the Jacobian so we require a minimum of three lines in order to have a Jacobian of full rank.◀

Interestingly a line feature provides two rows of the stacked Jacobian, yet two points which define a line would provide four rows.

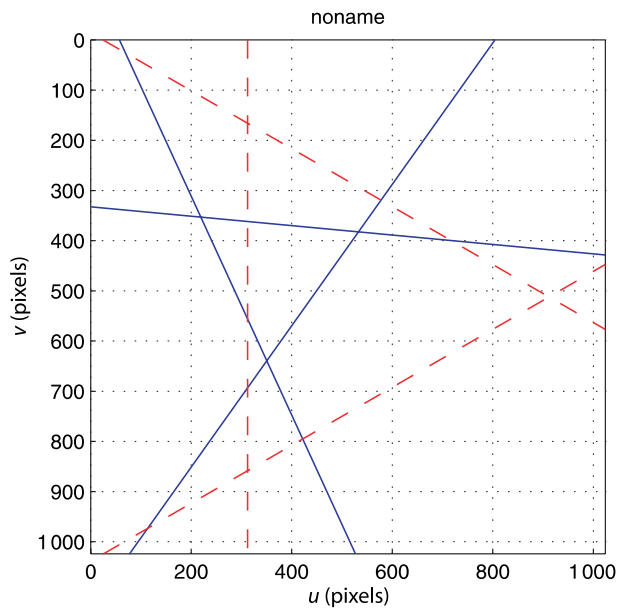


Fig. 15.14. IBVS using line features. The image plane showing the three current lines (solid) and desired (dashed)

We illustrate this with an example comprising three lines that all lie in the plane $Z = 3$, and we construct three points in that plane using the `circle` function with just three boundary points

```
>> P = circle([0 0 3], 0.5, 'n', 3);
```

and use the familiar `CentralCamera` class methods to project these to the image. For each pair of points we compute the equations of the line

$$\tan\theta = \frac{v_2 - v_1}{u_1 - u_2},\ \rho = u_1 \sin\theta + v_1 \cos\theta$$

The simulation is run in familiar fashion

```
>> ibvs = IBVS_l(cam, 'example');
>> ibvs.run()
```

and a snapshot of results is shown in Fig. 15.14. Note that we need to perform correspondence between the observed and desired lines.

15.3.2 Circle Features

A circle in the world will be projected, in the general case, to an ellipse in the image which is described by

This is different to the representation of an ellipse given in Appendix E, but the two forms are simply related by constant scale factors applied to the coefficients.

$$u^2 + E_1 v^2 - 2E_2 uv + 2E_3 u + 2E_4 v + E_5 = 0 \tag{15.14}$$

where E_i are parameters of the ellipse. The rate of change of the ellipse coefficients is related to camera velocity by

$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \end{pmatrix} = \boldsymbol{J}_e(\boldsymbol{E}, \rho)\boldsymbol{\nu}$$

where the Jacobian is

$$\boldsymbol{J}_e(\boldsymbol{E},\rho) = \begin{pmatrix} 2bE_2 - 2aE_1 & 2E_1(b - aE_2) & 2bE_4 - 2aE_1E_3 & 2E_4 & 2E_1E_3 & -2E_2(E_1+1) \\ b - aE_2 & bE_2 - a(2E_2^2 - E_1) & a(E_4 - 2E_2E_3) + bE_3 & E_3 & 2E_2E_3 - E_4 & E_1 - 2E_2^2 - 1 \\ c - aE_3 & a(E_4 - 2E_2E_3) + cE_2 & cE_3 - a(2E_3^2 - E_5) & -E_2 & 1 + 2E_3^2 - E_5 & E_4 - 2E_2E_3 \\ E_3b + E_2c - 2aE_4 & E_4b + E_1c - 2aE_2E_4 & bE_5 + cE_4 - 2aE_3E_4 & E_5 - E_1 & 2E_3E_4 + E_2 & -2E_2E_4 - E_3 \\ 2cE_3 - 2aE_5 & 2cE_4 - 2aE_2E_5 & 2cA5 - 2aE_3E_5 & -2E_4 & 2E_3E_5 + 2E_3 & -2E_2E_5 \end{pmatrix}$$

and where $\rho = (\alpha, \beta, \gamma)$ defines a plane in world coordinates $aX + bY + cZ + d = 0$ in which the ellipse lies and $\alpha = -a/d$, $\beta = -b/d$ and $\gamma = -c/d$. Just as was the case for point and line feature Jacobians we need to provide some depth information about the target. The Jacobian normally has a rank of five, but this drops to three when the projection is of a circle centred in the image plane, and a rank of two if the circle is a point.

