Chapter

14 Using Multiple Images

In the previous chapter we learned 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 characterized 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. 11.6. 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 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 $\lambda \boldsymbol{t} \in \mathbb{R}^3$ 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 nontrivial 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 for the case of a single camera. 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 to encompass multiple image planes and show the geometric relationship between pairs of images. Stereo vision is an important technique for robotics where information from two images of a scene, taken from different viewpoints, is combined to determine the 3-dimensional structure of the world. We discuss sparse and dense approaches to stereo, and reconstruction, in some detail in Sect. 14.3. Bundle adjustment is a very general approach to combining information from many cameras and is introduced in Sect. 14.4. The 3-dimensional information that is created is typically represented as a *point cloud*, a set of 3D points, and techniques for plane fitting and alignment of such data are introduced in Sect. 14.5. For some applications we can use RGBD cameras which return depth as well as color information and the underlying principle of structured light is introduced in Sect. 14.6.

We finish this chapter, and this part of the book, with four application examples based on the concepts we have learned. Section 14.7.1 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 14.7.2 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 14.7.3 describes image retrieval which is the problem of finding which image, from an existing set of images, is most similar to some new image. This can be used by a robot to determine whether it has visited a particular place, or seen the same object, before. Section 14.7.4 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 bodies 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 in the Toolbox can process an image sequence provided as a matrix with more than two dimensions. 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.

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

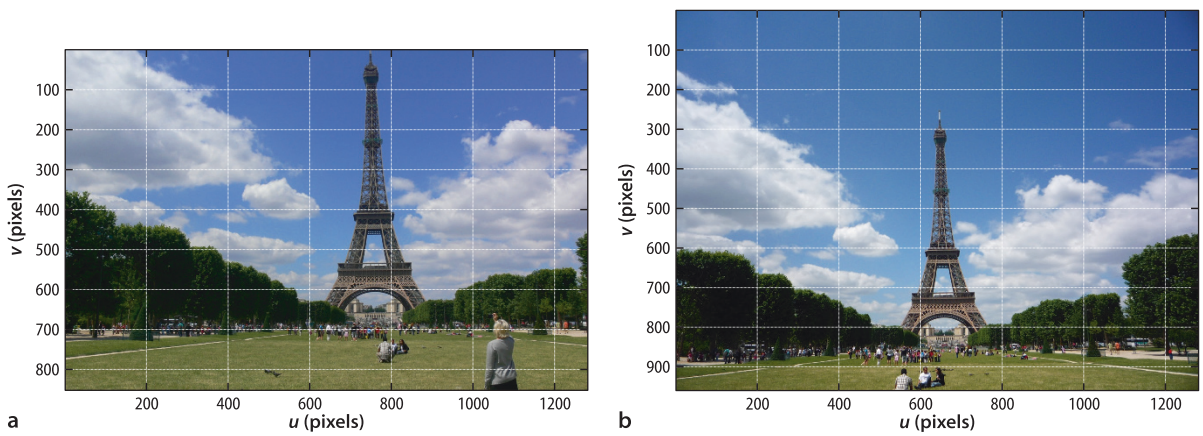
or SURF features

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

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

Consider the general case of two sets of features points: $\{{}^1\boldsymbol{p}_i \in \mathbb{Z}^2, i = 1 \cdots N_1\}$ in the first image and $\{{}^2\boldsymbol{p}_j \in \mathbb{Z}^2, j = 1 \cdots N_2\}$ in the second image. Since these are distinctive image points we would expect a significant number of points in image one would cor-

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 pixels tall in **a** and **b** respectively. The camera that captured image **b** is held by the person in the bottom-right corner of **a**



respond to points found in image two. The problem is to determine which $({}^2u_j, {}^2v_j)$, if any, corresponds to each $({}^1u_i, {}^1v_i)$.

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 highly unlikely that corresponding pixels will have the same value. Even if intensity variation was eliminated there are likely 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

```
>> hf(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

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

but it is difficult to know whether this value represents strong similarity or not since the units are not very intuitive. Typically we would compare feature ${}^1\boldsymbol{f}_i \in \mathbb{R}^M$ with all features in the other image $\{{}^2\boldsymbol{f}_j \in \mathbb{R}^M, j = 1 \cdots N_2\}$ and choose the one that is most similar. However a short descriptor vector like this is still insufficiently distinctive and prone to incorrect matching. Feature descriptors are often referred to as feature vectors.

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

```
>> hf = 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, rearranged into a vector and then normalized to create a unit vector. We can use the ZNCC similarity measure from Table 12.1 in 1-dimensional form to compare these descriptor vectors

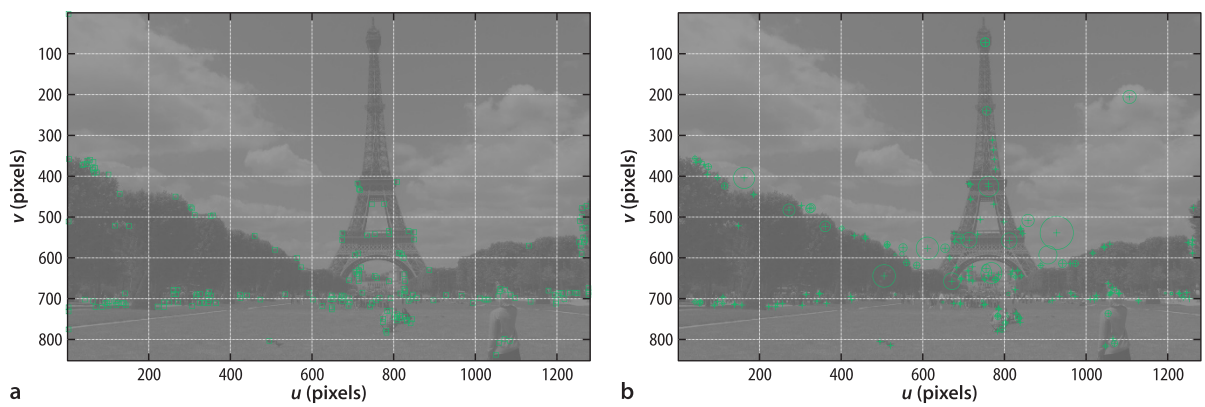


Fig. 14.2. Corner features computed for Fig. 14.1a. **a** Harris corner features; **b** SURF corner features showing scale

$$
\begin{aligned}
s &= \frac{\sum_{i=1}^{N}\left(I_1[i]-\overline{I}_1\right)\left(I_2[i]-\overline{I}_2\right)}{\sqrt{\sum_{i=1}^{N}\left(I_1[i]-\overline{I}_1\right)^2 \cdot \sum_{i=1}^{N}\left(I_2[i]-\overline{I}_2\right)^2}} \\
&= \underbrace{\frac{I_1[i]-\overline{I}_1}{\sqrt{\sum_{i=1}^{N}\left(I_1[i]-\overline{I}_1\right)^2}}}_{f_1} \cdot \underbrace{\frac{I_2[i]-\overline{I}_2}{\sqrt{\sum_{i=1}^{N}\left(I_2[i]-\overline{I}_2\right)^2}}}_{f_2}
\end{aligned}
\tag{14.1}
$$

which we have factored into the dot product of the descriptor unit-vectors associated with each image patch. Determining the similarity of two descriptors using normalized cross-correlation is simply the dot product of two descriptors and the resulting similarity measure $s \in [-1, 1]$ has some meaning – perfect match is $s = 1$ and $s \geq 0.8$ is typically considered a good match. For the example above

```
>> hf(1).ncc( hf(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 spatial arrangement have the same histogram.

The SURF algorithm computes a 64-element descriptor▸ vector to describe the feature point in a 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

```
>> sf1 = isurf(im1)
sf1 =
1288 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
>> sf2 = isurf(im2)
sf2 =
1426 features (listing suppressed)
  Properties: theta image_id scale u v strength descriptor
```

which results in two vectors of `SurfPointFeature` objects. Over a thousand corner features were found in each image.

Detectors versus descriptors. When matching world feature points, or landmarks, between different views we must first *find* points that are distinctive. This is the job of the detector and results in a coordinate (u, v) and perhaps a scale factor or orientation. The second task is to *describe* the region around the point in a way that allows it to be matched as decisively as possible with the region around the corresponding point in the other view. This is the descriptor which is typically a long vector formed from pixel values, histograms, gradients, histograms of gradient and so on. There are many detectors to choose from: Harris and variants, Shi-Tomasi, FAST, AGAST, MSER etc.; as well as many descriptors: ORB, BRISK, FREAK, CenSurE (aka STAR), HOG, ACF etc. Some algorithms such as SIFT and SURF define both a detector and a descriptor. The SIFT descriptor is a form of HOG descriptor.

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

```
>> m = sf1.match(sf2)
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.

```
>> 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}, 'dark')
>> 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 and we will deal with these shortly. 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] = sf1.match(sf2);
>> 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 `sf1(215)` and `sf2(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 = sf1.match(sf2, 'all');
>> histogram(m2.distance, 'Normalization', 'cdf')
```

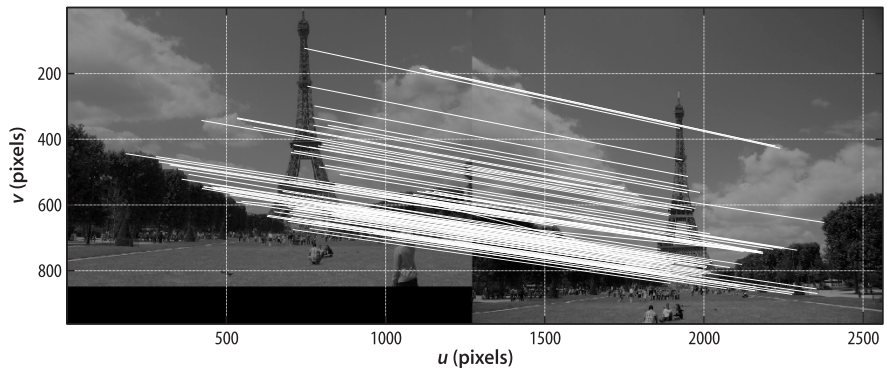


Fig. 14.3. Feature matching. Subset (100 out of 1 664) of matches based on SURF descriptor similarity. We note that a few are clearly incorrect

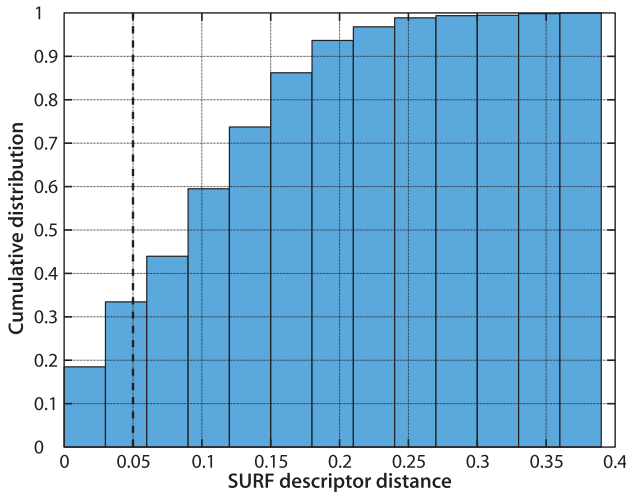


Fig. 14.4.
Cumulative distribution of feature distance

shown in Fig. 14.4. It shows that 35% of all matches have descriptor distances below 0.05 whereas the maximum distance can be over ten times larger – such matches are less likely to be valid. We can specify a distance threshold

```
>> mm = sf1.match(sf2, 'thresh', 0.05);
```

but choosing the threshold value is always problematic. By default the method selects all matches whose distance is less than the median of all distances. Alternatively, we could choose to take the N best matches

```
>> mm = sf1.match(sf2, 'top', N);
```

Feature matching is computationally expensive – it is an $O(N^2)$ problem since every feature descriptor in one image must be compared with every feature descriptor in the other image. More sophisticated systems store the descriptors in a data structure like a kd-tree so that similar descriptors – nearest neighbors in feature space – can be easily found.

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 right. 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 $\mathbf{P}$ 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 moving camera taking a picture from two different viewpoints.▸ The center of each camera, the origins of {1} and {2}, plus the world point $\mathbf{P}$ defines a plane in space – the epipolar plane. The world point $\mathbf{P}$ is projected onto the image planes of the two cameras at points ${}^1\mathbf{p}$ 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 $\mathbf{P}$. The camera center, ${}^1\mathbf{e}$ 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.

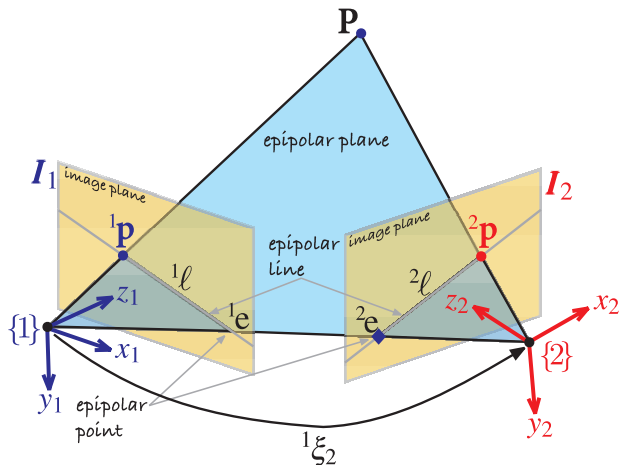


Fig. 14.5.
Epipolar geometry showing the two cameras with associated coordinate frames {1} and {2} and image planes. The world point **P** and the two camera centers form the epipolar plane, and the intersection of this plane with the image-planes form epipolar lines

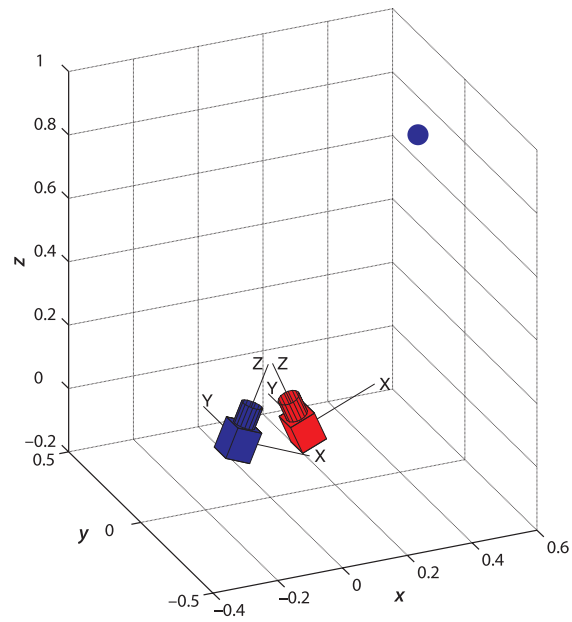


Fig. 14.6.
Simulation of two cameras and a target point. The origins of the two cameras are offset along the *x*-axis and the cameras are *verged*, that is, their optical axes intersect

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 = SE3(-0.1, 0, 0) * SE3.Ry(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. 11.1.2. Similarly for the second camera

```
>> T2 = SE3(0.1, 0,0)*SE3.Ry(-0.4);
>> cam2 = CentralCamera('name', 'camera 2', 'default', ...↵
    'focal', 0.002, 'pose', T2);
```

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]';
```

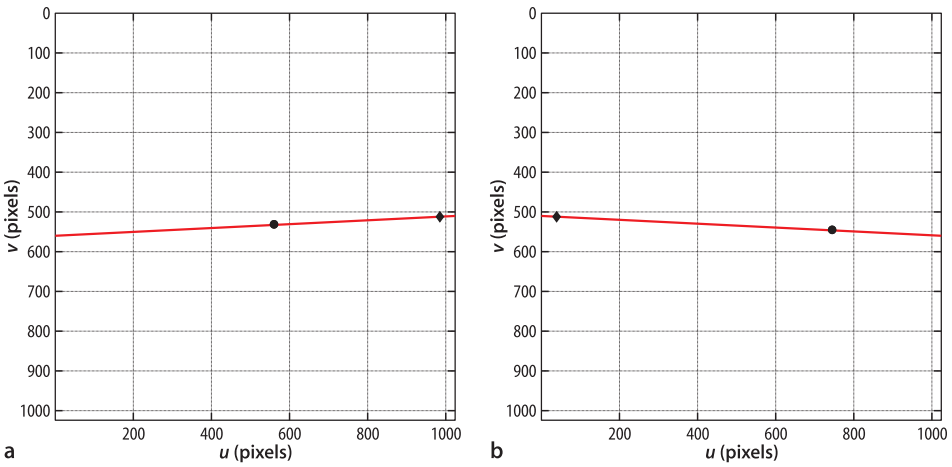


Fig. 14.7.
Epipolar geometry simulation showing the virtual image planes of two Toolbox `CentralCamera` objects. The perspective projection of point **P** is a *black circle*, the projection of the other camera's center is a *black* ♦*-marker*, and the epipolar line is shown in *red*

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 center of each camera to the other camera's image plane

```
>> 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 ${}^{1}\mathbf{p}$ and ${}^{2}\mathbf{p}$ expressed in homogeneous form and $\boldsymbol{F} \subset \mathbb{R}^{3\times 3}$ is known as the fundamental matrix. We can rewrite this as

$$ {}^{2}\tilde{\boldsymbol{p}}^{T}\ {}^{2}\tilde{\ell} = 0 \tag{14.3} $$

2D projective geometry in brief. The projective plane $\mathbb{P}^2$ is the set of all points $(x_1, x_2, x_3)^T$, $x_i \in \mathbb{R}$ and x_i not all zero. Typically the 3-tuple is considered a column vector. A point $\mathbf{p} = (u, v)$ is represented in $\mathbb{P}^2$ by homogeneous coordinates $\tilde{\boldsymbol{p}} = (u, v, 1)^T$. 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) nonzero scale factor. A point in $\mathbb{P}^2$ can be represented in nonhomogeneous, or Euclidean, form $\mathbf{p} = (x_1/x_3, x_2/x_3)^T$ in $\mathbb{R}^2$. The homogeneous vector $(u, v, f)^T$, where f is the focal length in pixels, is a vector from the camera's origin that points toward the world point $\mathbf{P}$. More details are given in Sect. C.2.

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).

where

$$ {}^2\tilde{\ell} \simeq \boldsymbol{F}\, {}^1\tilde{\boldsymbol{p}} \tag{14.4} $$

is the equation of a line, the epipolar line, along which conjugate point in image two must lie. This line is a function of the point coordinate ${}^1\tilde{\boldsymbol{p}}$ in image one and Eq. 14.3 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

$$ \boldsymbol{F} \simeq \boldsymbol{K}_2^{-T} [\boldsymbol{t}]_\times \boldsymbol{R} \boldsymbol{K}_1^{-1} \tag{14.7} $$

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

where $\boldsymbol{K}_1$ and $\boldsymbol{K}_2$ are the camera intrinsic matrices defined in Eq. 11.7, 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, but the mathematics can be expressed more simply this way. Toolbox functions always describe camera pose with respect to the world frame.

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

and for the two image points computed earlier

```
>> e2h(p2)' * F * e2h(p1)
ans =
   1.1102e-16
```

we see that Eq. 14.2 holds.

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

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

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 $\det(F) = 0$.

