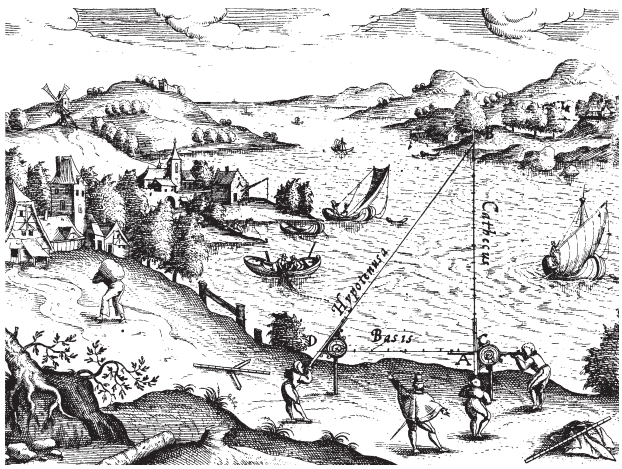
Chapter

14 Using Multiple Images



In the previous chapter we learnt about corner detectors which find particularly distinctive *points* in a scene. These points can be reliably detected in different views of the same scene irrespective of viewpoint or lighting conditions. Such points are characterised by high image gradients in orthogonal directions and typically occur on the corners of objects. However the 3-dimensional coordinate of the corresponding world point was lost in the perspective projection process which we discussed in Chap. 11 – we mapped a 3-dimensional world point to a 2-dimensional image coordinate. All we know is that the world point lies along some ray in space corresponding to the pixel coordinate, as shown in Fig. 11.1. To recover the missing third dimension we need additional information. In Sect. 11.2.3 the additional information was camera calibration parameters plus a geometric object model, and this allowed us to estimate the object's 3-dimensional pose from the 2-dimensional image data.

In this chapter we consider an alternative approach in which the additional information comes from *multiple* views of the same scene. As already mentioned the pixel coordinates from a single view constrain the world point to lie along some ray. If we can locate the same world point in another image, taken from a different but known pose, we can determine another ray along which that world point must lie. The world point lies at the intersection of these two rays – a process known as triangulation or 3D reconstruction. Even more powerfully, if we observe sufficient points, we can estimate the 3D motion of the camera between the views as well as the 3D structure of the world.◂

Almost! We can determine the translation of the camera only up to an unknown scale factor, that is, the translation is $\lambda \boldsymbol{t}$ where the direction of $\boldsymbol{t}$ is known but λ is not.

The underlying challenge is to find the same world point in multiple images. This is the *correspondence problem*, an important but non-trivial problem that we will discuss in Sect. 14.1. In Sect. 14.2 we revisit the fundamental geometry of image formation developed in Chap. 11. If you haven't yet read that chapter, or it's been a while since you read it, it would be helpful to (re)acquaint yourself with that material. We extend the geometry of single-camera imaging to the situation of multiple image planes and show the geometric relationship between pairs of images. Stereo vision is an important technique for robotic 3-dimensional perception and is discussed in some detail in Sect. 14.3. Information from two images of a scene, taken from different viewpoints, is combined to determine the 3-dimensional structure of the world. Section 14.4 introduces the topic of structure from motion where visual information from a sequence of images is used to determine the 3-dimensional structure of the world as well how the robot has moved through the world. The latter is known as visual odometry.

We finish this chapter, and this part of the book, with four application examples based on the concepts we have learned. Section 14.5 describes how we can transform an image with obvious perspective distortion into one without, effectively synthesizing the view from a virtual camera at a different location. Section 14.6 describes mosaicing which is the process of taking consecutive images from a moving camera

and *stitching* them together to form one large virtual image. Section 14.7 describes image retrieval which is the problem of finding which image in an existing set of images is most similar to some new image. This can be used by robot to determine whether it has visited a particular place, or seen the same object, before. Section 14.8 describes how we can process a sequence of images from a moving camera to locate consistent world points and to estimate the camera motion and 3-dimensional world structure.

14.1 Feature Correspondence

Correspondence is the problem of finding the pixel coordinates in two different images that correspond to the same point in the world. Consider the pair of real images

This is another example of the data association problem.

```
>> im1 = iread('eiffel2-1.jpg', 'mono', 'double');
>> im2 = iread('eiffel2-2.jpg', 'mono', 'double');
```

shown in Fig. 14.1. They show the same scene viewed from two different positions using two different cameras – the pixel size, focal length and number of pixels for each image are quite different. The scenes are complex and we see immediately that determining correspondence is not trivial. More than half the pixels in each scene correspond to blue sky and it is impossible to match a blue pixel in one image to the corresponding blue pixel in the other – these pixels are insufficiently distinct. This situation is common and can occur with homogeneous image regions such as dark shadows, smooth sheets of water, snow or smooth man-made objects such as walls or the sides of cars.

The solution is to choose only those points that are distinctive. We can use the interest point detectors that we introduced in the last chapter to find Harris corner features

The SURF detector cannot process a color image, it converts it to greyscale. The Harris detector computes the squared gradients for the individual color planes separately and then combines them. All detectors can process an image sequence provided as a matrix of dimension greater than two. There is ambiguity between a color image and an image sequence of length three. If the image's third dimension is three it is deemed to be a color image, not a sequence. A four-dimensional image is unambiguous as a sequence of color images.

```
>> harris = icorner(im1, 'nfeat', 200);
>> idisp(im1); harris.plot('gs');
```

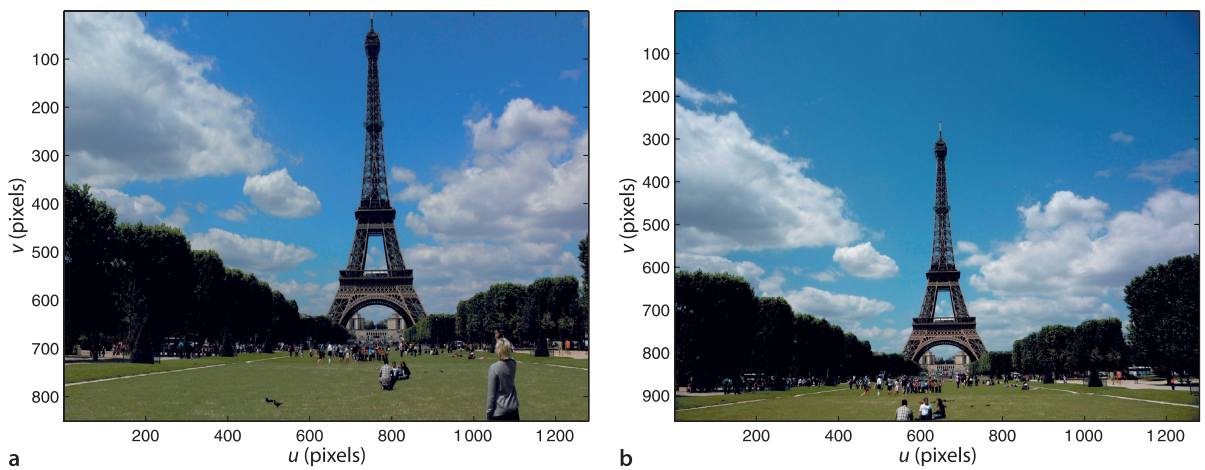
or SURF features

```
>> sf = isurf(im1, 'nfeat', 200);
>> idisp(im1); sf.plot_scale('g');
```

and these are shown in Fig. 14.2. We have simplified the problem – instead of millions of pixels we have just 200 distinctive points.

Consider the general case of two sets of features points: $\{{}^1\boldsymbol{p}_i,\ i = 1 \cdots N_1\}$ in the first image and $\{{}^2\boldsymbol{p}_j,\ j = 1 \cdots N_2\}$ in the second image. Since these are distinctive image points we would expect a significant number of points in image one would correspond to points found in image two. The problem is to determine which $({}^2u_j, {}^2v_j)$, if any, corresponds to each $({}^1u_i, {}^1v_i)$.

Fig. 14.1. Two views of the Eiffel tower. The images were captured approximately simultaneously using two different handheld digital cameras. **a** 7 Mpix camera with $f = 7.4$ mm; **b** 10 Mpix camera with $f = 5.2$ mm (photo by Lucy Corke). The images have quite different scale and the tower is 700 and 600 pixel tall in **a** and **b** respectively. The camera that captured image **b** is held by the person in the bottom-right corner of **a**



We cannot use the feature coordinates to determine correspondence – the features will have different coordinates in each image. For example in Fig. 14.1 we see that most features are lower in the right-hand image. We cannot use the intensity or color of the pixels either. Variations in white balance, illumination and exposure setting make it very unlikely that pixels that should correspond will have the same value. Even if intensity variation was eliminated there are likley to be tens of thousands of pixels in the other image with exactly the same intensity value – it is not sufficiently unique. We need some richer way of *describing* each feature.

In practice we describe the region of pixels *around* the corner point which provides a distinctive and unique description of the corner point and its immediate surrounds – the feature descriptor. In the Toolbox the feature descriptor for a corner point is a vector – the `descriptor` property of the `PointFeature` superclass. For the Harris corner feature the descriptor

```
>> harris(1).descriptor'
ans =
    0.0805    0.0821    0.0371
```

is a 3-vector that contains the unique elements of the structure tensor Eq. 13.14. This low-dimensional descriptor is computationally cheap since the elements were already computed in order to determine corner strength. These descriptor elements are gradients which have the advantage of being robust to offsets in image intensity. The similarity of two descriptors is based on Euclidean distance and is zero for a perfect match. For example, the similarity of corner features one and two is

```
>> harris(1).distance( harris(2) )
ans =
    0.0518
```

However it is difficult to know whether this value represents strong similarity or not since the units are somewhat arbitrary. Typically we would compare feature ${}^1\boldsymbol{p}_i$ with all features in the other image $\{{}^2\boldsymbol{p}_j,\ j = 1 \cdots N_2\}$ and choose the one that is most similar. ◀ However a short descriptor vector like this is still insufficiently distinctive and prone to incorrect matching.

If the world point is not visible in image two then the most similar feature will be an incorrect match.

We can create a large descriptor vector by representing the square window around the feature point as a vector. For example

```
>> harris = icorner(im1, 'nfeat', 200, 'color', 'patch', 5)
```

creates a 121-element descriptor vector for each corner point from the window of specified half-width around the feature point – in this case an 11×11 window. The pixel values are offset by the mean value and normalized to create a unit vector. We can rewrite the ZNCC similarity measure from Table 12.1 in 1-dimensional form as

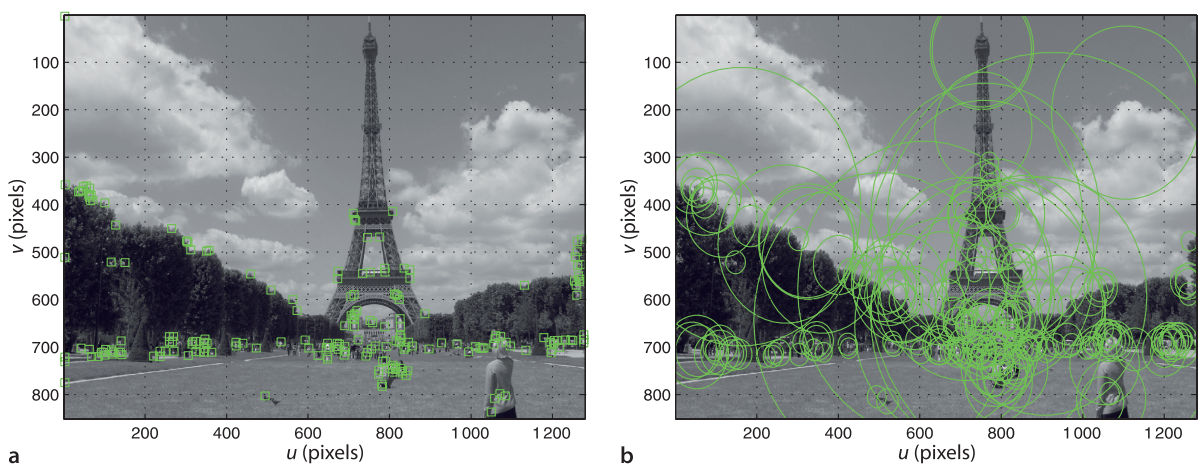


Fig. 14.2. Corner features computed for Fig. 14.1a. **a** Harris corner features; **b** SURF corner features showing scale

$$
\begin{aligned}
s &= \frac{\sum_{i=1}^{N}(I_1[i]-\bar{I}_1)\cdot(I_2[i]-\bar{I}_2)}{\sqrt{\sum_{i=1}^{N}(I_1[i]-\bar{I}_1)^2\cdot\sum_{i=1}^{N}(I_2[i]-\bar{I}_2)^2}} \\
&= \underbrace{\frac{\sum_{i=1}^{N}(I_1[i]-\bar{I}_1)}{\sqrt{\sum_{i=1}^{N}(I_1[i]-\bar{I}_1)^2}}}_{d_1}\cdot\underbrace{\frac{\sum_{i=1}^{N}(I_2[i]-\bar{I}_2)}{\sqrt{\sum_{i=1}^{N}(I_2[i]-\bar{I}_2)^2}}}_{d_2}
\end{aligned}
\tag{14.1}
$$

which we have factored into the dot product of a unit vector associated with each image patch. This normalized vector $\boldsymbol{d}_i$ can be used as the feature descriptor. Normalized cross-correlation is simply the dot product of two descriptors and the resulting similarity measure $s \in [-1, 1]$ has some meaning. A perfect match is $s = 1$ and $s \geq 0.8$ is typically considered a good match. For the example above

```
>> harris(1).ncc( harris(2) )
ans =
   -0.0292
```

the correlation score indicates a poor match. This descriptor is distinctive and invariant to changes in image intensity but is not invariant to scale or rotation. Other descriptors of the surrounding region that we could use include census and rank values as well as histograms of intensity or color. Histograms have the advantage of being invariant to rotation but they say nothing about the spatial relationship between the pixels, that is, the same pixel values in a completely different arrangement have the same histogram.

The SURF algorithm computes a 64-element descriptor vector to describe the feature point in way that is scale and rotationally invariant and based on the pixels within the feature's support region. It is created from the image in the scale-space sequence corresponding to the feature's scale and rotated according to the feature's orientation. The vector is normalized to a unit vector to increase its invariance to changes in image intensity. Similarity between descriptors is based on Euclidean distance. This descriptor is quite invariant to image intensity, scale and rotation. SURF is both a corner detector and a descriptor, whereas the Harris operator is just a corner detector which must be used with one of a number of different descriptors.

A 128-element vector can be created by passing the option `'extended'` to `isurf`.

It is conceivable to use the SURF descriptor with a Harris corner point.

For the remainder of this chapter we will use SURF features. They are computationally more expensive but pay for themselves in terms of the quality of matches between widely different views of the same scene. We compute SURF features for each image

```
>> s1 = isurf(im1)
s1 =
1288 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
>> s2 = isurf(im2)
s2 =
1426 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
```

which results in two vectors of `SurfPointFeature` objects. Many thousands of corner features were found in each image.

Next we match the two sets of SURF features based on the distance between the SURF descriptors

```
>> m = s1.match(s2)
m =
644 corresponding points (listing suppressed)
```

which results in a vector of `FeatureMatch` objects that represents 644 *candidate*-corresponding points. The first five candidate correspondences are

We refer to them as candidates because although they are very likely to correspond this has not yet been confirmed.

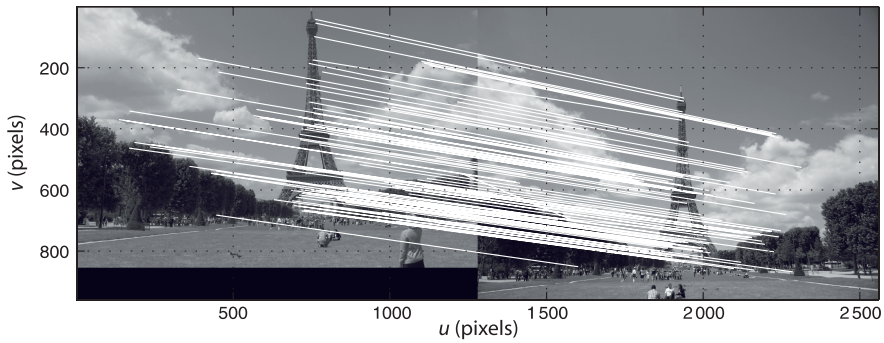


Fig. 14.3. Feature matching. Subset (100 out of 1664) of matches based on SURF descriptor similarity. We note that a few are clearly incorrect

```
>> m(1:5)
ans =
(819.56, 358.557) <-> (708.008, 563.342), dist=0.002137
(1028.3, 231.748) <-> (880.14, 461.094), dist=0.004057
(1027.6, 571.118) <-> (885.147, 742.088), dist=0.004297
(927.724, 509.93) <-> (800.833, 692.564), dist=0.004371
(854.35, 401.633) <-> (737.504, 602.187), dist=0.004417
```

which shows the feature coordinate in the first and second image, as well as the Euclidean distance between the two feature vectors. The matches are ordered by decreasing similarity, and a threshold on feature similarity has been applied.

We can overlay a subset of these matches on the original image pair

```
>> idisp({im1, im2})
>> m.subset(100).plot('w')
```

and the result is shown in Fig. 14.3. White lines connect the matched features in each image and the lines show a consistent pattern. Most of these connections seem quite sensible, but a few are quite obviously incorrect. Note that we passed a cell-array of images to `idisp` which it displays horizontally tiled as a single image. The `subset` method of the `FeatureMatch` class returns a vector with the specified number of `FeatureMatch` objects sampled evenly from the original vector. If all correspondences were shown we would just see a solid white mass.

The correspondences can be obtained via an optional return value

```
>> [m,corresp] = s1.match(s2);
>> corresp(:,1:5)
ans =
         215         389         357        1044         853
         246         418         312        1240         765
```

which is a matrix with one column per correspondence. The first column indicates that feature 215 in image one matches feature 246 in image two and so on. In terms of workspace variables this is `s1(215)` and `s2(246)`.

The Euclidean distance between the matched feature descriptors is given by the `distance` property and the distribution of these, with no thresholding applied, is

```
>> m2 = s1.match(s2, 'thresh', []);
>> ihist(m2.distance, 'normcdf')
```

shown in Fig. 14.4. It shows that 35% of all matches have descriptor distances below 0.05 which is the default threshold, whereas the maximum distance can be over ten times larger – such matches are less likely to be valid. Instead of a fixed threshold we could choose to take the matches with the N smallest distances

```
>> m = s1.match(s2, 'top', N);
```

or all those below a distance threshold

```
>> m = s1.match(s2, 'thresh', 0.04);
```

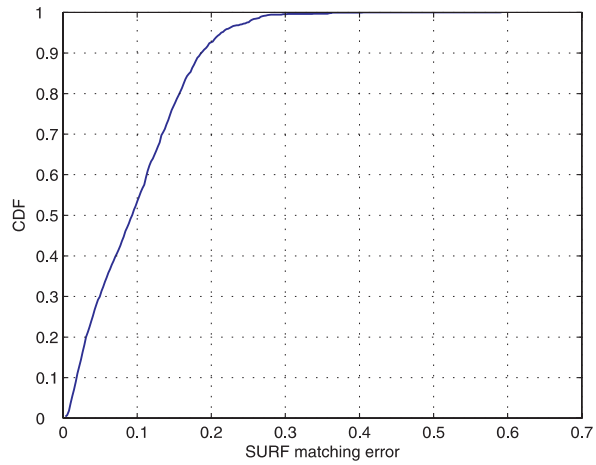


Fig. 14.4. Cumulative distribution of feature distance

or all those below the median

```
>> m = s1.match(s2, 'median');
```

Feature matching is computationally expensive – it is an $O(N^2)$ problem since every feature in one image must be compared with every feature in the other image.

Although the quality of matching shown in Fig. 14.3 looks quite good there are a few obviously incorrect matches in this small subset. We can discern a pattern in the lines joining the corresponding points, they are slightly converging and sloping down to the left. This pattern is a function of the relative pose between the two camera views, and understanding this is key to determining which of the candidate matches are correct. That is the topic of the next section.

14.2 Geometry of Multiple Views

We start by studying the geometric relationships between images of a single point **P** observed from two different viewpoints and this is shown in Fig. 14.5. This geometry could represent the case of two cameras simultaneously viewing the same scene, or one camera taking a picture from two different viewpoints.▶ The centre of each camera, the origins of {1} and {2}, plus the world point **P** defines a plane in space – the epipolar plane. The world point **P** is projected onto the image planes of the two cameras at pixel coordinates ${}^1\mathbf{p}$ and ${}^2\mathbf{p}$ respectively, and these points are known as conjugate points.

▶ Assuming the point does not move.

Consider image one. The image point ${}^1\mathbf{e}$ is a function of the position of camera two. The image point ${}^1\mathbf{p}$ is a function of the world point **P**. The camera centre, ${}^1\mathbf{e}$ and ${}^1\mathbf{p}$ define the epipolar plane and hence the epipolar line ${}^2\ell$ in image two. By definition the conjugate point ${}^2\mathbf{p}$ must lie on that line. Conversely ${}^1\mathbf{p}$ must lie along the epipolar line in image one ${}^1\ell$ that is defined by ${}^2\mathbf{p}$ in image two.

This is a very fundamental and important geometric relationship – given a point in one image we know that its conjugate is constrained to lie along a line in the other image. We illustrate this with a simple example that mimics the geometry of Fig. 14.5

```
>> T1 = transl(-0.1, 0, 0) * troty(0.4);
>> cam1 = CentralCamera('name', 'camera 1', 'default', ...
     'focal', 0.002, 'pose', T1)
```

which returns an instance of the `CentralCamera` class as discussed previously in Sect. 11.1. Similarly for the second camera

```
>> T2 = transl(0.1, 0,0)*troty(-0.4);
>> cam2 = CentralCamera('name', 'camera 2', 'default', ...
     'focal', 0.002, 'pose', T2);
```

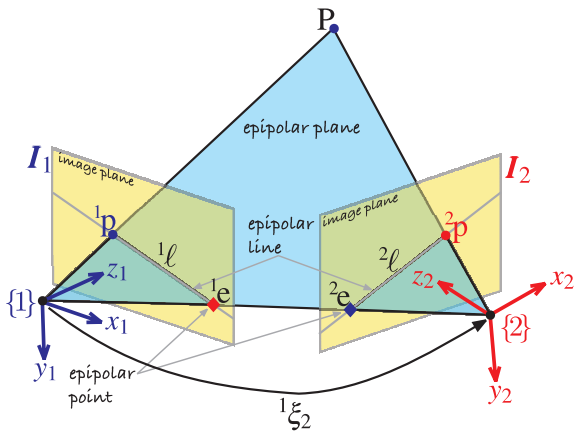


Fig. 14.5.
Epipolar geometry showing the two cameras with associated coordinate frames {1} and {2} and image planes. The world point **P** and the two camera centres form the epipolar plane, and the intersection of this plane with the image-planes form epipolar lines

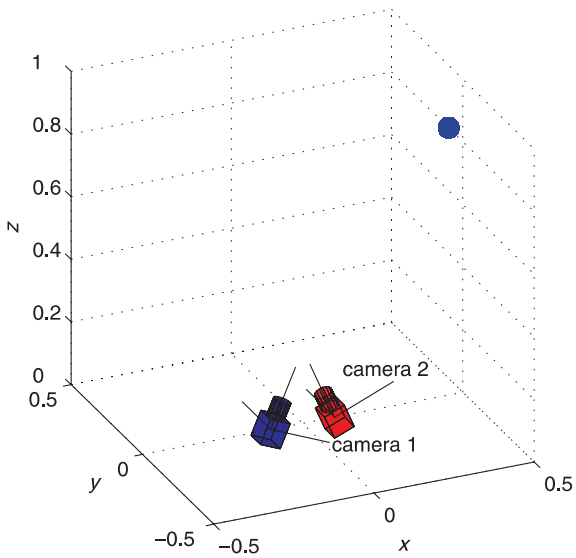


Fig. 14.6.
Simulation of two cameras and a target point. The origins of the two cameras are offset along the *x*-axis and the cameras are *verged*, that is, their optical axes intersect

and the pose of the two cameras is visualized by

```
>> axis([-0.5 0.5 -0.5 0.5 0 1])
>> cam1.plot_camera('color', 'b', 'label')
>> cam2.plot_camera('color', 'r', 'label')
```

which is also shown in Fig. 14.6. We define an arbitrary world point

```
>> P=[0.5 0.1 0.8]';
```

which we display as a small sphere

```
>> plot_sphere(P, 0.03, 'b');
```

which is shown in Fig. 14.6. We project this point to both cameras

```
>> p1 = cam1.plot(P)
p1 =
   561.6861
   532.6079
>> p2 = cam2.plot(P)
p2 =
   746.0323
   546.4186
```

and this is shown in Fig. 14.7. The epipoles are computed by projecting the centre of each camera to the other camera's image plane

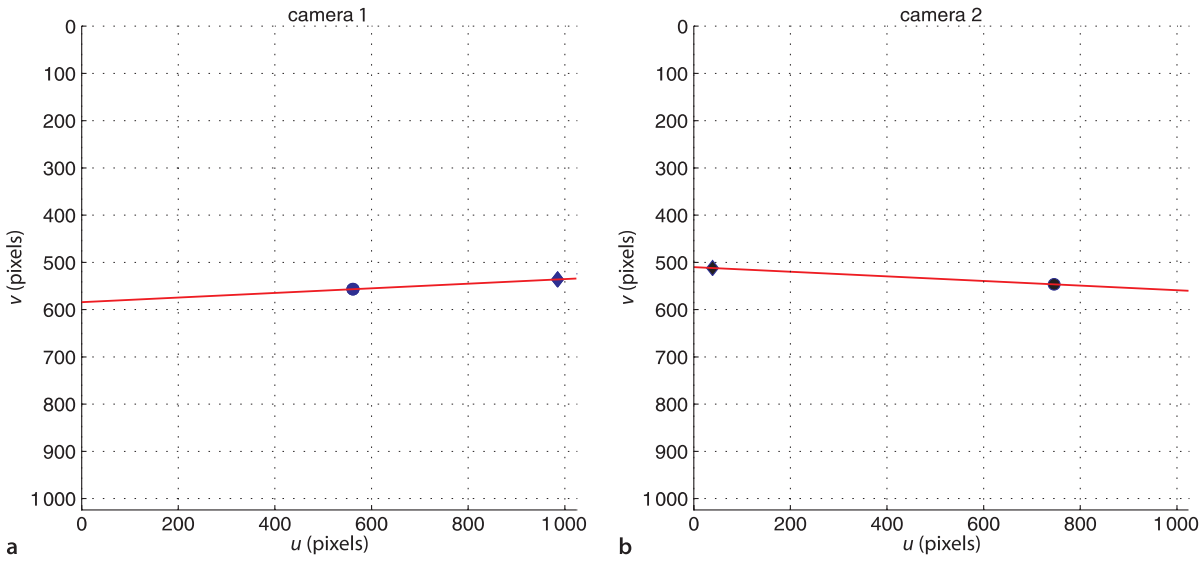


Fig. 14.7. Epipolar geometry simulation showing the virtual image planes of two Toolbox CentralCamera objects. The perspective projection of point **P** is a black circle, the projection of the other camera's centre is a black ♦-marker, and the epipolar line is shown in red

```
>> cam1.hold
>> e1 = cam1.plot( cam2.centre, 'Marker', 'd', 'MarkerFaceColor', 'k')
e1 =
  985.0445  512.0000
>> cam2.hold
>> e2 = cam2.plot( cam1.centre, 'Marker', 'd', 'MarkerFaceColor', 'k')
e2 =
   38.9555  512.0000
```

and these are shown in Fig. 14.7 as a black ♦-marker.

14.2.1 The Fundamental Matrix

The epipolar relationship shown graphically in Fig. 14.5 can be expressed concisely and elegantly as

$$ {}^2\tilde{\boldsymbol{p}}^T \boldsymbol{F}\, {}^1\tilde{\boldsymbol{p}} = 0 \tag{14.2} $$

where ${}^1\tilde{\boldsymbol{p}}$ and ${}^2\tilde{\boldsymbol{p}}$ are the image points ${}^1\mathbf{p}$ axnd ${}^2\mathbf{p}$ expressed in homogeneous form and $\boldsymbol{F}$ is a 3×3 matrix known as the fundamental matrix.

We can group the last two terms

$$ {}^2\tilde{\ell} \simeq \boldsymbol{F}\, {}^1\tilde{\boldsymbol{p}} \tag{14.3} $$

which is the equation of a line, the epipolar line, along which conjugate point in image two must lie

2D projective geometry in brief. The projective plane $\mathbb{P}^2$ is the set of all points (x_1, x_2, x_3), $x_i \in \mathbb{R}$ and x_i not all zero. Typically the 3-tuple is considered a column vector. A point $\boldsymbol{p} = (u, v)$ is represented in $\mathbb{P}^2$ by homogeneous coordinates $\tilde{\boldsymbol{p}} = (u, v, 1)$. Scale is unimportant for homogeneous quantities and we express this as $\tilde{\boldsymbol{p}} \simeq \lambda\tilde{\boldsymbol{p}}$ where the operator $\simeq$ means equal up to a (possibly unknown) non-zero scale factor. A point in $\mathbb{P}^2$ can be represented in non-homogeneneous, or Euclidean, form $\boldsymbol{p} = (x_1/x_3, x_2/x_3)$ in $\mathbb{R}^2$. The homogeneous vector (u, v, f), where f is the focal length in pixels, is a vector from the camera's origin that points toward the world point **P**. More details are given in Appendix I.