An advantage of the ellipse feature is that the ellipse can computed from the set of all boundary points without needing to solve the correspondence problem. The ellipse feature can also be computed from the moments of all the points within the ellipse boundary. We illustrate this with an example of a circle comprising ten points around its circumference

```
>> P = circle([0 0 3], 0.5, 'n', 10);
```

and the `CentralCamera` class project these to the image plane.

```
>> p = cam.project(P, 'Tcam', Tc);
```

where `Tc` is the current camera pose and we convert to normalized image coordinates

```
>> p = homtrans( inv(cam.K), p );
```

The parameters of an ellipse are calculated using the methods of Appendix E

```
>> a = [y.^2; -2*x.*y; 2*x; 2*y; ones(1,numcols(x))]';
>> b = -(x.^2)';
>> E = a\b;
```

which returns a 5-vector of ellipse parameters. The image Jacobian for an ellipse feature is computed by a method of the `CentralCamera` class

```
>> J = cam.visjac_e(E, plane);
```

where the plane containing the circle must also be specified. For this example the plane is $Z = 3$ so `plane = [0 0 1 -3]`.

The Jacobian is 5×6 and has a maximum rank of only 5 so we cannot uniquely solve for the camera velocity. We have at least two options. Firstly, if our final view is of a circle then we may not be concerned about rotation around the centre of the circle, and in this case we can delete the sixth column of the Jacobian to make it square and set ω_z to zero. Secondly, and the approach taken in this example, is to combine the features for the ellipse and a single point◂

Here we arbitrarily choose the first point, any one will do.

$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \\ \dot{u}_1 \\ \dot{v}_1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{J}_e(\boldsymbol{E}) \\ \boldsymbol{J}_p(\boldsymbol{p}_1) \end{pmatrix} \boldsymbol{\nu}$$

and the stacked Jacobian is now 7×6 and we can solve for camera velocity. As for the previous IBVS examples the desired velocity is proportional to the difference between the current and desired feature values