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 $\boldsymbol{F}$

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

in homogeneous coordinates or

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

in Euclidean coordinates – as shown in Fig. 14.7. The epipole for camera two is the left null space of the fundamental matrix

This is the right null space of the matrix transpose. The MATLAB function `null` returns the right null space.

```
>> 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 $\boldsymbol{E} \subset \mathbb{R}^{3\times 3}$ is the essential matrix and ${}^2\tilde{\boldsymbol{x}}$ and ${}^1\tilde{\boldsymbol{x}}$ are conjugate points in homogeneous normalized image coordinates. This matrix is a simple function of the relative camera pose

For a camera with a focal length of 1 and the coordinate origin at the principal point, see page 322.

$$ \boldsymbol{E} \simeq [\boldsymbol{t}]_\times \boldsymbol{R} \tag{14.9} $$

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 nonzero 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 B.

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

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

Equating terms with 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.

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

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

```
>> sol = cam1.invE(E)
sol(1) =
    1.0000         0         0   -0.1842
         0   -1.0000         0         0
         0         0        -1  -0.07788
         0         0         0         1

sol(2) =
    0.6967         0   -0.7174    0.1842
         0    1.0000         0         0
    0.7174         0    0.6967   0.07788
         0         0         0         1
```

The true relative pose from camera one to camera two is

```
>> inv(cam1.T) * cam2.T
ans =
    0.6967         0   -0.7174    0.1842
         0         1         0         0
    0.7174         0    0.6967   0.07788
         0         0         0         1
```

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

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 the scale is correct because the essential matrix was determined directly from the relative pose between the cameras.

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

In the general case we do not know the pose of the two cameras, so 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

```
>> cam1.project(Q)'
ans =
  429.7889  512.0000
```

is a reasonable value. We can test each of the possible relative poses in `sol` by using them to move the first camera. We can create an instance copy of the first camera with an arbitrary displacement using the `move` method

```
>> cam1.move(sol(1).T).project(Q)'
ans =
   NaN   NaN
```

and the values of `NaN` indicate that the point `Q` is not visible from this camera pose – in fact it is behind the camera. The second solution

```
>> cam1.move(sol(2).T).project(Q)'
ans =
  594.2111  512.0000
```

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

```
>> sol = cam1.invE(E, Q)
sol =
    0.6967         0   -0.7174    0.1842
         0    1.0000         0         0
    0.7174         0    0.6967   0.07788
         0         0         0         1
```

in which case only the valid solution is returned.

In summary these 3×3 matrices, the fundamental and the essential matrix, encode the parameters and relative pose 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 centers 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 from Real Image Data

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

The SE3 class, a 4×4 matrix is applied to a set of 3D points expressed as a 3×20 matrix. The $*$ operator for the SE3 class does the right thing here, it first converts the second matrix to homogeneous form, performs the matrix multiplication, and then converts back to Euclidean form.

```
>> P = SE3(-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 2.645e-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);
```

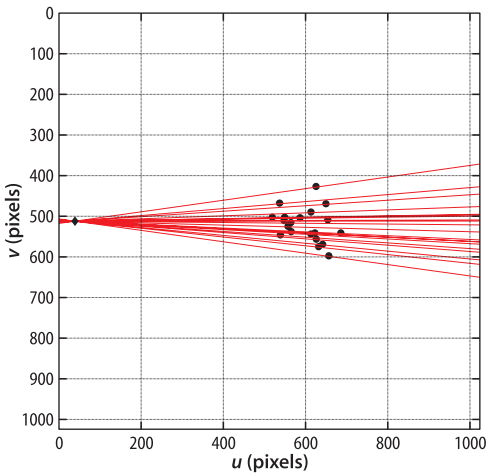


Fig. 14.8.
A pencil of epipolar lines on the camera two image plane. Note how all epipolar lines pass through the epipole which is the projection of camera one's center

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, that each camera can see the center 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]);
```

The fundamental matrix estimation

```
>> fmatrix(p1, p2)
maximum residual 0.000424
ans =
    0.0000   -0.0001    0.0628
    0.0000   -0.0000    0.0098
   -0.0192    0.0511  -29.7672
```

now has a residual that is over 20 orders of magnitude larger than previously. 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 =
   1.5356e-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
2.03262e-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 464 we had found *candidate* correspondences 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 therefore raising the limit on the 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 image sensor 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 by warping as discussed in Sect. 12.7.4 but this is computationally expensive. A cheaper alternative is to find the coordinates of the features in the distorted image and correct those using the inverse of the 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 also updated the elements of the `FeatureMatch` vector

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

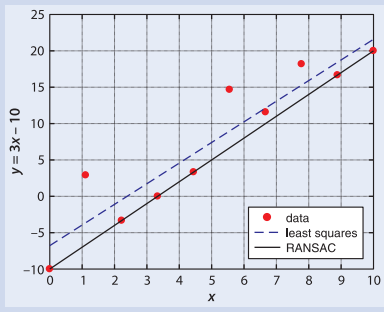
which now displays the total number of inliers and outliers. Compared to page 463 the elements of the vector

```
>> 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 +
```

Example of RANSAC fitting a line to data with a few erroneous, or outlier, points. The blue dashed line is the least squares best fit and is clearly biased away from the true line by the outlier data points. Despite 40% of the points not fitting the model RANSAC finds the parameters of the consensus line, the line that the largest number of points agree on

```
>> [theta,inliers] = ransac(@linefit, [x; y], 1e-3)
theta =
    3.0000  -10.0000
inliers =
     1     3     4     5     9    10
```

and the indices of the data points that support that model. [examples/linefit.m]



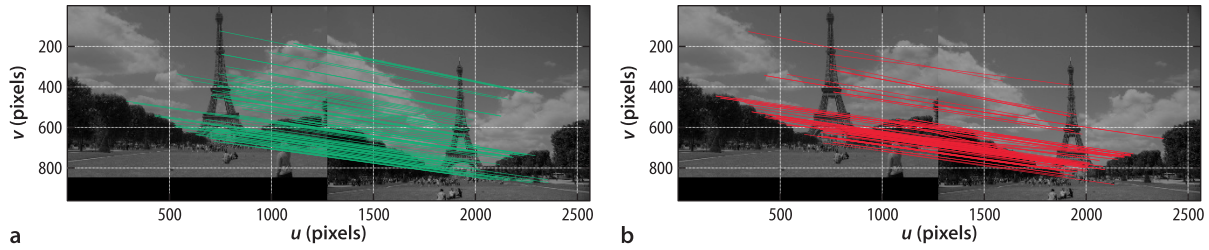


Fig. 14.9. Results of SURF feature matching after RANSAC. **a** Subset of all inlier matches; **b** subset of the outlier matches, some are quite visibly incorrect while others are more subtly wrong

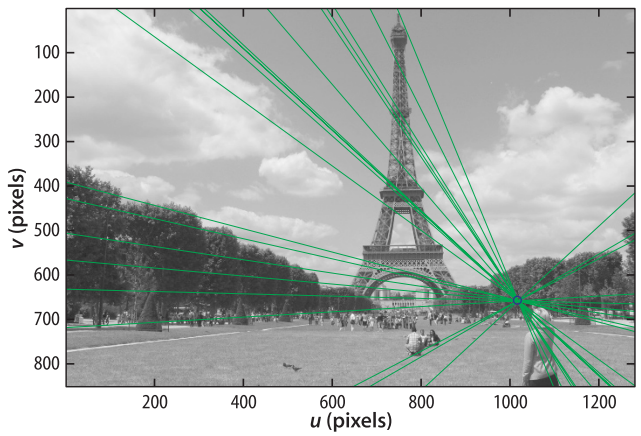


Fig. 14.10. Image from Fig. 14.1a showing epipolar lines converging on the projection of the second camera's center. In this case the second camera is clearly visible in the bottom right of the image

The second match has been determined to be an outlier even though it was the second strongest candidate based on descriptor similarity. Similarity alone is not enough, the corresponding points in the two images must be *consistent* with the epipolar geometry as represented by the consensus fundamental matrix.

now have a trailing plus or minus sign to indicate whether the corresponding match is an inlier or outlier respectively. 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');
```

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

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

```
>> 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°.

14.2.4 Planar Homography

In this section we will consider a camera viewing a set 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 $\boldsymbol{H} \subset \mathbb{R}^{3\times3}$ is a nonsingular matrix known as an homography, a planar homography, or the homography *induced* by the plane.

An homography matrix has arbitrary scale and therefore 8 degrees of freedom. With respect to Eq. 14.14 the rotation, translation and normal have 3, 3 and 2 degrees of freedom respectively, for a total of 8. Homographies form a group: the product of two homographies is another homography, the identity homography is a unit matrix and an inverse operation is defined.

For example consider again the pair of cameras from page 465 now observing a 3×3 grid of points

```
>> Tgrid = SE3(0,0,1)*SE3.Rx(0.1)*SE3.Ry(0.2);
>> P = mkgrid(3, 1.0, 'pose', Tgrid);
```

where `Tgrid` is the pose of the grid coordinate frame $\{G\}$ and the grid points are centered 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, '+')
```

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

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

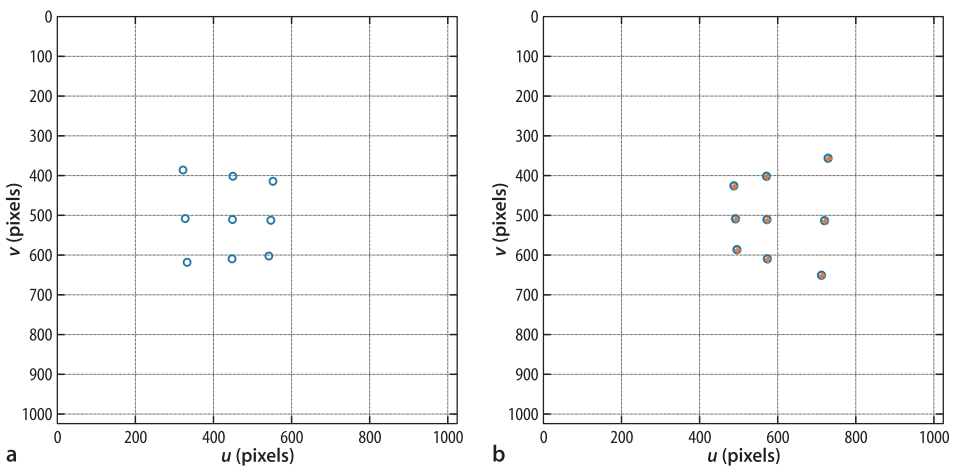


Fig. 14.11. Views of the oblique planar grid of points from two different view points. The grid points are projected as *open circles*. *Plus signs* in **b** indicate points transformed from the camera one image plane by the homography

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 center 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

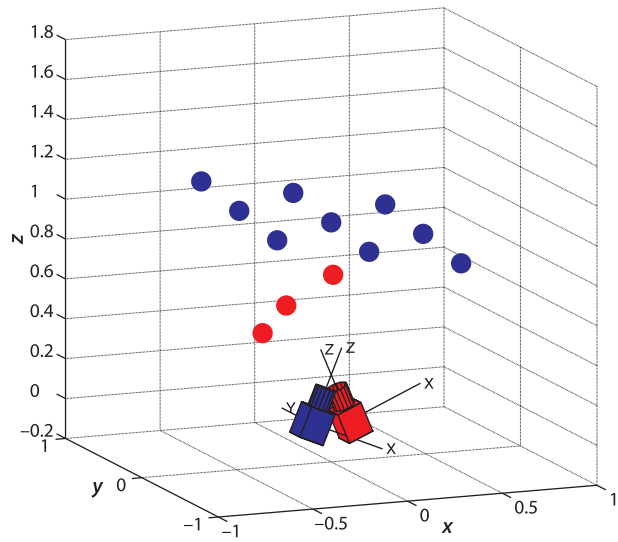


Fig. 14.12. World view of target points and two camera poses. *Blue points* lie in a planar grid, while the *red points* appear to lie in the grid from the viewpoint of camera one

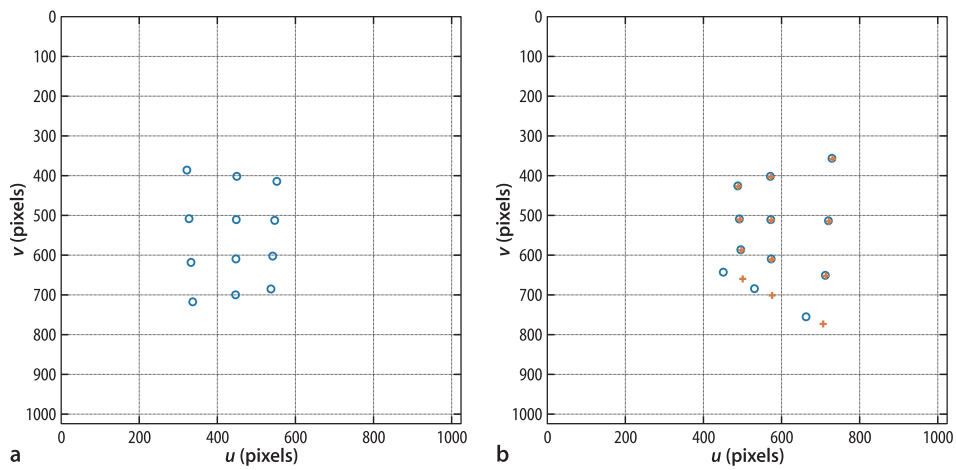


Fig. 14.13. Views of the oblique planar grid of points from two different view points. The grid points are projected as *open circles*. *Plus signs* in **b** indicate points transformed from the camera one image plane by the homography. The bottom of row of points in each case are not coplanar with the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

```
>> 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 the camera two image if they belonged to the plane implicit in the homography

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

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 =
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000   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*, points 10–12, are not on the inlier list.

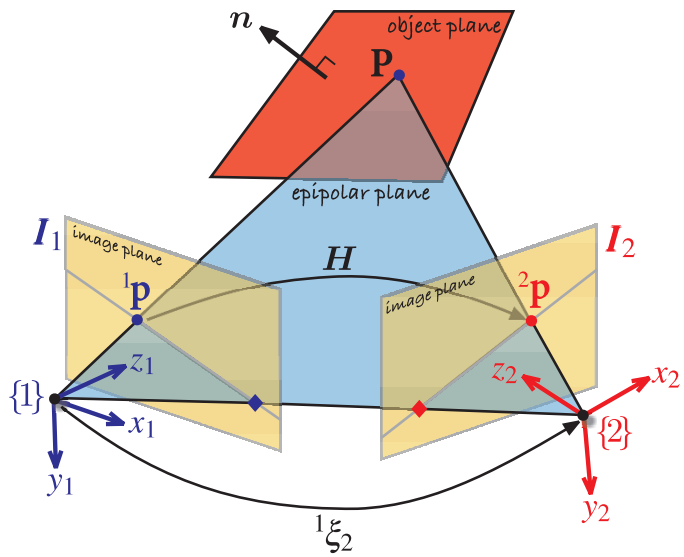


Fig. 14.14.
Geometry of homography showing two cameras with associated coordinate frames {1} and {2} and image planes. The world point **P** belongs to a plane with surface normal $\boldsymbol{n}$. $\boldsymbol{H}$ is the homography, a 3×3 matrix that maps ${}^1\mathbf{p}$ to ${}^2\mathbf{p}$

See Sect. 11.1.2.

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

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

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

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

in terms of the camera 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

$$ \boldsymbol{H}_E \simeq \boldsymbol{K}^{-1}\boldsymbol{H}\boldsymbol{K} $$

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

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

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

```
>> 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
```

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

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

We can apply this technique to a pair of real images

```
>> im1 = iread('walls-l.jpg',  'double', 'reduce', 2);
>> im2 = iread('walls-r.jpg',  'double', 'reduce', 2);
```

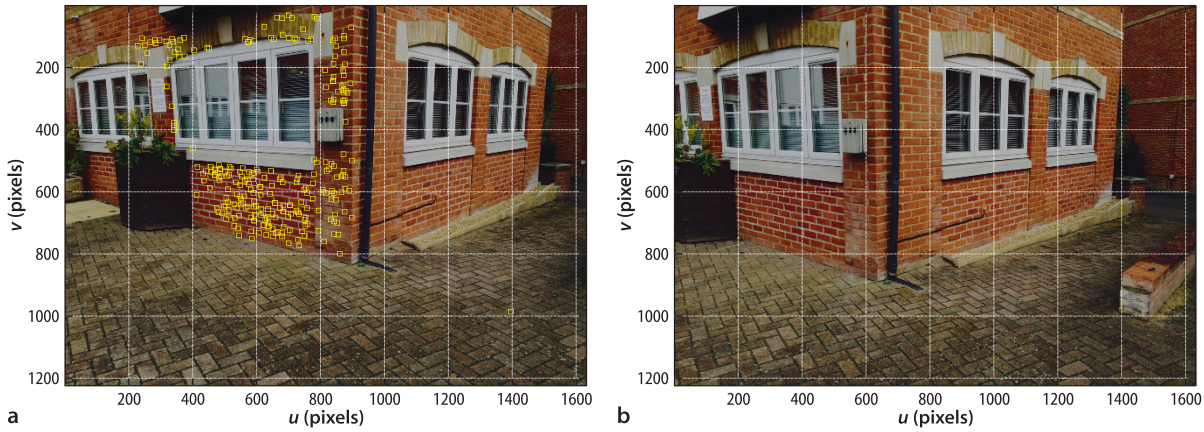


Fig. 14.15. Two pictures of a courtyard taken from different viewpoints. Image **b** was taken approximately 30 cm to the right of image **a**. Image **a** has superimposed features that fit a plane. The camera was handheld

which we have downsized by a factor of 2 in each dimension and are shown in Fig. 14.15. We start by finding the SURF features

```
>> sf1 = isurf(im1);
>> sf2 = isurf(im2);
```

and the candidate corresponding points

```
>> m = sf1.match(sf2, 'top', 1000)
m =
1000 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, 4)
H =
    0.8463    0.0164 -150.4748
    0.0050    1.0067   20.3413
   -0.0000   -0.0000    1.0000
r =
    1.6799
```

The number of inlier and outlier points is

```
>> m.show
ans =
1000 corresponding points
262 inliers (26.2%)
738 outliers (73.8%)
```

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 262 points *do* belong to the plane and we can superimpose them on the figure

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

as shown in Fig. 14.15a. RANSAC has found a consensus which is the plane containing the left-hand wall. The error tolerance was set to 4 pixels to account for lens distortion and the planes being not perfectly smooth. 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 right-hand wall. Planes are very common in man-made environments and we will revisit homographies and their decomposition in Sect. 14.7.1.

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 as for example shown in Fig. 14.15. Human eyes are separated by 50–80 mm and the difference between these two viewpoints is an important, but not the only, part of how we sense distance. 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 attempts to recover 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 images shown in Fig. 14.15. We have already found the SURF features and established candidate correspondences between them. Now we estimate the fundamental matrix

```
>> [F,r] = m.ransac(@fmatrix,1e-4, 'verbose');
102 trials
238 outliers
0.000132333 final residual
```

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

```
>> cam = CentralCamera('image', im1);
>> cam.plot_epiline(F', m.inlier.subset(40).p2, 'y');
```

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, $\mathbf{P}$ to $\mathbf{P}'$ the conjugate points in the right-hand image moves to the right along the epipolar line.

The points ${}^1\mathbf{p}$ and ${}^2\mathbf{p}$ 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 the two views. We have the

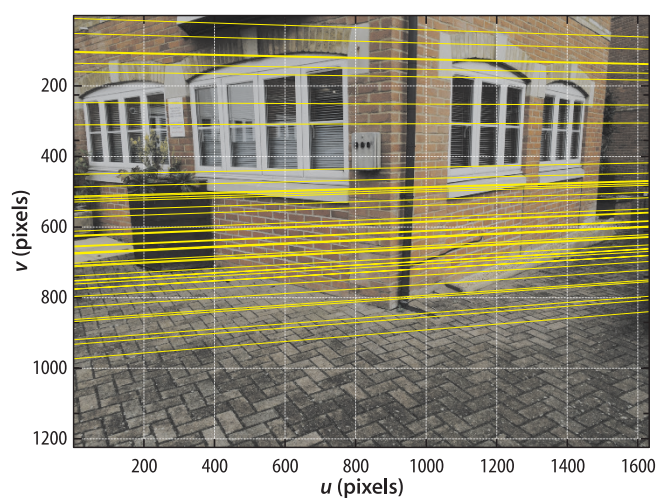


Fig. 14.16. Image of Fig. 14.15a with epipolar lines for a subset of right image points superimposed

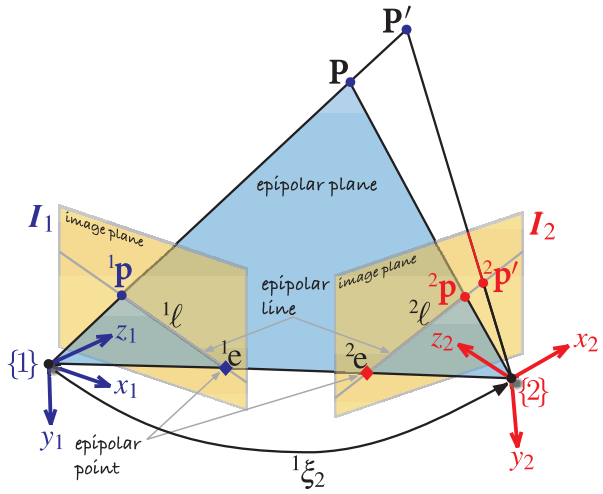


Fig. 14.17.
Epipolar geometry for stereo vision. We can see clearly that as the depth of the world point increases, from **P** to **P**′, the projection moves along the epipolar line in the second image plane

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 metadata of the image as we discussed on page 363 and can be examined

```
>> [~,md] = iread('walls-1.jpg');
```

where `md` is a structure of text strings that contains various characteristics of the image – its metadata. The element `DigitalCamera` is a structure that describes the camera

```
>> f = md.DigitalCamera.FocalLength
f =
    4.1500
```

from which we determine the focal length is 4.15 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

```
>> md.Model
ans =
iPhone 5s
```

suggests that this camera has an image sensor with 1.5 μm pixels. We create a `CentralCamera` object based on the known focal length, pixel size and image dimension▸

We have doubled the pixel dimensions to account for halving the image resolution when we loaded the images. A low-resolution image effectively has larger pixels.

```
>> cam = CentralCamera('image', im1, 'focal', f/1000, ...↵
    'pixel', 2*1.5e-6)
cam =
name: noname [central-perspective]
  focal length:   0.00415
  pixel size:     (3e-06, 3e-06)
  principal pt:   (816, 612)
  number pixels:  1632 x 1224
  pose:           t = (0,0,0), RPY/yxz = (0,0,0) deg
```

In the absence of any other information the principal point is assumed to be in the center of the image.