The Toolbox functions e2h and h2e convert between Euclidean and homogeneous coordinates for points (a column vector) or sets of points (a matrix with one column per point).

$$ {}^{2}\tilde{\boldsymbol{p}}^{T}\ {}^{2}\tilde{\ell} = 0 \tag{14.4} $$

This line is a function (Eq. 14.3) of the point ${}^{1}\tilde{\boldsymbol{p}}$ in image one and is a powerful test as to whether or not a point in image two is a possible conjugate. Taking the transpose of both sides of Eq. 14.2 yields

$$ {}^{1}\tilde{\boldsymbol{p}}^{T}\boldsymbol{F}^{T}\ {}^{2}\tilde{\boldsymbol{p}} = 0 \tag{14.5} $$

from which we can write the epipolar line for camera one

$$ {}^{1}\tilde{\ell} \simeq \boldsymbol{F}^{T}\ {}^{2}\tilde{\boldsymbol{p}} \tag{14.6} $$

in terms of a point viewed by camera two.

The fundamental matrix is a function of the camera parameters and the relative camera pose between the views

$$ \boldsymbol{F} \simeq \boldsymbol{K}^{-1}\boldsymbol{S}(\boldsymbol{t})\boldsymbol{R}\boldsymbol{K} \tag{14.7} $$

where $\boldsymbol{K}$ is the camera intrinsic matrix given in Eq. 11.7, $\boldsymbol{S}(\cdot)$ is the skew-symmetric matrix, and ${}^{2}\xi_{1} \sim (\boldsymbol{R}, \boldsymbol{t})$ is the relative pose of camera one with respect to camera two.◀ The fundamental matrix that relates the two views is returned by the method `F` of the `CentralCamera` class, for example

Note well that this is the inverse of what you might expect, camera two with respect to camera one.

```
>> F = cam1.F( cam2 )
F =
         0   -0.0000    0.0010
   -0.0000         0    0.0019
    0.0010    0.0001   -1.0208
```

where the relative pose is from `CentralCamera` object `cam1` to `cam2`.

The fundamental matrix has some interesting properties. It is singular with a rank of two

```
>> rank(F)
ans =
     2
```

and has seven degrees of freedom.◀ The epipoles are *encoded* in the null-space of the matrix. The epipole for camera one is the right null-space of $\boldsymbol{F}$

The matrix $F \subset \mathbb{R}^{3\times3}$ has seven underlying parameters so its nine elements are not independent. The overall scale is not defined, and there exists a constraint that $\det(F) = 0$.

```
>> null(F)'
ans =
   -0.8873   -0.4612   -0.0009
```

in homogeneous coordinates or

```
>> e1 = h2e(ans)'
e1 =
  985.0445  512.0000
```

in Euclidean coordinates – the same as determined above using the `plot` function. The epipole for camera two is the left null-space of the transpose of the fundamental matrix

```
>> null(F');
>> e2 = h2e(ans)'
e2 =
   38.9555  512.0000
```

The Toolbox can display epipolar lines using the `plot_epiline` methods of the `CentralCamera` class

```
>> cam2.plot_epiline(F, p1, 'r')
```

which is shown in Fig. 14.7 as a red line in the camera two image plane. We see, as expected, that the projection of **P** lies on this epipolar line. The epipolar line for camera one is

```
>> cam1.plot_epiline(F', p2, 'r');
```

14.2.2 The Essential Matrix

The epipolar geometric constraint can also be expressed in terms of normalized image coordinates

$$ {}^2\tilde{\boldsymbol{x}}^T \boldsymbol{E}\, {}^1\tilde{\boldsymbol{x}} = 0 \tag{14.8} $$

where $\boldsymbol{E}$ is the essential matrix and ${}^1\tilde{\boldsymbol{x}}$ and ${}^1\tilde{\boldsymbol{x}}$ are conjugate points in homogeneous normalized image coordinates. This matrix is a simple function of the relative camera pose

See page 254.

$$ \boldsymbol{E} \simeq \boldsymbol{S}(\boldsymbol{t})\boldsymbol{R} \tag{14.9} $$

where ${}^2\xi_1 \sim (\boldsymbol{R}, \boldsymbol{t})$ is the relative pose of camera one with respect to camera two. The essential matrix is singular, has a rank of two, and has two equal singular values and one of zero. The essential matrix has only 5 degrees of freedom and is completely defined by 3 rotational and 2 translational parameters. For pure rotation, when $\boldsymbol{t} = 0$, the essential matrix is not defined.

See Appendix D.

A 3-dimensional translation (x, y, z) with unknown scale can be considered as $(x', y', 1)$.

We recall from Eq. 11.7 that $\tilde{\boldsymbol{p}} \simeq \boldsymbol{K}\tilde{\boldsymbol{x}}$ and substituting into Eq. 14.8 we write

$$ {}^2\tilde{\boldsymbol{p}}^T \underbrace{\boldsymbol{K}_2^{-T}\boldsymbol{E}\boldsymbol{K}_1^{-1}}_{\boldsymbol{F}} {}^1\tilde{\boldsymbol{p}} = 0 \tag{14.10} $$

Similarity to Eq. 14.2 yields a relationship between the two matrices

$$ \boldsymbol{E} \simeq \boldsymbol{K}_2^T \boldsymbol{F} \boldsymbol{K}_1 \tag{14.11} $$

in terms of the intrinsic parameters of the two cameras involved. This is implemented by the `E` method of the `CentralCamera` class

If both images were captured with the same camera then $K_1 = K_2$.

```
>> E = cam1.E(F)
E =
         0   -0.0779         0
   -0.0779         0    0.1842
         0   -0.1842    0.0000
```

where the intrinsic parameters of camera one (which is the same as camera two) are used.

Like the camera matrix in Sect. 11.2.2 the essential matrix can be *decomposed* to yield the relative pose ${}^1\xi_2$ in homogeneous transformation form. The inverse is not unique and in general there are two solutions

Although Eq. 14.9 is written in terms of $S(R, t) \sim {}^2\xi_1$ the Toolbox function returns ${}^1\xi_2$.

```
>> sol = cam1.invE(E)
sol(:,:,1) =
    1.0000    0.0000   -0.0000   -0.1842
    0.0000   -1.0000   -0.0000   -0.0000
   -0.0000    0.0000   -1.0000   -0.0779
         0         0         0    1.0000
sol(:,:,2) =
    0.6967    0.0000   -0.7174    0.1842
    0.0000    1.0000    0.0000    0.0000
    0.7174   -0.0000    0.6967    0.0779
         0         0         0    1.0000
```

which returns a 3-dimensional matrix where the last index is the solution number. The true relative pose from view two to view one is

```
>> inv(cam1.T) * cam2.T
ans =
    0.6967         0   -0.7174    0.1842
         0    1.0000         0         0
    0.7174         0    0.6967    0.0779
         0         0         0    1.0000
```

which indicates that, in this case, solution two is the correct one.

As observed by Hartley and Zisserman (2003, p 259) not even the sign of $\boldsymbol{t}$ can be determined.

Unusually we have recovered the camera translation exactly but since $\boldsymbol{E} \simeq \lambda \boldsymbol{E}$ the translational part of the homogeneous transformation matrix has an unknown scale factor.◂ In this case it is correct because the essential matrix was determined directly from the relative pose between the cameras.

In this case we could choose the correct solution because we knew the pose of the two cameras, but how do we determine the correct solution in practice? One approach is to determine whether a world point is visible. Typically we would choose a point on the optical axis in front of the first camera

```
>> Q = [0 0 10]';
```

and its projection to the first camera

```
ans =
  429.7889
  512.0000
```

is a reasonable value. We can create an instance of the first camera with an arbitrary displacement using the `move` method

```
>> cam1.move(sol(:,:,1)).project(Q)
ans =
   NaN
   NaN
```

and the values of `NaN` indicate that the world point is behind the camera. The second solution

```
>> cam1.move(sol(:,:,2)).project(Q)
ans =
  594.2111
  512.0000
```

has a finite value and indicates that this solution is the valid one. We can perform this more compactly by providing a test point

```
>> sol = cam1.invE(E, Q)
sol =
    0.6967    0.0000   -0.7174    0.1842
    0.0000    1.0000    0.0000    0.0000
    0.7174   -0.0000    0.6967    0.0779
         0         0         0    1.0000
```

in which case only the valid solution is returned.

In summary these 3×3 matrices, the fundamental and the essential matrix, encode the geometry of the two cameras. The fundamental matrix and a point in one image defines an epipolar line in the other image along which its conjugate points must lie. The essential matrix encodes the relative pose of the two camera's centres and the pose can be extracted, with two possible values, and with translation scaled by an unknown factor. In this example the fundamental matrix was computed from known camera motion and intrinsic parameters. The real world isn't like this – camera motion is difficult to measure and the camera may not be calibrated. Instead we can estimate the fundamental matrix directly from corresponding image points.

14.2.3 Estimating the Fundamental Matrix

Assume that we have N pairs of corresponding points in two views of the same scene $({}^1\boldsymbol{p}_i, {}^2\boldsymbol{p}_i)$, $i = 1 \cdots N$. To demonstrate this we create a set of twenty random point features (within a $2 \times 2 \times 2$ m cube) whose center is located 3 m in front of the cameras

```
>> P = homtrans( transl(-1, -1, 2), 2*rand(3,20) );
```

and project these points onto the two camera image planes

```
>> p1 = cam1.project(P);
>> p2 = cam2.project(P);
```

If $N \geq 8$ the fundamental matrix can be estimated from these two sets of corresponding points

```
>> F = fmatrix(p1, p2)
maximum residual 1.932e-29
F =
    0.0000   -0.0000    0.0239
   -0.0000   -0.0000    0.0460
    0.0239    0.0018  -24.4896
```

where the residual is the maximum value of the left-hand side of Eq. 14.2 and is ideally zero. The value here is not zero, but it is very small, and this is due to the accumulation of errors from finite precision arithmetic. The estimated matrix has the required rank property

```
>> rank(F)
ans =
    2
```

For camera two we can plot the projected points

```
>> cam2.plot(P);
```

and overlay the epipolar lines generated by each point in image one

```
>> cam2.plot_epiline(F, p1, 'r')
```

which is shown in Fig. 14.8. We see a family or *pencil* of epipolar lines, and that every point in image two lies on an epipolar line. Note how the epipolar lines all converge on the epipole which is possible in this case▸ because the two cameras are verged as shown in Fig. 14.6.

The example has been contrived so that the epipoles lie within the images, that is, the each camera can see the centre of the other camera. A common imaging geometry is for the optical axes to be parallel, such as shown in Fig. 14.19 in which case the epipoles are at infinity (the third element of the homogeneous coordinate is zero) and all the epipolar lines are parallel.

To demonstrate the importance of correct point correspondence we will repeat the example above but introduce two *bad* data associations by swapping two elements in `p2`

```
>> p2(:,[8 7]) = p2(:,[7 8]);
```

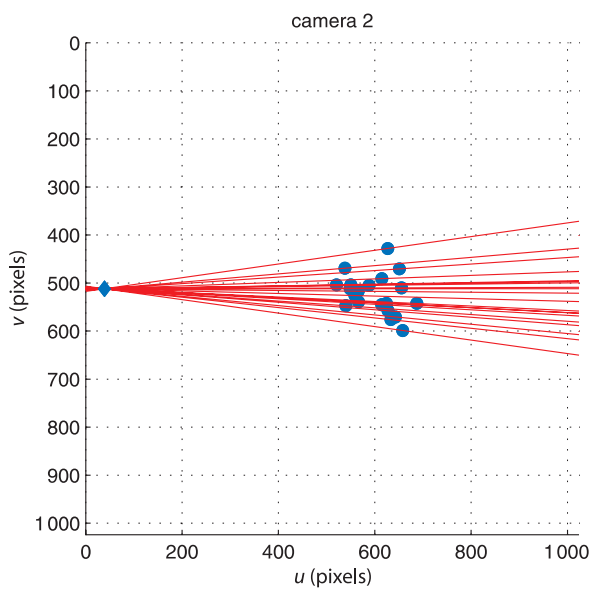


Fig. 14.8.
A pencil of epipolar lines on the camera two image plane. Note how all epipolar lines pass through the epipole which is the projection of camera one's centre

The fundamental matrix estimation

```
>> fmatrix(p1, p2)
maximum residual 236.2
ans =
    0.0000   -0.0000    0.0022
    0.0000    0.0000   -0.0023
   -0.0030    0.0014    1.0000
```

now has a very high residual – hundreds of pixels. This means that the point correspondence cannot be *explained* by the relationship Eq. 14.2.

If we knew the fundamental matrix we could test whether a pair of candidate corresponding points are in fact conjugates by measuring how far one is from the epipolar line defined by the other

```
>> epidist(F, p1(:,1), p2(:,1))
ans =
   2.3035e-13
>> epidist(F, p1(:,7), p2(:,7))
ans =
   18.8228
```

which shows that point 1 is a good fit, but point 7 (which we swapped with point 8), is a poor fit. However we have to first estimate the fundamental matrix and that requires that point correspondence is known. We break this deadlock with an ingenious algorithm called Random Sampling and Consensus or RANSAC.

The underlying principle is delightfully simple. Estimating a fundamental matrix requires eight points so we randomly choose eight candidate corresponding points (the sample) and estimate $\boldsymbol{F}$ to create a *model*. This model is tested against all the other candidate pairs and those that fit◂ vote for this model. The process is repeated a number of times and the model that had the most supporters (the consensus) is returned. Since the sample is small the chance that it contains all valid candidate pairs is high. The point pairs that support the model are termed inliers and those that do not are outliers.

◂ To within a defined threshold t. The Toolbox function `epidist` returns the distance between a point and an epipolar line.

RANSAC is remarkably effective and efficient at finding the inlier set, even in the presence of large numbers of outliers (more than 50%), and is applicable to a wide range of problems. Within the Toolbox we invoke RANSAC as a *driver* of the `fmatrix` function

```
>> [F,in,r] = ransac(@fmatrix, [p1; p2], 1e-6, 'verbose')
15 trials
2 outliers
3.48564e-29 final residual
```

and we obtain an excellent final residual. The set of inliers is also returned

```
>> in
in =
  Columns 1 through 14
     1     2     3     4     5     6     9    10    11    12    13    14    15    16
  Columns 15 through 18
    17    18    19    20
```

and the two incorrect associations, points 7 and 8, are notably absent from this list. The third parameter to `ransac` is the threshold t which is used to determine whether or not a point pair supports the model. If t is chosen to be too small RANSAC requires many more trials than its default maximum and this requires adjustment of additional parameters. Keep in mind also that the results of RANSAC will vary from run to run due to the random subsampling performed. Using RANSAC involve some trial and error to choose the correct threshold based on the final residual and the number of outliers. There are also a number of other options that are described in the online documentation.

We return now to the pair of images of the Eiffel tower shown in Fig. 14.3. When we left off at page 384 we had determined correspondence based on descriptor similarity

but there were a number of clearly incorrect matches. RANSAC is available as a method `ransac` that operates on a vector of `FeatureMatch` objects

```
>> F = m.ransac(@fmatrix, 1e-4, 'verbose')
1527 trials
312 outliers
0.000140437 final residual
F =
    0.0000   -0.0000    0.0098
    0.0000    0.0000   -0.0148
   -0.0121    0.0129    3.6393
```

A small amount of trial and error was required to settle on the tolerance of 10^{-4}. Making it smaller requires more RANSAC trials and requires raising the limit on maximum number of trials allowed but without any significant change in the result. It is also unrealistic to expect a very small residual since the real image data is subject to random error such as pixel noise and systematic error such as lens distortion.

Lens distortion causes points to be displaced on the image plane and this violates the epipolar geometry. Images can be corrected as discussed in Sect. 12.6.4 but this is computationally expensive. A cheaper alternative is to correct the coordinates of the features by mapping them through the inverse distortion model Eq. 11.13.

RANSAC identified 312 outliers or incorrect data associations from the SURF feature matching stage which is nearly 50% of the *candidate* matches – the preliminary matching was worse than it looked. Running RANSAC has updated the elements of the `FeatureMatch` vector

```
>> m.show
ans =
1667 corresponding points
644 corresponding points
332 inliers (51.6%)
312 outliers (48.4%)
```

which displays the total number of inliers and outliers. Compared to page 384 the elements of the vector

```
>> m(1:5)
ans =
(819.56, 358.557) <-> (708.008, 563.342), dist=0.002137 +
(1028.3, 231.748) <-> (880.14, 461.094), dist=0.004057 -
(1027.6, 571.118) <-> (885.147, 742.088), dist=0.004297 +
(927.724, 509.93) <-> (800.833, 692.564), dist=0.004371 +
(854.35, 401.633) <-> (737.504, 602.187), dist=0.004417 +
```

now have a trailing plus or minus sign to indicate whether the corresponding match is an inlier or outlier. We can plot some of the inliers

```
>> idisp({im1, im2});
>> m.inlier.subset(100).plot('g')
```

or some of the outliers

```
>> idisp({im1, im2});
>> m.outlier.subset(100).plot('r')
```

and these are shown in Fig. 14.9.

An alternative way to create a `CentralCamera` object is from an image

```
>> cam = CentralCamera('image', im1);
```

The size of the pixel array is inferred from the image and the intrinsic parameters are set to default values. As before, we can overlay the epipolar lines computed from the corresponding points found in the second image

```
>> cam.plot_epiline(F', m.inlier.subset(20).p2, 'g');
```

and the result is shown in Fig. 14.10. The epipolar lines intersect at the epipolar point which we can clearly see is the projection of the second camera in the first image. The epipole at

We only plot a small subset of the epipolar lines since they are too numerous and would obscure the image.

```
>> h2e( null(F))
ans =
   1.0e+03 *
    1.0359
    0.6709
>> cam.plot(ans, 'bo')
```

is also superimposed on the plot. With two handheld cameras and a common view we have been able to pinpoint the second camera in the first image. The result is not quite perfect – there is a horizontal offset of about 20 pixels which is likely to be due to a small pointing error in one or both cameras which were handheld and only approximately synchronized.◂

At the focal lengths used a 20 pix displacement on the image plane corresponds to a pointing error of less than 0.5°.

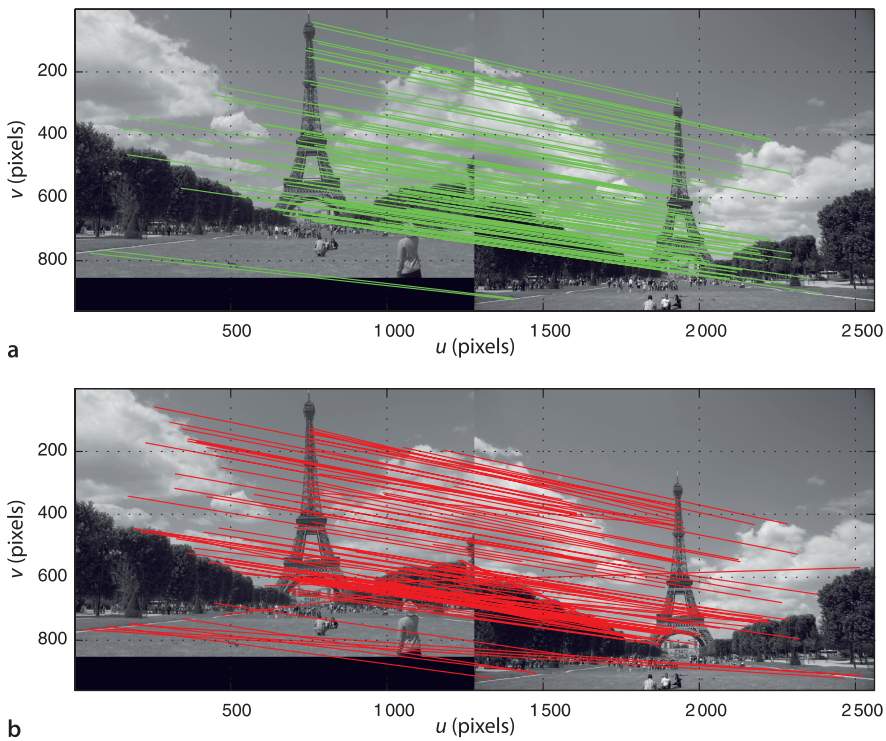


Fig. 14.9. Results of SURF feature matching after RANSAC. **a** Subset of all inlier matches; **b** subset of the outlier matches, some are quite visibly incorrect while others are more subtle

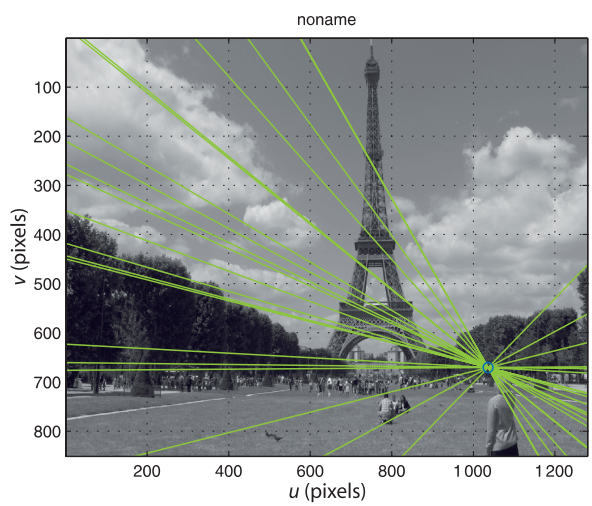


Fig. 14.10. Image from Fig. 14.1a showing epipolar lines converging on the projection of the second camera's centre. In this case the second camera is clearly visible in the bottom right of the image

14.2.4 Planar Homography

In this section we will consider a camera viewing a group of world points $\mathbf{P}_i$ that lie on a plane. They are viewed by two different cameras and the projection in the cameras are ${}^1\boldsymbol{p}_i$ and ${}^2\boldsymbol{p}_i$ respectively which are related by

$$ {}^2\tilde{\boldsymbol{p}}_i \simeq \boldsymbol{H}\, {}^1\tilde{\boldsymbol{p}}_i \tag{14.12}$$

where $\boldsymbol{H}$ is a non-singular 3×3 matrix known as an homography, a planar homography, or the homography *induced* by the plane.

For example consider again the pair of cameras from page 387 now observing a 3×3 grid of points

```
>> Tgrid = transl(0,0,1)*trotx(0.1)*troty(0.2);
>> P = mkgrid(3, 1.0, 'T', Tgrid);
```

where `Tgrid` is the pose of the grid coordinate frame $\{G\}$ and the grid points are centred in the frame's xy-plane. The points are projected to both cameras

```
>> p1 = cam1.plot(P, 'o');
>> p2 = cam2.plot(P, 'o');
```

and the images are shown in Fig. 14.11a and b respectively.

Just as we did for the fundamental matrix, if $N \geq 8$ we can estimate the matrix `H` from two sets of corresponding points

```
>> H = homography(p1, p2)
H =
   -0.4282   -0.0006  408.0894
   -0.7030    0.3674  320.1340
   -0.0014   -0.0000    1.0000
```

According to Eq. 14.12 we can predict the position of the grid points in image two from the corresponding image one coordinates

```
>> p2b = homtrans(H, p1);
```

which we can can superimpose on image two as +-symbols

```
>> cam2.hold()
>> cam2.plot(p2b, '+')
```

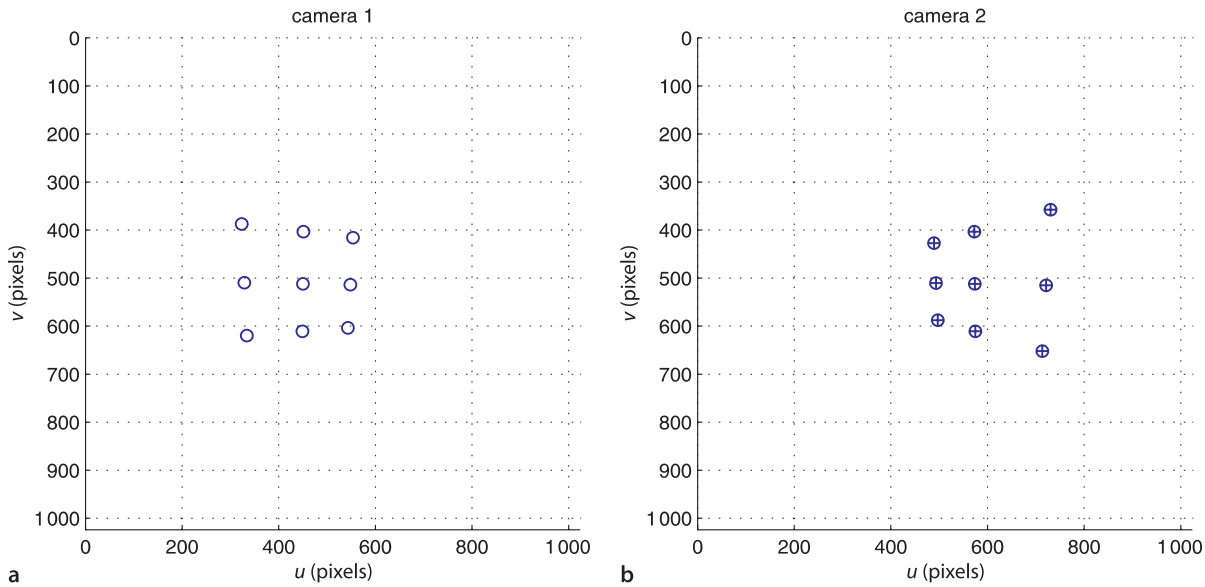


Fig. 14.11. Views of the oblique planar grid of points from two different view points. The grid points are projected as open circles. Plus signs in **b** indicate points transformed from the camera one image plane by the homography

This is shown in Fig. 14.11b and we see that the predicted points are perfectly aligned with the actual projection of the world points. The inverse of the homography matrix

$$ {}^1\tilde{\boldsymbol{p}}_i \simeq \boldsymbol{H}^{-1}\, {}^2\tilde{\boldsymbol{p}}_i \tag{14.13} $$

performs the inverse mapping, from image two coordinates to image one

```
>> p1b = homtrans(inv(H), p2);
```

The fundamental matrix constrains the conjugate point to lie along a line but the homography tells us *exactly* where the conjugate point will be in the other image – provided that the points lie on a plane.