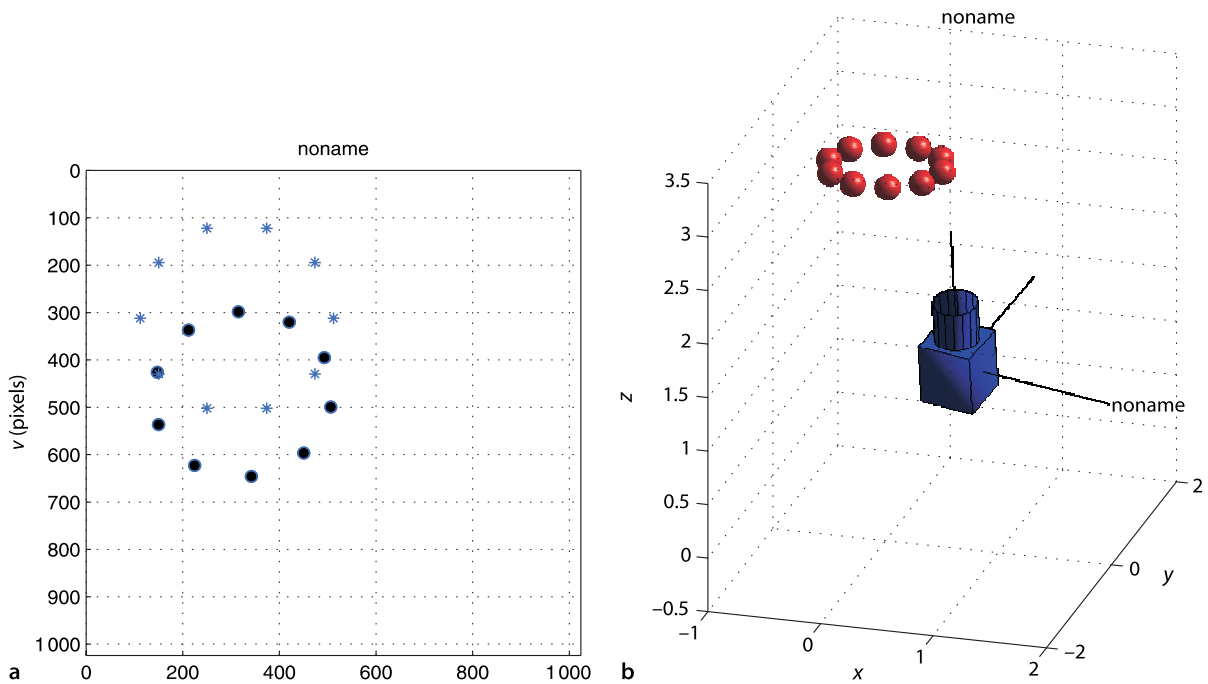


Fig. 15.15. IBVS using ellipse feature. **a** The image plane showing the current points (solid) and demanded (∗); **b** a world view showing the points and the camera

$$\begin{pmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \vdots \\ \dot{E}_5 \\ \dot{u}_1 \\ \dot{v}_1 \end{pmatrix} = \lambda \begin{pmatrix} E_1^* - E_1 \\ E_2^* - E_2 \\ \vdots \\ E_5^* - E_5 \\ u_1^* - u_1 \\ v_1^* - v_1 \end{pmatrix}$$

The simulation is run in the now familiar fashion

```
>> ibvs = IBVS_e(cam, 'example');
>> ibvs.run()
```

and a snapshot of results is shown in Fig. 15.15.

15.4 Wrapping Up

In this chapter we have learnt about the fundamentals of vision-based robot control, and the fundamental techniques developed over two decades up to the mid 1990s. There are two distinct configurations. The camera can be attached to the robot observing the target, eye-in-hand, or fixed in the world observing both robot and target. Another form of distinction is the control structure: Position-Based Visual Servo (PBVS) and Image-Based Visual Servo (IBVS). The former involves pose estimation based on a calibrated camera and a geometric model of the target, while the latter performs the control directly in the image plane. Each approach has certain advantages and disadvantages. PBVS performs efficient straight-line Cartesian camera motion in the world but may cause image features to leave the image plane. IBVS always keeps features in the image plane but may result in trajectories that exceed the reach of the robot, particularly if it requires a large amount of rotation about the camera's optical axis. IBVS also requires a touch of 3-dimensional information, the depth of the feature points, but is quite robust to errors in depth and it is quite feasible to estimate the depth as the robot moves. IBVS can be formulated to work with not only point features, but also for lines and ellipses.

So far in our simulations we have determined the required camera velocity and moved the camera accordingly, without consideration of the mechanism to move it. In the next chapter we consider cameras attached to arm-type robots, mobile ground robots and flying robots.

Further Reading

The tutorial paper by Hutchinson et al. (1996) was the first comprehensive articulation and taxonomy of the field. More recent articles by Chaumette and Hutchinson (2006) and Siciliano and Khatib (2008, § 24) provide excellent coverage of the fundamentals of visual servoing. Chapters on visual servoing are included in recent textbooks by Spong et al. (2006, § 12) and Siciliano et al. (2008, § 10).

The 1993 book edited by Hashimoto (1993) was the first collection of papers covering approaches and applications in visual servoing. The 1996 book by Corke (1996b) is now out of print but available free online and covers the fundamentals of robotics and vision for controlling the dynamics of an image-based visual servoing system. It contains an extensive, but dated, collection of references to visual servoing applications including industrial applications, camera control for tracking, high-speed planar micro-manipulator, road vehicle guidance, aircraft refuelling, and fruit picking. Another important collection of papers (Kriegman et al. 1998) stems from a 1998 workshop on the synergies between control and vision: how vision can be used for control and how