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

```
>> E = cam.E(F)
E =
    0.0143    1.1448    0.2380
   -1.1286   -0.1483    6.0273
   -0.2826   -6.0536   -0.1461
```

and we then decompose it to determine the camera motion

```
>> T = cam.invE(E, [0,0,10]')
T =
    0.9999    0.0115    0.0027     6.042
   -0.0115    0.9996   -0.0255   -0.3092
   -0.0030    0.0254    0.9997     1.124
         0         0         0         1
```

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

```
>> T.torpy('yxz', 'deg')
ans =
   -0.6569    1.4597    0.1561
```

is less than two degrees of rotation about any axis.◀

We have specified a different roll-pitch-yaw rotation order *YXZ*. Given the way we have defined our axes the camera orientation with respect to the world frame is a yaw about the vertical or *y*-axis, followed by a pitch about the *x*-axis followed by a roll about the optical axis or *z*-axis.

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 magnitude is quite wrong. We therefore scale the translation

```
>> t = T.t;
>> T.t = 0.3 * t/t(1)
T =
    0.9999    0.0115    0.0027       0.3
   -0.0115    0.9996   -0.0255  -0.01535
   -0.0030    0.0254    0.9997    0.0558
         0         0         0         1
```

and we have an estimate of ${}^1\xi_2$ – 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◀

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

Sometimes called a raxel. Many representations exist including Plücker coordinates which are described in Sect. C.1.2.2. Determining a ray from a pixel coordinate is covered on page 327.

where $\boldsymbol{P}_0$ is the principal point of the camera and $\boldsymbol{d} \in \mathbb{R}^3$, $\|\boldsymbol{d}\| = 1$ is a unit-vector in the direction of the ray. Consider now the first corresponding point pair `m(1)`. The ray from camera one is

```
>> r1 = cam.ray(m(1).p1)
r1 =
d=(0.37844, -0.0819363, 0.921992), P0=(0, 0, 0)
```

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

```
>> r2 = cam.move(T).ray(m(1).p2)
r2 =
d=(0.29936, -0.0826926, 0.95055), P0=(0.3, -0.0153494, 0.0557958)
```

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

```
>> [P,e] = r1.intersect(r2);
P'
ans =
    1.2134   -0.2627    2.9563
```

which is a point with a *z*-coordinate, or depth, of almost 3 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
```

nearly 5 mm apart. Considering the lack of rigor 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 meters. The closing error observed here would be induced by a rotational error of less than 1 deg.

We draw a subset of one hundred corresponding points from the inlier set

```
>> m2 = m.inlier.subset(100);
```

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 courtyard

```
>> idisp(im1)
>> plot_point(m2.p1, 'y+', 'textcolor', 'y', 'printf', {'%.1f', z});
```

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

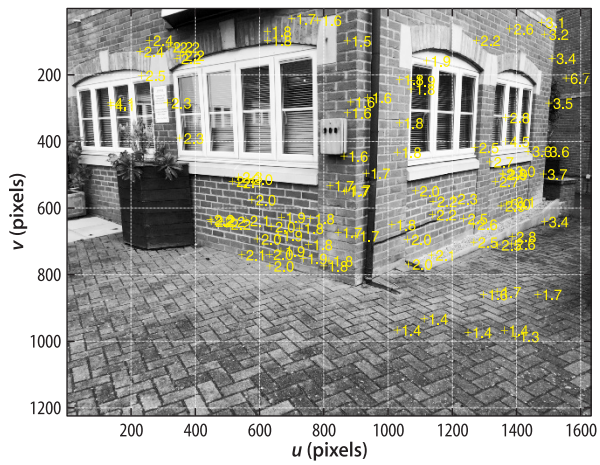


Fig. 14.18.
Image of Fig. 14.15a with depth of selected points indicated (units of meters)

Fig. 14.19.
A small stereo camera sensor mounted on a mobile robot and capable of real-time depth map generation (image courtesy of Stereolabs 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 more commonly taken simultaneously 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 the left and right images comprising a 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 relative to 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 point on the digit 5 written on one of the foreground rocks and we observe several things. Firstly the spot has the same vertical coordinate in both images, and this implies that the epipolar lines are horizontal. Secondly, in the right-hand image the spot has moved to the left by 70.9 pixels. If we probed more points we would see that disparity decreases for points that are further from the camera.

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 $({}^{L}u, {}^{L}v)$ and $({}^{R}u, {}^{R}v)$ then ${}^{R}v = {}^{L}v$. The displacement along the horizontal epipolar line $d = {}^{L}u - {}^{R}u$ where $d \geq 0$ is called disparity.

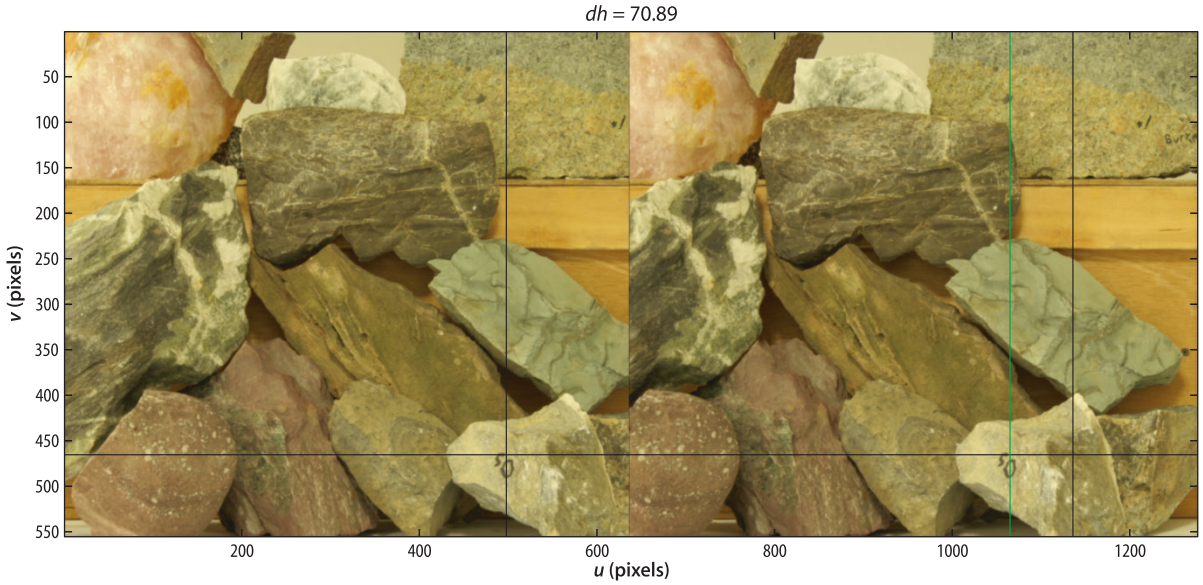


Fig. 14.20. The `stdisp` image browsing window. The *black cross hair* in the left-hand image has been positioned at the top right of the digit 5 on a foreground rock. Another *black cross hair* is automatically positioned at the same coordinate in the right-hand image. Clicking on the corresponding point in the right-hand image sets the *green cross-hair*, and the *panel* at the top indicates a horizontal shift of 70.9 pixels to the left. This stereo image pair is from the Middlebury stereo database (Scharstein and Pal 2007). The focal length f/ρ is 3 740 pixels, and the baseline is 160 mm. The images have been cropped so that the actual disparity should be offset by 274 pixels

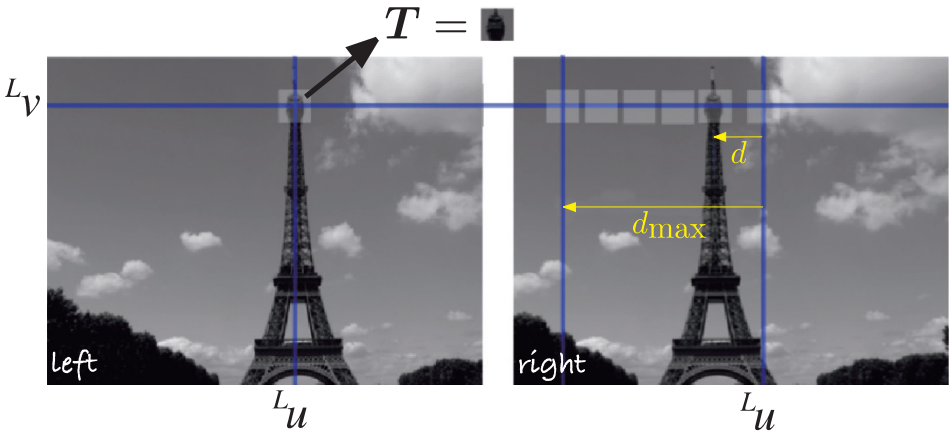


Fig. 14.21.
Stereo matching. A search window in the right image, starting at $u = {}^{L}u$, is moved leftward along the epipolar line $v = {}^{L}v$ until it matches the *template* window $\boldsymbol{T}$ from the left image

The dense stereo process is illustrated in Fig. 14.21. For the pixel at $({}^{L}u, {}^{L}v)$ in the left-hand image we know that its corresponding pixel is at some coordinate $({}^{L}u - d, {}^{L}v)$ 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 an $N \times N$ 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 the matching problem we discussed in Sect. 12.5.2, 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. 12.5.2, and for an $N \times N$ template these have a computational cost of $O(N^2)$. For a $W \times H$ image the total cost of dense stereo matching is $O(DWHN^2)$ 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 an image – 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` 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.

▸ 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.

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 some 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 red.

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

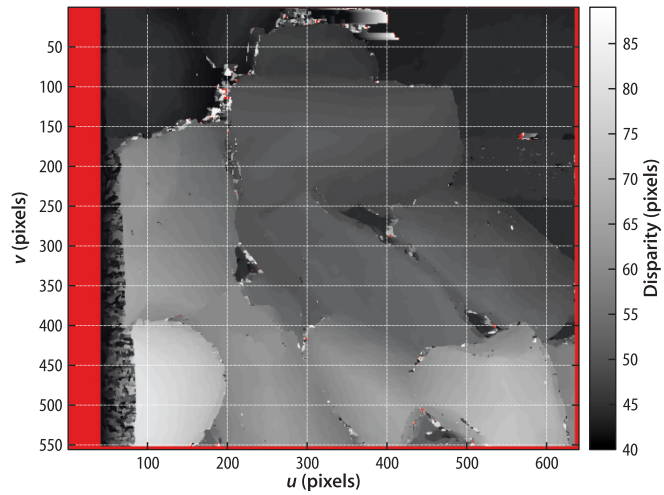


Fig. 14.22. Disparity image for the rock pile stereo pair, where *brighter* means higher disparity or shorter range. *Red* indicates `Inf` or `NaN` values in the result where disparity could not be computed. Note the quantization in grey levels since we search for disparity in steps of one pixel

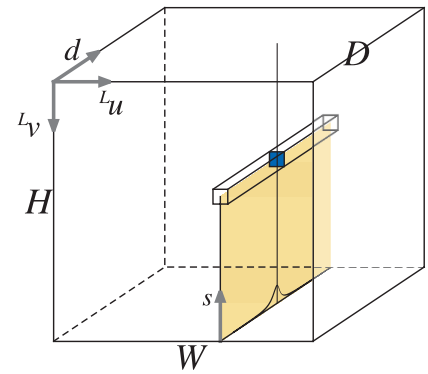


Fig. 14.23. The disparity space image (DSI) is a 3-dimensional image where element $D(u, v, d)$ is the similarity between the support regions centered at $({}^{L}u, {}^{L}v)$ in the left image and $({}^{L}u - d, {}^{L}v)$ in the right image

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);
```

where `sim` is an $H \times W$ matrix whose elements are the peak similarity score at the corresponding pixel and `DSI` is an $H \times W \times D$ matrix shown in Fig. 14.23

```
>> 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 (u, v, d) are the similarity measure between the templates centered 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 workable but 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, as shown in Fig. 14.23, holds 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 an almost perfect match occurs at a disparity of 80 pixels, since the horizontal axis is $d - d_{\min}$ and $d_{\min} = 40$. Such a strong and unambiguous peak is fortunately very common. However Fig. 14.22 shows that the stereo matching process is not perfect and plots of the template similarity metric versus disparity provide insight into the causes of error.

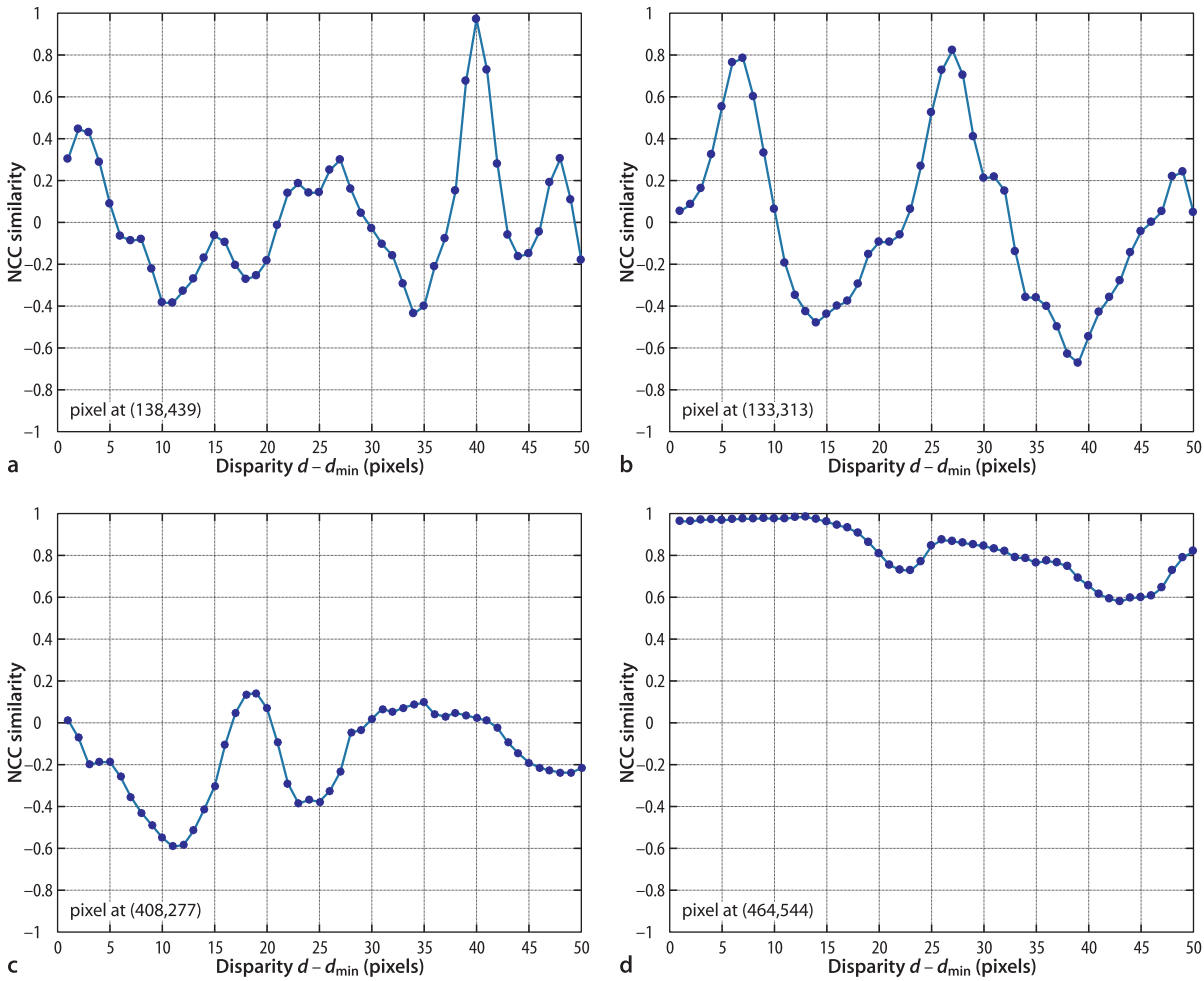


Fig. 14.24. Some typical ZNCC metric versus disparity curves. **a** Single strong peak; **b** multiple peaks; **c** weak peak; **d** broad peak

Figure 14.24b shows two peaks of almost similar amplitude and this means that the template pattern was found twice in the search region. This 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 effect 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 but this requires some knowledge of the expected range of objects.

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

A weak match 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 – also known as 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. The stereo matching algorithm will always return the best match so if the point is occluded it will return disparity to the most similar, but wrong, template. Even though the figure is an exaggerated depiction, real images suffer this problem where the depth changes rapidly. In our 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.

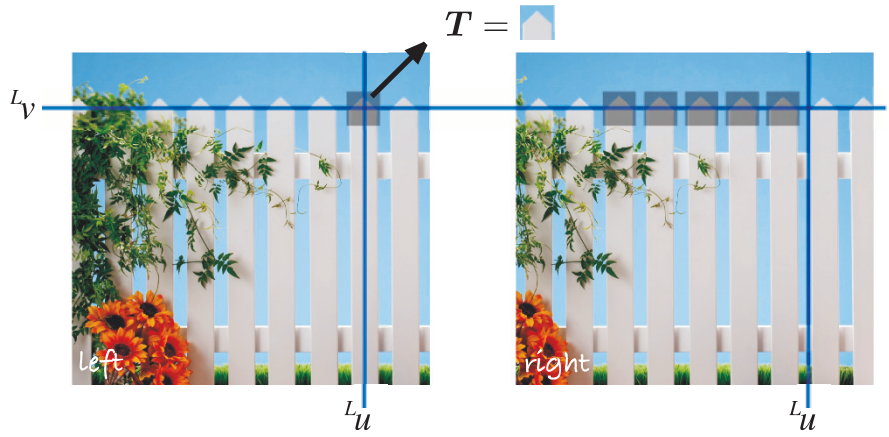


Fig. 14.25. Picket fence effect. The template will match well at a number of different disparities. This problem occurs in any scene with repeating patterns

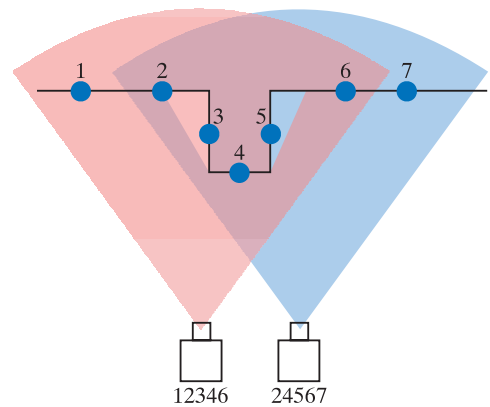


Fig. 14.26. Occlusion in stereo vision. The field of view of the two cameras are shown as *colored sectors*. *Points 1* and *7* fall outside the overlapping view area and are seen by only one camera each. *Point 5* is occluded from the left camera and *point 3* is occluded from the right camera. The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

The problem cannot be cured but it can be detected. The simplest method is to consider the similarity score returned by the `istereo` function