We can use this proviso to our advantage as a test for whether or not points lie on a plane. We will add some extra world points◂ to our example

These points lie along the ray from the camera one centre to an extra row of points in the grid plane. However their z-coordinates have been chosen to be 0.4, 0.5 and 0.6 m respectively.

```
>> Q = [
   -0.2302   -0.0545    0.2537
    0.3287    0.4523    0.6024
    0.4000    0.5000    0.6000  ];
```

which we plot in 3D

```
>> axis([-1 1 -1 1 0 2])
>> plot_sphere(P, 0.05, 'b')
>> plot_sphere(Q, 0.05, 'r')
>> cam1.plot_camera('color', 'b', 'label')
>> cam2.plot_camera('color', 'r', 'label')
```

and this is shown in Fig. 14.12. The new points, shown in red, are clearly not in the same plane as the original blue points. Viewed from camera one

```
>> p1 = cam1.plot([P Q], 'o');
```

as shown in Fig. 14.13a, these new points appear as an extra row in the grid of points we used above. However in the second view

```
>> p2 = cam2.plot([P Q], 'o');
```

as shown in Fig. 14.13b these *out of plane* points no longer form a regular grid. If we apply the homography to the camera one image points

```
>> p2h = homtrans(H, p1);
```

we find where they should be in camera two image if they belonged to the plane implicit in the homography

```
>> cam2.plot(p2h, '+')
```

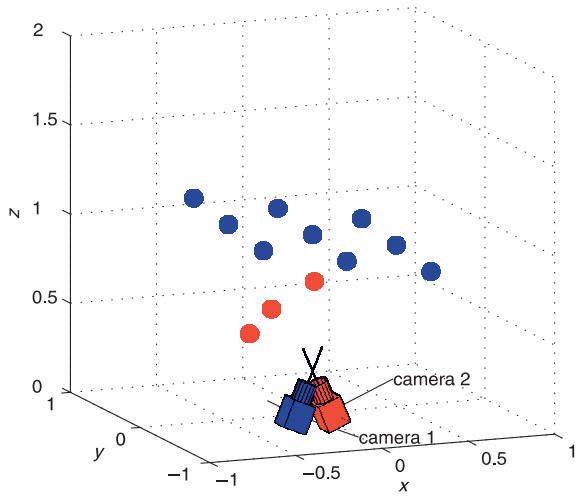


Fig. 14.12. World view of target points and two camera poses. Blue points lie in a planar grid, while the red points appear to lie in the grid from the viewpoint of camera one

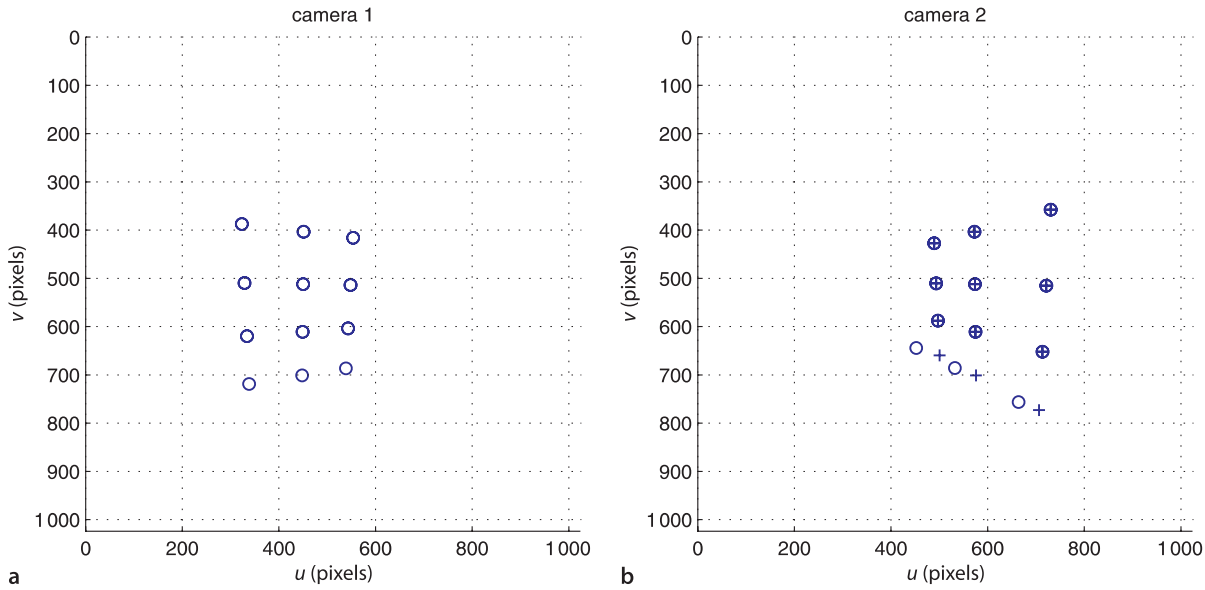


Fig. 14.13. Views of the oblique planar grid of points from two different view points. The grid points are projected as open circles. Plus signs in **b** indicate points transformed from the camera one image plane by the homography. The bottom of row of points in each case are not co-planar with the other points

We see that the original nine points overlap, but the three new points do not. We could make an automated test based on the prediction error

```
>> colnorm( homtrans(H, p1)-p2 )
ans =
  Columns 1 through 9
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000
  Columns 10 through 12
   50.5969   46.4423   45.3836
```

which is large for these last three points – they do not belong to the plane that induced the homography.

In this example we estimated the homography based on two sets of corresponding points which were projections of known planar points. In practice we do not know in advance which points belong to the plane so we can again use RANSAC

```
>> [H,in] = ransac(@homography, [p1; p2], 0.1)
resid =
   4.0990e-13
H =
   -0.4282   -0.0006  408.0894
   -0.7030    0.3674  320.1340
   -0.0014   -0.0000    1.0000
in =
     1     2     3     4     5     6     7     8     9
```

which finds the homography that best explains the relationship between the sets of image points. It has also identified those points which support the homography and the three *out of plane points* are not on the inlier list.

The geometry related to the homography is shown in Fig. 14.14. We can express the homography in normalized image coordinates

See page 254.

$$ {}^2\tilde{\boldsymbol{x}} \simeq \boldsymbol{H}_E \, {}^1\tilde{\boldsymbol{x}} $$

where $\boldsymbol{H}_E$ is the Euclidean homography which is written

$$ \boldsymbol{H}_E \simeq \boldsymbol{R} + \frac{\boldsymbol{t}}{d}\boldsymbol{n}^T \qquad (14.14) $$

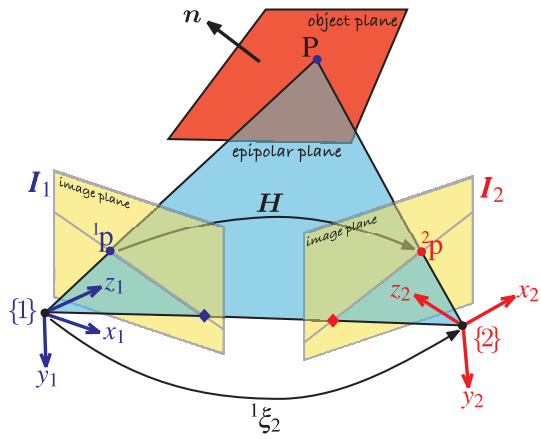


Fig. 14.14. Geometry of homography showing two cameras with associated coordinate frames {1} and {2} and image planes. The world point **P** belongs to a plane with surface normal $\boldsymbol{n}$. $\boldsymbol{H}$ is the homography, a 3×3 matrix that maps $^1\mathbf{p}$ to $^2\mathbf{p}$

in terms of the motion $(\boldsymbol{R}, \boldsymbol{t}) \sim {}^2\xi_1$ and the plane $\boldsymbol{n}^T\boldsymbol{P} + d = 0$ with respect to frame {1}. The Euclidean and projective homographies are related by

$$H_E \simeq K^{-1}HK$$

where $\boldsymbol{K}$ is the camera parameter matrix.

As for the essential matrix the projective homography can be *decomposed* to yield the relative pose $^1\xi_2$ in homogeneous transformation form◂ as well as the normal to the plane. We use the `invH` method of the `CentralCamera` class

Although Eq. 14.14 is written in terms of $(R, t) \sim {}^2\xi_1$ the Toolbox function returns $^1\xi_2$.

```
>> cam1.invH(H)
solution 1
     T =0.82478    -0.01907    -0.56513    -0.01966
        0.01907     0.99980    -0.00591    -0.01917
        0.56513    -0.00591     0.82498     0.19911
        0.00000     0.00000     0.00000     1.00000
    n = 0.95519     0.00998     0.29582
solution 2
     T =0.69671     0.00000    -0.71736     0.18513
        0.00000     1.00000     0.00000    -0.00000
        0.71736    -0.00000     0.69671     0.07827
        0.00000     0.00000     0.00000     1.00000
    n = -0.19676    -0.09784     0.97556
```

which returns a short structure array. Again there are multiple solutions and we need to apply additional information to determine the correct one. As usual the translational component of the transformation matrix has an unknown scale factor. We know from Fig. 14.12 that the camera motion is predominantly in the x-direction and that the plane normal is approximately parallel to the camera's optical- or z-axis and this knowledge helps us to choose solution two. The true transformation from camera one to two is

```
>> inv(T1)*T2
ans =
    0.6967         0   -0.7174    0.1842
         0    1.0000         0         0
    0.7174         0    0.6967    0.0779
         0         0         0    1.0000
```

The translation scale factor is quite close to one in this example, but in general it must be considered unknown.

and supports our choice.◂ The pose of the grid with respect to camera one is

```
>> inv(T1)*Tgrid
ans =
    0.9797   -0.0389   -0.1968   -0.2973
    0.0198    0.9950   -0.0978         0
    0.1996    0.0920    0.9756    0.9600
         0         0         0    1.0000
```

and the third column is the grid's normal which matches the estimated normal associated with solution two.

Since the points are in the *xy*-plane of the grid frame {*G*} the normal is the *z*-axis.

We can apply this technique to a pair of real images

```
>> im1=iread('garden-l.jpg',  'double');
>> im2=iread('garden-r.jpg',  'double');
```

shown in Fig. 14.15. We start by finding the SURF features

```
>> s1 = isurf(im1);
>> s2 = isurf(im2);
```

and the candidate corresponding points

```
>> m = s1.match(s2)
m =
323 corresponding points (listing suppressed)
```

then use RANSAC to find the set of corresponding points that best fits a plane in the world

```
>> [H,r] = m.ransac(@homography, 2)
H =
    0.9966    0.0061  -15.6385
   -0.0105    1.0081  -29.7619
   -0.0000    0.0000    1.0000
r =
    1.2228
```

The number of inlier and outlier points is

```
>> m.show
ans =
323 corresponding points
83 inliers (25.7%)
240 outliers (74.3%)
```

In this case the majority of point pairs do not fit the model, that is they do not belong to the plane that induces the homography $\boldsymbol{H}$. However 83 points *do* belong to the plane and we can superimpose them on the figure

```
>> idisp(im1)
>> plot_point(m.inlier.p1, 'w*')
```

as shown in Fig. 14.15a. RANSAC has found a consensus which is the plane containing the wall. The tolerance was set to 4 pixel error since the dominant planes in this scene, the wall and the lawn, are only approximately planar. The lawn is quite large and contains many SURF feature points but the number of corresponding feature points, found

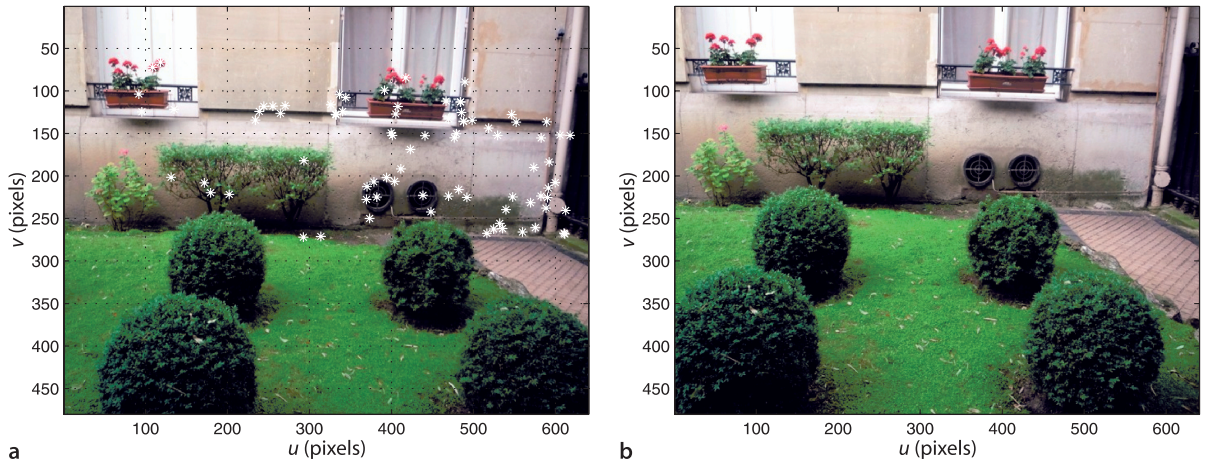


Fig. 14.15. Two pictures of a garden scene taken from different viewpoints. Image **b** approximately 30 cm to the right of image **a**. Image **a** has superimposed features that fit a plane. The camera was handheld with $f = 5.2$ mm

by `match`, on the lawn is very low. If we remove the inlier points from the `FeatureMatch` vector, that is, we keep the outliers

```
>> m = m.outlier
```

and repeat the RANSAC homography estimation step we will find the next most dominant plane in the scene which turns out to be the top of the front bushes. Planes are very common in man-made environments and we will revisit homographies and their decomposition in Sect. 14.5.

14.3 Stereo Vision

Stereo vision is the technique of estimating the 3-dimensional structure of the world from two images taken from different viewpoints. We will discuss two approaches known as sparse and dense stereo respectively. Sparse stereo is a natural extension of what we have learned about feature matching and recovers the world coordinate (X, Y, Z) for each corresponding point pair. Dense stereo recovers the world coordinate (X, Y, Z) for *every pixel* in the image.

14.3.1 Sparse Stereo

To illustrate sparse stereo we will return to the pair of garden images shown in Fig. 14.15. We have already found the SURF features so we will determine candidate matches based on descriptor similarity

```
>> m = s1.match(s2)
m =
323 corresponding points (listing suppressed)
```

and then estimate the fundamental matrix

```
>> [F,r] = m.ransac(@fmatrix,1e-4, 'verbose');
77 trials
82 outliers
0.000131656 final residual
```

which captures the relative geometry of the two views. We can display the epipolar lines for a subset of points overlaid on the the left-hand image

```
>> cam = CentralCamera('image', im1);
>> cam.plot_epiline(F', m.inlier.subset(30).p2, 'r');
```

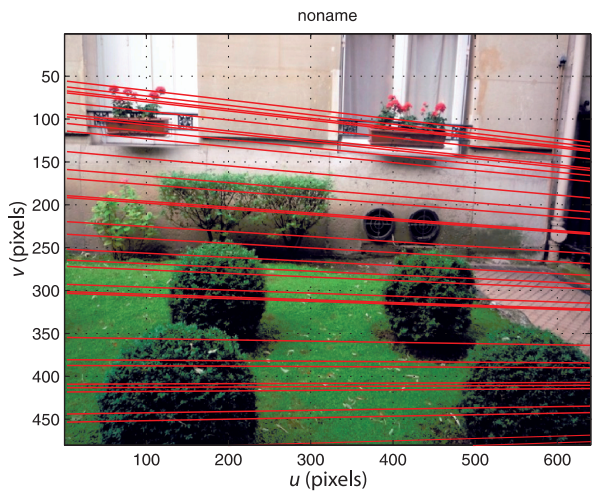


Fig. 14.16. Image of Fig. 14.15a with epipolar lines for selected right image points superimposed

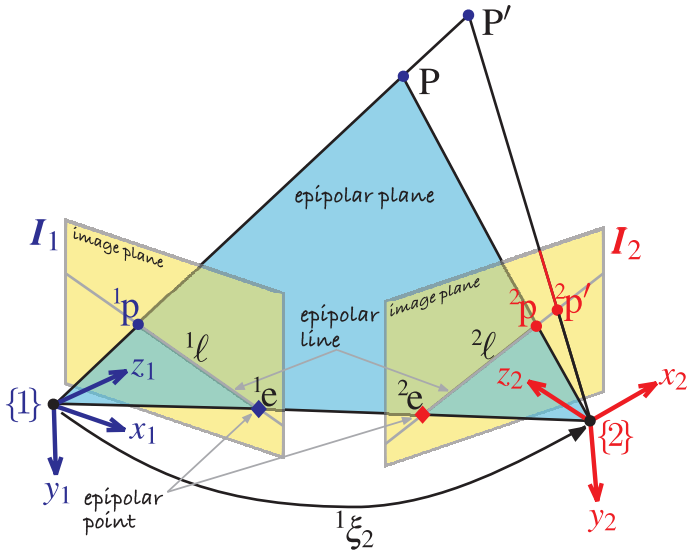


Fig. 14.17.
Epipolar geometry for stereo vision. We can see clearly that as the depth of the world point increases, from **P** to **P**′, the projection moves along the epipolar line in the second image plane

which is shown in Fig. 14.16. In this case the epipolar lines are approximately horizontal and parallel which is expected for a camera motion that is a pure translation in the x-direction. Figure 14.17 shows the epipolar geometry for stereo vision. It is clear that as points move away from the camera, **P** to **P**′ the conjugate points in the right-hand image moves to the right along the epipolar line.

The points ${}^1\boldsymbol{p}$ and ${}^2\boldsymbol{p}$ each define a ray in space which intersect at the world point. However to determine these rays we need to know the two poses of the camera and its intrinsic parameters. We can consider that the camera one frame {1} is the origin but the pose of camera two remains unknown. However we could estimate its pose by decomposing the essential matrix computed between between the two views. We have the fundamental matrix, but to determine the essential matrix according to Eq. 14.11 we need the camera's intrinsic parameters. With a little sleuthing we can find them!

The focal length used when the picture was taken is stored in the header of the EXIF-format file that holds the image and can be examined

```
>> [im1,tags] = iread('garden-l.jpg', 'double', 'mono');
```

where `tags` is a MATLAB® struct that contains various characteristics of the image as a structure of text strings. The element `DigitalCamera` describes the camera

```
>> tags.DigitalCamera
ans =
               ExposureTime: 0.0667
                    FNumber: 3.3000
                          .
                          .
                FocalLength: 5.2000
                          .
                          .
```

from which we determine the focal length is 5.2 mm.

The dimensions of the pixels $\rho_w \times \rho_h$ are not included in the image header but some web-based research on this model camera gives the answer. This camera is reported to have an image sensor that measures 7.18 × 5.32 mm and has 3 264 × 2 448 pixel – both sets of numbers are consistent with a 4:3 aspect ratio. From this we can determine that the pixels are square and have a side length of 2.2 μm. We create a `CentralCamera` object based on the known focal length, pixel size and image dimension

```
>> cam = CentralCamera('image', im1, 'focal', 5.2e-3, ...
    'sensor', [7.18e-3,5.32e-3])
cam =
name: noname [central-perspective]
  focal length:   0.0052
  pixel size:     (1.122e-05, 1.108e-05)
  principal pt:   (320, 240)
  number pixels:  640 x 480
  Tcam:
      1  0  0  0
      0  1  0  0
      0  0  1  0
      0  0  0  1
```

The effective pixel dimension is computed from the sensor dimensions and the pixels dimensions of the image. The image has been subsampled to 640×480 pixels so the effective pixel size of 11.2 μm is correspondingly larger. In the absence of any other information the principal point is assumed to be in the centre of the image.

The essential matrix is obtained by applying the camera intrinsic parameters to the fundamental matrix

```
>> E = cam.E(F)
E =
   -0.0064   -0.5401    0.2240
    0.6914   -0.2065   -4.0732
   -0.1902    4.0445   -0.2493
```

and we then decompose it to determine the camera motion

```
>> sol = cam.invE(E, [0,0,10]')
sol =
    0.9995   -0.0011   -0.0331   -4.0549
    0.0029    0.9985    0.0554   -0.2324
    0.0329   -0.0555    0.9979   -0.6766
         0         0         0    1.0000
>> [R,t] = tr2rt(sol);
```

We chose a test point ${}^1\mathbf{P} = (0, 0, 10)$, a distant point along the optical axis, to determine the correct solution for the relative camera motion. Since the camera orientation was kept fairly constant the rotational part of the transformation is expected to be close to the identity matrix as we observe, and the actual rotation

```
>> tr2rpy(R, 'deg')
ans =
   -3.1781   -1.8940    0.0602
```

is just a couple of degrees of rotation about the x- and y-axes.

The estimated translation $\boldsymbol{t}$ from {1} to {2} has an unknown scale factor. Once again we bring in an extra piece of information – when we took the images the camera position changed by approximately 0.3 m in the positive x-direction. The estimated translation has the correct direction, dominant x-axis motion, but the sign and magnitude are quite wrong. We therefore scale the translation

```
>> t = 0.3 * t/t(1);
>> T = rt2tr(R, t)
T =
    0.9995   -0.0011   -0.0331    0.3000
    0.0029    0.9985    0.0554    0.0172
    0.0329   -0.0555    0.9979    0.0501
         0         0         0    1.0000
```

and we have ${}^1\xi_2$ – the relative pose of camera two with respect to camera one represented as a homogeneous transformation.

Each point $\mathbf{p}$ in an image corresponds to a ray in space

$$\mathbf{P} = \alpha \boldsymbol{d} + \mathbf{P}_0, \quad \forall \alpha > 0$$

where $\mathbf{P}_0$ is the centre of the camera and $\boldsymbol{d}$ is the direction of the ray. If the camera matrix is $\boldsymbol{C} = (\boldsymbol{\Pi}, \boldsymbol{c}_4)$ where $\boldsymbol{\Pi} \in \mathbb{R}^{3\times3}$ and $\boldsymbol{c}_4 \in \mathbb{R}^3$ then the parameters of the ray are

$$\boldsymbol{d} \simeq \boldsymbol{\Pi}\tilde{\boldsymbol{p}}, \quad \mathbf{P}_0 = \boldsymbol{\Pi}\boldsymbol{c}_4$$

Consider now the first corresponding point pair `m(1)`. The ray from camera one is

```
>> r1 = cam.ray(m(1).p1)
r1 =
d=(0.429152, -0.210292, 0.878411), P0=(0, 0, 0)
```

which is an instance of a `Ray3D` object with properties `P0` and `d` representing $\mathbf{P}_0$ and $\boldsymbol{d}$ respectively. The corresponding ray from the second camera is

```
>> r2 = cam.move(T).ray(m(1).p2)
r2 =
d=(0.38276, -0.21969, 0.897347), P0=(0.3, 0.017194, 0.0500546)
```

where the `move` method returns an instance copy of the `CentralCamera` object `cam` with the specified relative pose. The two rays intersect at

```
>> [x,e] = r1.intersect(r2);
>> x'
ans =
    2.2031   -1.0796    4.5094
```

which is a point with a z-coordinate, or depth, of 4.51 m. Due to errors in the estimate of camera two's pose the two rays do not actually intersect, but their closest point is returned. At their closest point the rays are

```
>> e
e =
    0.0049
```

around 5 cm apart. Considering the lack of rigour in this exercise, two handheld camera shots and only approximate knowledge of the magnitude of the camera displacement, the recovered depth information is quite remarkable.▸

Even small errors in the estimated rotation between the camera poses will lead to large closing errors at distances of several metres. The closing error observed here would be induced by a rotational error of less than 1 deg.

We draw a subset of twenty corresponding points from the inlier set

```
>> m2 = m.inlier.subset(20);
```

and then compute the rays in world space from each camera

```
>> r1 = cam.ray( m2.p1 );
>> r2 = cam.move(T).ray( m2.p2 );
```

which are each vectors of `Ray3D` objects. Their intersection points are

```
>> [P,e] = r1.intersect(r2);
```

where `P` is a matrix of closest points, one per column, and the last row

```
>>  z = P(3,:);
```

is the depth coordinate. The columns of the vector `e` contains the distance between the rays at their closest points. We can superimpose the distance to each point on the image of the garden

```
>> idisp(im1)
>> plot_point(m.inlier.subset(20).p1, 'w*', 'textcolor', 'w', ...
    'printf', {'%.1f', z});
```

which is shown in Fig. 14.18 and the feature markers are annotated with the estimated depth.

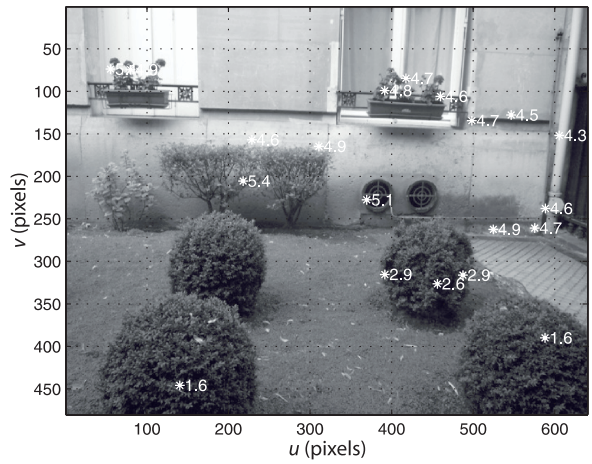


Fig. 14.18.
Image of Fig. 14.15a with depth of selected points indicated (units of metres)

Fig. 14.19.
A stereo vision sensor which with PC-based software can compute depth maps at 25 frames per second. The cameras and lenses are identical and the relative pose is constant (image courtesy of Point Grey Research Inc.)

This is an example of stereopsis where we have used information from two overlapping images to infer the 3-dimensional position of points in the world. For obvious reasons the approach used here is referred to as sparse stereo because we only compute distance at a tiny subset of pixels in the image. More commonly the relative pose between the cameras would be known as would the camera intrinsic parameters.

14.3.2 Dense Stereo Matching

A stereo pair is taken by two cameras, generally with parallel optical axes, and separated by a known distance referred to as the camera baseline. Figure 14.19 shows a typical stereo camera system which simultaneously captures images from both cameras and transfers them to a host computer for processing.

To illustrate we load an example stereo pair

```
>> L = iread('rocks2-l.png', 'reduce', 2);
>> R = iread('rocks2-r.png', 'reduce', 2);
```

We can interactively examine these two images together

```
>> stdisp(L, R)
```

as shown in Fig. 14.20. Clicking on a point in the left-hand image updates a pair of cross hairs that mark the *same* coordinate in the right-hand image. Clicking in the right-hand image sets another vertical cross hair and displays the difference between the horizontal coordinate of the two crosshairs. The cross hairs as shown are set to a small white spot on the front leftmost rock and we observe several things. Firstly the spot has the same vertical coordinate in both images, and this implies that the epipolar line is horizontal. Secondly, in the right-hand image the spot has moved to the left by 82.4 pixel. If we probed more points we would see that disparity is greater for points that are closer to the camera.

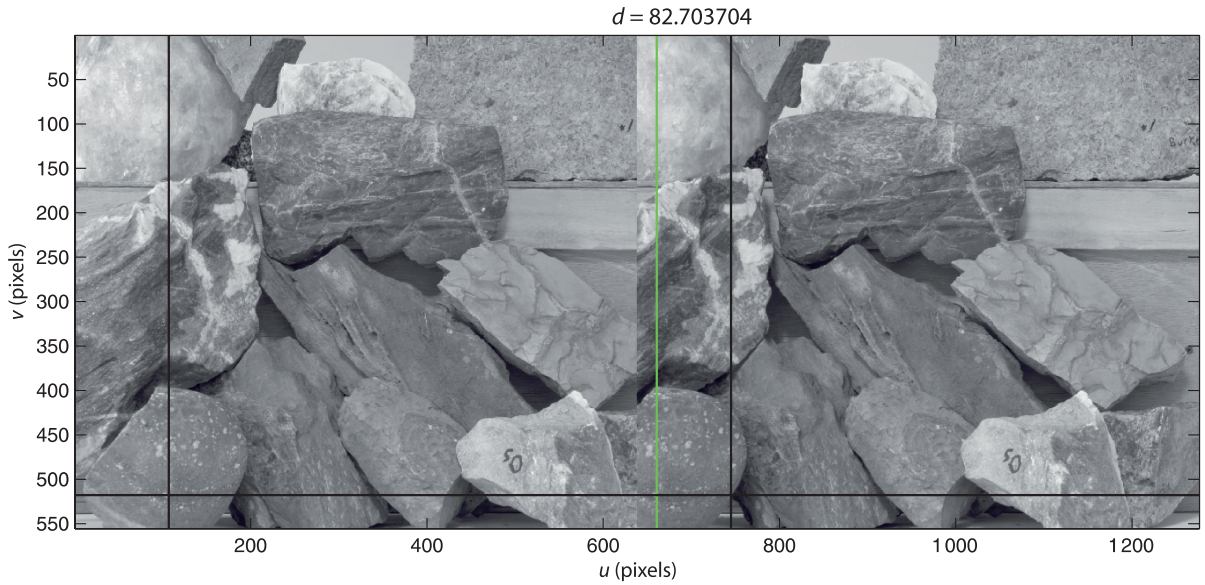


Fig. 14.20. The `stdisp` image browsing window. The black cross hair in the left-hand image has been positioned on a small white spot on the bottom-left rock. Another black cross hair is automatically positioned at the same coordinate in the right-hand image. Clicking on the corresponding point in the right-hand image sets the green cross-hair, and the panel at the top indicates a horizontal shift of 82.7 pixel to the left. This stereo image pair is from the Middlebury stereo database (Scharstein and Pal 2007). The focal length f/ρ is 3740 pixel, and the baseline is 160 mm. The images have been cropped so that the actual disparity should be offset by 274 pixel

As shown in Fig. 14.17 the conjugate point in the right-hand image moves rightward along the epipolar line as the point depth increases. For the parallel-axis camera geometry the epipolar lines are parallel and horizontal, so conjugate points have the same v-coordinate. If the coordinates of two corresponding points are $({}^Lu, {}^Lv)$ and $({}^Ru, {}^Rv)$ then ${}^Rv = {}^Lv$. The displacement along the horizontal epipolar line $d = {}^Lu - {}^Ru$ where $d \geq 0$ is called disparity.

The dense stereo process is illustrated in Fig. 14.21. For the pixel at $({}^Lu, {}^Lv)$ in the left-hand image we know that its corresponding pixel is at some coordinate $({}^Lu - d, {}^Lv)$ in the right-hand image where $d \in [d_{\min}, d_{\max}]$. To reliably find the corresponding point for a pixel in the left-hand image we create a $W \times W$ pixel *template* region $\boldsymbol{T}$ about that pixel. As shown in Fig. 14.21 we *slide* the template window horizontally across the right-hand image. The position at which the template is most similar is considered to be the corresponding point from which disparity is calculated. Compared to other matching problems we have encountered this one is much simpler because there is no change in relative scale or orientation between the two images.

The epipolar constraint means that we only need to perform a 1-dimensional search for the corresponding point. The template is moved in D steps of 1 pixel in the range $d_{\min} \cdots d_{\max}$. At each template position we perform a template matching operation, such as we discussed in Sect. 12.4.2, and for a $W \times W$ window these have a computational cost of $O(W^2)$. The total cost of dense stereo matching is $O(DW^2N^2)$ which is high but feasible in real time.