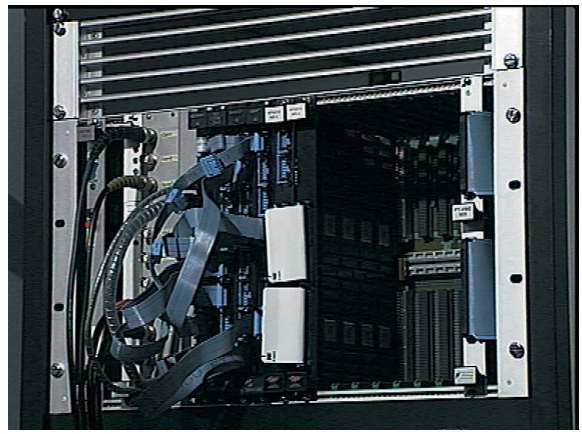


Fig. 15.16. A 19 inch VMEbus rack of hardware image processing cards, capable of 10 Mpix s^{-1} throughput or framerate for 512×512 images. Used by the author circa in the early 1990s

control can be used for vision. More recent algorithmic developments and application are covered in a collection of workshop papers by Chesi and Hashimoto (2010).

Visual servoing has a very long history – the earliest reference is by Shirai and Inoue (1973) who describe how a visual feedback loop can be used to correct the position of a robot to increase task accuracy. They demonstrated a system with a servo cycle time of 10 s, and this highlights a harsh reality for the field which has been the problem of real-time feature extraction. Until the late 1990s this required bulky and expensive special-purpose hardware such as that shown in Fig. 15.16. Other significant early work on industrial applications occurred at SRI International during the late 1970s (Hill and Park 1979; Makhlin 1985).

In the 1980s Weiss et al. (1987) introduced the classification of visual servo structures as either position-based or image-based. They also introduced a distinction between visual servo and dynamic look and move, the former uses only visual feedback whereas the latter uses joint feedback and visual feedback. This latter distinction is now longer in common usage and most visual servo systems today make use of joint-position *and* visual feedback. Weiss (1984) applied adaptive control techniques for IBVS of a robot arm without joint-level feedback, but the results were limited to low degree of freedom arms due to the low-sample rate vision processing available at that time. Others have looked at incorporating the manipulator dynamics Eq. 9.1 into controllers that command motor torque directly (Kelly 1996; Kelly et al. 2002a,b) but all still require joint angles in order to evaluate the manipulator Jacobian, and the joint rates to provide damping. Control and stability in closed-loop visual control systems was addressed by several researchers (Corke and Good 1992; Espiau et al. 1992; Papanikolopoulos et al. 1993) and feedforward predictive, rather than feedback, controllers were proposed by Corke (1994) and Corke and Good (1996).

Feddema (Feddema and Mitchell 1989; Feddema 1989) used closed-loop joint control to overcome problems due to low visual sampling rate and demonstrated IBVS for 4-DOF. Chaumette, Rives and Espiau (Chaumette et al. 1991; Rives et al. 1989) describe a similar approach using the task function method (Samson et al. 1990) and show experimental results for robot positioning using a target with four features. Feddema et al. (1991) describe an algorithm to select which subset of the available features give the best conditioned square Jacobian. Hashimoto et al. (1991) have shown that there are advantages in using a larger number of features and using a pseudo-inverse to solve for velocity.

It is well known that IBVS is very tolerant to errors in depth and its effect on control performance is examined in detail in Marey and Chaumette (2008). Feddema and Mitchell (1989) performed a partial 3D reconstruction to determine point depth based on observed features and known target geometry. Papanikolopoulos and Khosla (1993) described adaptive control techniques to estimate depth, as used in this chapter. Hosoda and Asada (1994), Jägersand et al. (1996) and Piepmeier et al. (1999) have shown how the image Jacobian matrix itself can be estimated online from measurements of robot and image motion.

The most common image Jacobian is based on the motion of points in the image, but it can also be derived for the parameters of lines in the image plane (Chaumette 1990; Espiau et al. 1992) and the parameters of an ellipse in the image plane (Espiau et al. 1992). More recently moments have been proposed for visual servoing of planar scenes (Chaumette 2004; Tahri and Chaumette 2005).