```
>> idisp(sim)
```

as shown in Fig. 14.27a and we see that the erroneous disparity values correspond to low similarity scores. Disparity results where similarity is low can be discarded

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

and this is shown in Fig. 14.27b where pixels with similarity $s < 0.7$ are displayed as yellow. 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 0.6, and that around 80% of pixels have a similarity score greater than 0.9.

A simple but effective way to test for occlusion is to perform the matching in two directions – left-right consistency checking. 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 where we started the match is considered valid. However if the corresponding point was occluded in the right image the first match will be a weak one to a different feature, and there is a high probability that the second match will be to a different pixel in the left image.

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 for example 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 (90 in this case) of the disparity image.

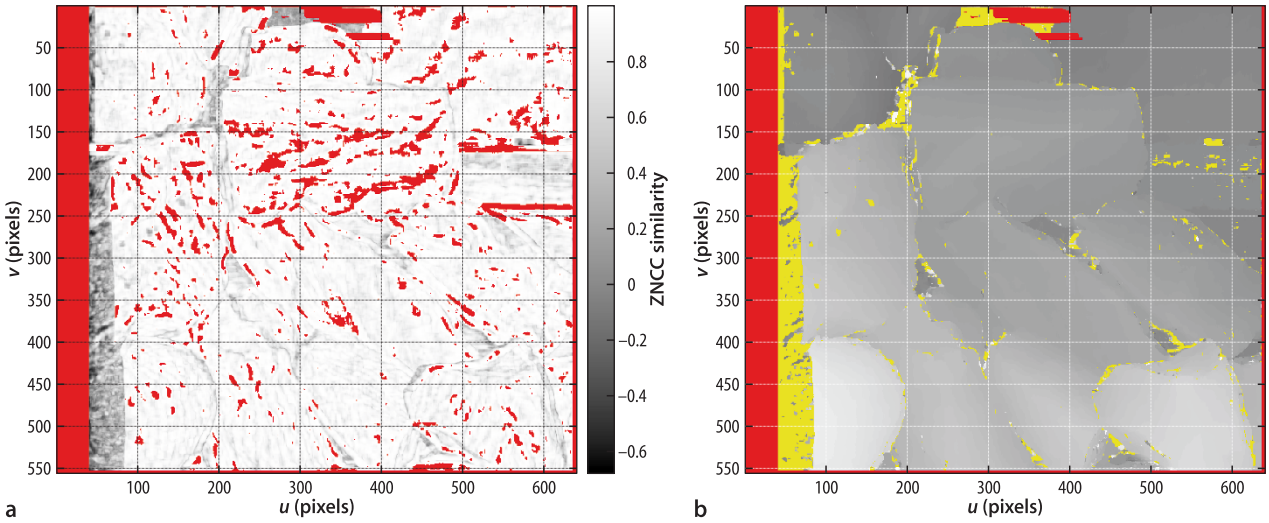


Fig. 14.27. Stereo template similarity. **a** Similarity image where *brighter* means higher similarity; **b** disparity image with pixels having low similarity score marked in *yellow*. *Red* indicates `Inf` or `NaN` values in the result where disparity could not be computed

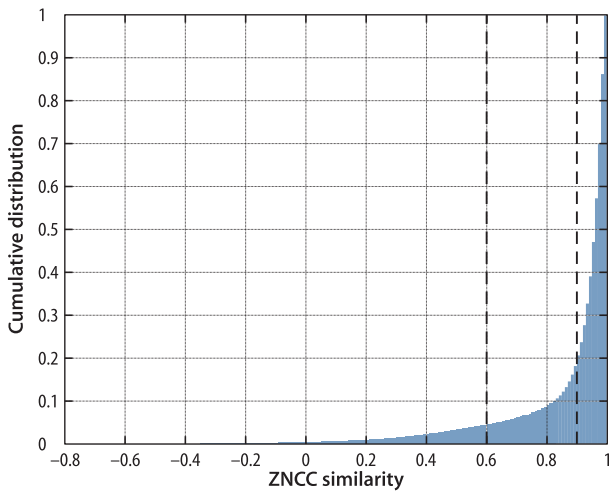


Fig. 14.28. Cumulative probability of ZNCC scores. The probability of a score less than 0.9 is 45%

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 result in 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 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. Second, 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

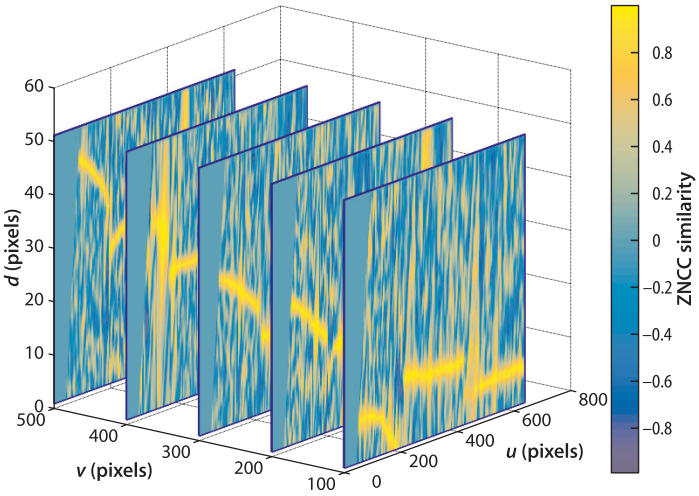


Fig. 14.29. The disparity space image is a 3-dimensional image where element $D(u, v, d)$ is the similarity between the support regions centered at $({}^{L}u, {}^{L}v)$ in the left image and $({}^{L}u - d, {}^{L}v)$ in the right image

points close to the camera will generate incorrect and weak matches. A large disparity range also increases the computation time. Third, 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 requires greater computation. It also increases the chance that the template will contain pixels belonging 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. One solution is to use a nonparametric local transform such as the rank or census transform prior to performing correlation. Since these rely on the ordering of intensity values not the values themselves they give better performance at object boundaries.

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], [])
>> shading interp; colorbar
```

which is shown in Fig. 14.29. These are slices at constant v-coordinate, effectively horizontal cross sections of the scene. Within each of the ud-planes we see a bright path (high similarity values) that represents 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 greatest similarity was found. Figure 14.24a shows a single unambiguous strong peak and we can use the peak and adjacent points to refine the estimate of the peak's position.◂ A parabola

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

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

is defined by three points and is fitted to the peak value and its two neighbors. For the ZNCC similarity measure, a maxima corresponds to the best match which means that the parabola is inverted and $A < 0$. The maximum value of the fitted parabola occurs

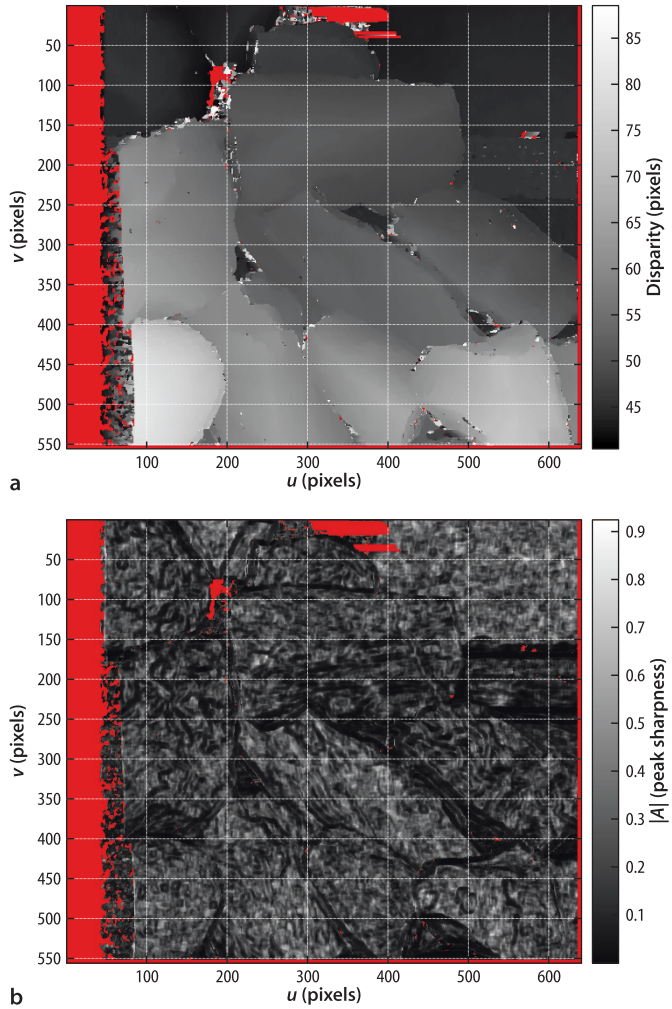


Fig. 14.30.
a Disparity image with peak refinement; **b** magnitude of the d^2 coefficient for every pixel. High values (*bright*) correspond to sharp peaks and occur where image texture is high. Broad peaks (*dark*) occur where image texture is low

when its derivative equals zero, from which we can obtain a more precise estimate of the position of the peak which is the disparity

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

The A coefficient will have a large magnitude 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 magnitude of the A coefficient is shown as an image in Fig. 14.30b.

14.3.4 Cleaning up and Reconstruction

The result of stereo matching, such as shown in Fig. 14.22 or 14.30a, have 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 one

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

The elements are set to different values if they correspond to specific failure conditions

```
>> [U,V] = imeshgrid(L);
>> status(isnan(d)) = 5;      % search template off the edge
>> status(U<=90) = 2;         % no overlap
>> status(sim<0.8) = 3;       % weak match
>> status(peak.A>=-0.1) = 4; % broad peak
```

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(:) == 1) / prod(size(status)) * 100
ans =
   57.7223
```

nearly 60% 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 nonrock 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>1) = NaN;
```

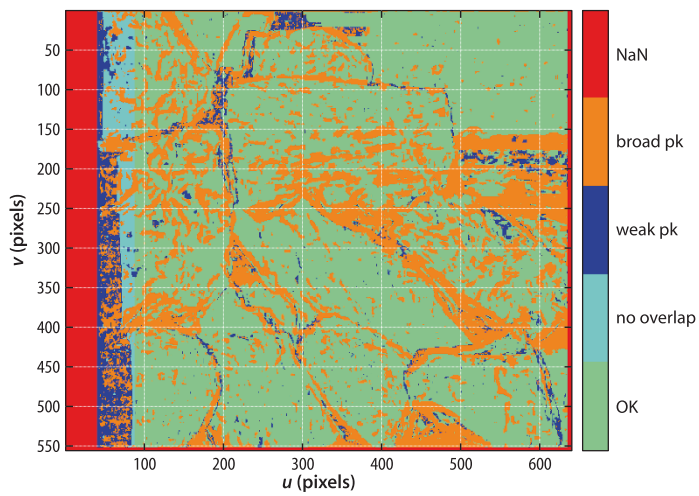


Fig. 14.31. Stereo matching status on a per pixel basis

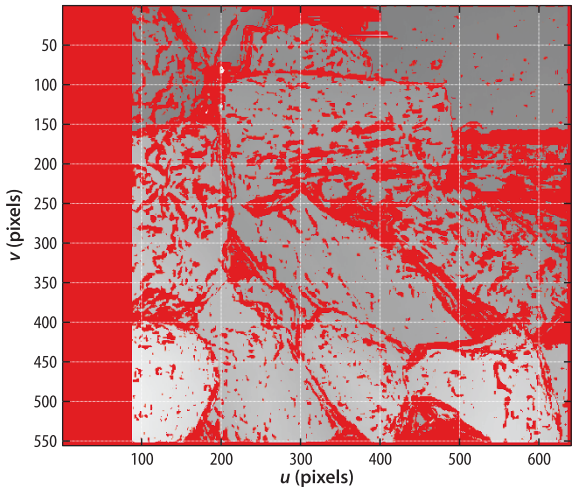


Fig. 14.32.
Interpolated disparity image with unreliable estimates indicated in *red*

by setting them to the value `NaN`.▸ 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.◂ 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 meters – 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 ${}^{i}u$, $i = \{L, R\}$

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

Substituting and eliminating X gives

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

which shows that depth is inversely proportional to disparity $d = {}^{L}u - {}^{R}u$ and $d > 0$. We can also recover the X- and Y-coordinates so the 3D point coordinate is

$$X = \frac{b\left({}^{L}u - u_0\right)}{d}, \quad Y = \frac{b\left({}^{L}v - v_0\right)}{d}, \quad Z = \frac{fb}{\rho_u d} \tag{14.16}$$

A good stereo system can estimate disparity with an accuracy of 0.2 pixels. Distant points have a small disparity and the error in the estimated 3D coordinate will be significant. A rule of thumb is that stereo systems typically have a maximum range of `50b`.

▸ 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.

◂ 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].

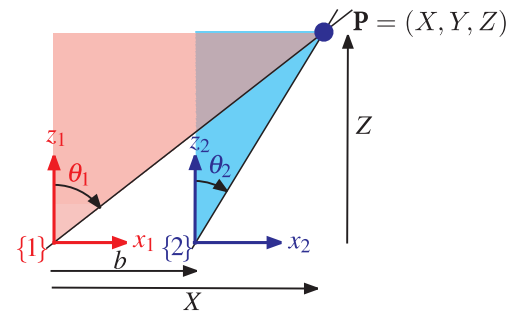


Fig. 14.33. Stereo geometry for parallel camera axes. X and Z are measured with respect to camera one, b is the baseline

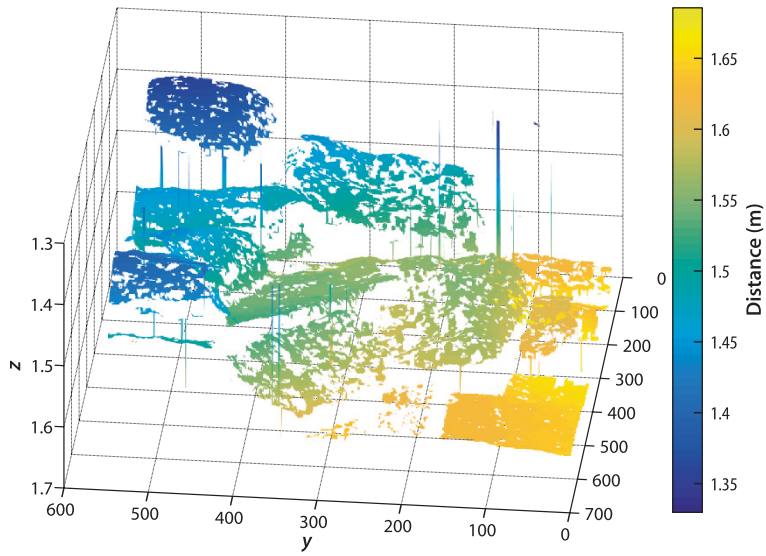


Fig. 14.34. 3-dimensional reconstruction for parallel stereo cameras. *Hotter colors* indicate parts of the surface that are further from the camera

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 only, 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));
```

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 we see that it 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')
```

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.

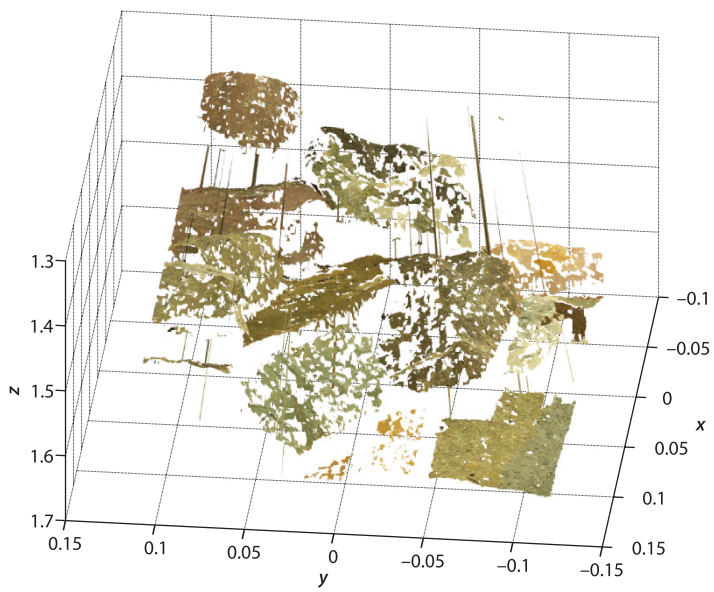
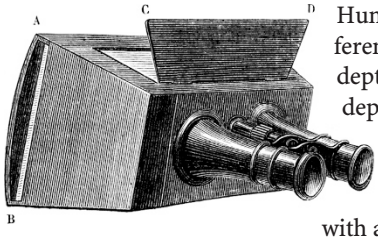


Fig. 14.35.
3-dimensional reconstruction for parallel stereo cameras with image texture mapped onto the surface

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 current popularity of 3D movies and availability of 3D television is further evidence of this.

The key in all 3D display technologies is 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 (gaming) glasses or polarized glasses which allow full-color stereo movie viewing, or head mounted displays.

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.

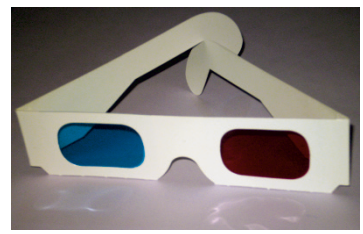
The rock pile stereo pair can be displayed as an anaglyph

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

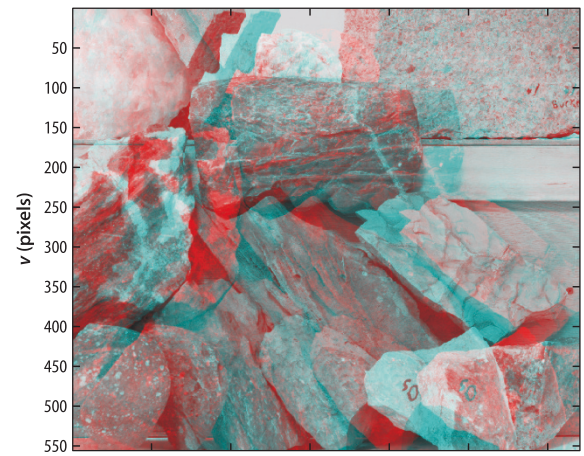
which is shown in Fig. 14.36b. The argument `'rc'` indicates that left and right images are encoded in red and cyan respectively. Other color options include: blue, green, magenta and orange.

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.

Today high-resolution panoramic anaglyphs can be found on http://gigapan.com and anaglyphs of Mars can be found at http://mars.nasa.gov/mars3d.



a



b

Fig. 14.36. Anaglyphs for stereo viewing. **a** Anaglyph glasses shown with red and blue lenses, **b** anaglyph rendering of the rock scene from Fig. 14.20 with left in *red* and right in *cyan*

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 – they are an epipolar-aligned image pair. Stereo cameras, such as shown in Fig. 14.19, 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 courtyard stereo pair from Fig. 14.15

```
>> L = iread('walls-l.jpg', 'mono', 'double', 'reduce', 2);
>> R = iread('walls-r.jpg', 'mono', 'double', 'reduce', 2);
```

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, 'top', 1000);
```

then determine the epipolar relationship

```
>> F = m.ransac(@fmatrix,1e-4, 'verbose');
96 trials
309 outliers
0.000305205 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 red.