To perform stereo matching for the image pair in Fig. 14.20 using the Toolbox is quite straightforward

```
>> d = istereo(L, R, [40, 90], 3);
```

The result is a matrix the same size as `L` and the value of each element $\boldsymbol{d}[u, v]$, or `d(v,u)` in MATLAB®, is the disparity at that coordinate in the left image. The corresponding pixel in the right image would be at $(u - \boldsymbol{d}[u, v], v)$. We can display the disparity as a disparity image

```
>> idisp(d, 'bar')
```

which is shown in Fig. 14.22. Disparity images have a distinctive ghostly appearance since all surface color and texture is absent. The third argument to `stereo` is the range of disparities to be searched, in this case from 40 to 90 pixel so the pixel values in the disparity image lie in the range [40, 90]. The disparity range was determined by examining some far and near points using `stdisp`.▸ The fourth argument to `istereo`

▸ We could chose a range such as `[0, 90]` but this increases the search time: 91 disparities would have to be evaluated instead of 51. It also increases the possibility of matching errors.

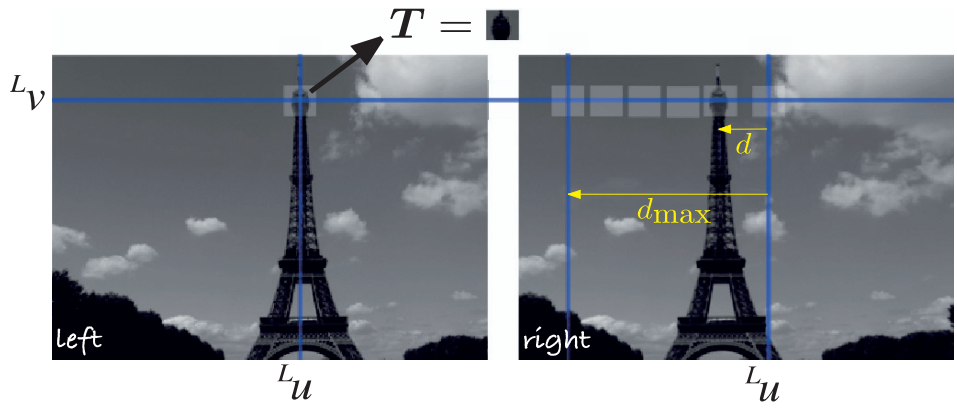


Fig. 14.21. Stereo matching. A search centred at (u_L, v_L) in the right image is swept horizontally until it matches the *template* window T from the left image

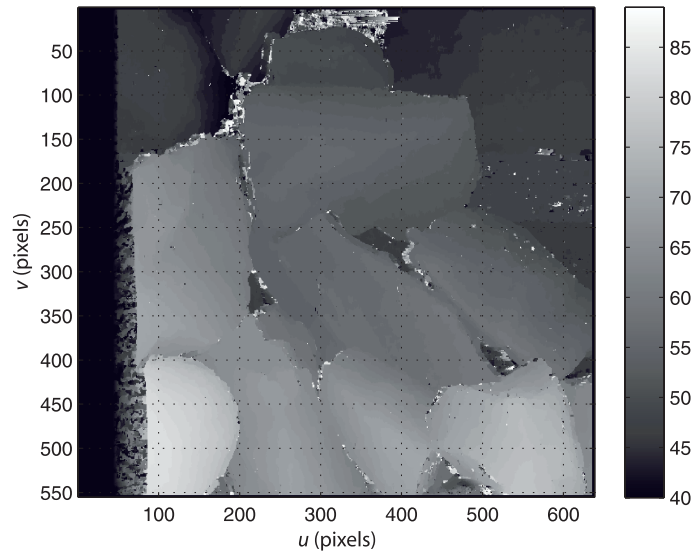


Fig. 14.22. Disparity image for the rock pile stereo pair, where brighter means higher disparity or shorter range

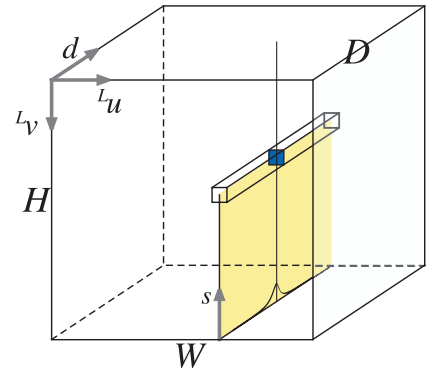


Fig. 14.23. The disparity space image is a 3-dimensional image where element $D(u, v, d)$ is the similarity between the support regions centered at $({}^Lu, {}^Lv)$ in the left image and $({}^Lu - d, {}^Lv)$ in the right image

is the half-width of the template, in this case we are using a 7×7 window. By default `stereo` uses the ZNCC similarity measure.

In the disparity image we can clearly see that the rocks at the bottom of the pile have a larger disparity and are closer to the camera than those at the top. There are also some errors, such as the anomalous bright values around the edges of the rocks. These pixels are indicated as being nearer than they really are. The similarity score is set to `NaN` around the edge of the image where the similarity matching template falls off the edge of the image and to `Inf` for the case where the denominator of the ZNCC similarity metric (Table 12.1) is equal to zero.◂ The values `NaN` and `Inf` are both displayed as black.

◂ This occurs if all the pixels in either template have exactly the same value.

14.3.2.1 Stereo Failure Modes

The stereo function can also return the disparity space image (DSI)

```
>> [d,sim,DSI] = istereo(L, R, [40 90], 3);
```

which is an $H \times W \times D$ matrix

```
>> about(DSI)
DSI [double] : 555x638x51 (144468720 bytes)
```

This is a large matrix (144 Mbyte) which is why the images were reduced in size when loaded.

whose elements (v, u, d) are the similarity measure between the templates centred at (u, v) in the left image and $(u - d, v)$ in the right image. The disparity image we saw earlier is simply the position of the maximum value in the d-direction evaluated at every pixel and the matrix `sim` is the value of those maxima.

This is a simplistic approach. A better approach is to apply regularization and estimate a function $g(u, v)$ that fits the points of maximum similarity while maintaining smoothness and continuity.

Each column in the d-direction gives the similarity measure versus disparity for the corresponding pixel in the left image. For the pixel at (138, 439) we can plot this

```
>> plot( squeeze(DSI(439,138,:)), 'o-');
```

which is shown in Fig. 14.24a. We are using the ZNCC measure and we see a perfect match occurs at a disparity of 80 pixel, since the horizontal axis is $d - d_{\min}$ and $d_{\min} = 40$.

Figure 14.24 shows some very typical plots of similarity metric as a function of disparity and this provides insight into the causes of error in stereo matching. Figure 14.24a shows a single unambiguous strong peak nearly equal to one which is the value for a perfect match. Fortunately this case is very common.

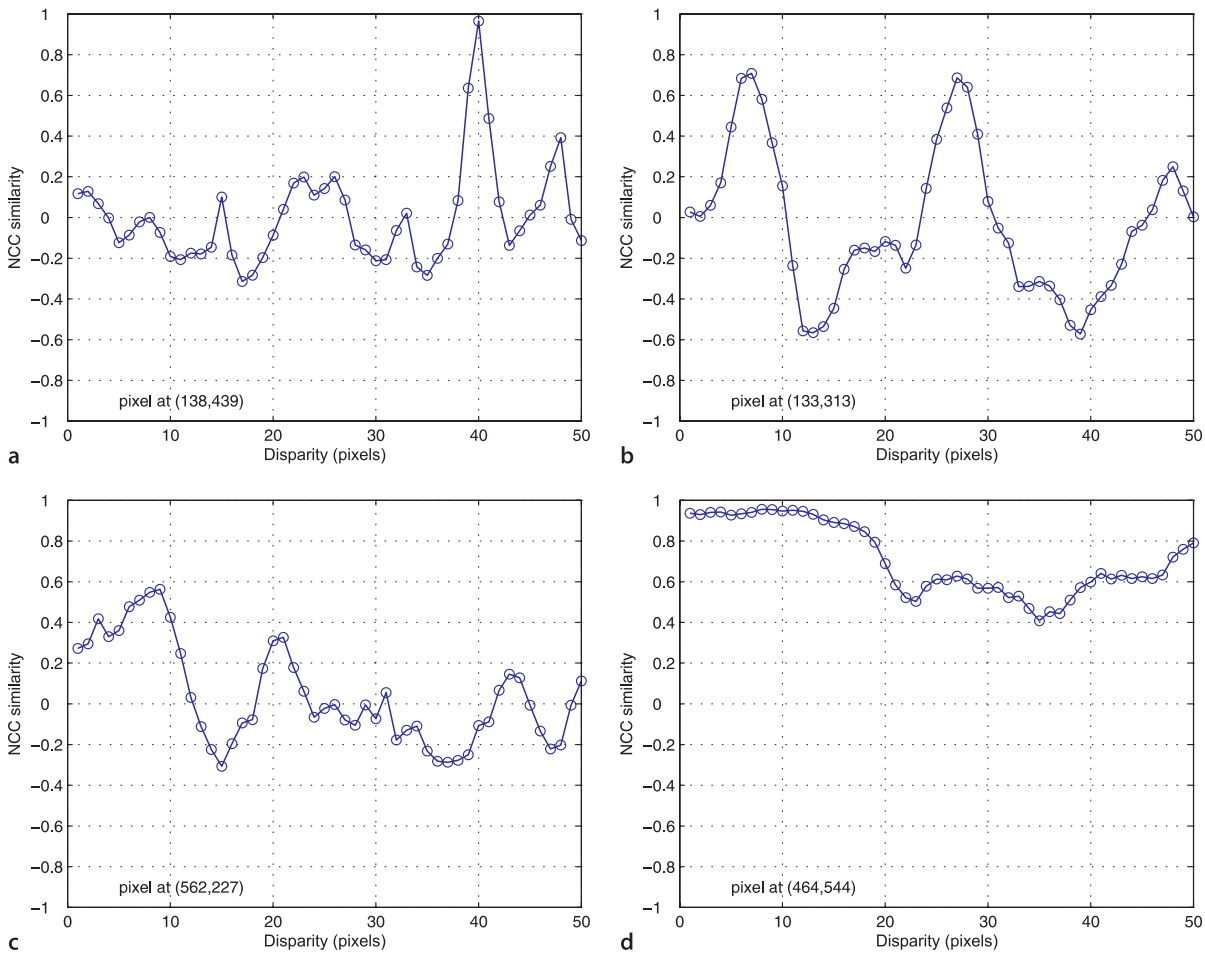


Fig. 14.24. Some typical ZNCC metric versus disparity curves. **a** Single strong peak; **b** multiple peaks; **c** weak peak; **d** broad peak

Two peaks of almost similar amplitude are shown in Fig. 14.24b. This means that the template pattern was found twice in the search region. This typically occurs when there are regular vertical features in the scene as is often the case in man-made scenes: brick walls, rows of windows, architectural features or a picket fence. The problem, illustrated in Fig. 14.25, is commonly known as the picket fence effector and more properly as spatial aliasing. There is no real cure for this problem◂ but we can detect its presence. The ambiguity ratio is the ratio of the height of second peak to the height of the first peak – a high-value indicate that the result is uncertain and should not be used. The chance of detecting incorrect peaks can be reduced by ensuring that the disparity range used in `istereo` is as small as possible and this requires some knowledge of the expected range of objects.

Multi-camera stereo, more than two cameras, is a powerful method to solve this ambiguity.

A weak peak is shown in Fig. 14.24c. This typically occurs when the corresponding scene point is not visible in the right-hand view due to occlusion or the missing parts problem. Occlusion is illustrated in Fig. 14.26 and it is clear that point 3 is only visible to the left camera. Dense stereo matching will attempt to find the best match in the right-hand image, but it will not find a point corresponding to 3. Even though the figure is an exaggerated depiction, real images suffer this problem where the depth changes rapidly. For example, this occurs at the edges of the rocks which is exactly where we observe the incorrect disparities in Fig. 14.22. The problem becomes more prevalent as the baseline increases. The problem also occurs when the corresponding point does not lie within the disparity search range, that is, the disparity search range is too small.

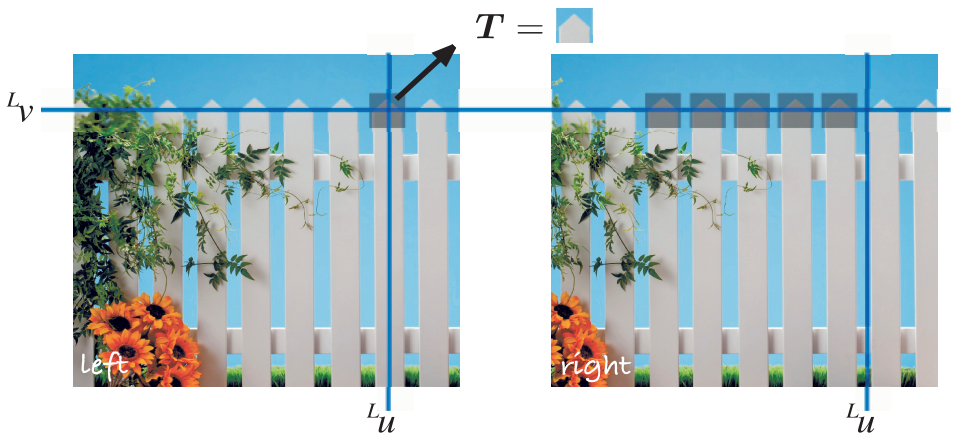


Fig. 14.25. Picket fence effect. The template will match well at a number of different disparities. This problem occurs in any scene with repeating patterns

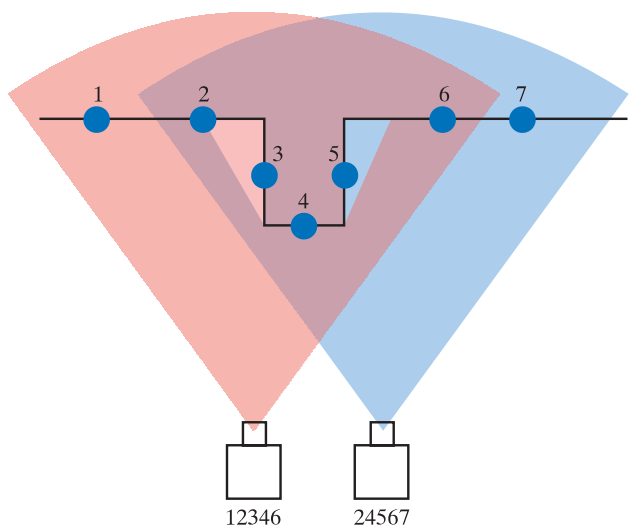


Fig. 14.26. Occlusion in stereo vision. The field of view of the two cameras are shown as colored sectors. Points 1 and 7 fall outside the overlapping view area and are seen by only one camera each. Point 5 is occluded from the left camera and point 3 is occluded from the right camera. The order of points seen by the cameras is given

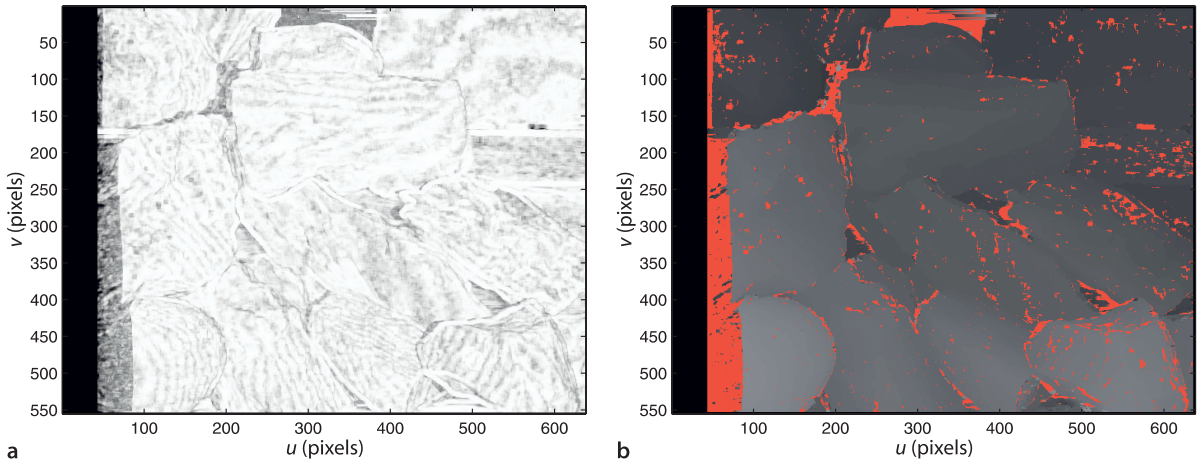


Fig. 14.27. Stereo template similarity. **a** Similarity image where brighter means higher similarity; **b** disparity image with pixels having low similarity score marked in red

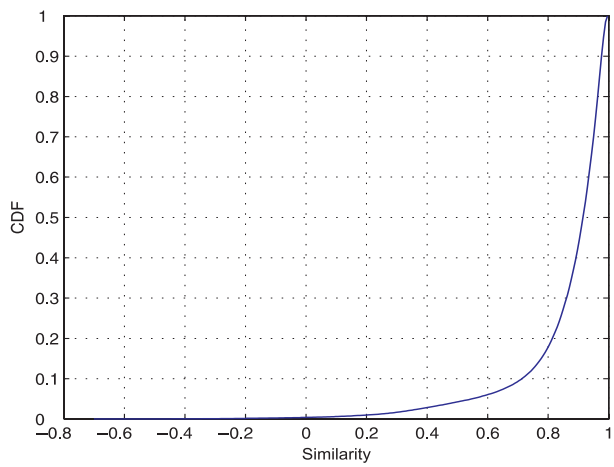


Fig. 14.28. Cumulative probability of ZNCC scores. The probability of a score less than 0.9 is 45%

The problem cannot be cured but it can be detected. The simplest method is to apply a threshold to the similarity score and ignore those disparity results where similarity is low. The `istereo` function can return the value of the similarity score at the peak

```
>> [d,sim] = istereo(L, R, [40 90], 3);
>> idisp(sim)
```

and this is shown in Fig. 14.27a. We see that the erroneous disparity values correspond to low maximum similarity score. For example

```
>> ipixswitch(sim<0.7, 'red', d/90);
```

shown in Fig. 14.27b displays pixels with similarity $s < 0.7$ as red. The distribution of maximum similarity scores

```
>> ihist(sim(isfinite(sim)), 'normcdf');
```

is shown in Fig. 14.28. We see that only 5% of pixels have a similarity score less than 0.5, and that 55% of pixels have a similarity score greater than 0.9.

A more powerful means to test for occlusion is to perform the matching in two directions which is known as the left-right consistency check. Starting with a pixel in the left-hand image the strongest match in the right-image is found. Then the strongest match to that pixel is found in the left-hand image. If this is the original left-hand image point the match is considered valid, otherwise it is discarded.

From Fig. 14.26 it is clear that pixels on the left-side of the left-hand image may not overlap at all with the right-hand image – point 1 is outside the field of view of the

right-hand camera. This is the reason for the large number of incorrect matches on the left hand side of the disparity image in Fig. 14.22. It is common practice to discard the $d_{\max}$ left-most columns of the disparity image.

The final problem that can arise is a similarity function with a very broad peak as shown in Fig. 14.24d. The breadth makes it difficult to precisely estimate the maxima. This generally occurs when the template region has very low texture for example corresponding to the sky, dark shadows, sheets of water, snow, ice or smooth man-made objects. Simply put, in a region that is all grey, a grey template matches equally well with any number of grey candidate regions. One approach to detect this is to look at the variability of pixel values in the template using measures such as the difference between the maximum and minimum value or the variance of the pixel values. If the template has too little variance it is less likely to give a strong peak. Measures of peak sharpness can also be used to eliminate these cases and this is discussed in the next section.

For the various problem cases just discussed disparity cannot be determined, but the problem can be detected. This is important since it allows those pixels to be marked as having no known range and this allows a robot path planner to be prudent with respect to regions whose 3-dimensional structure cannot be reliably determined.

The design of a stereo-vision system has three degrees of freedom. The first is the baseline distance between the cameras. As it increases the disparities become larger making it possible to estimate depth to greater precision, but the occlusion problem becomes worse. The disparity search range needs to be set carefully. If the maximum is too large the chance of spatial aliasing increases but if too small then points close to the camera will generate incorrect and weak matches. A large disparity range also increases the computation time. Template size involves a tradeoff between computation time and quality of the disparity image. A small template size can pick up fine depth structure but tends to give results that are much noisier since a small template is more susceptible to ambiguous matches. A large template gives a smoother disparity image but results in greater computation time. It also more likely that the template will contain pixels that belong to objects at different depths which is referred to as the mixed pixel problem. This tends to cause poor quality matching at the edges of objects, and the resulting disparity map appears blurred.

An alternative way to look at the failure modes is to use MATLAB's volume visualization functions to create horizontal slices through the disparity space image

```
>> slice(DSI, [], [100 200 300 400 500], []); view(-52,18)
>> shading interp; colorbar
```

which is shown in Fig. 14.29. These are slices at constant v-coordinate, and within each of the ud-planes we see a bright path (high similarity values) that represents

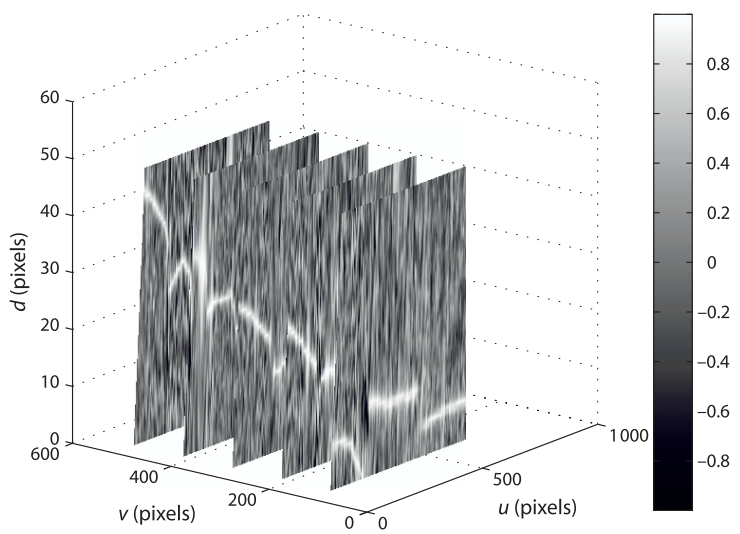


Fig. 14.29. The disparity space image is a 3-dimensional image where element $D(u, v, d)$ is the similarity between the support regions centered at $({}^{L}u, {}^{L}v)$ in the left image and $({}^{L}u - d, {}^{L}v)$ in the right image

disparity $d(u)$. Note the significant discontinuities in the path for the plane at $v = 100$ which correspond to sudden changes in depth. The planes at $v = 200, 300, 400$ show that the path also fades away in places. In these regions the maximum similarity is low, there is no strong match in the right-hand image, and the most likely cause is occlusion.

14.3.3 Peak Refinement

The disparity at each pixel is an integer value $d \in [d_{\min}, d_{\max}]$ at which the optimal similarity was found. Figure 14.24a shows a single unambiguous strong peak and we can use the disparity values adjacent to the peak to refine the estimate of the peak's position.▸ A parabola

This two-dimensional peak refinement is discussed in Appendix K.

$$s = Ad^2 + Bd + C \tag{14.15}$$

is fitted to the optimal disparity and its immediate neighbours. The optimal ZNCC similarity measure is a maxima which means that the parabola is inverted and $A < 0$. The interpolated maximum occurs when the derivative of Eq. 14.15 equals zero, from which we can estimate the horizontal position of the peak of the fitted parabola

$$\hat{d} = \frac{-B}{2A}$$

The A coefficient will be large for a sharp peak, and a simple threshold can be used to reject broad peaks, as we will discuss in the next section.

Disparity peak refinement is enabled with the `'interp'` option

```
>> [di,sim,peak] = istereo(L, R, [40 90], 3, 'interp');
>> idisp(di)
```

and the resulting disparity image is shown in Fig. 14.30a. We see that it is much smoother than the one shown previously in Fig. 14.22. The additional optional output argument `peak` is a structure

```
>> peak
peak =
     A: [555x638 double]
     B: [555x638 double]
```

that contains the per-pixel values of the parabola coefficients. The A coefficient is shown as an image in Fig 14.30b.

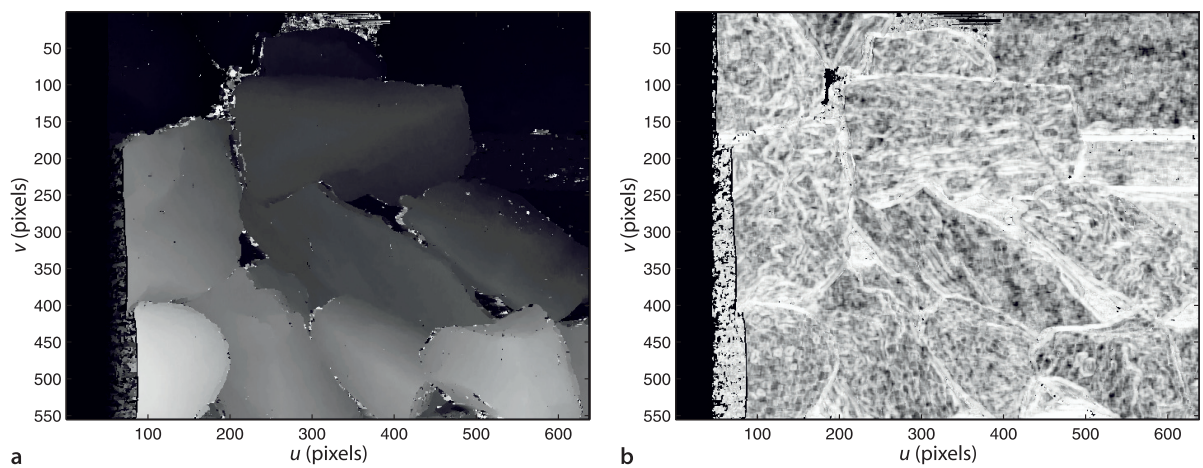


Fig. 14.30.a Disparity image with peak refinement; **b** coefficient of d^2 term at each peak. High values (bright) correspond to sharp peaks and occur where image texture is high. Broad peaks (dark) occur where image texture is low

14.3.4 Cleaning up and Reconstruction

The result of stereo matching, such as shown in Fig. 14.22, has a number of imperfections for the reasons we have just described. For robotic applications such as path planning and obstacle avoidance it is important to know the 3-dimensional structure of the world, but it is also critically important to know what we don't know. Where reliable depth information from stereo vision is missing a robot should be prudent and treat it differently to free space. We use a number of simple measures to mark elements of the disparity image as being invalid or unreliable.

We start by creating a matrix `status` the same size as `d` and initialized to zero

```
>> status = zeros(size(d));
```

The elements are set to non-zero values if they correspond to specific failure conditions

```
>> [U,V] = imeshgrid(L);
>> status(isnan(d)) = 4;
>> status(U<=90) = 1;
>> status(sim<0.8) = 2;
>> status(peak.A>=-0.1) = 3;
```

We can display this matrix as an image

```
>> idisp(status)
>> colormap( colorname({'lightgreen', 'cyan', 'blue', 'orange', 'red'}) )
```

which is shown in Fig. 14.31. The colormap is chosen to display the status values as light green for a good stereo match, cyan if the disparity search range extends beyond the left edge of the right image, blue if the peak similarity is too small, orange if the peak is too broad, and red for `NaN` values where the search template would fall off the edge of the image. The good news is that there are a lot of light green pixels! In fact

```
>> sum(status(:)) / prod(size(status)) * 100
ans =
   86.0674
```

nearly 90% of disparity values pass our battery of quality tests. The blue pixels, indicating weak similarity, occur around the edges of rocks and are due to occlusion. The orange pixels, indicating a broad peak, occur in areas that are fairly smooth, either deep shadow between rocks or the non-rock background.