The literature on PBVS is much smaller, but the paper by Westmore and Wilson (1991) is a good introduction. They use an EKF to implicitly perform pose estimation, the target pose is the filter state and the innovation between predicted and feature coordinates updates the target pose state. Hashimoto et al. (1991) present simulations to compare position-based and image-based approaches.

Visual servoing has been applied to a diverse range of problems that normally require human hand-eye skills such as ping-pong (Andersson 1989), juggling (Rizzi and Koditschek 1991) and inverted pendulum balancing (Dickmanns and Graefe 1988a; Andersen et al. 1993), catching (Sakaguchi et al. 1993; Buttazzo et al. 1993; Bukowski et al. 1991; Skofteland and Hirzinger 1991; Skaar et al. 1987; Lin et al. 1989), and controlling a labyrinth game (Andersen et al. 1993).

Exercises

1. Position-based visual servoing
   a) Run the PBVS example. Experiment with varying parameters such as the initial camera pose, the path fraction λ and adding pixel noise to the output of the camera.
   b) Create a Simulink® model for PBVS.
   c) Use a different camera model for the pose estimation (slightly different focal length or principal point) and observe the effect on final end-effector pose.
   d) Implement an EKF based PBVS system as described in Westmore and Wilson (1991).
2. Optical flow fields
   a) Plot the optical flow fields for cameras with different focal lengths.
   b) Plot the flow field for some composite camera motions such as x- and y-translation, x- and z-translation, and x-translation and z-rotation.
3. For the case of two points the image Jacobian is 4×6 and the nullspace has two columns. What camera motions do they correspond to?
4. Image-based visual servoing
   a) Run the IBVS example, either command line or Simulink® version. Experiment with varying the gain λ. Remember that λ can be a scalar or a diagonal matrix which allows different gain settings for each degree of freedom.
   b) Implement the function to limit the maximum norm of the commanded velocity.
   c) Experiment with adding pixel noise to the output of the camera.
   d) Experiment with different initial camera poses and desired image plane coordinates.
   e) Experiment with different number of target points, from three up to ten. For the cases where $N > 3$ compare the performance of the pseudo-inverse with just selecting a subset of three points (first three or random three). Can you design an algorithm that chooses a subset of points which results in the stacked Jacobian with the best condition number?
   f) Create a set of desired image plane points that form a rectangle rather than a square. There is no perspective viewpoint from which a square appears as a rectangle. What does the IBVS system do?
   g) Create a set of desired image plane points that cannot be reached, for example swap two adjacent world or image points. What does the IBVS system do?
   h) Use a different camera model for the image Jacobian (slightly different focal length or principal point) and observe the effect on final end-effector pose.

   i) For IBVS we generally force points to move in straight lines but this is just a convenience. Use a trajectory generator to move the points from initial to desired position with some sideways motion, perhaps a half or full cycle of a sine wave. What is the effect on camera Cartesian motion?
5. Derive the image Jacobian for a pan/tilt camera head.
6. When discussing motion perceptibility we used the identity $(\boldsymbol{J}_p{}^+)^T\boldsymbol{J}_p{}^+ = (\boldsymbol{J}_p\boldsymbol{J}_p^T)^{-1}$. Prove this. Hint, use the singular value decomposition $\boldsymbol{J} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ and remember that $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices.
7. End-point open-loop visual servo systems have not been discussed in this book. Consider a group of target points on the robot end-effector as well as the those on the target object, both being observed by a single camera (challenging).
   a) Create an end-point open-loop PBVS system.
   b) Use a different camera model for the pose estimation (slightly different focal length or principal point) and observe the effect on final end-effector relative pose.
   c) Create an end-point open-loop IBVS system.
   d) Use a different camera model for the image Jacobian (slightly different focal length or principal point) and observe the effect on final end-effector relative pose.
8. Run the line-based visual servo example.
9. Ellipse-based visual servo
   a) Run the ellipse-based visual servo example.
   b) Modify to servo five degrees of camera motion using just the ellipse parameters (without the point feature).
   c) For an arbitrary shape we can compute its equivalent ellipse which is expressed in terms of an inertia matrix and a centroid. Determine the ellipse parameters of Eq. 15.14 from the inertia matrix and centroid. Create an ellipse-feature visual servo to move to a desired view of the arbitrary shape (challenging).