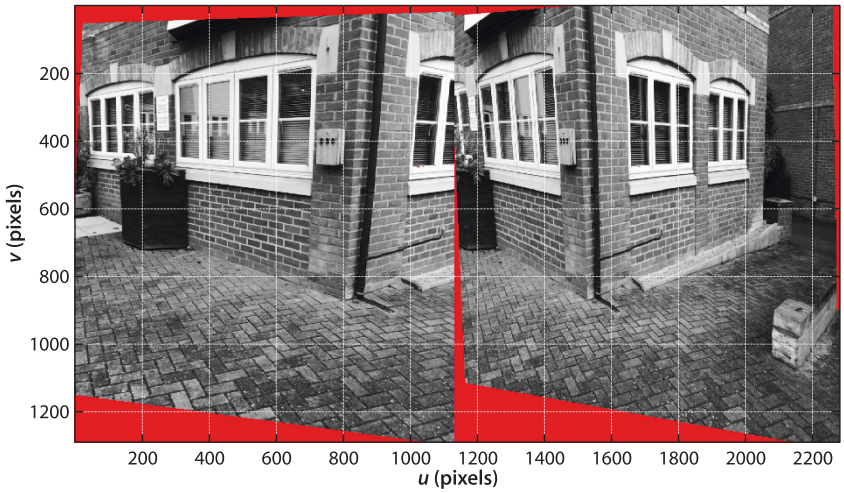


Fig. 14.37.
Rectified images of the courtyard. *Red pixels* have no correspondance in the other image

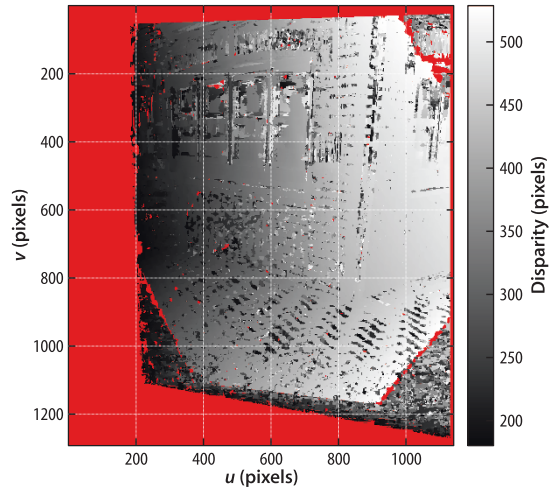


Fig. 14.38. Dense stereo disparity image for the courtyard. The walls and ground show a clear depth gradient

We can think of these images as having come from a virtual stereo camera with parallel axes and aligned pixel rows, and they can now be used for dense stereo matching

```
>> d = istereo(Lr, Rr, [180 530], 7, 'interp');
```

and the result is shown in Fig. 14.38. The disparity range parameters were determined interactively using `stdisp(Lr, Rr)` to check the disparity at near and far points in the rectified image pair. The window half size of 7 was arrived at with a little trial and error, this value corresponding to a 15×15 window and produces a reasonably smooth result, at the expense of computation. The noisy patches at the bottom and top right are due to occlusion – world points in one image are not visible in the other. Nevertheless this is quite an impressive result – using only two images taken from a handheld camera we have been able to create a dense 3-dimensional representation of the scene.

14.4 Bundle Adjustment

In Sect. 14.3.1 we used triangulation to estimate the 3D coordinates of a number of landmark points in the world, but this was an approximation based on a guesstimate of the relative pose between the cameras. To assess the quality of our solution we can "back project" the estimated 3D landmark points onto the image planes based on the estimated camera poses◀ and the known camera model. The back-projection error is the image-plane distance between the back-projected landmark and its observed position on the image plane.

In this case we only estimated the relative pose ${}^1\xi_2$ but we can consider the first camera pose as the reference coordinate frame $\xi_1 = 0$ and $\xi_2 = {}^1\xi_2$.

For the previous example the back projection is

```
>> p1 = cam.project(P);
>> p2 = cam.move(T).project(P);
```

for the first and second camera respectively. The distances between the back projections and observations across both cameras is

```
>> e = colnorm( [p1-m2.p1 p2-m2.p2] );
```

with statistics

```
>> mean(e)
ans =
    0.9942
>> max(e)
ans =
    6.6765
```

which clearly indicates the approximate nature of our solution – each back-projected point is in error by up to 7 pixels. Unfortunately we do not know whether the error is in the estimated camera poses, the landmark coordinates or both. However we do know that a good estimate is one where this total back-projection error is low – ideally zero.

Bundle adjustment is an optimization process that simultaneously adjusts the camera poses and the landmark coordinates so as to minimize the total back-projection error. It uses 2D measurements from a set of images of the same scene to recover information related to the 3D geometry of the imaged scene as well as the locations and optical characteristics of the cameras. This is also called Structure from Motion (SfM) or Structure and Motion Estimation (SaM) – *structure* being the 3D landmarks in the world and *motion* being a sequence of camera poses. It is also called visual SLAM (VSLAM) since it is very similar to the pose-graph SLAM problem discussed in Sect. 6.6. That was a planar problem solved in the three dimensions of $\mathbf{SE}(2)$ whereas bundle adjustment involves camera poses in $\mathbf{SE}(3)$ and points in $\mathbb{R}^3$.

To formalize the problem consider a camera with known intrinsic parameters at N different poses $\xi_i \in \mathbf{SE}(3)$, $i = 1 \cdots N$ and a set of M landmark points $\boldsymbol{P}_j \in \mathbb{R}^3$, $j = 1 \cdots M$. At pose $\{i\}$ the camera observes $\boldsymbol{P}_j$ and the measured image-plane projection is ${}^i\boldsymbol{p}_j^{\#} \in \mathbb{R}^2$, but not all landmarks are necessarily visible from each camera pose. The notation is shown in Fig. 14.39 for two cameras.

The estimated value of the image-plane projection of the j^{th} landmark in the i^{th} image plane is

$$ {}^i\hat{\boldsymbol{p}}_j = \mathcal{P}\left(\hat{\xi}_i, \hat{\boldsymbol{P}}_j; \boldsymbol{K}\right) $$

and the back-projection error is ${}^i\hat{\boldsymbol{p}}_j - {}^i\boldsymbol{p}_j^{\#}$.

Using the Toolbox we start by creating a `BundleAdjust` object

```
>> ba = BundleAdjust(cam);
```

which is passed an intrinsic model of the camera which we assume is known. Next we add estimates of the two camera poses

```
>> c1 = ba.add_camera( SE3(), 'fixed' );
>> c2 = ba.add_camera( T );
```

and indicate that the first camera pose is known and that we do not wish to optimize for it. The second camera's estimated pose is that derived earlier from the essential matrix. The method returns an integer handle to the particular camera pose which we will use below.

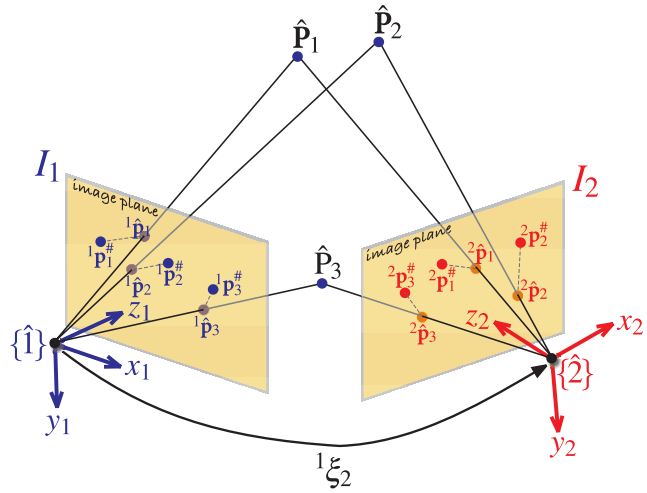


Fig. 14.39.
Bundle adjustment notation illustrated with a simple problem comprising only two cameras and three world points. The estimated camera poses and point positions are indicated, as are the estimated and measured image-plane coordinates. The reprojection errors are shown as *dashed grey lines*. The problem is solved when the variables are adjusted so that the total reprojection error is as small as possible

Next we add the estimated landmarks

```
>> for j=1:length(m2)
     lm = ba.add_landmark( P(:,j) );

     ba.add_projection(c1, lm, m2(j).p1);
     ba.add_projection(c2, lm, m2(j).p2);
   end
```

where `lm` is another integer handle, in this case to the particular landmark coordinate. Finally, we add the measurements by specifying the camera, the landmark and its projection on the image plane. The problem is now fully defined and a summary can be displayed

```
>> ba
ba =
Bundle adjustment problem:
  2 cameras
    locked cameras: 1
  100 landmarks
  200 projections
  306 dimension linear problem
  landmarks per camera: min=100.0, max=100.0, avg=100.0
  cameras per landmark: min=2.0, max=2.0, avg=2.0
```

In general only a subset of landmarks are visible from any camera, and this visibility information can be represented elegantly using a graph as shown in Fig. 14.40 where each camera pose and each landmark coordinate is a node. Edges between camera and landmark nodes represent observations, and the value of the edge is the observed image-plane coordinate. Such a graph, a Toolbox `PGraph` object, is held inside the `BundleAdjust` object and can be plotted by

```
>> ba.plot
```

and an example is shown in Fig. 14.41.

To solve this optimization problem we put all the variables we wish to adjust into a single state vector. For bundle adjustment the state vector contains camera poses and landmark coordinates

$$\boldsymbol{x} = \left\{ \xi_1, \xi_2 \cdots \xi_N \middle| \boldsymbol{P}_1, \boldsymbol{P}_2 \cdots \boldsymbol{P}_M \right\} \in \mathbb{R}^{6N+3M}$$

where the **SE**(3) camera pose is represented in a vector format $\xi_i \sim (\boldsymbol{t}, \boldsymbol{r}) \in \mathbb{R}^6$ comprising translation $\boldsymbol{t} \in \mathbb{R}^3$ and rotation $\boldsymbol{r} \in \mathbb{R}^3$; and $\boldsymbol{P}_j \in \mathbb{R}^3$.

Possible representations of rotation include Euler angles, roll-pitch-yaw angles, angle-axis or exponential coordinate representations. For bundle adjustment it is common to use the vector component of a unit-quaternion which is singularity free and has only three parameters. The double cover property of unit-quaternions means that any unit-quaternion can be written with a nonnegative scalar component. By definition

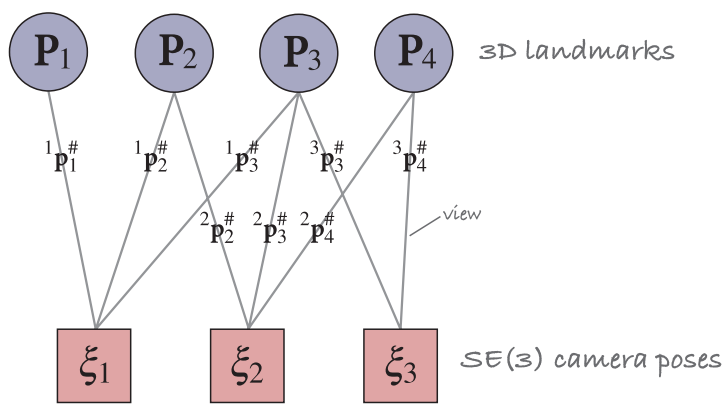


Fig. 14.40. A simple visibility graph showing camera nodes (*red*) and landmark nodes (*blue*). Lines connecting nodes represent a view of that node on that camera, and the edge value is the observed image-plane coordinate. The landmarks here are viewed by 1, 2 or 3 cameras

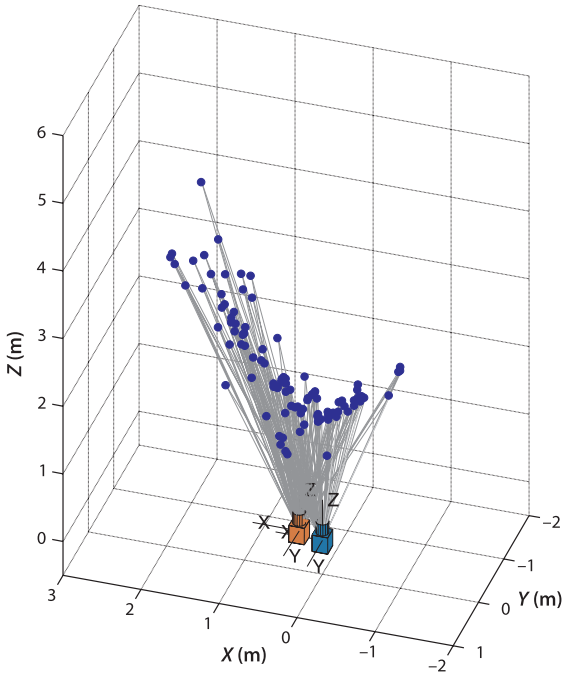


Fig. 14.41.
Bundle adjustment problem shown as an embedded graph. *Blue dots* represent landmark positions, *camera icons* represent camera pose, and *grey lines* denote observations. Camera 1 is *blue* and camera 2 is *red*

the unit-quaternion has a unit norm, so the scalar component can be easily recovered $s = \sqrt{1 - v_x^2 - v_y^2 - v_z^2}$ given the vector component.

The number of unknowns in this system is $6N + 3M$: 6 unknowns for each camera pose and 3 unknowns for the position of each landmark point. However we have up to $2NM$ equations due to the measured projections of the points on the image planes. Typically the pose of one camera is assumed to be the reference coordinate frame, and this reduces the number of unknowns to $6(N-1) + 3M$.

In the problem we are discussing $N = 2$, but one camera is locked, and $M = 100$ so we have $6 \times (2-1) + 3 \times 100 = 306$ unknowns and $2 \times 2 \times 100 = 400$ equations – an overdetermined set of equations for which a solution should be possible. For our problem we can extract the state vector

```
>> x = ba.getstate;
>> about x
x [double] : 1x312 (2.5 kB)
```

which includes the pose of the fixed camera, although that will remain constant. The pose of camera two is stored in the second block of 6 elements

```
>> x(7:12)
ans =
    0.3000   -0.0153    0.0558    0.0127    0.0014   -0.0057
```

as translation followed by rotation, and the first landmark is stored in

```
>> x(13:15)
ans =
    1.2134   -0.2627    2.9563
```

Bundle adjustment is a minimization problem that finds the camera poses and landmark positions that minimize the reprojection error across all the edges

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \sum_k F_k(\boldsymbol{x})$$

where $F_k(\cdot) > 0$ is a nonnegative scalar cost associated with the graph edge k from camera i to landmark j. The reprojection error of a landmark at $\boldsymbol{P}_j$ onto the camera at pose ξ_i is

$$\boldsymbol{f}_k(\boldsymbol{x}) = \mathcal{P}\left(\hat{\xi}_i, \hat{\boldsymbol{P}}_j; \boldsymbol{K}\right) - {}^i\boldsymbol{p}_j^{\#} \in \mathbb{R}^2$$

and the scalar cost is the squared Euclidean reprojection error

$$F_k(\boldsymbol{x}) = \boldsymbol{f}_k(\boldsymbol{x})^T \boldsymbol{f}_k(\boldsymbol{x})$$

Although written as a function of the entire state vector $F_k(\cdot)$ only depends on two elements of that vector: ξ_i and P_j. The total error, the sum of the squared back-projection error for all edges, can be computed for any value of the state vector and for the initial conditions is

```
>> ba.errors(x)
ans =
  553.2853
```

The bundle adjustment task is to adjust the camera and landmark parameters to reduce this value. We have framed bundle adjustment as a sparse nonlinear least squares problem and this can be solved numerically if we have a sufficiently good initial estimate of $\boldsymbol{x}$.

The first step in solving this class of problem is to linearize it. The reprojection error $\boldsymbol{f}_k(\boldsymbol{x})$ can be linearized about the current state $\boldsymbol{x}_0$ of the system

$$\boldsymbol{f}_k'(\boldsymbol{\Delta}) \approx \boldsymbol{f}_{0,k} + \boldsymbol{J}_k \boldsymbol{\Delta}$$

where $\boldsymbol{f}_{0,k} = \boldsymbol{f}_k(\boldsymbol{x}_0)$ and

$$\boldsymbol{J}_k = \frac{\partial \boldsymbol{f}_k(\boldsymbol{x})}{\partial \boldsymbol{x}} \in \mathbb{R}^{2\times(6N+3M)}$$

is a Jacobian matrix which depends only on the camera pose ξ_i and the landmark position $\boldsymbol{P}_j$ so is therefore mostly zeros

$$\boldsymbol{J}_k = \left(0 \cdots \boldsymbol{A}_i \cdots \boldsymbol{B}_j \cdots 0\right), \quad \text{where } \boldsymbol{A}_i = \frac{\partial \boldsymbol{f}_k(\boldsymbol{x})}{\partial \xi_i} \in \mathbb{R}^{2\times 6},\ \boldsymbol{B}_j = \frac{\partial \boldsymbol{f}_k(\boldsymbol{x})}{\partial \boldsymbol{P}_j} \in \mathbb{R}^{2\times 3}$$

The structure of the Jacobian matrix $\boldsymbol{A}_i$ is specific to the chosen representation of camera pose. The Jacobians, particularly $\boldsymbol{A}_i$, are quite complex to derive but can be automatically generated using the MATLAB Symbolic Math Toolbox™ and the script `vision/symbolic/bundleAdjust`. Code derived from this is implemented by the `derivs` method of the `CentralCamera` class

```
[p,A,B] = cam.derivs(t, r, P);
```

which returns the image-plane projection and the two Jacobians in a single call, and where `t` and `r` are the camera pose and `P` is the landmark coordinate. Linearization and Jacobians are discussed in Appendix E, and solution of sparse nonlinear equations in Appendix F.

Translating the camera by $\boldsymbol{d}$ and translating the point by $-\boldsymbol{d}$ have an equivalent effect on the image. Therefore $\boldsymbol{B}_j$ is the negative of the first three columns of $\boldsymbol{A}_j$.

Now that everything is in place we can solve our bundle adjustment problem

```
>> baf = ba.optimize(x);
Initial cost 553.285
  total cost 33.5955 (solved in 0.15 sec)
  total cost 33.5459 (solved in 0.051 sec)
  total cost 33.5459 (solved in 0.04 sec)
  total cost 33.5459 (solved in 0.038 sec)
  total cost 33.5459 (solved in 0.041 sec)
  total cost 33.5459 (solved in 0.037 sec)
 * 6 iterations in 0.5 seconds
 * 0.41 pixels RMS error
```

and the displayed text shows how the total cost (squared reprojection error) decreases at each iteration, reducing by over an order of magnitude. The final result has an RMS reprojection error better than half a pixel for each landmark which is impressive given

that the images were captured with a phone camera and we have completely ignored lens distortion.

The result is another `BundleAdjust` object but with updated camera poses and landmark positions. We can compare the initial and final pose of camera 2

```
>> ba.getcamera(2).print('camera')
t = (0.3, -0.0153, 0.0558), RPY/yxz = (-0.657, 1.46, 0.156) deg
>> baf.getcamera(2).print('camera')
t = (0.303, -0.0158, 0.0649), RPY/yxz = (-0.685, 1.38, 0.128) deg
```

and the final coordinate of landmark 5 is

```
>> baf.getlandmark(5)'
ans =
   -0.3861   -0.0968    2.0744
```

We can also plot the result graphically

```
>> ba.plot()
```

and this is shown in Fig. 14.41. While the overall RMS error is low we can look at the final reprojection error in more detail

```
>> e = sqrt( baf.getresidual() );
>> about e
e [double] : 2x100 (1.6 kB)
```

where element (i, j) is the reprojection error in pixels for camera i and landmark j. The median error is

```
>> median( e(:) )
ans =
    0.2540
```

around a quarter of a pixel, but there are a handful of landmarks with a final reprojection error in camera one that are greater than 1 pixel

```
>> find( e(1,:) > 1 )
ans =
    90    97
```

and the worst error for camera 1

```
>> [mx,k] = max( e(1,:) )
mx =
    1.2129
k =
    90
```

of 1.2 pixels occurs for landmark 90.

Bundle adjustment finds the optimal *relative* pose and positions – not absolute. For example if all the cameras and landmarks moved 1 m in the x-direction the total reprojection error would be the same. To remedy this we can fix or *anchor* one or more cameras or landmarks – in this example we fixed the first camera. The values of the fixed poses and positions are kept in the state vector but they are not updated during the iterations – their Jacobians do not need to be computed and the Hessian matrix used to solve the update at each iteration is smaller since the rows and columns corresponding to those fixed parameters can be deleted.

The fundamental issue of scale ambiguity with monocular cameras has been mentioned a number of times and it applies here as well. A scale model of the same world with a similarly scaled camera translation is indistinguishable from the real thing. More formally, if the whole problem was scaled so that $\boldsymbol{P}_j' = \lambda\, \boldsymbol{P}_j$, $[\xi_i']_t = \lambda[\xi_i]_t$ and $\lambda \neq 0$ the total reprojection error would be the same. The solution we obtained above has an arbitrary scale or value of λ – changing the initial condition for the camera poses or landmark coordinates will lead to a solution with a different scale. We can remedy this by anchoring the pose of at least two cameras, one camera and one landmark, or two landmarks.