Earlier we created an interpolated disparity image `di` and now we will invalidate the disparity values that we have determined to be unreliable

```
>> di(status>0) = NaN;
```

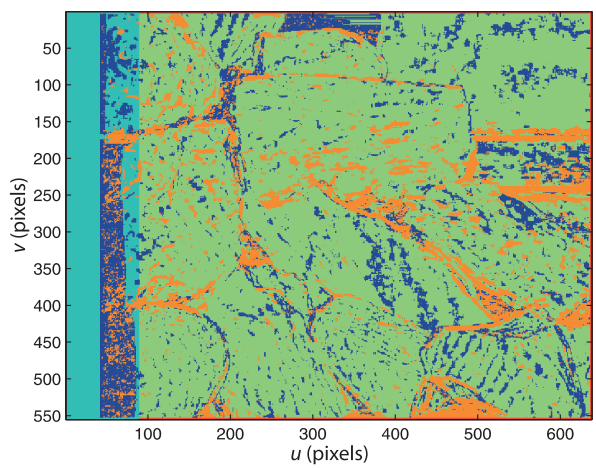


Fig. 14.31. Stereo matching status on a per pixel basis. Good stereo match (green); disparity search range extends beyond the left edge of the right image (cyan); low maximum similarity (blue); no-sharp peak (orange); search template beyond edge of image (red)

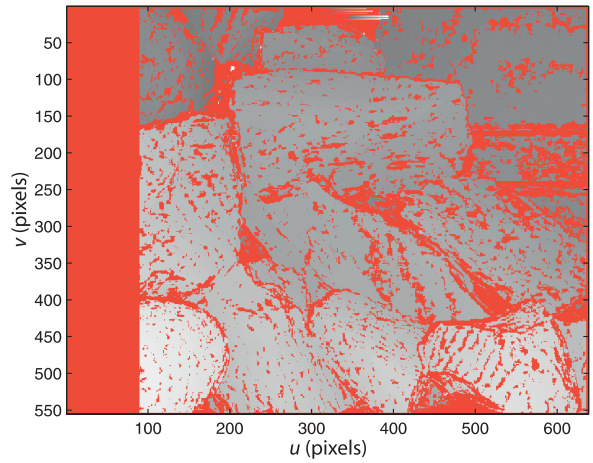


Fig. 14.32.
Interpolated disparity image with unreliable estimates indicated in red

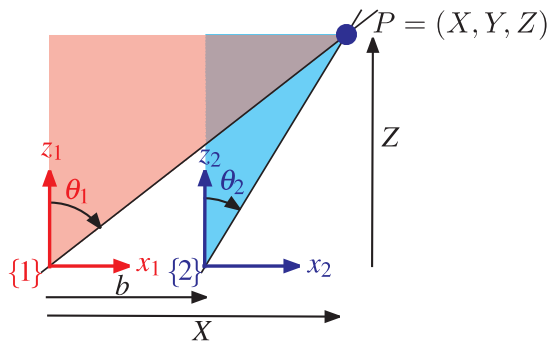


Fig. 14.33.
Stereo geometry for parallel camera axes. X and Z are measured with respect to camera one, b is the baseline

by setting them to the value `NaN`.[1] We can display this with the *unreliable* pixels marked in red by

```
>> ipixswitch(isnan(di), 'red', di/90);
```

which is shown in Fig. 14.32.[2] This is now in useful form for a robot – it contains disparity values interpolated to better than a pixel and all unreliable values are clearly marked.

The final step is to convert the disparity values in pixels to world coordinates in metres – a process known as 3D reconstruction. In the earlier discussion on sparse stereo we determined the world point from the intersection of two rays in 3-dimensional space. For a parallel axis stereo camera rig as shown in Fig. 14.19 the geometry is much simpler as illustrated in Fig. 14.33. For the red and blue triangles we can write

$$X = Z\tan\theta_1, \quad X - b = Z\tan\theta_2$$

where b is the baseline and the angles of the rays correspond to the horizontal image coordinate ${}^{i}u$, $i = \{L, R\}$

$$\tan\theta_i = \frac{\rho_u({}^{i}u - u_0)}{f}$$

Substituting and eliminating X gives

$$Z = \frac{fb}{\rho_u({}^{L}u - {}^{R}u)} = \frac{fb}{\rho_u d}$$

which shows that depth is inversely proportional to disparity $d = {}^{L}u - {}^{R}u$ and $d > 0$.

[1] The special floating point value `NaN` (for not a number) has the useful property that the result of any arithmetic operation involving `NaN` is always `NaN`. Many MATLAB® functions such as `max` or `min` ignore `NaN` values in the input matrix, and plotting and graphics functions do not display this value, leaving a hole in the graph or surface.

[2] The division by 90 is to convert the floating point disparity values in the range [40, 90] into valid greyscale values in the range [0, 1].

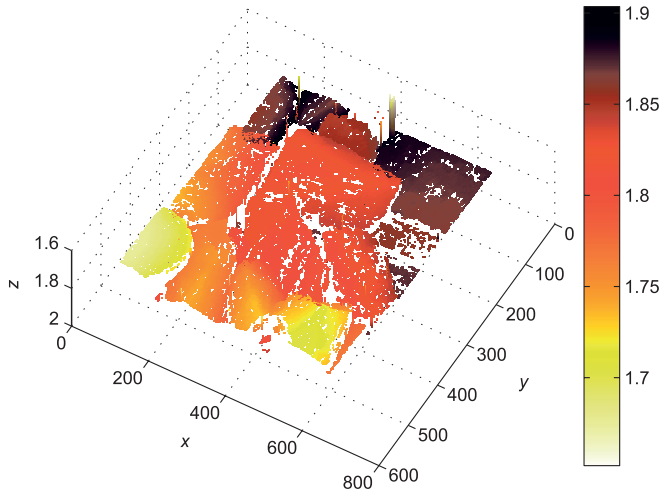


Fig. 14.34. 3-dimensional reconstruction for parallel stereo cameras. Hotter colors indicate parts of the surface that are closer to the camera

We can also recover the X- and Y-coordinates

$$X = \frac{b({}^{L}u - u_0)}{d}, \quad Y = \frac{b({}^{L}v - v_0)}{d}$$

The images shown in Fig. 14.20, from the Middlebury dataset, were taken with a very wide camera baseline. The left edge of the left-image and the right edge of the right-image have no overlap and have been cropped. Cropping N pixels from the left of the left-hand image reduces the disparity by N. For this stereo pair the actual disparity must be increased by 274 to account for the cropping.

The true disparity is

```
>> di = di + 274;
```

and we compute the X-, Y- and Z-coordinate of each pixel as separate matrices to exploit MATLAB's efficient matrix operations◂

A process known as vectorizing. Using matrix and vector operations instead of `for` loops greatly increases the speed of MATLAB® code execution. See http://www.mathworks.com/support/tech-notes/1100/1109.html for details.

```
>> [U,V] = imeshgrid(L);
>> u0 = size(L,2)/2; v0 = size(L,1)/2;
>> b = 0.160;
>> X = b*(U-u0) ./ di; Y = b*(V-v0) ./ di; Z = 3740 * b ./ di;
```

which can be displayed as a surface

```
>> surf(Z)
>> shading interp; view(-150, 75)
>> set(gca,'ZDir', 'reverse'); set(gca,'XDir', 'reverse')
>> colormap(flipud(hot))
```

as shown in Fig. 14.34. This is somewhat unimpressive in print but by using the mouse to rotate the image using the MATLAB® figure toolbar *3D rotate* option the 3-dimensionality becomes quite clear. The axis reversals are required to have z increase from our viewpoint and to maintain a right-handed coordinate frame. There are many *holes* in this surface which are the `NaN` values we inserted to indicate unreliable disparity values.

14.3.5 3D Texture Mapped Display

For human, rather than robot, consumption it would be nice to enhance the surface representation so that it looks less ragged. We create a median filtered image

```
>> dimf = irank(di, 41, ones(9,9));
```

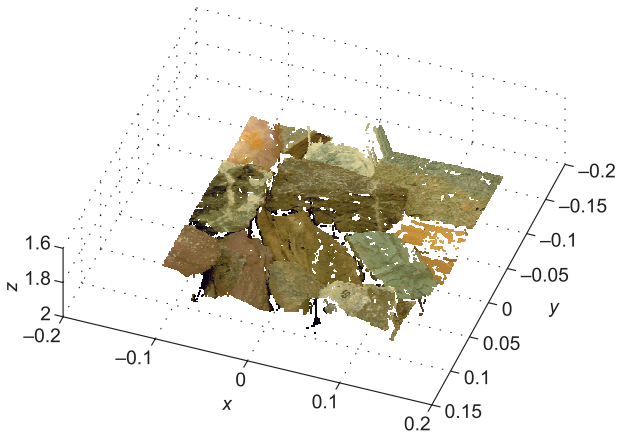


Fig. 14.35.
3-dimensional reconstruction for parallel stereo cameras with image texture mapped onto the surface

where each output pixel is the median value over a 9×9 window. This has *patched* many of the smaller holes but has the undesirable side effect of blurring the underlying disparity image. Instead we will keep the original interpolated disparity image and insert the median filtered values only where a `NaN` exists

```
>> di = ipixswitch(isnan(di), dimf, di);
```

We perform the reconstruction again

```
>> X = b*(U-u0) ./ di;  Y = b*(V-v0) ./ di; Z = 3740 * b ./ di;
```

and plotting this as a surface displays a surface that looks significantly less ragged.

However we can do even better. We can *drape* the left-hand image over the 3-dimensional surface using a process called texture mapping. We reload the left-hand image, this time in color

```
>> Lcolor = iread('rocks2-l.png');
```

and render the surface with the image texture mapped

```
>> surface(X, Y, Z, Lcolor, 'FaceColor', 'texturemap', ...
    'EdgeColor', 'none', 'CDataMapping', 'direct')
>> xyzlabel
>> set(gca,'ZDir', 'reverse'); set(gca,'XDir', 'reverse')
>> view(-160, 75)
```

which creates the image shown in Fig. 14.35. Once again it is easier to get an impression of the 3-dimensionality by using the mouse to rotate the image using the MATLAB® figure toolbar *3D rotate* option.

14.3.6 Anaglyphs

Human stereo perception of depth works because each eye views the scene from a different viewpoint. If we look at a photograph of a 3D scene we still get a perception of depth, albeit reduced, because our brain uses many visual cues besides stereo to infer depth. Since the invention of photography in the 19th century people have been fascinated by 3D photographs and movies, and the recent introduction of 3D television is further evidence of this.

The key in all 3D display technologies to take the image from two cameras, with a similar baseline to the human eyes (approximately 8 cm) and present those images again to the corresponding eyes. Old fashioned stereograms required a binocular viewing device or could, with difficulty, be viewed by squinting at the stereo pair and crossing your eyes. More modern and convenient means of viewing stereo pairs are LCD shutter glasses or polarized glasses which allow full-color stereo movie viewing.

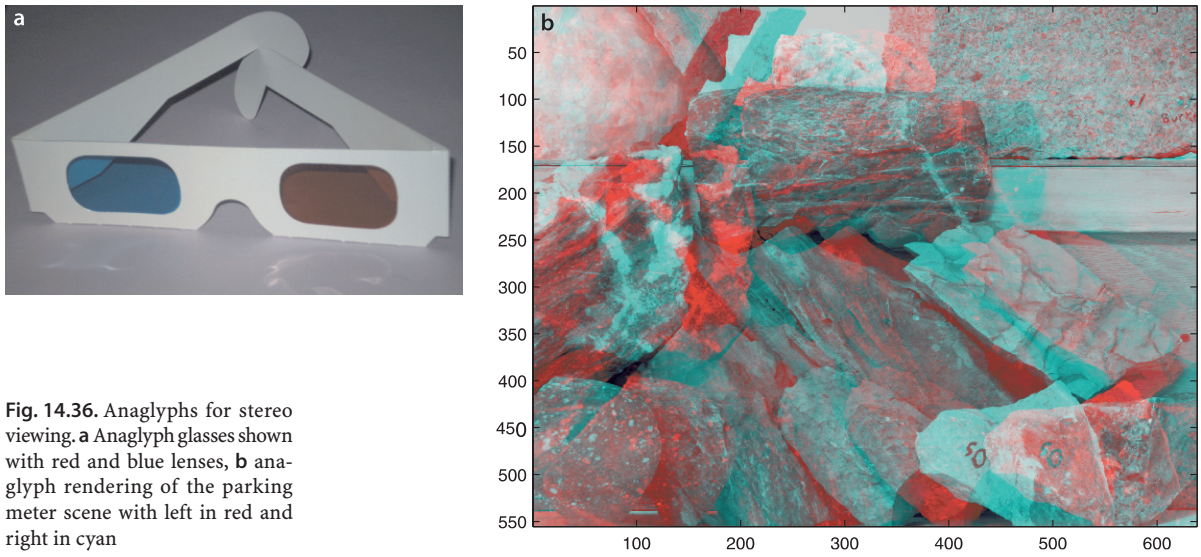


Fig. 14.36. Anaglyphs for stereo viewing. **a** Anaglyph glasses shown with red and blue lenses, **b** anaglyph rendering of the parking meter scene with left in red and right in cyan

Anaglyphs. The earliest developments occurred in France. In 1858 Joseph D'Almeida projected 3D magic lantern slide shows as red-blue anaglyphs and the audience wore red and blue goggles. Around the same time Louis Du Hauron created the first printed anaglyphs. Later, around 1890 William Friese-Green created the first 3D anaglyphic motion pictures using a camera with two lenses. Anaglyphic films called plasticons or plastigrams were a craze in the 1920s.

A century later stereo images from the surface of Marse are available on the web in anaglyph form at http://marsprogram.jpl.nasa.gov/MPF/mpf/anaglyph-arc.html.

An old but inexpensive method of viewing and distributing stereo information is through anaglyph images in which the left and right images are overlaid in different colors. Typically red is used for the left eye and cyan (greeny blue) for the right eye but many other color combinations are used. The red lens allows only the red part of the anaglyph image through to the left eye, while the cyan lens allows only the cyan parts of the image through to the right eye. The disadvantage is that only the scene intensity, not its color, can be portrayed. The big advantage of anaglyphs is that they can be printed on paper or imaged onto ordinary movie film and viewed with simple and cheap glasses such as those shown in Fig. 14.36a.

The rock pile stereo pair can be displayed as an anaglyph

```
>> anaglyph(L, R, 'rc')
```

which is shown in Fig. 14.36b. The argument `'rc'` indicates that left and right images are encoded in red and cyan respectively. Other color options include: blue, green, magenta and orange.

14.3.7 Image Rectification

The rock pile stereo pair of Fig. 14.20 has corresponding points on the same row in the left- and right-hand images. These are referred to as an epipolar-aligned image pair. Stereo cameras such as shown on page 405 are built with precision to ensure that the optical axes of the cameras are parallel and that the u- and v-axes of the two sensor chips are parallel. However there are limits to the precision of mechanical alignment and lens distortion will introduce error. Typically one or both images are warped to correct for these errors – a process known as rectification.

We will illustrate rectification using the garden stereo pair shown from Fig. 14.15

```
>> L = iread('garden-l.jpg', 'mono', 'double');
>> R = iread('garden-r.jpg', 'mono', 'double');
```

which we recall are far from being epipolar aligned. We first find the SURF features

```
>> sL = isurf(L);
>> sR = isurf(R);
```

and determine the candidate matches

```
>> m = sL.match(sR);
```

then determine the epipolar relationship

```
>> F = m.ransac(@fmatrix,1e-4, 'verbose');
46 trials
81 outliers
9.60297e-05 final residual
```

The rectification step requires the fundamental matrix as well as the set of corresponding points which is embedded in the `FeatureMatch` object `m`

```
>> [Lr,Rr] = irectify(F, m, L, R);
```

and returns rectified versions of the two input images. We display these using `stdisp`

```
>> stdisp(Lr, Rr)
```

which is shown in Fig. 14.37. We see that corresponding points in the scene now have the same vertical coordinate. The function `irectify` works by computing unique homographies to warp the left and right images. As we have observed previously when warping images not all of the output pixels are mapped to the input images which results in undefined pixels which are displayed here as black. This pair of rectified images could now be used for dense stereo matching

```
>> d = istereo(Lr, Rr, [5 120], 4);
```

and the result is shown in Fig. 14.38. We have been able to create a dense 3-dimensional representation of the scene using just two images taken from a handheld camera.

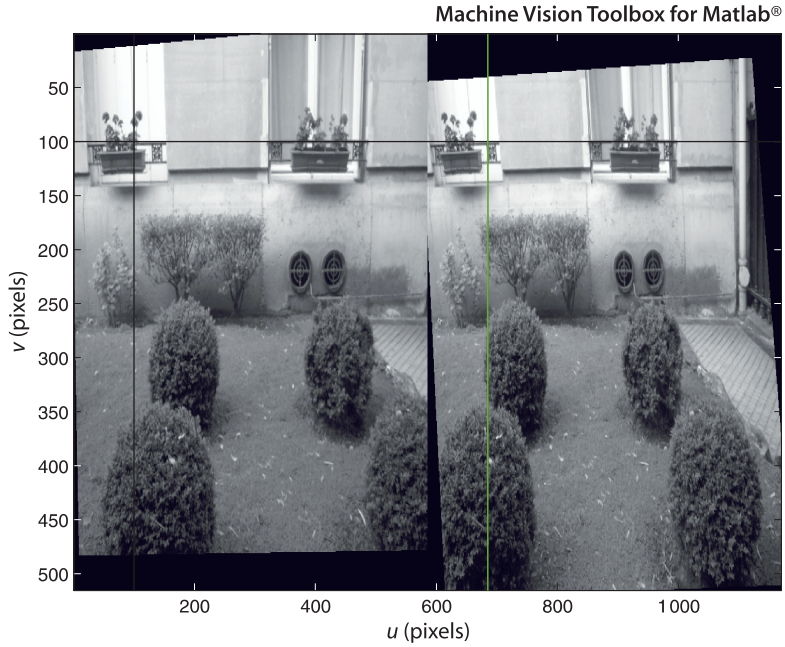


Fig. 14.37.
Rectified images of the garden

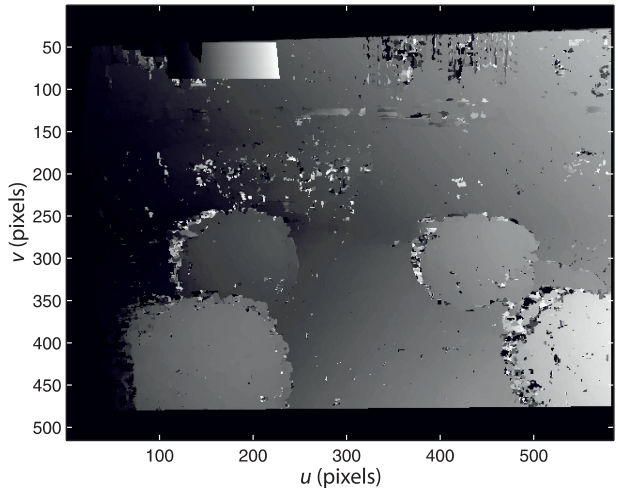


Fig. 14.38.
Dense stereo disparity image for the garden shows near objects as brighter than far objects

14.3.8 Plane Fitting

Stereo vision results in a set of 3-dimensional world points $\mathbf{P}_i$ which are referred to as a point cloud. A common problem is fitting a plane to such a set of points. One simple and effective approach is to fit an ellipsoid to the data, and the ellipsoid will have one very small radius in the direction normal to the plane – that is, it will be an elliptical plate. The inertia matrix of the points can be calculated by

$$\boldsymbol{J} = \sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{x}_i^T \tag{14.16}$$

where $\boldsymbol{x} = \boldsymbol{P}_i - \overline{\boldsymbol{P}}$ are the coordinates of the points with respect to the centroid of the points $\overline{\boldsymbol{P}} = {}^{1}/_{N} \sum_{i=1}^{N} \boldsymbol{P}_i$. The ellipsoid is centred at the centroid of the point cloud. The radii of the ellipsoid are the eigenvalues of $\boldsymbol{J}$ and the eigenvector corresponding to the smallest eigenvalue is the direction of the minimum radius which is the normal to the plane.

To illustrate this we create a 10×10 grid of points in a plane

```
>> T = transl(1,2,3) * rpy2tr(0.3, 0.4, 0.5);
>> P = mkgrid(10, 1, T);
>> P = P + 0.02*randn(size(P));
```

with an arbitrary orientation represented by the homogeneous transformation `T`, and to which some Gaussian noise has been added.

The mean of the point cloud is

```
>> x0 = mean(P')
ans =
    0.9980    1.9979    2.9973
```

and this is subtracted from all the data points

```
>> P = bsxfun(@minus, P, x0');
```

and then the moments are computed

```
>> J = 0;
>> for x = P,  J = J + x*x';  end
>> J
J =
    8.4143    1.0722   -3.3781
    1.0722    9.6187    2.5577
   -3.3781    2.5577    2.3441
```

The eigenvalues are

```
>> [x,lambda] = eig(J);
>> diag(lambda)'
ans =
    0.0357    9.9135   10.4279
```

and we see two large eigenvalues corresponding to the spread of points within the plane, and one eigenvalue which is the *thickness* of the plane. The eigenvector corresponding to the first, and smallest, eigenvalue is

```
>> n = x(:,1)'
n =
    0.3896   -0.2779    0.8781
```

which is the estimated normal to the plane.

The true direction of the plane's normal is given by the third column of the transformation matrix

Since the points lie in the frame's *xy*-plane, the normal is the frame's *z*-axis.

```
>> T
T =
    0.8083   -0.4416    0.3894    1.0000
    0.5590    0.7832   -0.2722    2.0000
   -0.1848    0.4377    0.8799    3.0000
         0         0         0    1.0000
```

and we see that it is very close to the estimated normal.

The equation of a plane is the set of points $\boldsymbol{x}$ such that

$$\boldsymbol{n}^T(\boldsymbol{x} - \boldsymbol{x}_0) = 0 \tag{14.17}$$

where $\boldsymbol{n}$ is the normal and $\boldsymbol{x}_0$ is the centroid.

Outlier data points are problematic with this type of estimator since they significantly bias the solution. A number of approaches are commonly used but a simple one is to modify Eq. 14.16 to include a weight

$$\boldsymbol{J} = \sum_{i=1}^{N} w_i \boldsymbol{x}_i \boldsymbol{x}_i^T$$

which is inversely related to the distance of $\boldsymbol{x}_i$ from the plane and solve iteratively. Initially all weights $w_i = 1$, and on subsequent iterations the weights are set according to the distance of $\mathbf{P}_i$ from the plane estimated at the previous step. Alternatively we could apply RANSAC by taking samples of three points to solve for Eq. 14.17. Appendix E has more details about ellipses.

Commonly a Cauchy-Lorentz function $w = r^2/(x^2 + r^2)$ is used which is smooth over the range of point distance $0 \rightarrow \infty$ and has a value of ½ when $x = r$.

14.3.9 Matching Sets of 3D Points

Consider a model of some object represented by a set of points in 2- or 3-dimensions with respect to the world frame. Now consider an example of that object somewhere in the world and we observe a set of 2- or 3-dimensional points on the object. The task is to determine the relative pose ξ that will transform the model points to the observed data points by matching the two sets of points.

The dual problem is that the camera has moved, not the object. The same technique can be applied to determine the camera motion.

More formally, given two sets of points $\mathbf{M}_i, \mathbf{D}_j \in \mathbb{R}^3$, $i, j \in 1 \cdots N$ in the world frame determine the relative pose ξ such that

$$\boldsymbol{D}_i = \xi \cdot \boldsymbol{M}_i$$

The points $\mathbf{M}_i$ are the *model* of a 3-dimensional object which we want to fit to the observed *data* $\mathbf{D}_j$.

To illustrate we will create a model which is a cloud of 100 random points in a $1 \times 1 \times 1$ m cube centred at the origin

The technique can work for 2- or 3-dimensional data but we will illustrate it here for 3-dimensional data.

```
>> M = rand(3,100) - 0.5;
```

The data is a copy of the model that has been rotated and translated

```
>> T_unknown = transl(1, 2, 3) * rpy2tr(0.2, 0.3, 0.4);
>> D = homtrans(T_unknown, M);
```

by an unknown relative pose ξ.

At first glance this looks like a problem where we need to establish correspondence between the points in the two sets but we will introduce an alternative approach called iterative closest point or ICP. For each data point $\mathbf{D}_i$, the corresponding model point $\mathbf{M}_j$ is assumed to be the closest one, that is $\mathbf{M}_j$ which minimizes $|\mathbf{D}_i - \mathbf{M}_j|$. Correspondence is not unique and quite commonly several data points can be associated with a single model point, and consequently some data points $\mathbf{D}_j$ will be unpaired. This approach to correspondence is far from perfect but it is (generally) good enough to *improve* the alignment of the point clouds so that in the next iteration the computed correspondences will be a little more accurate.

The first step is to compute a translation that makes the centroids of the two point clouds coincident

$$\bar{\boldsymbol{M}} = \frac{1}{N_M} \sum_{i=1}^{N_M} \boldsymbol{M}_i$$

$$\bar{\boldsymbol{D}} = \frac{1}{N_D} \sum_{i=1}^{N_D} \boldsymbol{D}_i$$

from which we compute a displacement

$$\boldsymbol{t} = \bar{\boldsymbol{D}} - \bar{\boldsymbol{M}}$$

Next we compute correspondence. For each data point $\mathbf{D}_i$ we find the closest model point $\mathbf{M}_j$, and for this we use the Toolbox function `closest`

```
>> corresp = closest(D, M);
```

where `corresp(i)` is the column of `M` that corresponds to column `i` of `D`. The next step is to compute the 3×3 moment matrix

$$\boldsymbol{W} = \sum (\boldsymbol{M}_i - \bar{\boldsymbol{M}})(\boldsymbol{D}_i - \bar{\boldsymbol{D}})^T$$

This is the sum of a number of rank 1 matrices.

which encodes the rotation between the two point sets.◂ The singular value decomposition is

$$\boldsymbol{W} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

from which the rotation matrix is determined to be

$$\boldsymbol{R} = \boldsymbol{V}\boldsymbol{U}^T$$

The estimated relative pose between the two point clouds is $\xi \sim (\boldsymbol{R}, \boldsymbol{t})$. The model points are transformed so that they are now closer to the data points

$$\boldsymbol{M}_i\langle k+1 \rangle = \xi\langle k \rangle \cdot \boldsymbol{M}_i\langle k \rangle, \; i = 1 \cdots N_M$$

and the process repeated until it converges. The correspondences used are unlikely to have all been correct and therefore the estimate of the relative orientation between the sets is only an approximation.

The Toolbox provides an implementation of ICP

```
>> [T,d] = icp(M, D, 'plot');
```

which returns the pose ξ

```
>> trprint(T, 'rpy', 'radian')
t=(1 2 3),R=(0.2 0.3 0.4) rad
```

which is the exactly the unknown relative pose of the second point cloud that we chose above. The residual

```
>> d
d =
   6.8437e-08
```

is the root mean square of the errors between the transformed model points and the data. The option `'plot'` shows the model and data points at each step as well as the closest-point correspondences. ICP is a popular algorithm because it is both fast and robust.

We can demonstrate the robustness of ICP by simulating some realistic sensor errors. Firstly we will randomly remove twenty points from the data

```
>> D(:,randi(100, 20,1)) = [];
```

which are points in the model not observed by the sensor. Then we will add ten spurious points that are not part of the model

```
>> D = [D rand(3,10)];
```

and finally we will add Gaussian noise with $\sigma = 0.05$ to the data

```
>> D = D + randn(size(D)) * 0.05;
```

Now we fit the corrupted data to the model

```
>> [T,d] = icp(M, D, 'plot', 'distthresh', 1.5);
```

using an additional option to eliminate incorrect closest-point correspondences. The correspondences are established as described above and the median of the distances between the corresponding points is computed. In this case the correspondence is not made if the distance between the points is more than 1.5 times the median distance. The estimated pose ξ is now

```
>> trprint(T, 'rpy', 'radian')
t = (1.02161,1.98456,2.98566), R = (0.164436,0.299455,0.418697) rad
```

which is still close to the value computed for the ideal case but the residual

```
>> d
d =
    0.6780
```

is higher since an exact fit between the model and noise corrupted data is no longer possible.▸ ICP is fast and robust for modest sized point clouds but the correspondence determination is an $O(N^2)$ problem which leads to computational bottlenecks for very large data sets.▸

We would expect the residual to be approximately equal to $\sqrt{N}\sigma$ where N is the number of corresponding points and σ is the standard deviation of the additive noise.

For large-scale problems the data would be kept in a *kd*-tree which reduces the time required to find the closest point.

14.4 Structure and Motion

In the sparse stereo example of Sect. 14.3.1 we estimated the *motion* of the camera, its change in pose, from corresponding image points. Using this as a baseline we were then able to estimate the 3-dimensional position of some points in the world, the *structure* of the scene.

For robotic applications we would like to perform these operations on the fly, that is, as each new observation is made we wish to update the estimates of the robot's pose and the structure of its world. We will illustrate this with an example using a sequence of image from a single moving camera.

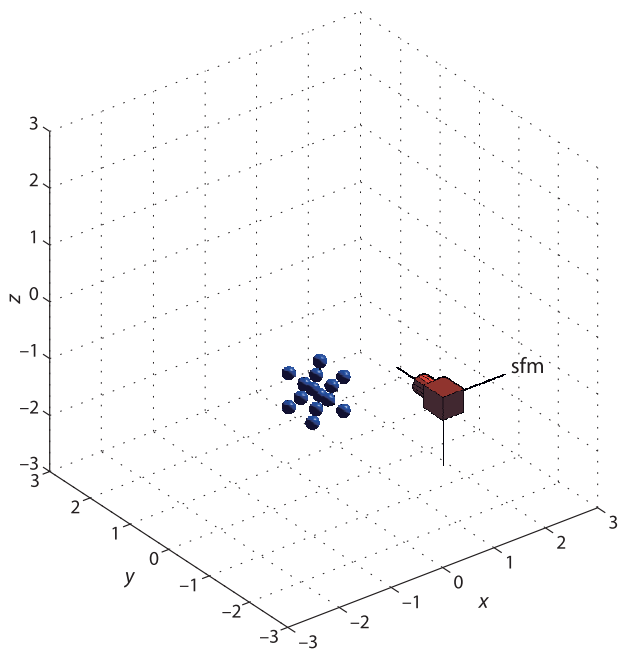


Fig. 14.39. Structure from motion example showing camera and world points

We create a central perspective camera

```
>> camera = CentralCamera('name', 'sfm', 'default');
```

and some points positioned at the vertices and face centres of a cube

```
>> cube = mkcube(0.6, 'facepoint')
```

A subsequent step estimates the fundamental matrix which requires at least eight points. A cube with only eight points leads to poor estimates of the fundamental matrix, hence the addition of six extra points in the centre of each face.

a total of 14 points.◂ The camera moves in a circle of radius 3 m around the cube in the *xy*-plane while always facing the origin

```
>> nsteps = 50;
>> theta = tpoly(0, 2*pi, nsteps);
>> T = [];
>> for th=theta'
>>    T = cat(3, T, trotz(-th) * transl(0, -3, 0) * trotx(-pi/2) );
>> end
```

where `T` is the path, a sequence of camera poses, represented by a $4 \times 4 \times 50$ matrix where the first two indices represent a homogeneous transformation and the last index corresponds to the camera viewpoint.

The image plane projection is

```
>> p = camera.plot(cube, 'Tcam', T);
```

which returns a $2 \times 14 \times 50$ matrix where the second index corresponds to the world point, and the last index corresponds to the camera view.

At each step along the path we perform the following algorithm, illustrated here for step twenty

```
>> k = 20;
```

of the path. The corresponding points between the current and previous image are used to estimate the fundamental matrix

```
>> F = fmatrix( p(:,:,k-1), p(:,:,k) );
```

and then *upgrade* it to an essential matrix

```
>> E = camera.E(F);
```

and decompose that to an estimate of the motion from the last camera pose to the current

```
>> relpose = camera.invE(E, [0 0 10]');
solution 2 is good
relpose =
    0.9775   -0.0000    0.2108  -11.2645
    0.0000    1.0000    0.0000   -0.0000
   -0.2108   -0.0000    0.9775    1.2008
         0         0         0    1.0000
```

where we use a world point along the optical axis to determine which of the two solutions is valid.

The true relative pose determined from the known camera trajectory is

```
>> inv(T(:,:,k-1))*T(:,:,k)
ans =
    0.9775    0.0000    0.2108   -0.6324
   -0.0000    1.0000   -0.0000    0.0000
   -0.2108   -0.0000    0.9775    0.0674
         0         0         0    1.0000
```

The rotational part of the estimated pose is accurate but the translational part suffers from the unknown scale factor problem that we have discussed previously. Figure 14.40 shows the magnitude of the estimated and true translation versus time. We see that the translation is always overestimated and that there is no obvious relationship to the true translation magnitude.

The true translation magnitude shows the smoothly changing velocity created by the `tpoly` function.

Without knowledge of the scale factor we are somewhat stuck – we cannot estimate the incremental motion between views. As we have seen before the solution involves incorporating other sources of information. For example other sensors such as odometry or GPS can provide an estimate for the magnitude of the translational motion. The odometer is only required to provide the distance travelled, since the rotational component of the motion is determined without ambiguity. Alternatively we could use a stereo camera to provide the depth information directly and the problem at each time step is then determining the relative pose that aligns the 3D point clouds from the current and previous time step, which is can be solved using ICP. Another option, and the one we consider next is to consider that we know the height above ground of just one observed world point. In a robotic scenario we might know that the robot moves on a plane and that a particular feature point lies on the ground or the top of a doorway.

For this example we will assume that we know the height of point j and we arbitrarily choose

```
>> j = 1;
```

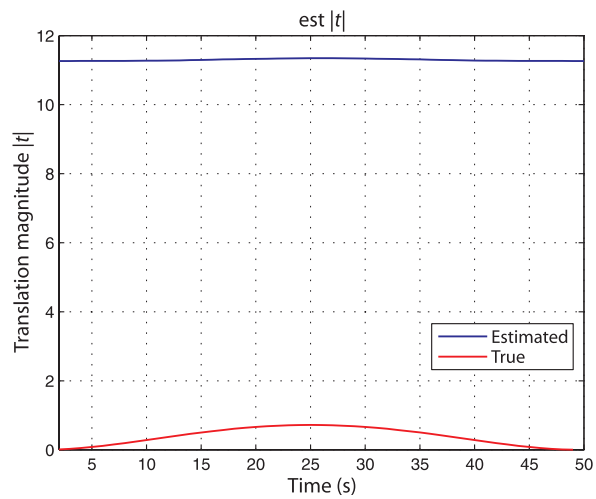


Fig. 14.40.
Magnitude of camera translational motion at each time step as estimated from the essential matrix and the ground truth

and the point's world coordinate is

```
>> P(:,j)'
ans =
   -0.3000   -0.3000   -0.3000
```

Since the camera lies in the *world* xy-plane with its y-axis pointing downward, then ${}^{C}Y_j = 0.3$

```
>> Yj = 0.3;
```

This is the *only* extra piece of information that we need but it does require that this particular point can be found in all future images.

We will again illustrate the algorithm for step twenty of the path. We consider that {1} is the camera frame in the previous time step and frame {2} the current time step. A ray corresponding to the projection of the known point in {1} is

```
>> r1 = camera.ray( p(:,j,k-1) )
r1 =
d=(0.117002, 0.108684, 0.987167), P0=(0, 0, 0)
```

which intersects the plane ${}^{C}Y = {}^{C}Y_j$ at the point◂

In vector form the plane is (0, 1, 0) $(X, Y, Z)^T = {}^{C}Y_j$ or in homogeneous form (0, 1, 0, $-{}^{C}Y_j$) $(X, Y, Z, 1)^T = 0$.

```
>> P1 = r1.intersect([0 1 0 -Yj])
P1 =
    0.3230
    0.3000
    2.7249
```

The projection of this world point in {2} which we observe with the camera is given by Eq. 11.5

$$ {}^{2}\tilde{\boldsymbol{p}} \simeq \boldsymbol{C}\, {}^{2}\boldsymbol{T}_1\, {}^{1}\tilde{\boldsymbol{P}} $$

which we can write in terms of an explicit homogeneous scale factor λ

$$ \begin{aligned} \lambda\, {}^{2}\tilde{\boldsymbol{p}} &= (\boldsymbol{\Pi} \quad \boldsymbol{c}_4) \begin{pmatrix} \boldsymbol{R} & \sigma \boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{P} \\ 1 \end{pmatrix} \\ &= \sigma \boldsymbol{\Pi} \boldsymbol{t} + \boldsymbol{\Pi} \boldsymbol{R} \boldsymbol{P} + \boldsymbol{c}_4 \\ &= \sigma \boldsymbol{a} + \boldsymbol{b} \end{aligned} $$

where σ is the unknown translational scale factor and $\boldsymbol{a}$ and $\boldsymbol{b}$ are vectors that are a function of the known camera matrix $\boldsymbol{C}$ and the estimated camera rotation $\boldsymbol{R}$. The difference between the observed and actual image plane coordinate is known as reprojection error and its magnitude is

$$ e = (\sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}})^T (\sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}}) $$

To determine values of the translation scale factor σ and the homogeneous scale factor λ we minimize this error by taking the partial derivatives and setting them to zero

$$ \begin{aligned} \frac{\partial e}{\partial \sigma} &= 2(\sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}}) \cdot \boldsymbol{a} = 0 \\ \frac{\partial e}{\partial \alpha} &= -2(\sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}}) \cdot {}^{2}\tilde{\boldsymbol{p}} = 0 \end{aligned} $$

which implies that the vector term in parentheses is orthogonal to both $\boldsymbol{a}$ and ${}^{2}\tilde{\boldsymbol{p}}$. This is the same as being proportional to their cross product

$$ \sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}} \propto \boldsymbol{a} \times {}^{2}\tilde{\boldsymbol{p}} $$

and by introducing another scale factor α we write this as an equality

$$ \sigma \boldsymbol{a} + \boldsymbol{b} - \lambda\, {}^{2}\tilde{\boldsymbol{p}} = \alpha (\boldsymbol{a} \times {}^{2}\tilde{\boldsymbol{p}}) $$

that we rearrange as

$$(-\boldsymbol{a} \quad {}^2\tilde{\boldsymbol{p}} \quad \boldsymbol{a} \times {}^2\tilde{\boldsymbol{p}}) \begin{pmatrix} \sigma \\ \lambda \\ \alpha \end{pmatrix} = \boldsymbol{b}$$

which is a linear equation that can be solved for the unknown parameters σ, λ and α. For this example

```
>> C = cam.C; PI = C(:,1:3); c4 = C(:,4);
>> [R,t] = tr2rt( inv(relpose) )
>> a = PI*t; b = PI*R*P1+c4;
>> p2 = e2h( p(:,j,k) );
```

and the parameters are estimated

```
>> phi = [-a p2 cross(a, p2)] \ b
phi =
    0.0561
    2.7991
         0
```

from which the displacement ${}^1\xi_2$ can be computed

```
>> dT = inv( [R phi(1)*t; 0 0 0 1] )
dT =
    0.9775    0.0000    0.2108   -0.6324
   -0.0000    1.0000   -0.0000    0.0000
   -0.2108    0.0000    0.9775    0.0674
         0         0         0    1.0000
```

This is the same, to four significant figures, as the known camera motion shown earlier. The missing scale factor has been recovered using the known height above ground of a single observed point, known robot motion and a calibrated camera. The pose of the camera in the world frame is estimated by accumulating the frame-to-frame pose changes

$$\hat{\boldsymbol{T}}\langle k \rangle = \hat{\boldsymbol{T}}\langle k-1 \rangle {}^1\hat{\boldsymbol{T}}_2\langle k-1 \rangle$$

to obtain an estimate of the camera's pose at each time step with respect to the initial camera pose. Figure 14.41 shows the true and estimated position of camera's centre

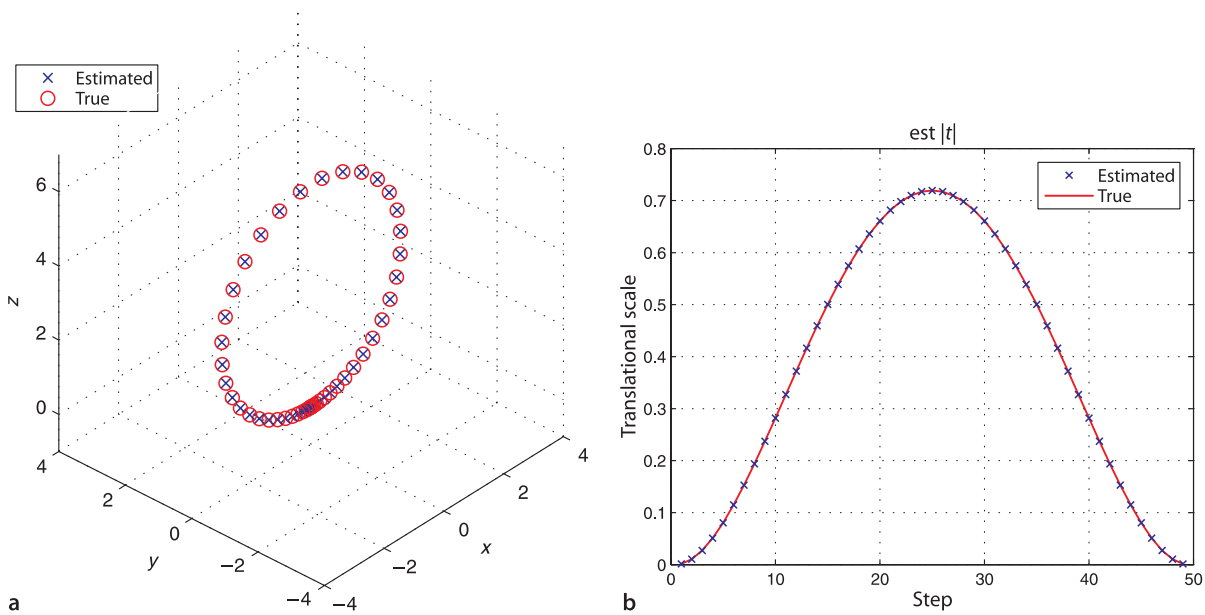


Fig. 14.41. Structure from a motion with estimated scale from a single world point. **a** Estimated and true camera centre; **b** estimated and true translational scale factor

along the path as well as the estimated scale factor. This is a dead-reckoning approach to position estimation and, as we discussed in Sect. 6.1, the error in the estimate of each incremental motion will accumulate over time. The results for the case with noise added to the feature coordinates is shown in Fig. 14.42 and we see that the scale estimate is less accurate and that the dead-reckoned camera position has diverged from the true position. The full code for this example is available as `sfm1` in the examples directory.

Next we will estimate the structure of the scene with the sparse stereo approach we used previously in Sect. 14.3.1. The rays in 3-dimensional space corresponding to the projected points at the previous position are

```
>> r1 = camera.ray(p(:,:,k-1))
```

and at the current position they are

```
>> r2 = camera.move(dT).ray(p(:,:,k))
```

The rays intersect at

```
>> x = r1.intersect(r2)
```

which is a matrix where each column is the 3-dimensional location of the point with respect to the previous camera pose. Using our estimate of the camera pose with respect to the world these points can be transformed into the world coordinate frame

```
>> xc = homtrans(inv(T_est(:,:,k-1), x)
```

This approach is simple but has a number of drawbacks. The biggest problem is that the 3-dimensional coordinates of every world point is estimated at every time step, and the quality of the estimates depends strongly on the relative motion between the camera and the world point. The triangulation involves three points, the two camera centres and the world point as shown in Fig. 14.17. The side of the triangle between the camera centres is the baseline, and if the baseline is small the estimates will be susceptible to noise. This will be the case here since the camera motion between images is small. However if the second camera centre is collinear with the first camera centre and the world point – the motion is along a line toward or away from the point – there is a zero base line and we can learn nothing about that point's depth.

Figure 14.42 showed the effect of image noise on the estimate of camera position, and Fig. 14.43 shows the effect of this noise on the estimated location of a single world

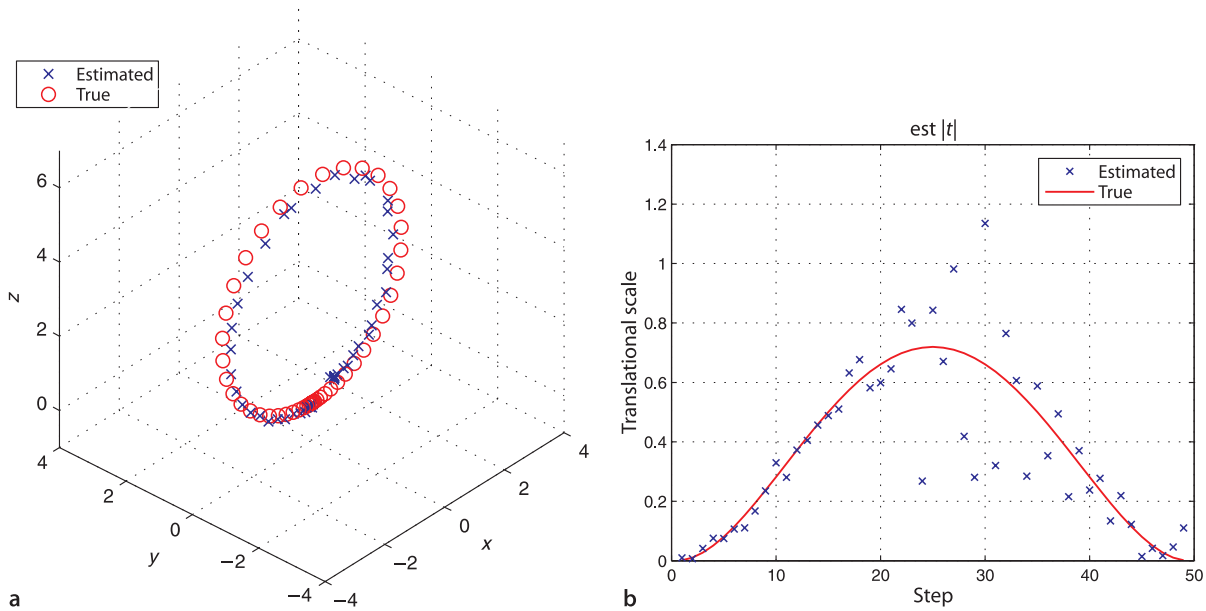


Fig. 14.42. Structure from motion with estimated scale from single world point and pixel noise $\sigma = 0.5$. **a** Estimated and true camera centre; **b** estimated and true translational scale factor

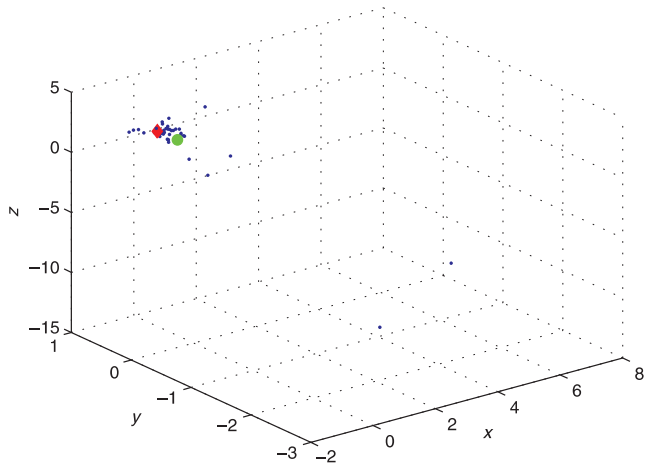


Fig. 14.43. Estimated spread, pixel noise $\sigma = 0.5$. Blue dots are the pairwise estimated point position, red ◊-marker is the true location of the world point (in camera frame), and the green circle is the mean of the estimated positions

point. We see that the estimates are scattered with some significant outliers that are in error by more than 1 m. One solution is to increase the effective baseline by triangulating between every N frames instead of between every consecutive frame. Another option is to apply some low-pass filter▸ to the estimates, and Fig. 14.43 shows that the mean of the estimates is close to the true position.▸ A better solution is to invert the problem and use an extended Kalman filter as we did for the localization problem of Sect. 6.1.2.▸ The filter's state comprises the pose of the camera and the world coordinates of landmarks. The observations are the pixel coordinates of the features. The difference between the observed and predicted pixel coordinates is the innovation which updates the state vector via the Kalman gain matrix. This framework allows for improved estimates of the world point locations since it incorporates measurements over many time steps and allows for explicit modeling of the uncertainty associated with the world points. The Kalman filter framework also allows for the incorporation of any number of other sensors such as odometry, GPS or laser range measurements which can help to resolve the scale problem. Some book keeping is required to keep track of features as they enter the camera view and later leave.

Such as a moving average filter, a Kalman filter or an $\alpha-\beta$ tracking filter.

The errors do not appear to be zero-mean so a mean estimate would be biased.

The Kalman filter assumes that the noise is zero-mean and Gaussian, which is unlikely to be the case here.

14.5 Application: Perspective Correction

Consider the image

```
>> im = iread('notre-dame.jpg', 'double');
>> idisp(im)
```

shown in Fig. 14.44. The shape of the building is significantly distorted because the camera's optical axis was not normal to the plane of the building and we see evidence of perspective foreshortening or keystone distortion. We manually pick four points, clockwise from the bottom left, that are the corners of a large rectangle on the planar face of the building

```
>> p1 = ginput(4)'
ans =
   44.1364   94.0065  537.8506  611.8247
  377.0654  152.7850  163.4019  366.4486
```

which has one column per point that contains the u- and v-coordinate. We overlay this on the image of the cathedral

```
>> plot_poly(p1, 'wo', 'fill', 'b', 'alpha', 0.2);
```

with translucent blue fill.▸ We use the extrema of these points to define the vertices of a rectangle in the image

In computer graphics terminology alpha is the transparency of a surface, and varies from 0 completely transparent to 1 which is opaque.

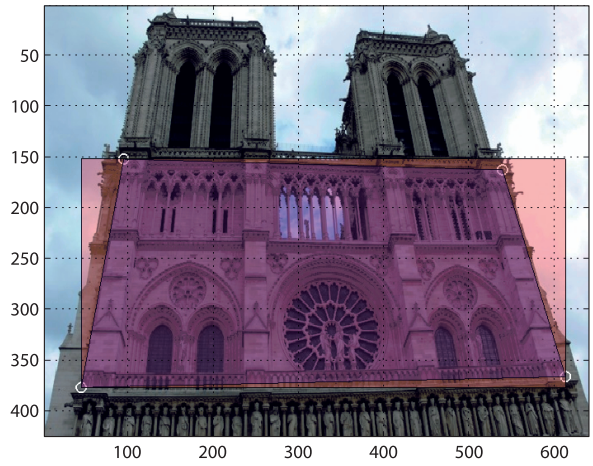


Fig. 14.44. Photograph taken from the ground shows the effect of foreshortening which gives the building a trapezoidal appearance (also known as keystone distortion). Four points on the approximately planar face of the building have been manually picked as indicated by the white ○-markers (Notre Dame de Paris)

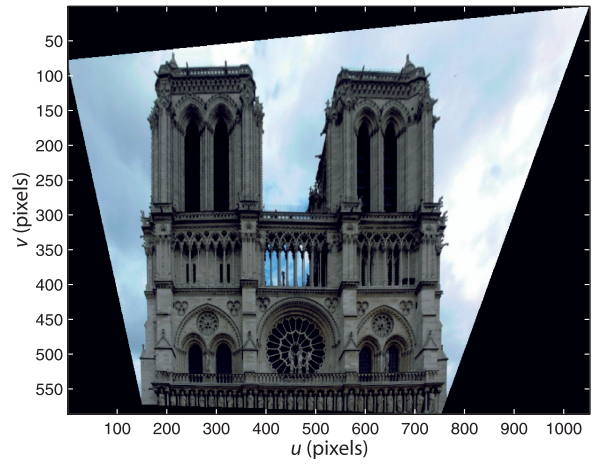


Fig. 14.45. A fronto-parallel view synthesized from Fig. 14.44. The image has been transformed so that the marked points become the corners of a rectangle in the image

```
>> mn = min(p1');
>> mx = max(p1');
>> p2 = [mn(1) mx(2); mn(1) mn(2); mx(1) mn(2); mx(1) mx(2)]';
```

which we overlay on the image

```
>> plot_poly(p2, 'k', 'fill',  'r', 'alpha', 0.2)
```

in red.

The sets of points `p1` and `p2` are projections of world points that lie approximately in a plane so we can compute an homography

```
>> H = homography(p1, p2)
H =
    1.4003    0.3827 -136.5900
   -0.0785    1.8049  -83.1054
   -0.0003    0.0016    1.0000
```

An homography can also be computed from four lines in the plane, but this is not supported by the Toolbox.

that will transform the vertices of the blue trapezoid to the vertices of the red rectangle.◂

$$\tilde{\boldsymbol{p}}_2 \simeq \boldsymbol{H}\tilde{\boldsymbol{p}}_1$$

That is, the homography maps image coordinates from the distorted keystone shape to an undistorted rectangular shape.

We can apply this homography to the coordinate of every pixel in an output image in order to warp the input image. We use the Toolbox generalized image warping function

```
>> homwarp(H, im, 'full')
```

and the result shown in Fig. 14.45 is a synthetic fronto-parallel view. This is equivalent to the view that would be seen by a camera high in the air with its optical axis normal to the face of the cathedral. However points that are not in the plane, such as the left-hand side of the right bell tower have been distorted. The black pixels in the output image are due to the corresponding pixel coordinates not being present in the input image. Note that with no output argument specified the warped image is displayed using `idisp`.

In addition to creating this synthetic view we can decompose the homography to recover the camera motion from the actual to the virtual viewpoint and also the surface normal of the cathedral. As we saw in Sect. 14.2.4 we need to determine the camera calibration matrix so that we can convert the projective homography into a Euclidean homography. We obtain the focal length from the header of the EXIF-format file that holds the image

```
>> [im,tags] = iread('notre-dame.jpg', 'double');
>> tags.DigitalCamera
ans =
              ExposureTime: 0.0031
                   FNumber: 5.6000
                         .
                         .
                FocalLength: 7.4000
                         .
                         .
```

and the pixel dimensions are the same as for the example on page 402. We create a calibrated camera

```
>> cam = CentralCamera('image', im, 'focal', 7.4e-3, ...
    'sensor', [7.18e-3,5.32e-3])
cam =
name: noname [central-perspective]
  focal length:   0.0074
  pixel size:     (5.609e-06, 6.251e-06)
  principal pt:   (640, 425.5)
  number pixels:  1280 x 851
  Tcam:
      1  0  0  0
      0  1  0  0
      0  0  1  0
      0  0  0  1
```

Now we use the camera model to compute and decompose the Euclidean homography

```
>> sol = cam.invH(H, 'verbose');
solution 1
     T = 0.98415    -0.15470     0.08667     0.23438
         0.15902     0.98623    -0.04534     0.92578
        -0.07846     0.05841     0.99520    -1.29375
         0.00000     0.00000     0.00000     1.00000
    n = -0.23041     0.88969     0.39416
solution 2
     T = 0.95800     0.23545     0.16369     0.07289
        -0.20987     0.18671     0.95974    -1.53146
         0.19541    -0.95378     0.22828     0.48488
         0.00000     0.00000     0.00000     1.00000
    n = -0.28109     0.11855     0.95233
```

which returns a structure array of two possible solutions. The coordinate frames for this example are sketched in Fig. 14.46 and shows the actual and virtual camera poses. In this case the second solution is the correct one since it represents considerable rotation about the *x*-axis. The camera translation vector, which is not to scale but has the

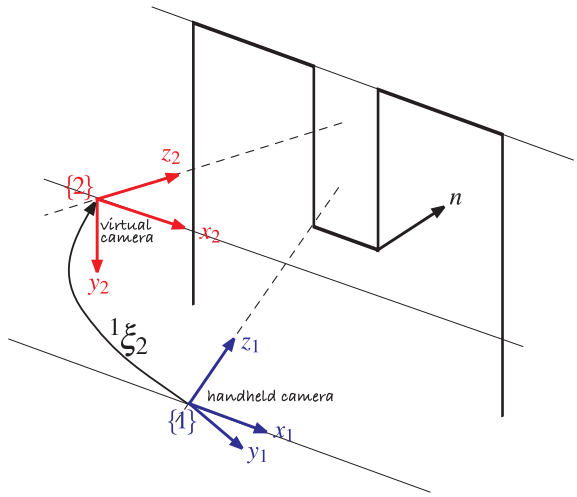


Fig. 14.46. Notre-Dame example showing the two camera coordinate frames. The blue frame {1} is that of the camera that took the image, and the red frame {2} is the viewpoint for the synthetic fronto-parallel view

See Malis and Vargas (2007).

correct sign,◂ is dominantly in the negative y- and positive z-direction with respect to the frame {1}. The rotation matrix in XYZ-angle form

```
>> tr2rpy(sol(2).T, 'deg')
ans =
  -76.6202    9.4210  -13.8081
```

indicates that the camera needs to be tilted downward (roll is rotation about the camera's x-axis) by 76 degrees to achieve the attitude of the virtual camera. The normal to the frontal plane of the church $\boldsymbol{n}$ is defined with respect to {1} and is essentially in the camera z-direction as expected.

14.6 Application: Mosaicing

Mosaicing or image stitching is the process of creating a large-scale composite image from a number of overlapping images. It is commonly applied to aerial and satellite images to create a seemingly continuous single picture of the earth's surface. It can also be applied to images of the ocean floor captured from downward looking cameras on an underwater robot. The panorama generation software supplied with digital cameras is another example of mosaicing.

As a rule of thumb images should overlap by 60% of area in the forward direction and 30% sideways.

The input to the mosaicing process is a sequence of overlapping images.◂ It is not necessary to know the camera calibration parameters or the pose of the camera where the images were taken – the camera can rotate arbitrarily between images and the scale can change. However for the approach that we will use the scene is assumed to be planar which is reasonable for high-altitude photography where the vertical relief◂ is small.

The ratio of the height of points above the plane to the distance of the camera from the plane.

We will illustrate our discussion with a real example using the pair of images

```
>> im1 = iread('mosaic/aerial2-1.png', 'double', 'grey');
>> im2 = iread('mosaic/aerial2-2.png', 'double', 'grey');
```

which are each 1 280 × 1 024. We create an empty composite image that is 2 000 × 2 000

```
>> composite = zeros(2000,2000);
```

that will hold the mosaic. The essentials of the mosaicing process are shown in Fig. 14.47.

The first image is easy and we simply paste it into the top left corner

```
>> composite = ipaste(composite, im1, [1 1]);
```

of the composite image as shown in red in Fig. 14.47. The next image, shown in blue, is more complex and needs to be rotated, scaled and translated so that it correctly overlays the red image.