The bundle adjustment technique, but not this implementation, allows for constraints between cameras. For example, a multi-camera rig moving through space would use constraints to ensure the fixed relative pose of the cameras at each time step. Odometry from wheels or inertial sensing could be used to constrain the distance between camera coordinate frames to enforce the correct scale, or orientation from an IMU could be used to constrain the camera attitude. In the underlying graph representation of the problem as shown in Fig. 14.40 this would involve adding additional edges between the camera nodes. Constraints could also be added between landmarks that had a known relative position, for example the corners of a window – this would involve adding additional edges between the relevant landmark nodes.

The particular problem we studied is unusual in that every camera views every landmark. In a more common situation the camera might be moving in a very large environment so any one camera will only see a small subset of landmarks. In a real-time system a limited bundle adjustment might be performed with respect to occasional frames known as key frames, and a bundle adjustment over all frames, or all keyframes, performed at a low rate in the background.

In this example we have assumed the camera intrinsic parameters are known and constant. Theoretically bundle adjustment can solve for intrinsic as well as extrinsic parameters. We simply add additional parameters for each camera in the state vector and adjust the Jacobian $\boldsymbol{A}$ accordingly. However given the coupling between intrinsic and extrinsic parameters◀ this may lead to poor performance. If we chose to estimate the elements of the camera matrix $\boldsymbol{C}$ directly then the state vector would contain 11◀ rather than 6 elements for each camera. However if $\boldsymbol{C}_i$ and $\boldsymbol{P}_j$ are solutions so to is $\boldsymbol{C}_i\boldsymbol{Q}^{-1}$ and $\boldsymbol{Q}\tilde{\boldsymbol{P}}_j$ for any nonsingular matrix $\boldsymbol{Q} \in \mathbb{R}^{4\times4}$. Fortunately, projection matrices for realistic cameras have well defined structure and properties as described on page 327, and these provide constraints that allow us to estimate $\boldsymbol{Q}$. Estimating an arbitrary $\boldsymbol{C}_i$ is referred to as a projective reconstruction. This can be *upgraded* to an affine reconstruction (using an affine camera model) or a metric reconstruction (using a perspective camera model) by suitable choice of $\boldsymbol{Q}$.

The camera matrix has an arbitrary scale factor.

Changes in focal length and z-axis translation have similar image-plane effects as do change in principal point and camera x- and y-axis translation.

14.5 Point Clouds

Stereo vision results in a set of 3-dimensional world points $\mathbf{P}_i$ which are often referred to as a point cloud. For a robotics application we need to extract some concise meaning from the thousands or millions of points.

14.5.1 Fitting a Plane

Planes are common in our built world and for robotics useful planes include the ground (for wheeled mobile robot driving or UAV landing) and walls. Given a set of 3-dimensional coordinates, a point cloud, a simple and effective approach for finding the plane of best fit is to fit the data to an ellipsoid. 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

$$\boldsymbol{J} = \sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{x}_i^T \tag{14.17}$$

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 centered 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 = SE3(1,2,3) * SE3.rpy(0.3, 0.4, 0.5);
>> P = mkgrid(10, 1, 'pose', 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 with $\sigma = 0.02$ m.

The mean of the point cloud is

```
>> x0 = mean(P')
ans =
    0.9967    2.0009    3.0013
```

which we subtract from all the data points

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

and the inertia Eq. 14.17 is simply a matrix multiplication

```
>> J = P*P'
J =
    7.8769    0.3239   -4.2585
    0.3239   10.0076    0.6153
   -4.2585    0.6153    2.4271
```

The eigenvalues are

```
>> [x,lambda] = eig(J);
>> diag(lambda)'
ans =
    0.0478   10.0553   10.2085
```

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 =
    0.4789   -0.0696    0.8751
```

which is the estimated normal to the plane.

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

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

```
>> T.SO3.a'

ans =
    0.4682   -0.0810    0.8799
```

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

$$\boldsymbol{n}^T(\boldsymbol{x} - \boldsymbol{x}_0) = 0 \tag{14.18}$$

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. 14.17 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.

Commonly a Cauchy-Lorentz function $w = \beta^2 / (d^2 + \beta^2)$ is used where d is the distance of the point from the plane and β is the half-width. The function is smooth for $d = [0, \text{inf})$ and has a value of ½ when $d = \beta$.

Alternatively we could apply RANSAC by taking samples of three points to solve for Eq. 14.18. Section C.1.4 has more details about ellipses.

14.5.2 Matching Two Sets of 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 with a different pose 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 point coordinate vectors: the model $\boldsymbol{M}_i \in \mathbb{R}^n$, $i \in [1, N_M]$ and some noisy observed data $\boldsymbol{D}_j \in \mathbb{R}^n$, $j \in [1, N_D]$ determine the rigid-body motion from the data coordinate frame to the model frame

$$ {}^{D}\xi_M^* = \arg\min_{\xi} \sum_{i,j} \left\| \boldsymbol{D}_j - \xi \bullet \boldsymbol{M}_i \right\| $$

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 iterated closest point or ICP. For each data point $\boldsymbol{D}_j$, the corresponding model point $\boldsymbol{M}_i$ is assumed to be the closest one, that is $\boldsymbol{M}_i$ which minimizes $\|\boldsymbol{M}_i - \boldsymbol{D}_j\|$. Correspondence is not unique and quite commonly several points in one set can be associated with a single point in the other set, and consequently some points will be unpaired. Often the sensor returns only a subset of points in the model, for instance a laser scanner can see the front but not the back of an object. This approach to correspondence is far from perfect but it is surprisingly good in practice and *improves* the alignment of the point clouds so that in the next iteration the computed correspondences will be a little more accurate.

In robotics the problem is often considered as comprising a *model* $\boldsymbol{M}$ of a 3-dimensional object which we want to fit to the observed sensor data $\boldsymbol{D}$. To illustrate we will load a version of the famous Stanford bunny◀

This model is well known in the computer graphics community. It was created by Greg Turk and Marc Levoy in 1994 at Stanford University using a Cyberware 3030 MS scanner and a ceramic rabbit figurine. The original scan has over 30 000 points, here we use a low-resolution version.

```
>> load bunny
>> about bunny
bunny [double] : 3x453 (10.9 kB)
```

which is a cloud of 453 3-dimensional points and this will be our model

```
>> M = bunny;
```

We simulate a sensor that is observing the model with respect to a different coordinate frame by making a copy of the model and applying a transformation

```
>> T_unknown = SE3(0.2, 0.2, 0.1) * SE3.rpy(0.2, 0.3, 0.4);
>> D = T_unknown * M;
```

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

We consider the general case where the two points clouds have different numbers of points, that is, $N_D \neq N_M$.

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

$$ \bar{\boldsymbol{D}} = \frac{1}{N_D} \sum_{j=1}^{N_D} \boldsymbol{D}_j $$

from which we compute a displacement

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

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

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

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

$$\boldsymbol{W} = \sum_{i,j} \left(\boldsymbol{M}_i - \bar{\boldsymbol{M}}\right)\left(\boldsymbol{D}_j - \bar{\boldsymbol{D}}\right)^T$$

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

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

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

from which the rotation matrix is determined ▸ to be

See Sect. F.1.1.

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

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

$$\begin{aligned} \boldsymbol{M}_i &\leftarrow \xi \bullet \boldsymbol{M}_i, \forall i \\ \xi &\leftarrow \xi \oplus \xi_\Delta \end{aligned}$$

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 (Fig. 14.42)

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

which returns the pose ${}^D\xi_M$

```
>> trprint(T, 'rpy', 'radian')
t = (0.2, 0.2, 0.1), RPY/zyx = (0.2, 0.3, 0.4) rad
```

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

```
>> d
d =
   1.7619e-15
```

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 forty points from the data

```
>> D(:,randi(numcols(D), 40,1)) = [];
```

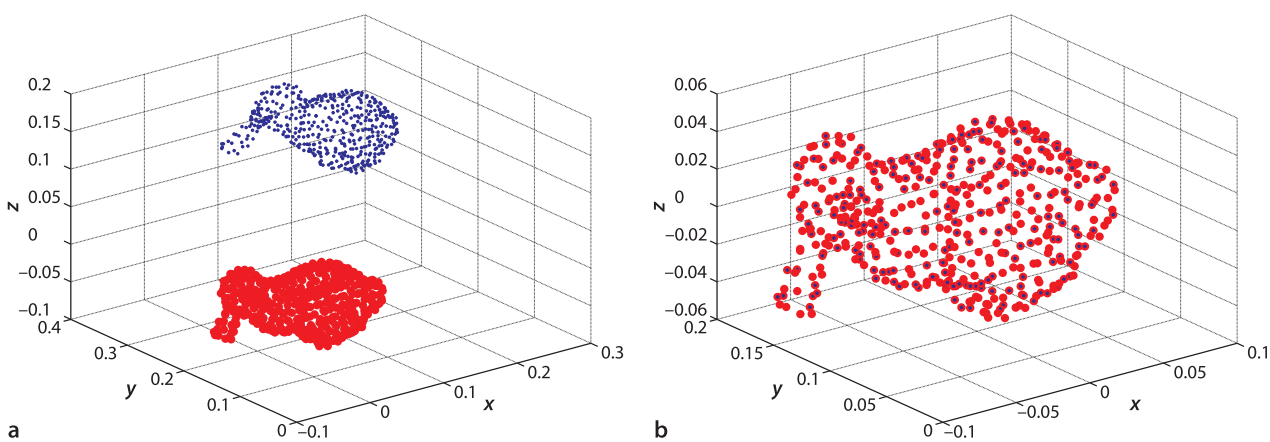


Fig. 14.42. Iterated closest point (ICP) matching of two point clouds: model (*red*) and data (*blue*) **a** before registration, **b** after registration; observed data points have been transformed to the model coordinate frame using the inverse of the identified transformation (Stanford bunny model courtesy Stanford Graphics Laboratory)

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

```
>> D = [D 0.1*rand(3,20)+0.1];
```

and finally we will add Gaussian noise with $\sigma = 0.01$ to the data

```
>> D = D + 0.01*randn(size(D));
```

Now we fit this imperfect sensor data to the model

```
>> [T,d] = icp(M, D, 'plot', 'distthresh', 3);
```

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 3 times the median distance. The estimated pose ${}^{D}\xi_{M}$ is now

```
>> trprint(T, 'rpy', 'radian')
t = (0.186, 0.194, 0.108), RPY/zyx = (0.125, 0.287, 0.298) rad
```

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

```
>> d
d =
    0.2114
```

is higher since an exact fit between the model and noise corrupted data is no longer possible. ICP is popular, 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.

14.6 Structured Light

An old, yet simple and effective, method of estimating the 3D structure of a scene is structured light. It is conceptually similar to stereo vision but we replace the left camera with a projector that emits a vertical plane of light as shown in Fig. 14.43a. This is equivalent, in a stereo system, to a left-hand image that is a vertical line. The image of the line projected onto the surface viewed from the right-hand camera will be a distorted version of the line, as shown in Fig. 14.43b. The disparity between the virtual left-hand image and the actual right-hand image is a function of the depth of points along the line.

Laser-based line projectors, so called "laser stripe"' or "line laser"', are available for just a few dollars. They comprise a low-power solid-state laser and a cylindrical lens or diffractive optical element.

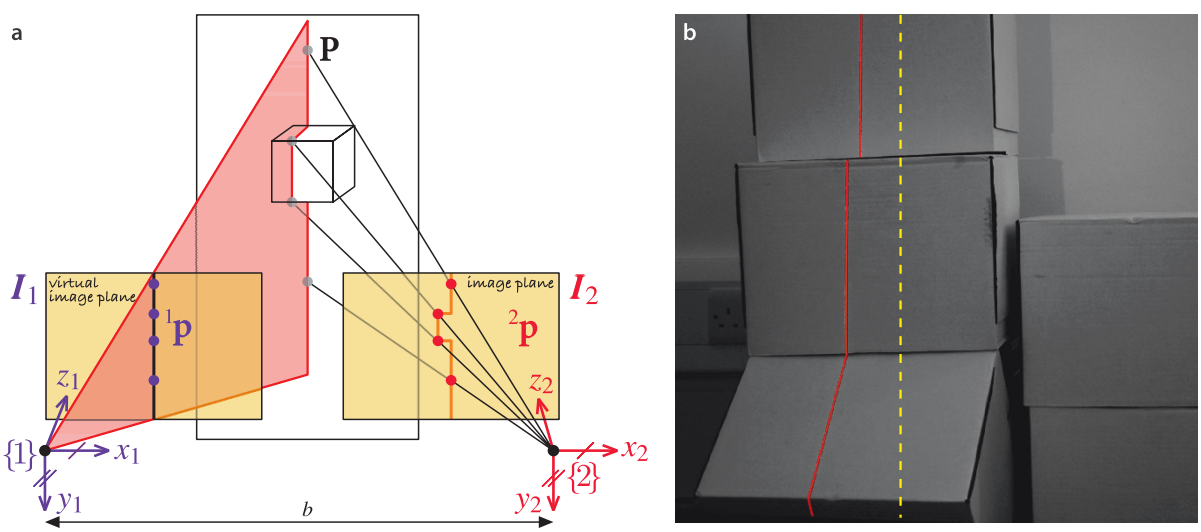


Fig. 14.43. a Geometry of structured light showing a light projector on the left and a camera on the right; four corresponding points are marked with *dots* on the left and right images and the scene; **b** a real structured light scenario showing the light stripe falling on a simple 3D scene. The superimposed *dashed line* represents the stripe position for a plane at infinity. Disparity, left shift of the projected line relative to the dashed line, is inversely proportional to depth

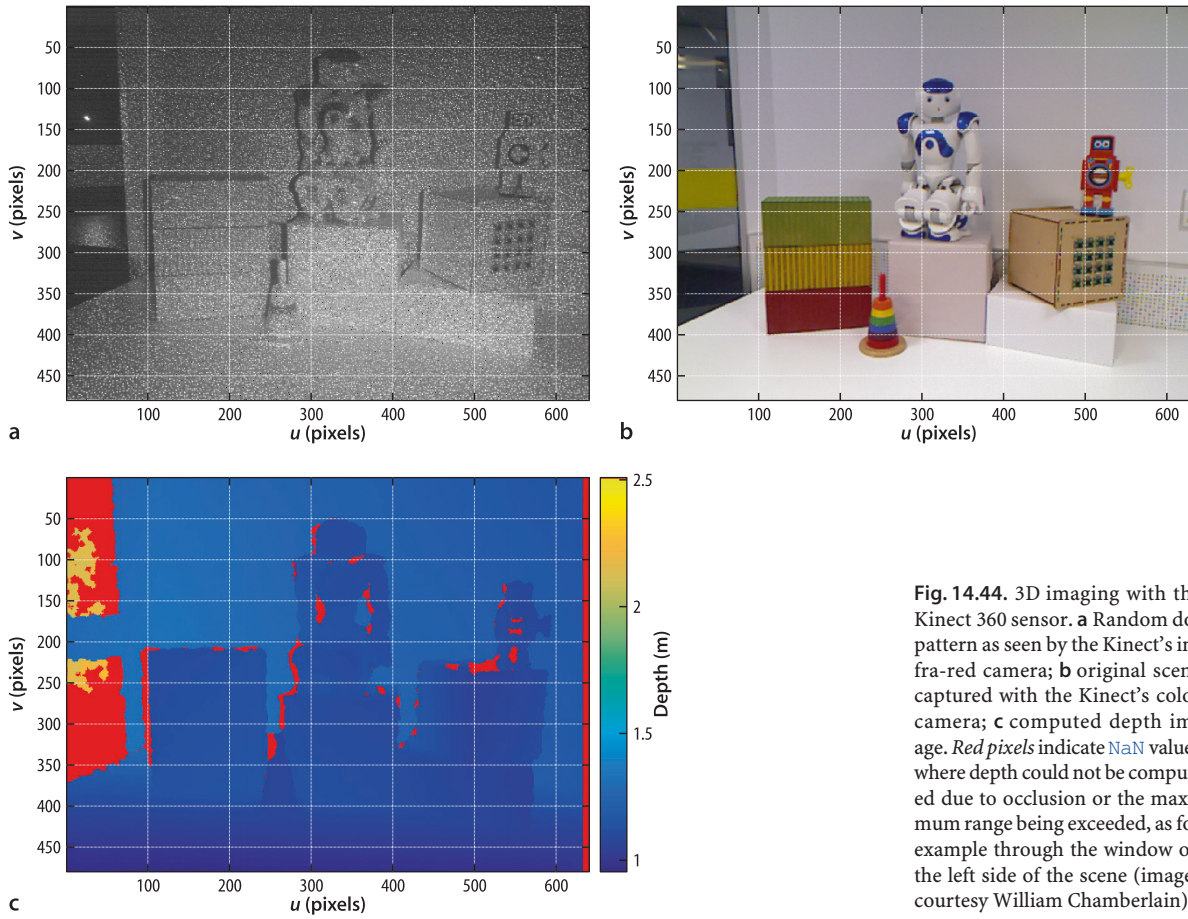


Fig. 14.44. 3D imaging with the Kinect 360 sensor. **a** Random dot pattern as seen by the Kinect's infra-red camera; **b** original scene captured with the Kinect's color camera; **c** computed depth image. *Red pixels* indicate `NaN` values where depth could not be computed due to occlusion or the maximum range being exceeded, as for example through the window on the left side of the scene (images courtesy William Chamberlain)

Finding the light stripe on the scene is a relatively simple vision problem. In each image row we search for the pixel corresponding to the projected stripe based on intensity or color. If the camera coordinate frames are parallel then depth is computed by Eq. 14.16.

To achieve depth estimates over the whole scene we need to move the light plane horizontally across the scene and there are many ways to achieve this: mechanically rotating the laser stripe projector, using a moving mirror to deflect the stripe or using a data projector and software to create the stripe image. However sweeping the light plane across the scene is slow and fundamentally limited by the rate at which we can acquire successive images of the scene. One way to speed up the process is to project multiple lines on the scene but then we have to solve the correspondence problem which is not simple if parts of some lines are occluded. Many solutions have been proposed but generally involve coding the lines in some way – using different colors or using a sequence of binary or grey-coded line patterns which can match 2^N lines in just N frames.

A related approach is to project a known but random pattern of *dots* onto the scene as shown in Fig. 14.44a. Each dot can be identified by the unique pattern of dots in its surrounding window. The original Kinect sensor▸ uses this approach: its left-most lens◂ projects an infra-red dot pattern using a laser with a diffractive optical element which is viewed, see Fig. 14.44a, by an infra-red sensitive camera behind the right-most lens from which the depth image shown in Fig. 14.44c is computed. The shape of the dots also varies with distance, due to imperfect focus, and this provides additional cues about the distance of a point. The middle lens is a regular color camera which provides

The Kinect for Xbox 360 and Kinect for Windows is now known as the Kinect 1, as well as sensors such as PrimeSense Carmine and Asus Xtion. The newer Kinect for Xbox One, or Kinect 2, uses per pixel time-of-flight measurement.

Looking at the front of the device.

the view shown in Fig. 14.44b. This is an example of an RGBD camera, returning an RGB color values as well as depth (D) at every pixel.

Structured light approaches work well for ranges of a few meters indoors, for textureless surfaces, and they also work in the dark. However outdoors the projected pattern is overwhelmed by ambient illumination from the sun.

Some stereo systems, such as the Intel RealSense R200, also employ a dot pattern projector, sometimes known as a speckle projector. This provides artificial texture which helps the stereo vision system when it is looking at textureless surfaces where matching is frequently weak and ambiguous as discussed in Sect. 14.3.2.1. Such a sensor has the advantage of working on textureless surfaces which are common indoors where the sun is not a problem, and outdoors using pure stereo where scene texture is usually rich.

14.7 Applications

14.7.1 Perspective Correction

Consider the image

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

shown in Fig. 14.45. 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 mark these on the image of the cathedral and overlay a translucent blue keystone shape

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

We use the extrema of these points to define the vertices 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)]';
```