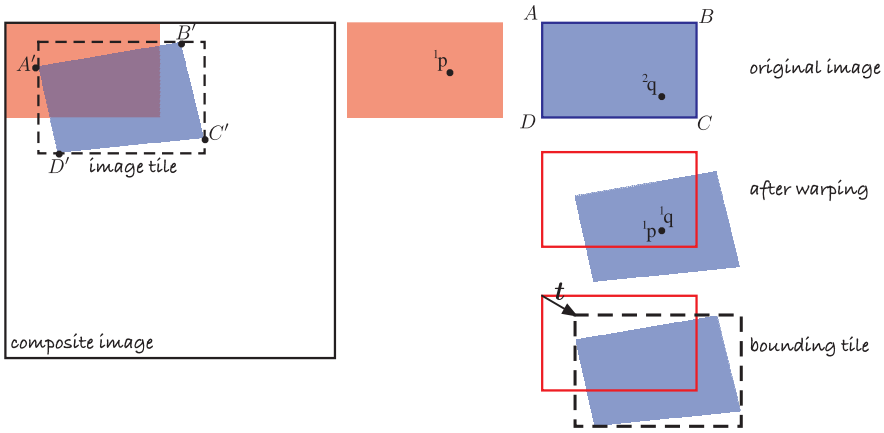


Fig. 14.47.
The first image in the sequence is shown as red, the second as blue. The second image is warped into the image tile and then blended into the composite image

The first step is to identify common feature points which are known as tie points, and we use now familiar tools

```
>> surf = isurf(im1)
>> surf = isurf(im2)
>> m = surf.match(surf);
```

and then RANSAC to estimate the homography

```
>> [H,in] = m.ransac(@homography, 0.2)
```

Since we assume the features lie on a plane the homography maps ${}^1\mathbf{p}$ to ${}^2\mathbf{p}$. Now we wish to map ${}^2\mathbf{p}$ to its corresponding coordinate in the first image

$$ {}^1\mathbf{p} \simeq \boldsymbol{H}^{-1}\, {}^2\mathbf{p} $$

We do this for every pixel in the new image by warping

```
>> [tile,t] = homwarp(inv(H), im2, 'full', 'extrapval', 0);
```

The bounding box of the tile is computed by applying the homography to the image corners $A = (1, 1), B = (W, 1)$, $C = (W, H)$ and $D = (1, H)$, where W and H are the width and height respectively, and finding the bounds in the u- and v-directions.

As shown in Fig. 14.47 the warped blue image falls outside the bounds of the original blue image and the option `'full'` specifies that the returned image is the minimum containing rectangle of the warped image. This image is referred to as a *tile* and shown with a dashed black line. The vector $\boldsymbol{t}$ is returned by `homwarp` and gives the offset of the tile's coordinate frame with respect to the original image. In general not every pixel in the tile has a corresponding point in the input image and those pixels are set to zero, as specified by the fifth argument.

The default is `NaN`.

Now the tile has to be *blended* into the composite image

```
>> canvas = ipaste(canvas, tile, t, 'add');
```

and the result is shown in Fig. 14.48. We can clearly see several images overlaid and with excellent alignment. The non-mapped pixels in the warped image are set to zero so adding them causes no change to the existing pixel values in the composite image.

Simply *adding* the tile into the composite image means that overlapping pixels are necessarily brighter and a number of different strategies can be used to remedy this. We could instead set pixels in the composite image from the tile only if the composite image pixels have not yet been set. Conversely we could *always* set pixels in the composite image from the non-zero pixels in the tile. Alternatively we set the composite image pixels to the mean of the tile and the composite image. This requires that we tag the tile pixels that are not mapped

```
>> [tile,t] = homwarp(inv(H), im2, 'full', 'extrapval', NaN);
```

and then blend using the `'mean'` option

Which ignores any pixels with the value NaN.

```
>> canvas = ipaste(canvas, tile, t, 'mean');
```

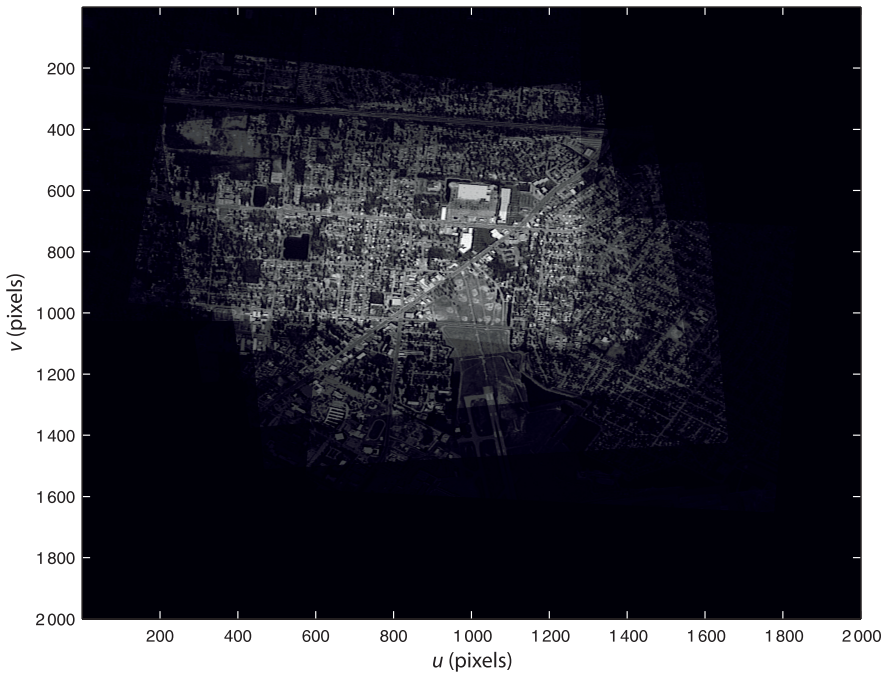


Fig. 14.48. Example image mosaic. At the bottom of the frame we can clearly see three overlapping views of the airport runway which shows good alignment between the frames

If the images were taken with the same exposure then the edges of the tiles would not be visible. If the exposures were different the two sets of overlapping pixels have to analyzed to determine the average intensity offset and scale factor which can be used to correct the tile before blending – a process known as tone matching.

Finally, we need to consider the effect of points in the image that are not in the ground plane such as those on a tall building. An image taken from directly overhead will show just the roof of the building, but an image taken from further away will be an oblique view that shows the side of the building. In a mosaic we want to create the illusion that we are directly above every point in the image so we should not see the sides of any building.◀ This type of image is known as an orthophoto and unlike a perspective view, where rays converge on the camera's focal point, the rays are all parallel which implies a viewpoint at infinity. At every pixel in the composite image we can choose a pixel from any of the overlapping tiles. To best approximate an orthophoto we should choose the pixel that is closest to overhead, that is, prior to warping the pixel was closest to the principal point.

Google Earth provides an imperfect orthophoto. When looking at cities it is very common to see oblique views of buildings.

In photogrammetry this type of mosaic is referred to as an uncontrolled digital mosaic since it does not use explicit control points – manually identified corresponding features in the images. The result is an orthophoto which has a viewpoint at infinity. The full code is given by `mosaic1` in the examples directory. The principles illustrated here can also be applied to the problem of image stabilization. The homography is used to map features in the new image to the location they had in the previous image.

14.7 Application: Image Matching and Retrieval

Given a set of images $\{\mathbf{I}_j,\ j = 1 \cdots N\}$ and a new image $\mathbf{I}'$ the image matching problem is to determine j such that $\mathbf{I}'$ and $\mathbf{I}_j$ are most similar. This is a difficult problem when we consider the effect of changes in viewpoint and exposure. Pixel-level similarity measures such as SSD or ZNCC that we used previously are not suitable for this problem since quite small changes in viewpoint will result in almost zero similarity.

Image matching can be used by a robot to determine if it has visited a particular place before, or seen the same object before. If those previous images have some asso-

ciated semantic data such as the name of an object or the name of a place then by inference that semantic data applies to the new image. For example if a new image matches an existing image that has the semantic tag "lobby" then it implies the robot is seeing the same scene and is therefore in or close to, the lobby.

The particular technique that we will introduce is commonly referred to as "bag of words" and has become very popular in robotics in the last few years. It builds on techniques we have previously encountered such as SURF point features and k-means clustering.

We start by loading a set of twenty images

```
>> images = iread('campus/*.jpg', 'mono');
```

as a $426 \times 640 \times 20$ array and for each of these we compute the SURF features

```
>> sf = isurf(images);
```

which returns a MATLAB® cell array whose elements are vectors of SURF features that correspond to the input images. For example

```
>> sf{1}
ans =
663 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
```

is a vector of 663 SURF feature objects corresponding to the first image in the sequence. The set of all SURF features across all images is

```
>> sf = [sf{:}]
sf =
17945 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
```

which is a vector of 17 945 SURF features objects.

Consider a particular SURF feature

```
>> sf(380)
ans =
   (48.1448,219.771), theta=4.4736, image_id=1, scale=3.48386,
     strength=0.000615617, descrip= ..
```

and we see the `SurfPointFeature` properties discussed earlier such as centroid, scale and orientation. The property `image_id` indicates that this feature was extracted from the first image in the original image sequence. We can display that image and superimpose the feature

```
>> idisp(images(:,:,1))
>> sf(380).plot('g+')
>> sf(380).plot_scale('g', 'clock')
```

which is shown in Fig. 14.49a. The support region for this feature

```
>> sf(380).support(images)
```

is shown in Fig. 14.49b. The support region shows bricks and the edge of a window. The `support` method uses the `image_id` property to determine which of the passed images contains the feature.

The key insight behind the bag of words technique is that many of these features will describe visually similar scene elements such as leaves, corners of windows, bricks, chimneys and so on. If we consider each SURF feature descriptor as a point in a 64-dimensional space then similar descriptors will form clusters, and this a k-means problem. To find 2 000 feature clusters

```
>> bag = BagOfWords(sf, 2000)
```

returns a `BagOfWords` object that contains the original features, the centre of each cluster, and various other information.▸ Each cluster is referred to as a visual word and is described by a 64-element SURF descriptor. The set of all visual words, 2 000 in this case, is a visual vocabulary. Just as a document comprises a set of words drawn

▸ The `BagOfWords` class use the MEX-file *k*-means implementation from http://www.vlfeat.org/. This uses its own random number generator and to initialize it to a known state use `vl_twister('STATE', 0.0);`.

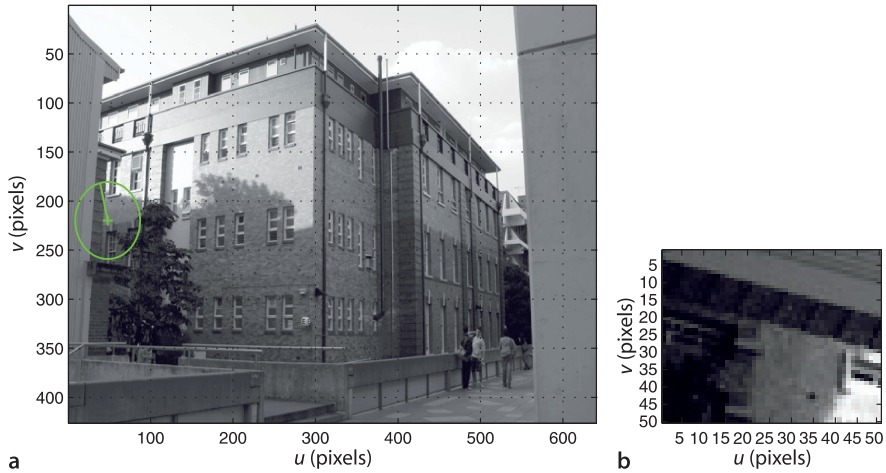


Fig. 14.49. **a** Image 1 with visual word SURF feature 380 indicated by green circle showing scale and a radial line showing orientation direction; **b** the square support region has the same area as the circle and the horizontal axis is parallel to the orientation direction

Fig. 14.50. Exemplars of visual word 1057 from the various images in which it appears. The annotation is of the form word/image

from some vocabulary, each image comprises a collection (or *bag*) of words drawn from the visual vocabulary.

The clustering step assigns a visual word index to every SURF feature. For the particular feature shown above

```
>> bag.words(380)
ans =
        1057
```

we find that the *k*-means clustering has assigned this image feature to word 1 057 in the vocabulary – it is an instance of visual word 1 057. That particular visual word appears

```
>> bag.occurrence(1057)
ans =
     6
```

times across the set of images, and it appears at least once in each of the images

```
>> bag.contains(1057)
ans =
     1     2     3     4    14    16
```

We can display some of the different instances of word 1 057 by

```
>> bag.exemplars(1057, images)
```

which is shown in Fig. 14.50. These exemplars actually look quite different, but we need to keep in mind that we are viewing them as patterns of pixels whereas the similarity is in terms of the descriptor.◀ The exemplars do however share some dominant horizontal and vertical structure.

The descriptor comprises responses of Haar wavelet detectors computed over multiple windows within the support region.

Visual words occur with quite different frequencies

```
>> [word,f] = bag.wordfreq()
```

where `word` is a vector containing all unique words and `f` are their corresponding frequencies. We can display these in descending order of frequency

```
>> bar( sort(f, 'descend') )
```

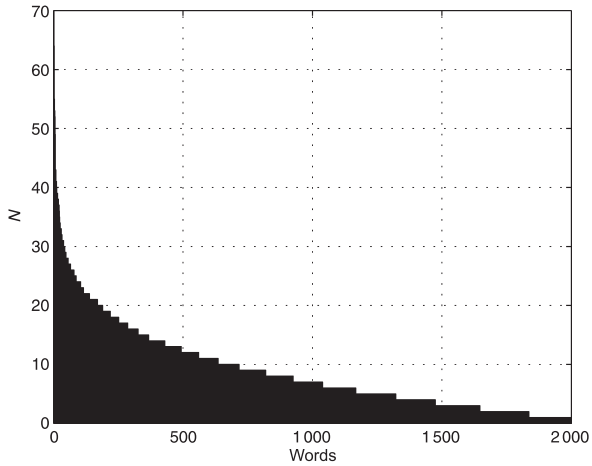


Fig. 14.51. Histogram of the number of occurrences of each word (sorted). Note the small number of words that occur very frequently

which is shown in Fig. 14.51. Words that occur very frequently have less meaning or power to discriminate between images. They are analogous to English words that are considered stop words in text document retrieval. The visual stop words are removed from the bag of words

Search engines ignore words such as 'a', 'and', 'the' and so on.

```
>> bag.remove_stop(50)
Removing 1850 features associated with 50 most frequent words
>> bag
bag =
BagOfWords: 16095 features from 20 images
            1950 words, 50 stop words
```

which leaves some 16 000 SURF features behind. This method performs relabelling so that word labels are now in the interval $1 \cdots 1\,950$.

Our visual vocabulary comprises K visual words and in this case $K = 1\,950$. We apply a technique from text document retrieval and describe *each* image by a K-element vector

$$\boldsymbol{v}_i = (t_1, \cdots t_j \cdots t_K)$$

whose elements describes the frequency of the corresponding visual words in the image.

$$t_j = \frac{n_{ij}}{n_i} \underbrace{\log \frac{N}{N_j}}_{\text{idf}} \tag{14.18}$$

where j is the visual word label, N is the total number of images in the database, N_j is the number of images which contain word j, n_i is the number of words in image i, and n_{ji} is the number of times word j appears in image i. The inverse document frequency (idf) term is a weighting that reduces the significance of words that are common across all images and which are thus less discriminatory. The weighted word frequency vectors are a property of the `BagOfWords` object but can be accessed by

```
>> M = bag.wordvector;
```

which is a $1\,950 \times 20$ matrix and each column is a 1 950-element vector that concisely describes the corresponding image.

This might seem like a very large vector but it contains less than 1% of the number of elements of the original image.

The similarity between two images is the cosine of the angle between their corresponding word-frequency vectors

$$s(\boldsymbol{v}_1, \boldsymbol{v}_2) = \frac{\boldsymbol{v}_1 \boldsymbol{v}_2^T}{|\boldsymbol{v}_1||\boldsymbol{v}_2|}$$

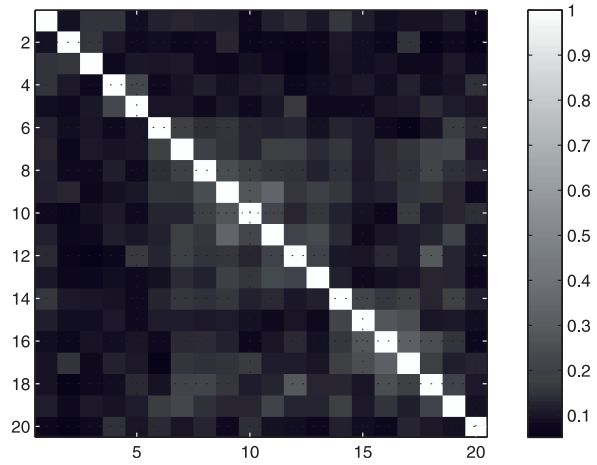


Fig. 14.52. Similarity matrix for 20 images where light colors indicate strong similarity. Element (i, j) indicates the similarity between image i and image j

and is implemented by the `similarity` method. A value of one indicates maximum similarity. To compute the mutual similarity between two bags of words is simply

```
>> S = bag.similarity(bag)
```

which returns a 20×20 similarity matrix where the elements `S(i,j)` indicates the similarity between the i^{th} column and j^{th} columns of `M`, or between image i and image j. Such a similarity matrix is best interpreted visually

```
>> idisp(S, 'bar')
```

which is shown in Fig. 14.52. The bright diagonal indicates, as a useful cross check, that image i is identical to image i.

Consider image 11 shown in Fig. 14.53a. Its similarity to other images is given by row, or column, 11 of the similarity matrix

```
>> s = S(:,11);
```

which we sort into descending order of similarity

```
>> [z,k] = sort(s, 'descend');
>> [z k]
ans =
    1.0000   11.0000
    0.3448    9.0000
    0.2377   13.0000
    0.1948   10.0000
    0.1719    7.0000
      .
      .
```

where each row comprises the similarity measure and the corresponding image. Image 11 is identical to image 11 as expected, and in decreasing order of similarity we have images 9, 13, 10 and so on. These are shown in Fig. 14.53 and we see that the algorithm has recalled quite different views of the same building.

Now consider that we have some new images and we wish to determine which of the previous images is the most similar. Perhaps the robot has taken a picture and wishes to compare it to its database of existing images. The steps are broadly similar to the previous case

```
>> images2 = iread('campus/holdout/*.jpg', 'mono');
>> sf2 = isurf(images2)
```

but rather than perform clustering we want to assign the features to the existing set of visual words, that is, to determine the closest visual word for each of the new feature descriptors

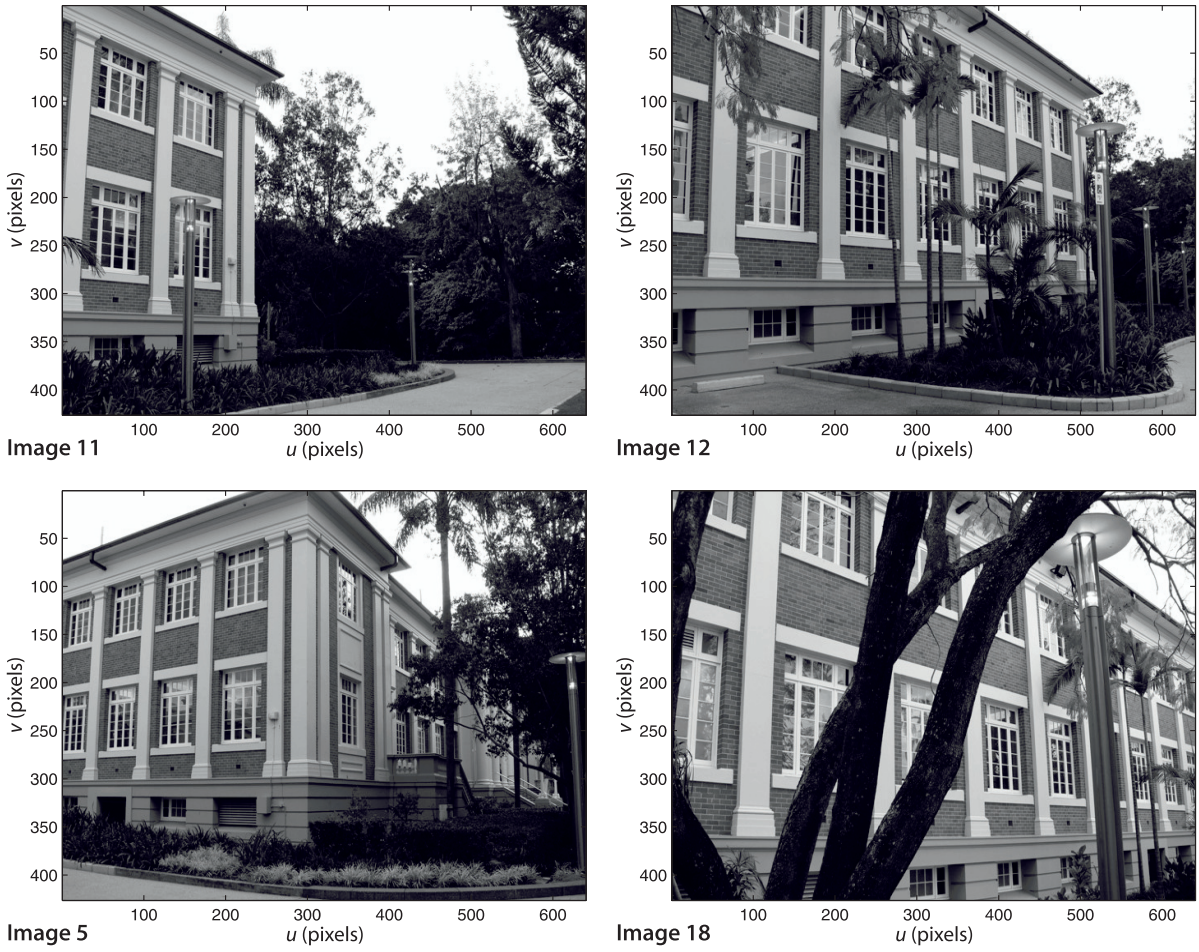


Fig. 14.53. Image recall. Image 11 is the query, and in decreasing order of match quality we have recalled images 9, 13 and 10

```
>> bag2 = BagOfWords(sf2, bag)
bag2 =
BagOfWords: 4708 features from 5 images
            1950 words, 50 stop words
```

This operation also removes any features words that were previously determined to be stop words, and computes the word frequency vectors▸ according to Eq. 14.18.

▸ Which requires the image-word statistics from the existing bag of words to compute the idf weighting terms.

Finally the similarity between the images in the two bags of words is

```
>> S2 = bag.similarity(bag2);
```

which returns a 20×5 matrix where the elements `S(i,j)` indicates the similarity between the existing image i and new image j. The maxima in each *column* corresponds to the most similar image in the previously observed set

```
>> [z,i] = max(S2)
z =
    0.2751    0.690    0.5121    0.5017    0.3177
i =
     7    11    16    18    20
```

New image 1 best matches image 7 in the original sequence, new image 2 matches image 11 and so on. The new images and their closest existing images are shown in Fig. 14.54. The first recall has a low similarity score and is incorrect – it shows a very different building. However the two scenes do have some similar features such as windows, roof line and gutters.

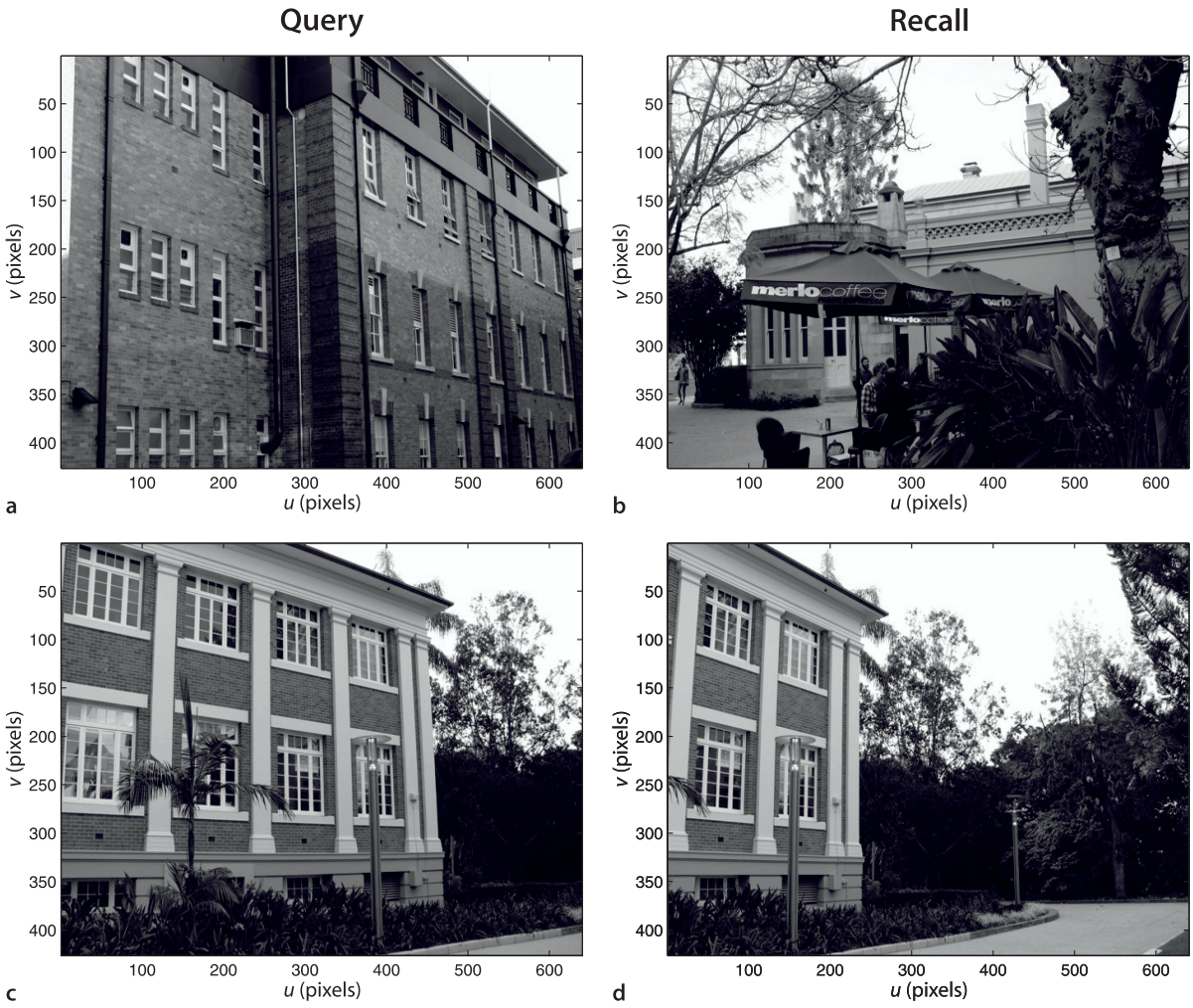


Fig. 14.54. Image recall for new images. The new query images **a** and **c** recall the database images **b** and **d** respectively

14.8 Application: Image Sequence Processing

In this final section of the chapter, and this part of the book, we look at how to use the Toolbox for processing image sequences. We load a sequence of images taken from a car driving along a road

```
>> im = iread('bridge-l/*.png', 'roi', [20 750; 20 480]);
```

and the option `'roi'` selects a region of interest from each image to eliminate an irregular black border.◀ These images are unusual in having 16-bit pixels

The black border is the result of image rectification.

```
>> about(im)
im [uint16] : 461x731x251 (169169482 bytes)
```

and the image `im` belongs to the class `'uint16'`. Since this sequence is already nearly 200 Mbyte we do not convert it to double precision since this would quadruple the amount of memory required.

The image sequence can be displayed as an animation

```
>> ianimate(im, 'fps', 10);
```

at 10 frames per second.

For each frame we compute corner features

```
>> c = icorner(im, 'nfeat', 200, 'patch', 7);
```

and for a change we have used Harris corners since they are computationally cheaper. For this application the change in orientation and scale from frame to frame is small and Harris corner features are well suited for this purpose. The function returns a cell array with one element per input image, and each element is a vector of the 200 strongest Harris corner features per image. The `'patch'` option specifies a 15×15 local neighbourhood descriptor according to Eq. 14.1 which is a 225-element unit vector. The image sequence can be displayed as an animation with the features overlaid

```
>> ianimate(im, c, 'fps', 10);
```

at 10 frames per second and a single frame of this sequence is shown in Fig. 14.55. The features are associated with regions of high gradient such as the edges of trees, as well as the corners of signs and cars. Watching the animation we see that the corner features *stick* reliably to world points for many frames. The motion of features in the image is known as optical flow and is a function of the camera's motion through the world and the 3-dimensional structure of the world.

We will revisit optical flow in the next chapter.

The Toolbox class `Tracker` maintains a list of *tracks*, which are the motion of individual features over time

```
>> t = Tracker(im, c);
```

Figure 14.56a shows one frame from the operation of `Tracker` where the features are annotated with a unique identifier associated with a particular track. For each track the object maintains the track number, the current position of the feature, the feature descriptor and some statistics such as the number of times it has been seen and how many frames since it was last seen.

For each new frame every track finds a subset of features within a specified radius of its position in the previous frame. The descriptors of these features are compared with the track's feature descriptor from the previous frame, and if a match is found the track is updated. Any feature that is not claimed by a track is considered to be the start of a new track – a new entry is created in the table and a unique track identifier is assigned to it. If a track doesn't find a matching feature in the current frame a counter is incremented, and when it exceeds some threshold the track is retired.

The descriptor associated with the track is updated at each time step. In this way, even if the support region changes in appearance from frame to frame, we always compare new features to the most recent appearance of the tracked feature.

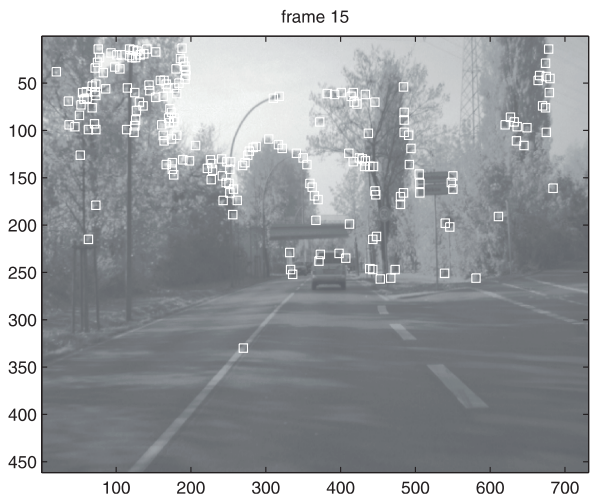


Fig. 14.55.
Frame number 15 from the `bridge-l` image sequence with overlaid features (image from .enpeda. project, Klette et al. 2011)

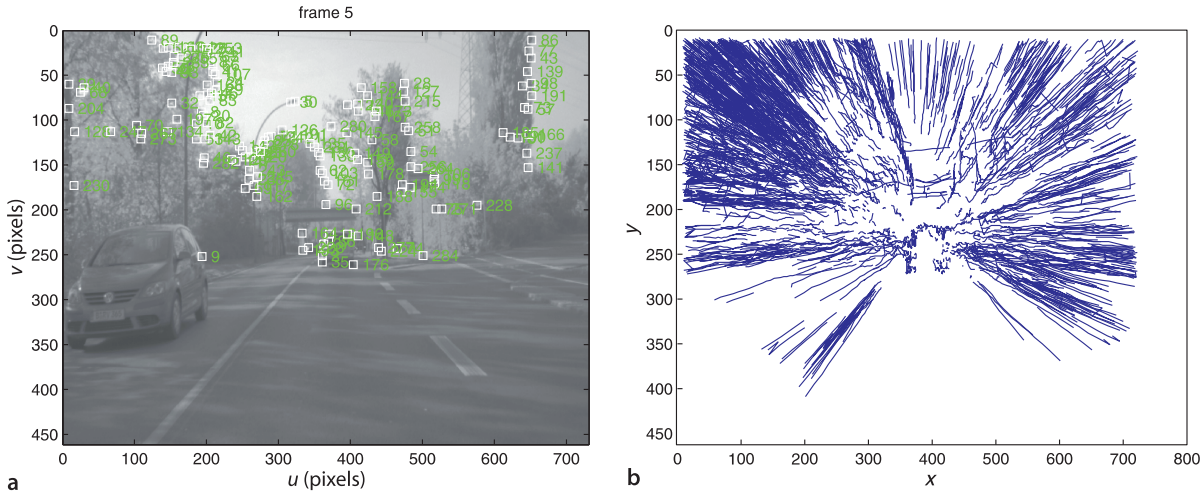


Fig. 14.56. Temporal tracking of features. **a** Features with unique track identifiers shown; **b** all feature tracks

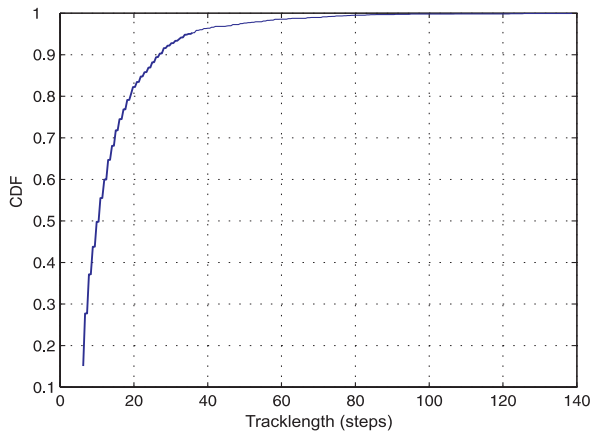


Fig. 14.57. Cumulative histogram of track length. 50% of the tracks are longer than 10 frames, and 5% are longer than 40 frames

Data about the tracks is stored as a property of the class. The feature trajectories can be displayed as individual lines

```
>> t.plot()
```

as shown in Fig. 14.56b. The motion of features outward from the centre of the image is very clear and the point from which features appear to expand is called the centre of expansion. In terms of 2-camera epipolar geometry the second camera is in front of the first camera, so the epipole will be close the middle of the image and the epipolar lines all pass through it. The corresponding points must lie on the epipolar lines so therefore move along these radiating lines, and the amount of motion is inversely related to the depth of the world point. In the next part of the book we will explore the relationship between feature motion and camera motion in more detail.

We see that there are some tracks that are not radial and these are due to the rather simplistic tracker which is not enforcing frame-to-frame epipolar constraints and which RANSAC could eliminate. One of the horizontal tracks is due to a car driving across a bridge which has violated our assumption that the world is rigid, that is, points in the world do not move with respect to each other. The number of frames over which features are tracked varies considerably. The method `tracklengths` returns a vector of the number of frames that each feature was tracked for

```
>> ihist( t.tracklengths(), 'normcdf' )
```

and Fig. 14.57 shows the distribution of feature track lengths. The majority are quite short, the mean is 14 frames, but the longest is 140 frames.

14.9 Wrapping Up

This chapter has covered many topics but the aim has been to demonstrate a multiplicity of concepts that are of use in real robotic vision systems. There have been two common threads through this chapter. The first has been the use of corner features to find distinctive points in images, and matching them to the same world point in another image. The second thread has been the use of additional sources of information to recover the depth of a point or translation scale that is lost in the perspective projection process.

We extended the geometry of single camera imaging to the case of two cameras and showed how corresponding points in the two images are constrained by the fundamental matrix. We showed how the fundamental matrix can be estimated from image data, the effect of incorrect data association, and how to overcome this using the RANSAC algorithm. Using camera intrinsic parameters the essential matrix can be computed and then decomposed to give the camera motion between the two view, but the translation has an unknown scale factor. With some extra information such as the magnitude of the translation the camera motion can be estimated completely. Given the camera motion then the 3-dimensional coordinates of points in the world can be estimated.

World points that lie on a plane induce an homography that is a linear mapping of image points between images. The homography can be used to detect points that do not lie in the plane and can be decomposed to give the camera motion between the two views (translation again has an unknown scale factor) and the normal to the plane.

If the fundamental matrix is known then a pair of overlapping images can be rectified to create an epipolar-aligned stereo pair and dense stereo matching can be used to recover the world coordinates for every point. Errors due to effects such as occlusion and lack of texture were discussed as were techniques to detect these situations.

These multi-view techniques were then used in a number of application examples such as perspective correction, mosaic creation, image retrieval and image sequence analysis.

Further Reading

3-dimensional reconstruction and camera pose estimation has been studied by the photogrammetry community since the mid nineteenth century. 3-dimensional computer vision or *robot vision* has been studied by the computer vision and artificial intelligence communities since the 1960s. This book follows the language and nomenclature associated with the computer vision literature, but the photogrammetric literature can be comprehended with only a little extra difficulty.

Significant early work on multi-view geometry was conducted at laboratories such as Stanford, SRI International, MIT AI laboratory, CMU, JPL, INRIA, Oxford and ETL Japan in the 1980s and 1990s and led to a number of text books being published in the early 2000s.

The definitive references for multiple-view geometry are Hartley and Zisserman (2003) and Ma et al. (2003). These books present quite different approaches to the same body of material. The former takes a more geometric approach while the latter is more mathematical. Unfortunately they use quite different notation, and each differs from the notation used in this book – a summary of the important notational elements is given in Table 14.1. These books all cover feature extraction (using Harris corner features, since they were published before scale invariant feature detectors such as SIFT and SURF corner detectors were developed); the geometry of one, two and N views; fundamental and essential matrix; homographies; and the recovery of 3-dimensional scene structure and camera motion through offline batch techniques. Both provide the key algorithms in pseudo-code and have some supporting MATLAB® code on their associated web sites. The slightly earlier book by Faugeras et al. (2001) covers much of the same material using a fairly mathematical approach and with different

Table 14.1. Rosetta stone. Summary of notational differences between two other popular textbooks and this book

Object	Hartley and Zisserman 2003	Ma et al. 2003	This book
World point	$\mathbf{X}$	P	P
Image point	$\mathbf{x}, \mathbf{x}'$	x_1, x_2	${}^1\boldsymbol{p}, {}^2\boldsymbol{p}$
i^{th} image point	$\mathbf{x}_i, \mathbf{x}'_i$	x_1^i, x_2^i	${}^1p_i, {}^2p_i$
Camera motion	$\mathsf{R}, \mathbf{t}$	R, T	$\boldsymbol{R}, \boldsymbol{t}$
Normalized coordinates	$\mathbf{x}, \mathbf{x}'$	x_1, x_2	$(\bar{u}, \bar{v})$
Camera matrix	P	Π	$\boldsymbol{C}$
Homogeneous quantities	$\mathbf{x}, \mathbf{X}$	x, P	$\tilde{\boldsymbol{p}}, \tilde{\boldsymbol{P}}$
Homogeneous equivalence	$\mathbf{x} = \mathsf{P}\mathbf{X}$	$\lambda x = \Pi P$ $x \sim \Pi P$	$\tilde{\boldsymbol{p}} \simeq \boldsymbol{C}\tilde{\boldsymbol{P}}$

notation again. The older book by Faugeras (1993) focuses on sparse stereo from line features. The recent book by Szeliski (2010) provides a very readable and deeper discussion of the topics in this chapter.

References related to SURF and other feature detectors were previously discussed on page 377. The performance of feature detectors and their matching performance is covered in Mikolajczyk and Schmid (2005) which reviews a number of different feature descriptors including spin images and local jets. ◀

A jet is a vector of higher order derivatives such as $\mathsf{I}_{uu}, \mathsf{I}_{vv}, \mathsf{I}_{uv}, \mathsf{I}_{uuu}, \mathsf{I}_{uuv}, \mathsf{I}_{uvv}, \mathsf{I}_{vvv}, \mathsf{I}_{uuuu}, \mathsf{I}_{uuuv}, \mathsf{I}_{uuvv}, \mathsf{I}_{uvvv}, \mathsf{I}_{vvvv}$ and so on (Mikolajczyk and Schmid 2005).

The RANSAC algorithm described by Fischler and Bolles (1981) is the workhorse of all the feature-based methods discussed in this chapter. A more recent development is PROSAC (Chum and Matas 2005) which exploits the ordering of corresponding points. Pilu (1997) discusses how SVD can be applied to a matrix formed from the distances between features to determine correspondence. Dellaert et al. (2000) describe a probabilistic approach to determining structure from a group of images not necessarily in order.

The term fundamental matrix was defined in the thesis of Luong (1992). The book by Xu and Zhang (1996) is a readable introduction to epipolar geometry. Epipolar geometry can also be formulated for non-perspective cameras in which case the epipolar line becomes an epipolar curve (Mičušík and Pajdla 2003; Svoboda and Pajdla 2002). For three views the geometry is described by the trifocal tensor $\mathcal{T}$ which is a $3 \times 3 \times 3$ tensor with 18 degrees of freedom that relates a point in one image to epipolar lines in two other images (Hartley and Zisserman 2003; Ma et al. 2003). An important early paper on epipolar geometry for an image sequence is Bolles et al. (1987).

The essential matrix was first described a decade earlier in a letter to Nature (Longuet-Higgins 1981) by the eminent theoretical chemist and cognitive scientist Christopher Longuet-Higgins (1923–2004). The paper describes a method of estimating the essential matrix from eight corresponding point pairs. The decomposition of the essential matrix was first described in Faugeras (1993, § 7.3.1) but is also covered in the texts Hartley and Zisserman (2003) and Ma et al. (2003). In this chapter we have estimated camera motion by first computing the essential matrix and then decomposing it. The first step requires at least eight pairs of corresponding points but algorithms such as Nistér (2003), Li and Hartley (2006) compute the motion directly from just five pairs of points. Decomposition of an homography is described by Faugeras and Lustman (1988), Hartley and Zisserman (2003), Ma et al. (2003), and the comprehensive technical report by Malis and Vargas (2007). The relationships between these matrices, camera motion, and the relevant Toolbox functions are summarized in Fig. 14.58.

Stereo cameras and stereo matching software are available today from several vendors and can provide high-resolution depth maps at more than 10 Hz on standard computers. A decade ago this was difficult and custom hardware including FPGAs was required to achieve real-time operation (Corke et al. 1999; Woodfill and Von Herzen 1997). The application of stereo vision for planetary rover navigation is discussed by

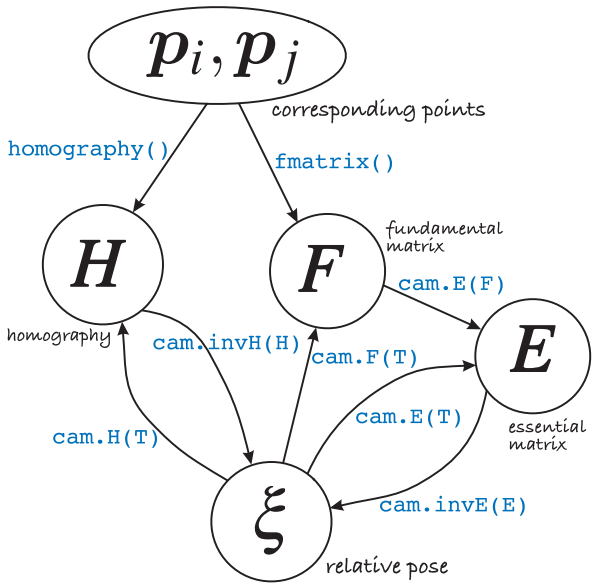


Fig. 14.58.
Toolbox functions and camera object methods, and their inter-relationship

Matthies (1992). More than two cameras can be used, and multi-camera stereo was introduced by Okutomi and Kanade (1993) and provides robustness to problems such as the picket fence effect.

Brown et al. (2003) provide a readable review of stereo vision techniques with a focus on real-time issues. An old but clearly written book on the principles of stereo vision is Shirai (1987). Scharstein and Szeliski (2002) consider the stereo process as four steps: matching, aggregation, disparity computation and refinement. The cost and performance of different algorithms for each step are compared. The example in this chapter would be described as: NCC matching, box filter aggregation, winner takes all, and sub-pixel refinement. More sophisticated approaches are described that take similarity of neighbouring pixels into account using iterative regularization, dynamic programming or graph-based methods which can give improved performance at discontinuities and can explicitly model occlusion. However improving the smoothness and appearance of disparity data involves making assumptions about the world for which the images do not provide strong evidence, and should be used with caution for robotic control. The similarity of a stereo camera to our own two eyes is very striking, and while we do make strong use of stereo vision it is not the only technique we use to infer distance (Cutting 1997).

The ICP algorithm (Besl and McKay 1992) is used for a wide range of applications from robotics to medical imaging. ICP is fast but determing the correspondences via nearest neighbours is an expensive $O(N^2)$ operation. Many variations have been developed that make the approach robust to outlier data and to improve computational speed for large datasets. Salvi et al. (2007) provide a recent review and comparison of some different algorithms. Determining the relative orientation between two sets of points is a classical problem and the SVD approach used here is described by Arun et al. (1987). Solutions based on quaternions and orthornormal rotation matrices were described by Horn (Horn et al. 1988; Horn 1987).

Structure from motion (SfM), the simultaneous recovery of world structure and camera motion, is a classical problem in computer vision. Two useful review papers are by Huang and Netravali (1994) which provides a taxonomy of approaches, and Jebara et al. (1999). Broida et al. (1990) describe an early recursive SfM technique for a monocular camera sequence using an EKF where each world point is represented by its (X, Y, Z) coordinate. McLauchlan provides a detailed description of a variable-length state estimator for SfM (McLauchlan 1999). Azarbayejani and Pentland (1995) present a recursive approach where each world point is parameterized by a scalar, its depth with respect to the first image. A more recent algorithm with bounded estimation error is described

by Chiuso et al. (2002) and also discusses the problem of scale variation. The MonoSlam system by Davison et al. (2007) is an impressive monocular SfM system that maintains a local map that includes features even when they are not currently in the field of view. The application of SfM to large-scale urban mapping is becoming increasing popular and Pollefeys et al. (2008) describe a system for offline processing of large image sets.

The offline SfM problem, in which a sequence of images is processed to recover the motion and structure, is not covered in this book. The approach typically involves estimating the camera matrix for each view, computing a projective reconstruction, and then upgrading it to a Euclidean reconstruction (Hartley and Zisserman 2003; Ma et al. 2003). A key part of these offline techniques is bundle adjustment which is an optimization process that adjusts the estimated camera poses and world points to minimize the error between estimated and actual image points. A good introduction to bundle adjustment is Triggs et al. (2000).

The SfM problem can be simplified by using stereo rather than monocular image sequences (Molton and Brady 2000; Zhang et al. 1992), or by incorporating inertial data (Strelow and Singh 2004). A related technique is visual odometry (VO) which is concerned only with recovering the camera motion. Nistér et al. (2006) describe a point feature-based system for monocular or stereo vision. Maimone et al. (2007) describe experience with stereo-camera VO on the Mars rover and Corke et al. (2004) describe catadioptric VO for a prototype planetary rover.

Mosaicing is a process as old as photography. In the past it was highly skilled and labour intensive requiring photographs, scalpels and sandpaper. The surface of the Moon and nearby planets was mosaiced manually in the 1960s using imagery sent back by robotic spacecraft. Today a number of high-quality mosaicing tools are available for creating panoramas, for example the Hugin open source project http://hugin.sourceforge.net and the proprietary AutoStitch.

The "bag of words" technique for image retrieval was first proposed by Sivic and Zisserman (2003) and has been used by many other researchers since. A notable extension for robotic applications is FABMAP (Cummins and Newman 2008) which explicitly accounts for the joint probability of feature occurrence and associates a probability with the image match.

Image sequence analysis is the core of many real-time robotic vision systems. Real-time feature tracking is described by Hager and Toyama (1998), Lucas and Kanade (1981) and is typically based on the computationally cheaper Harris detectors or the pyramidal Kanade-Lucas-Tomasi (KLT) tracker. SURF detectors are still too time consuming to use for this purpose although some C-based implementations and GPU implementations are capable of modest real-time perofrmance.

Resources

The field of computer vision has progressed through the availability of standard datasets. These have enabled researchers to quantitatively compare the performance of different algorithms on the same data. One of the earliest collections of stereo image pairs was the JISCT dataset (Bolles et al. 1993) named for the research groups that contributed to it: JPL, INRIA, SRI, CMU, and Teleos. It is available at http://vasc.ri.cmu.edu/idb/html/jisct. The more recent Middlebury dataset (Scharstein and Szeliski 2002) at http://vision.middlebury.edu/stereo provides an extensive collection of stereo images, at high resolution, taken at different exposure settings and including ground truth data. Stereo images from NASA's Mars exploration rovers Spirit and Opportunity are available online at http://marsrover.nasa.gov/gallery/3d. These are in anaglyph format with left and right images encoded as red and cyan respectively. The red and blue color planes of the anaglyph image are rectified left and right images. Motion datasets include classic motion sequences of indoor scenes http://vasc.ri.cmu.edu//idb/html/motion, people moving inside a building http://

homepages.inf.ed.ac.uk/rbf/CAVIARDATA1, traffic scenes http://i21www.ira.uka.de/image_sequences, and from a moving vehicle http://www.mi.auckland.ac.nz/EISATS.

An implementation of the KLT feature tracker, in C, written by Stan Birchfield is available at http://www.ces.clemson.edu/~stb/klt. A GPU-based version of KLT, in C, by Christoper Zach is available at http://www.cs.unc.edu/~cmzach/opensource.html. Pointers to SIFT and SURF implementations are given on page 377. The Epipolar Geometry Toolbox (Mariottini and Prattichizzo 2005) for MATLAB® by Gian Luca Mariottini and Domenico Prattichizzo is available at http://egt.dii.unisi.it/#tth_sEc7 and handles perspective and catadioptric cameras. Andrew Davison's MonoSLAM system for C and MATLAB® is available at http://www.doc.ic.ac.uk/~ajd/software.html.

There are several offline or batch SfM Toolboxes available. The Structure and Motion Toolkit in MATLAB® by Philip Torr (2002) is available at http://www.mathworks.com/matlabcentral/fileexchange/4576-structure-and-motion-toolkit-in-MATLAB®. The *Structure from Motion Toolbox* by Vincent Ribaud is a collection of popular SFM algorithms (Rabaud, http://vision.ucsd.edu/~vrabaud/toolbox) and is available at http://code.google.com/p/vincents-structure-from-motion-matlab-toolbox. That in turn makes use of the Sparse Bundle Adjustment (SBA) tool by Lourakis and Argyros (2009). An alternative bundle adjustment package is Simple Sparse Bundle Adjustment (SSBA) by Christoper Zach which is available at http://www.cs.unc.edu/~cmzach/opensource.html.

The fundamental matrix song can be found at http://danielwedge.com/fmatrix/.

Exercises

1. Corner features. Examine the cumulative distribution of corner strength for Harris and SURF features. What is an appropriate way to choose strong corners for feature matching?
2. Feature matching. We could define the quality of descriptor-based feature matching in terms of the percentage of inliers after applying RANSAC.
   a) Take any image. We will match this image against various transforms of itself to explore the robustness of SURF and Harris features. The transforms are: *(a)* scale the intensity by 70%; *(b)* add Gaussian noise with standard deviation of 0.05, 0.5 and 2 greyvalues; *(c)* scale the size of the image by 0.9, 0.8, 0.7, 0.6 and 0.5; *(d)* rotate by 5, 10, 15, 20, 30, 40 degrees.
   b) For the Harris detector compare the performance for the structure-tensor-based feature and and the patch descriptor sizes of $3 \times 3, 7 \times 7$ and 11×11 and 15×15.
   c) Try increasing the suppression radius for SURF and Harris corners. Does the lower density of matches improve the matching performance?
   d) The Harris detector can process a color image. Does this lead to improved performance compared to the greyscale version of the same image.
   e) Is there any correlation between outlier matches and strength of the corner features involved?
3. Write the equation for the epipolar line in image two, given a point in image one.
4. Show that the epipoles are the null-space of the fundamental matrix.
5. Can you determine the camera matrix $\boldsymbol{C}$ for camera two given the fundamental matrix and the camera matrix for camera one?
6. Estimating the fundamental matrix (page 391)
   a) For the synthetic data example vary the number of points and the additive Gaussian noise and observe the effect on the residual.
   b) For the Eiffel tower data observe the effect of varying the parameter to RANSAC. Repeat this with SURF features computed with a lower strength threshold (the default is 0.002).
   c) What is the probability of drawing 8 inlier points in a random sample (without replacement)?

7. Essential matrix (page 390)
   a) Create a set of corresponding points for a camera undergoing pure rotational motion, and compute the fundamental and essential matrix. Can you recover the rotational motion?
   b) For a case of translational and rotational motion visualize both poses that result from decomposing the essential matrix. Sketch it or use `trplot`.
8. Sparse stereo (page 401)
   a) The ray intersection method can return the closest distance between the rays (which is ideally zero). Plot a histogram of the closing error and compute the mean and maximum error.
   b) The assumed camera translation magnitude was 30 cm. Repeat for 25 and 35 cm. Are the closing error statistics changed? Can you determine what translation magnitude minimizes this error?
9. Derive a relationship for depth in terms of disparity for the case of verged cameras. That is, cameras with their optical axes intersecting similar to the cameras shown in Fig. 14.6.
10. Stereo vision. Using the rock piles example (page 405)
    a) Use `idisp` to zoom in on the disparity image and examine pixel values on the boundaries of the image and around the edges of rocks.
    b) Experiment with different similarity measures and window sizes. What effects do you observe in the disparity image and computation time?
    c) Experiment with changing the disparity range. Try `[50,90]`, `[30,90]`, `[40,80]` and `[40,100]`. What happens to the disparity image and why?
11. Using the rock piles example (page 405) obtain the disparity space image D
    a) For selected pixels (u, v) plot $D(u, v, d)$ versus d. Look for pixels that have a sharp peak, broad peak and weak peak. Repeat this for stereo computed using ZSSD similarity. For a selected row v display $D(u, v, d)$ as an image. What does this represent?
    b) For a particular pixel plot s versus d, fit a parabola around the maxima and overlay this on the plot.
    c) Use raw data from the DSI, find the second peak at each pixel and compute the ambiguity ratio
    d) Display the epipolar lines on image two for selected points in image one.
12. Download an analglyph image and convert it into a pair of greyscale images, then compute dense stereo.
13. Epipolar geometry
    a) Create two central cameras, one at the origin and the other translated in the x-direction. For a sparse fronto-parallel grid of world points display the family of epipolar lines in image two that correspond to the projected points in image one. Describe these epipolar lines? Repeat for the case where camera two is translated in the y- and z-axes and rotated about the x-, y- and z-axes. Repeat this for combinations of motion such as x- and z-translation or x-translation and y-rotation.
    b) The garden example of Fig. 14.16 has epipolar lines that slope slightly downward. What does this indicate about the two camera views?
14. Homography (page 396)
    a) Compute Euclidean homographies for translation in the x-, y- and z-directions and for rotation about the x-, y- and z-axes. Convert these to projective homographies and apply to a fronto-parallel grid of points. Is the resulting image motion what you would expect? Apply these homographies as a warp to a real image such as Lena.
    b) Decompose the homography of Fig. 14.15, the garden image, to determine the plane of the wall with respect to the camera. You will need the camera intrinsic parameters.

15. Plane fitting (page 419)
    a) Test the robustness of the plane fitting algorithm to additive noise and outlier points.
    b) Implement an iterative approach with weighting to minimize the effect of outliers.
    c) Create a RANSAC-based plane fit algorithm that takes random samples of three points to solve for Eq. 14.17. Use the `fmatrix` and `homography` code to guide you. You will need to create a number of functions that are invoked by the `ransac_driver`.
16. ICP (page 420)
    a) Run the ICP example on your computer and watch the animation.
    b) Change the initial relative pose between the point clouds. Try some very large rotations.
    c) Increase the noise added to the data points.
    d) For the case where there are missing and/or spurious data points experiment with different values of the `'distthresh'` option.
17. Structure from motion (page 422)
    a) Experiment with different levels of camera noise and see the effect on estimated camera position.
    b) What is the effect of noise on estimated camera orientation?
    c) Modify the simulation to model an odometer, based on the known distance travelled between frames, and use that information to correct the translational scale. Repeat with a systematic scale error in the odometer. Repeat with additive noise in the odometer.
    d) Use a moving average filter to smooth the frame to frame world point estimates.
    e) Use an extended Kalman filter to smooth the frame to frame world point estimates (challenging).
18. Perspective correction (page 428)
    a) Create a virtual view looking downward at 45° to the front of the cathedral.
    b) Create a virtual view from the original camera height but with the camera rotated 20° to the left.
    c) Find another real picture with perspective distortion and attempt to correct it.
19. Mosaicing (page 431)
    a) Run the example file `mosaic` and watch the whole mosaic being assembled.
    b) Modify the way the tile is pasted into the composite image to use pixel averaging rather than addition.
    c) Modify the way the tile is pasted into the composite image so that pixels closest to the principal point are used.
    d) Run the software on a set of your own overlapping images and create a panorama.
20. Bag of words (page 433)
    a) Examine the different support regions of different visual words using the `exemplars` method.
    b) Investigate the effect of changing the number of stop words.
    c) Investigate the effect of changing the size of the vocabulary. Try 1 000, 1 500, 2 500, 3 000.
    d) Build a bag of words from a set of your own images.
21. Image sequence tracking (page 439)
    a) Run the example on your computer.
    b) Repeat using the default Harris feature descriptor, that is, without the `'patch'` option. What happens to the quality of the tracks?
    c) Repeat with different patch sizes. What happens to the quality of the tracks?
    d) Repeat using SURF rather than Harris corner features. Is the quality of the tracks improved? How has compute time changed?
    e) Modify the tracker so that it keeps an estimate of the velocity of the feature in the image (based on its motion over two or more previous frames). Modify the association logic so that the search disk is centered on the predic-ted position of the feature rather than its previous position. The filter could be based on a simple constant-

velocity model, an $\alpha - \beta$ tracker (use the `AlphaBetaFilter` class) or subclass the Toolbox Kalman filter (`KalmanFilter`) abstract superclass.

f) At each frame estimate the fundamental matrix using RANSAC. Modify the association logic to use the epipolar line constraint.

g) Estimate the essential matrix from frame to frame. The required camera calibration data is in the `README` file. Decompose the matrix to determine frame to frame change in pose. Integrate the change in orientation over time to provide a visual compass or gyroscope function. How could you recover the unknown scale factor on frame to frame translation? Can you plot the car's trajectory as seen from overhead?

h) The bridge sequence was recorded in stereo and so far we have used just the left-hand camera. The folder `bridge-r` contains the corresponding right-hand images. Using the camera calibration data in the `README` file tp perform a sparse stereo reconstruction for every image pair. Use ICP to determine the change in pose of the car from frame to frame.

i) Perform a dense stereo reconstruction for every image pair.

j) Compute Harris features for live video captured on your computer and overlay these on the captured frame. How many frames per second will this run on your computer? Use the `step` method of the `Tracker` class to track these live features.