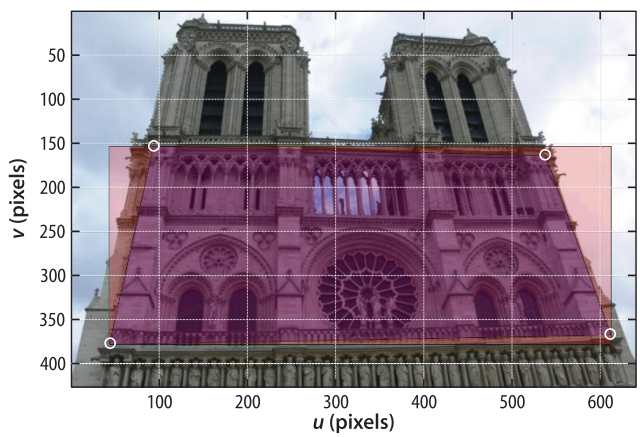


Fig. 14.45. Photograph taken from the ground shows the effect of foreshortening which gives the building a trapezoidal appearance (also known as keystone distortion). Four points on the approximately planar face of the building have been manually picked as indicated by the *white* ○*-markers* (Notre Dame de Paris)

which we overlay on the image in red

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

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
```

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

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

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

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. 14.46 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 metadata in the EXIF-format file that holds the image

```
>> [~,md] = iread('notre-dame.jpg', 'double');
>> f = md.DigitalCamera.FocalLength
f =
    7.4000
```

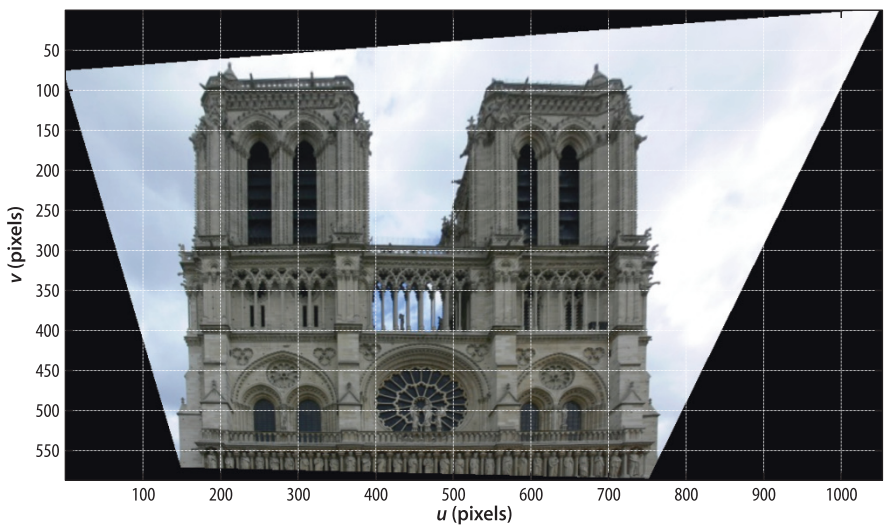


Fig. 14.46. A fronto-parallel view synthesized from Fig. 14.45. The image has been transformed so that the marked points become the corners of a rectangle in the image

which is in units of millimeters, and the sensor is known to be 7.18×5.32 mm. We create a calibrated camera

```
>> cam = CentralCamera('image', im, 'focal', f/1000, ...
    'sensor', [7.18e-3,5.32e-3])
name: image [central-perspective]
  focal length:   0.0074
  pixel size:     (1.122e-05, 1.249e-05)
  principal pt:   (320, 213)
  number pixels:  640 x 426
  pose:           t = (0, 0, 0), RPY/yxz = (0, 0, 0) deg
```

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

```
>> sol = cam.invH(H, 'verbose');
solution 1
     T = 0.99958    -0.01394     0.02526    -0.07271
         0.01431     0.99979    -0.01453    -0.00041
        -0.02505     0.01488     0.99958     0.68149
         0.00000     0.00000     0.00000     1.00000
    n = 0.21602    -0.95261     0.21420
solution 2
     T = 0.98872     0.10353    -0.10820     0.10448
        -0.01647     0.79331     0.60859    -0.57151
         0.14885    -0.59994     0.78607     0.36357
         0.00000     0.00000     0.00000     1.00000
    n = -0.18131     0.32802     0.92711
```

which returns a structure array of two possible solutions for ${}^1\xi_2$. The coordinate frames for this example are sketched in Fig. 14.47 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 correct sign,◂ is dominantly in the negative y- and positive z-direction with respect to the frame {1}. The rotation in YXZ-angle form

See Malis and Vargas (2007).

```
>> tr2rpy(sol(2).T, 'deg', 'camera')
ans =
   -1.1893  -37.4876   -7.8375
```

indicates that the camera needs to be pitched downward (pitch is rotation about the camera's x-axis) by 37 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.

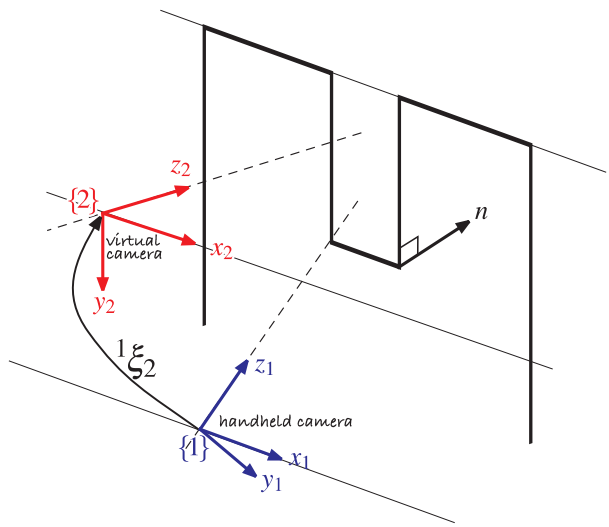


Fig. 14.47. Notre-Dame example showing the two camera coordinate frames. The *blue frame {1}* is that of the camera that took the image, and the *red frame {2}* is the viewpoint for the synthetic fronto-parallel view

14.7.2 Mosaicing [examples/mosaic]

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 drone 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, or built into, digital cameras is another example of mosaicing.

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.

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

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. 14.48.

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. 14.48. 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.

For this problem we assume that the scene is planar. This means that we can use an homography to relate the various camera views. The first step is to identify common feature points which are known as tie points, and we use now familiar tools

```
>> f1 = isurf(im1)
>> f2 = isurf(im2)
>> m = f1.match(f2);
```

and then RANSAC to estimate the homography

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

which maps ${}^1\mathbf{p}$ to ${}^2\mathbf{p}$. Now we wish to map ${}^2\mathbf{p}$ to its corresponding coordinate in the first image

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

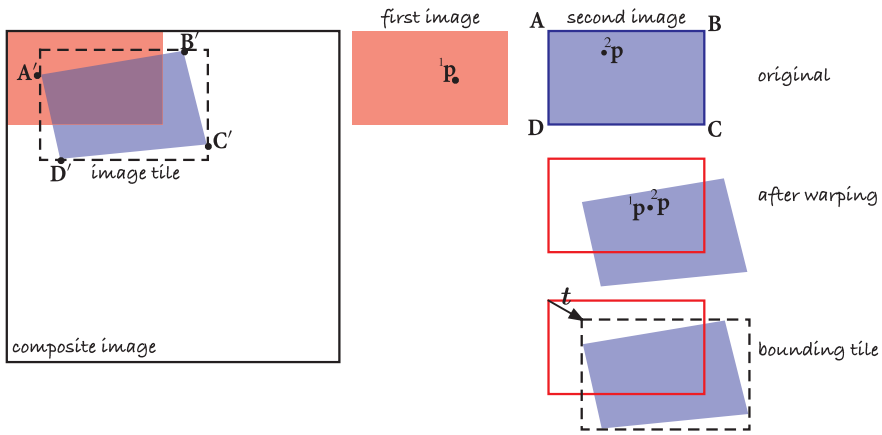


Fig. 14.48. The first image in the sequence is shown as *red*, the second as *blue*. The second image is warped into the image tile and then blended into the composite image

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. 14.48 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 mosaic image

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

and the result is shown in Fig. 14.49. We can clearly see several images overlaid with good alignment. The nonmapped 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 nonzero 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);
```

Which ignores any pixels with the value `NaN`.

and then blend using the `'mean'` option◀

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

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 be 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.

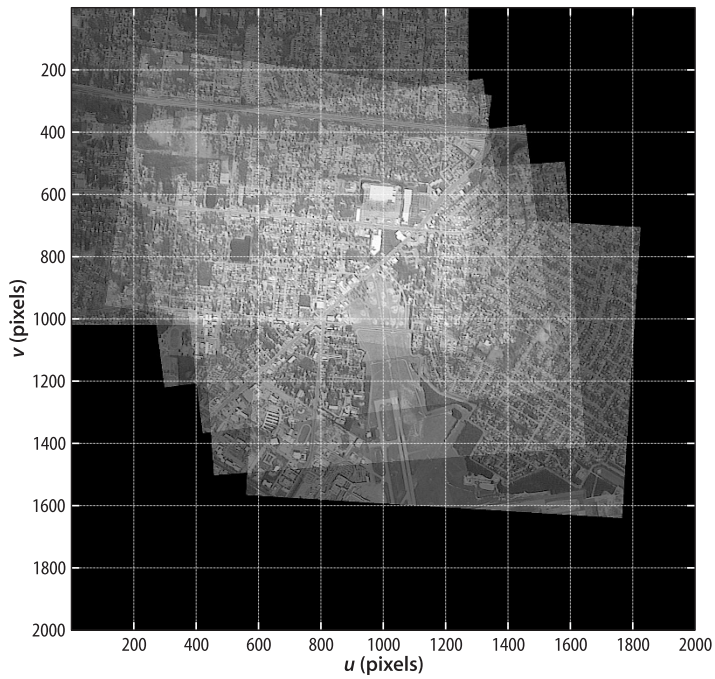


Fig. 14.49. Example image mosaic. At the bottom of the frame we can clearly see three overlapping views of the airport runway which shows good alignment between the frames

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 sometimes provides an imperfect orthophoto. When looking at cities we might 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 full code is given by `mosaic` 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.

14.7.3 Image Matching and Retrieval [examples/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 is useful to a robot to determine if it has visited a particular place before, or seen the same object before. If those previous images have some associated 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 been used successfully in a number of robotic applications. 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, 'thresh', 0);
```

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

```
>> sf{1}
ans =
1407 features (listing suppressed)
  Properties: theta scale u v strength descriptor image_id
```

is a vector of 1 407 SURF feature objects corresponding to the first image in the sequence. The set of all SURF features across all images is

```
>> sf = [sf{:}]
sf =
28644 features (listing suppressed)
  Properties: theta scale u v strength descriptor image_id
```

which is a vector of nearly 30 000 SURF features objects.

Consider a particular SURF feature

```
>> sf(259)
ans =
(207.101,300.162), theta=2.31733, scale=2.1409,↵
 strength=0.00114015, image_id=1, 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(259).plot('g+')
>> sf(259).plot_scale('g', 'clock')
```

which is shown in Fig. 14.50a. The support region for this feature

```
>> sf(259).support(images)
```

is shown in Fig. 14.50b. 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 is a *k*-means problem. To find 2 000 feature clusters

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

returns a `BagOfWords` object that contains the original features, the center 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 from some vocabulary, each image comprises a collection (or *bag*) of visual words drawn from the visual vocabulary.

The `BagOfWords` class uses 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);`.

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

```
>> w = bag.words(259)
w =
        1962
```

we find that the *k*-means clustering has assigned this image feature to word 1 962 in the vocabulary – it is an instance of visual word 1 962. That particular visual word appears

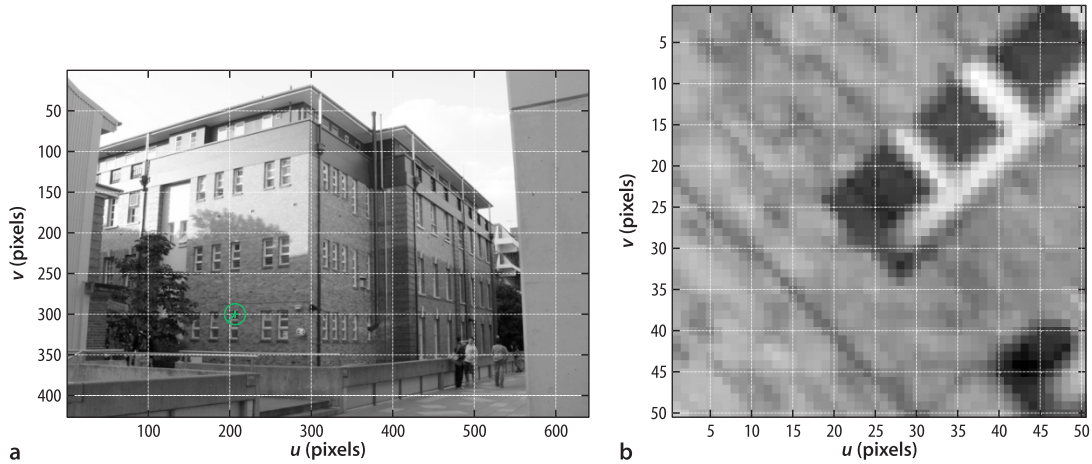


Fig. 14.50. a Image 1 with visual word SURF feature 380 indicated by *green circle* showing scale and a radial line showing orientation direction; **b** the square support region has the same area as the circle and the horizontal axis is parallel to the orientation direction

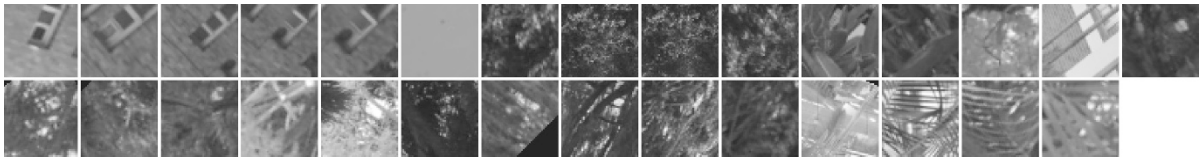


Fig. 14.51. Exemplars of visual word 1 962 from the various images in which it appears. The annotation is of the form word/image

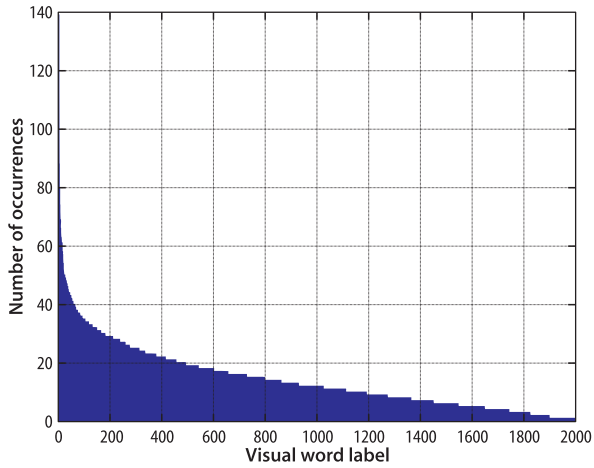


Fig. 14.52. Histogram of the number of occurrences of each word (sorted). Note the small number of words that occur very frequently

```
>> bag.occurrence(w)
ans =
    29
```

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

```
>> bag.contains(w)
ans =
   1     5     7     8     9    11    12    15    16    18
```

We can display some of the different instances of word 1 962 by

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

which is shown in Fig. 14.51. 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') )
```

which is shown in Fig. 14.52. 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 2863 features associated with 50 most frequent words
>> bag
bag =
BagOfWords: 25781 features from 20 images
           1950 words, 50 stop words
```

which leaves some 26 000 SURF features behind. This method performs relabelling so that word labels are now in the interval 1 to 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 word frequency vector. This is a K-element vector

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

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

$$t_j = \frac{n_{ij}}{n_i} \underbrace{\log \frac{N}{N_j}}_{\text{idf}} \tag{14.19}$$

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 and 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

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

and is implemented by the `similarity` method. A value of one indicates maximum similarity. To compute the mutual similarity across this set of images (bags of words) is simply

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

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

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

which is shown in Fig. 14.53. The bright diagonal indicates, as a useful cross check, that image i is identical to image i. We also see that there is also some nonzero similarity between images 12 and 18, among others.

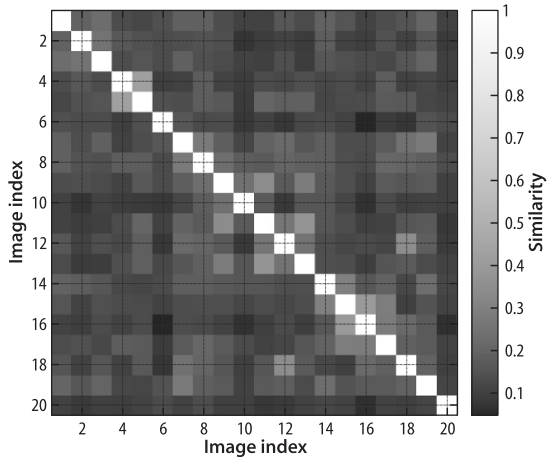


Fig. 14.53. Similarity matrix for 20 images where *light colors* indicate strong similarity. Element (i, j) indicates the similarity between image i and image j

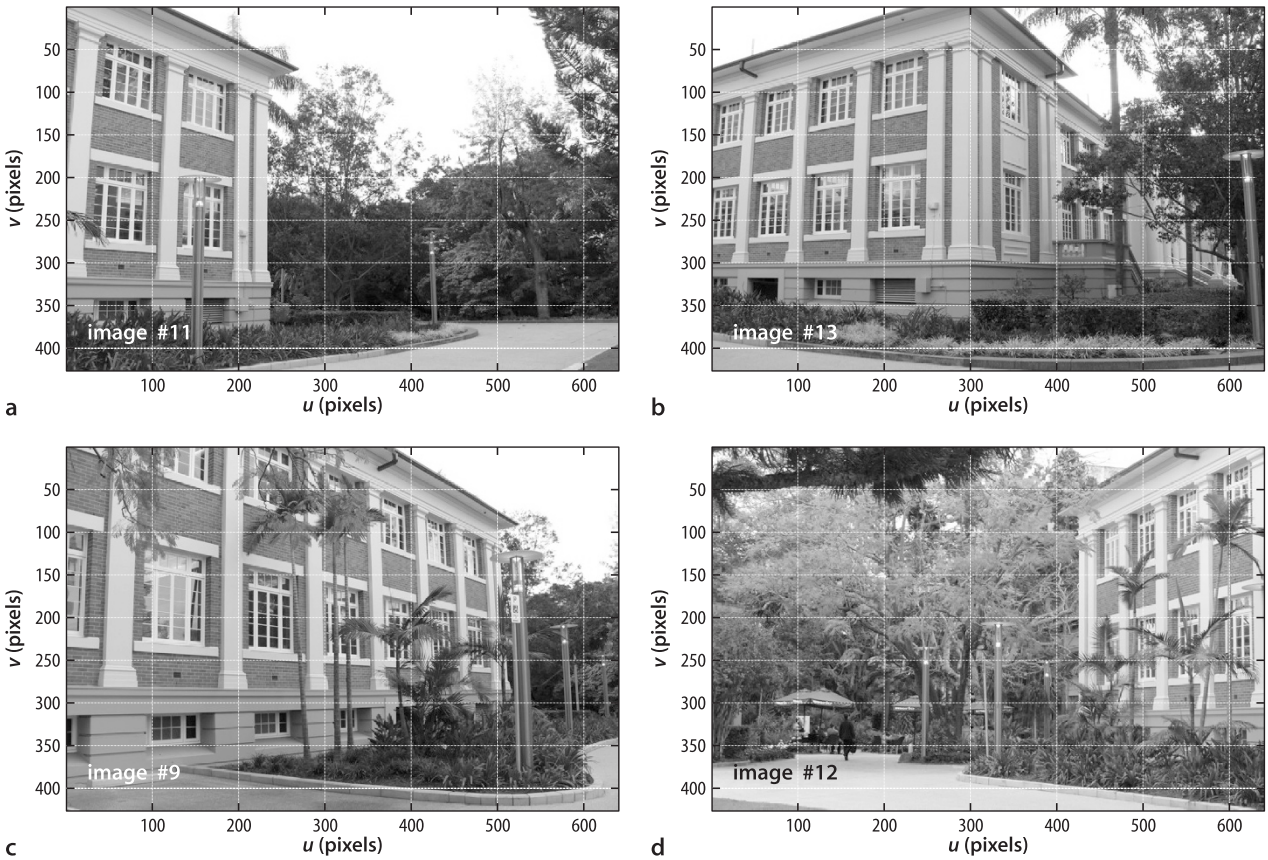


Fig. 14.54. Image recall. Image 11 is the query, and in decreasing order of match quality we have recalled images 13, 9 and 12

Consider image 11 shown in Fig. 14.54a. 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
    0.3722   13.0000
    0.3394    9.0000
    0.2610   12.0000
    0.2038    5.0000
      .
      .
```

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 13, 9, 12 and so on. These are shown in Fig. 14.54 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, 'thresh', 0)
```

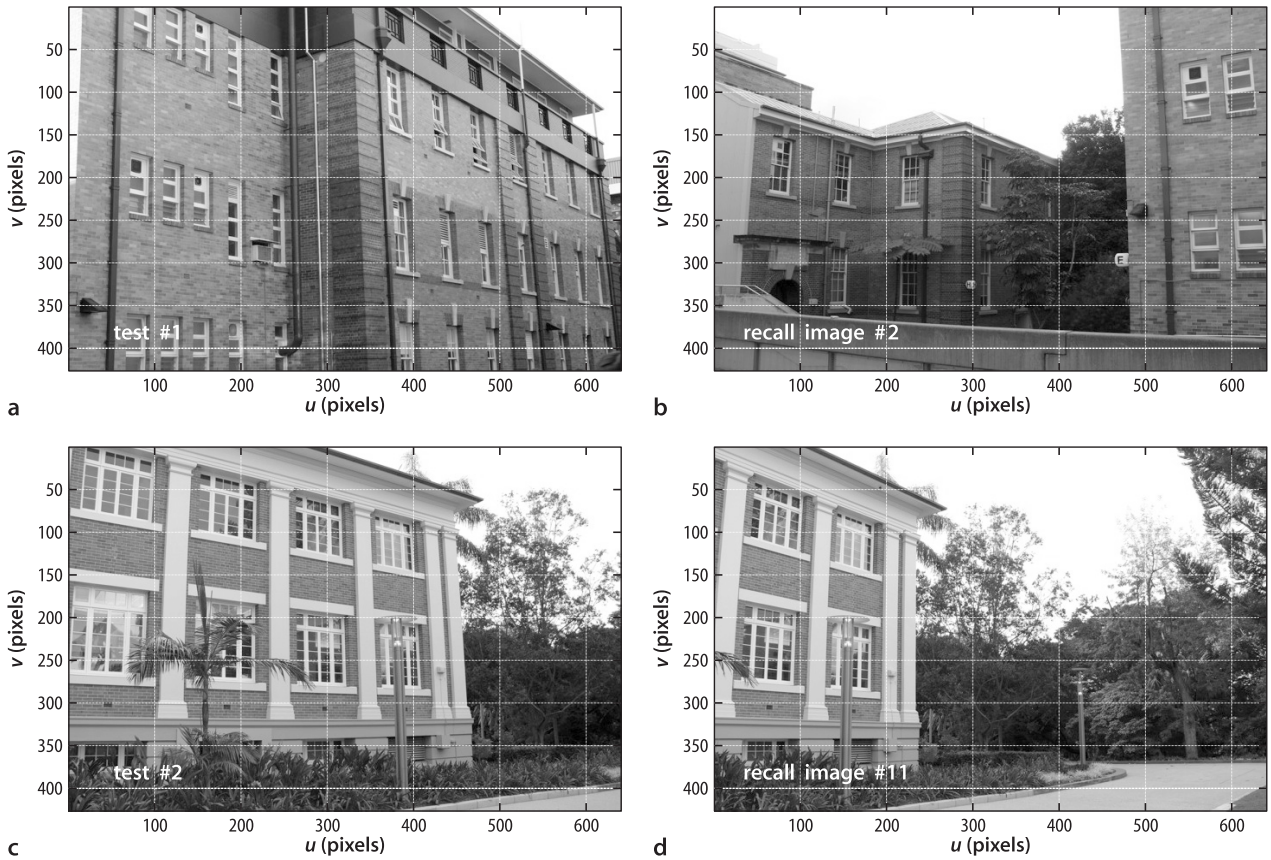


Fig. 14.55. Image recall for new images. The new query images **a** and **c** recall the database images **b** and **d** respectively

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

```
>> bag2 = BagOfWords(sf2, bag)
BagOfWords: 6530 features from 5 images
          1950 words, 50 stop words
```

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

This operation also removes any features words that were previously determined to be stop words, and computes the word frequency vectors◂ according to Eq. 14.19.

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 `S2(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,k] = max(S2)
z =
    0.3435    0.6948    0.5427    0.5521    0.3627
k =
     2    11    16    18    20
```

New image 1 best matches image 2 in the original sequence, new image 2 matches image 11 and so on. Two of the new images and their closest existing images are shown in Fig. 14.55. The first recall has a low similarity score but is a reasonable result – the recall image includes the building from the test image at the right and another building that has many similarities.

14.7.4 Visual Odometry [examples/vodemo]

A common problem in robotics is to estimate the distance a robot has traveled, and this is a key input to all of the localization algorithms discussed in Chap. 6. For a wheeled robot we can use information from the wheel encoders but these are subject to random errors (slippage) as well as systematic errors (imprecisely known wheel radius). However for a flying or underwater robot the problem of odometry is much more difficult. Visual odometry (VO) is the process of using information from consecutive images to estimate the robot's relative motion from one camera image to the next.

We load a sequence of images taken from a car driving along a road

This image sequence is bulky and not distributed with the main toolbox, but it can be found in the contrib2 zip file on the Toolbox website. This sequence is dataset 4 of the .enpeda.. Image Sequence Analysis Test Site (EISATS).

```
>> left = iread('bridge-l/*.png', 'roi', [20 750; 20 440]);
```

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(left)
left [uint16] : 421x731x251 (154.5 MB)
```

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(left, '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 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. 14.56. The features are associated with regions of high gradient such as the edges of trees, as

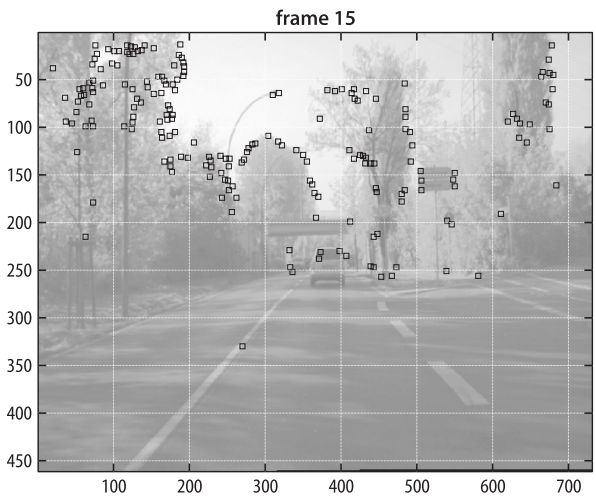


Fig. 14.56. Frame number 15 from the `bridge-l` image sequence with overlaid features (image from .enpeda. project, Klette et al. 2011)

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 magnitude of optical flow – the speed of a world point on the image plane – is proportional to camera velocity divided by distance to the world point and therefore has a scale ambiguity – a camera moving quickly through a world with distant points yields the same flow magnitude as a slower camera moving past closer points. To resolve this we need to use additional information. For example if we knew that the points were on the road surface, that the road was flat, and the height of the camera above the road then we can resolve this unknown scale. However this assumption is quite strict and would not apply for something like a drone moving over unknown terrain. Instead we will use information from a different view of the world – the right image from a stereo camera fitted to the vehicle.

```
>> right = iread('bridge-r/*.png', 'roi', [20 750; 20 440]);
```

For each pair of left and right images we extract features, and determine correspondence by robustly matching features using descriptor similarity and the epipolar constraint implied by a fundamental matrix. Next we compute horizontal disparity between corresponding features, and assuming the cameras are fully calibrated we triangulate the image-plane coordinates to determine the world coordinates of the landmark points with respect to the left-hand camera on the vehicle. We can match the 3D point clouds at the current and previous time step using a technique like iterated closest point (ICP) in order to determine the camera pose change. This is the so-called 3D-3D approach to visual odometry and while the principle is sound it works poorly in practice. Firstly, some of the 3D points may be on other moving objects and this violates the assumption of ICP that the sensor or the object moves, but not both. Secondly, the estimated range to distant points is quite inaccurate since errors in estimated disparity become significant when disparity is small.

An alternative approach, 3D-2D matching, projects the 3D points at the current time step into the previous image and finds the camera pose that minimizes the error with respect to the observed feature coordinates – this is bundle adjustment. Typically this is done for just one image and we will choose the left image. To establish correspondence of features over time we find correspondences between left-image features that had a match with the right image and a match with features from the previous left image – again enforcing an epipolar constraint. We now know the correspondence between points in the three views of the scene as shown in Fig. 14.57.

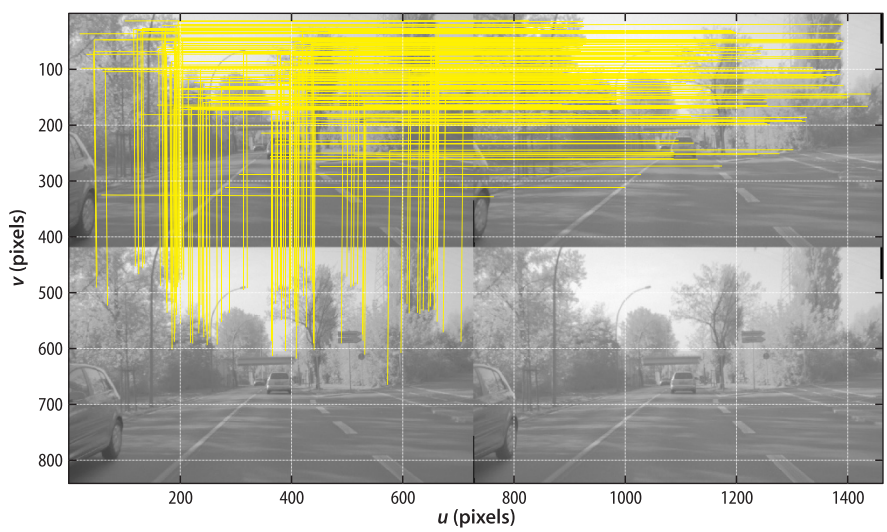


Fig. 14.57. Feature correspondence for visual odometry. The top row is a stereo pair at the current time step, and the bottom row is a stereo pair at the previous time step. Epiplolar consistent correspondences between three of the image images are shown in *yellow*

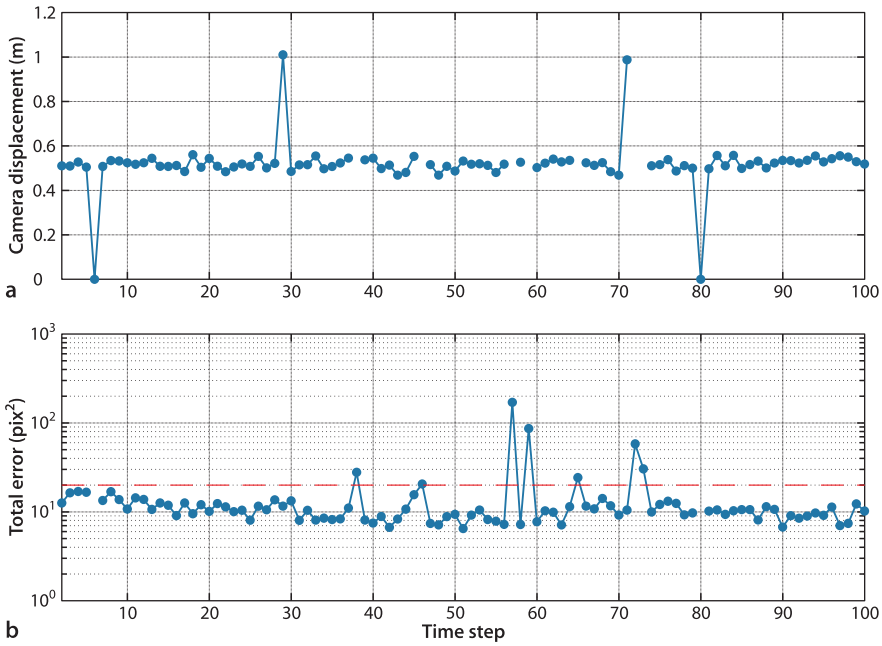


Fig. 14.58.
Visual odometry results. **a** Estimated displacement of the camera its z-direction (forward); **b** bundle adjustment final error per frame, shown with a *logorithmic vertical scale*

At each time step we set up a bundle adjustment problem that has two cameras and a number of landmarks determined from stereo triangulation. The first camera is associated with the previous time step and is fixed at the reference frame origin. The second camera is associated with the current time step and would be expected to have a translation in the positive z-axis direction. We could obtain an initial estimate of the second camera's pose by estimating and decomposing an essential matrix, but we will instead set it to the origin.

The details can be found in the example script

```
>> visodom
```

which processes 100 frames and displays graphics like Fig. 14.57 for every frame. The final results for z-axis translation are shown in Fig. 14.58a and we notice a value of around 0.5 m at each time step, but there are also some missing data points and two incorrect looking results. The bundle adjustment process returns the final squared error and this is plotted in Fig. 14.58b for each frame. For 8% of the frames that error was over 20 pix^2 (*red dashed line*) and we exclude those results. The likely source of error is incorrect point correspondences. Bundle adjustment assumes that all points in the world are fixed but in this sequence there are numerous moving objects. We used the epipolar constraint between current and previous frame to ensure that only features points consistent with a moving camera and a fixed world are in the inliner set. However when the script runs we see quite a lot of points on the car in front which are being incorrectly included in the inlier set – that car is moving but because it is a large and constant distance away those points are not inducing enough error to be considered outliers. A more sophisticated bundle adjustment algorithm would detect and reject such points. Finally there is a preponderance of points in the top part of the scene which tend to be quite distant from the cameras. A more sophisticated approach to feature detection would choose features more uniformly over the image.

The erroneous points at timesteps 29 and 71 highlight a common problem with using video data for robots. The clue is that those values are suspiciously close to exactly twice the other values. Each image in the sequence was assigned a timestamp when it was received by the computer and those timestamps can be loaded

```
>> ts = load('timestamps.dat');
```

and if we plot the difference between timestamps

```
>> plot(diff(ts))
```

we see that the average time between images is 44.6 ms but there are two spikes where the interval is twice that. The computer logging the images has skipped a frame, perhaps it was unable to write image data to memory or disk as quickly as it was arriving. So the interval between the frames was twice as long, the vehicle traveled twice as far, and the spikes on our estimated displacement are in fact correct. This is not an uncommon situation – in a robot system all data should be timestamped and timestamps should be checked to detect problems like this.

The median velocity over the valid estimates is

```
>> median(tz(ebundle<20))
ans =
    0.5201
```

in units of meters which, with the camera frame interval of 44.6 ms, indicates a vehicle speed of around 40 km h^{-1}. The variable `tz` is a vector of frame-to-frame displacement computed by the script, and `ebundle` is a vector of bundle adjustment errors at each time step. The residuals from estimating the fundamental matrix between the current and previous left image are saved in the vector `efund`.

For a vehicle or robot the estimated displacements over time are not independent and are related by vehicle kino-dynamic model, and we can use this to smooth the results and discount erroneous velocity estimates. If the bundle adjuster included constraints on camera pose we could set the weighting to penalize infeasible motion in the lateral and vertical directions as well as roll and pitch motion.

14.8 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 loss of scale in the perspective projection process and techniques based on additional sources of information to recover scale such as stereo vision, structured light or bundle adjustment.

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.

For the special case where world points lie on a plane they *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.

We used bundle adjustment to solve the structure and motion estimation problem – using 2D measurements from a set of images of the scene to recover information related to the 3D geometry of the scene as well as the locations of the cameras. Stereo vision is a simple case where the motion is known – fixed by the stereo baseline – and we are interested only in structure. The visual odometry problem is complementary and we are interested only in the motion of the camera, not the scene structure.

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

MATLAB and Toolbox Notes

The Toolbox uses open-source code to support SIFT (VLFeat) and SURF (OpenSURF http://www.mathworks.com/matlabcentral/fileexchange/28300) features. VLFeat (http://www.vlfeat.org) includes a number of feature detectors and other useful functions. The OpenCV library implements many feature detectors and descriptors and can be accessed in MATLAB using mexopencv (https://kyamagu.github.io/mexopencv).

The MATLAB Computer Vision System Toolbox™ (CVST) has support for stereo rectification; stereo matching; SURF, FAST and Harris feature detectors; a range of descriptors (BRISK, HOG, MSER); and point cloud processing including kd-trees, model fitting and visualization. Many CVST functions can be used inside Simulink and support automatic code generation for real-time hardware such as FPGAs.

Further Reading

3-dimensional reconstruction and camera pose estimation has been studied by the photogrammetry community since the mid nineteenth century, see page 354. 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. 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).

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 matrices; 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 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.

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	$\mathbf{P}$
Image point	$\mathbf{x}, \mathbf{x}'$	x_1, x_2	${}^1\mathbf{p}, {}^2\mathbf{p}$
i^{th} image point	$\mathbf{x}_i, \mathbf{x}'_i$	x_1^i, x_2^i	${}^1\mathbf{p}_i, {}^2\mathbf{p}_i$
Camera motion	$\mathrm{R}, \mathbf{t}$	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	$\mathbf{x} = \mathrm{P}\mathbf{X}$	$\lambda x = \Pi P$ $x \sim \Pi P$	$\tilde{\boldsymbol{p}} \simeq \boldsymbol{C}\tilde{\boldsymbol{P}}$

References related to SURF and other feature detectors were previously discussed on page 456. 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.◀ Arandjelović and Zisserman (2012) discuss some important points when matching feature vectors.

A jet is a vector of higher order derivatives such as $\mathbf{I}_{uu}$, $\mathbf{I}_{vv}$, $\mathbf{I}_{uv}$, $\mathbf{I}_{uuu}$, $\mathbf{I}_{uuv}$, $\mathbf{I}_{uvv}$, $\mathbf{I}_{vvv}$, $\mathbf{I}_{uuuu}$, $\mathbf{I}_{uuuv}$, $\mathbf{I}_{uuvv}$, $\mathbf{I}_{uvvv}$, $\mathbf{I}_{vvvv}$ 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 but fails with very small inlier ratios. A recent more robust development is vector field consensus (VFC) by Ma et al. (2014). 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 nonperspective 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. 14.59.

Stereo cameras and stereo matching software are available today from many 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 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.

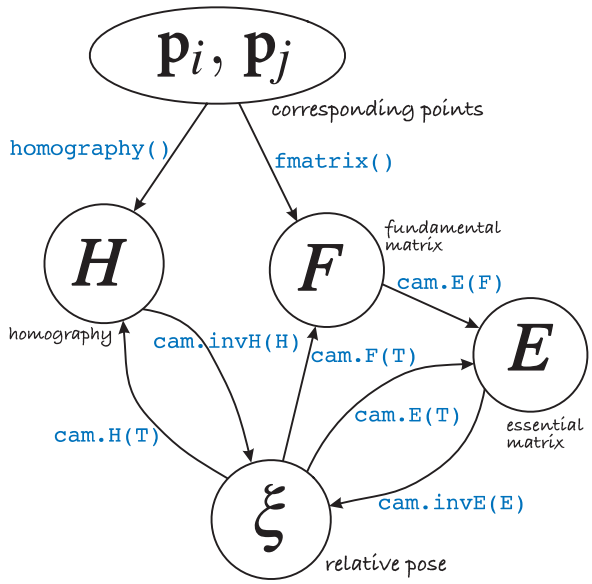


Fig. 14.59. Toolbox functions and camera object methods, and their interrelationship

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 subpixel refinement. The dense stereo matching algorithm presented in Sect. 14.3.2 is a very conventional correlation-based stereo algorithm. The disparity computed at each pixel is independent of other pixels but for most real scenes adjacent pixels belong to the same surface and disparity will be quite similar – this is referred to as the *smoothness constraint*. Of course disparity will be discontinuous at the edges of surfaces. Finding the shortest best-fit path through a slice of the disparity space image as shown in Fig. 14.29 will enforce the smoothness constraint in the horizontal direction. Ideally we wish to also ensure vertical smoothness as well and this can be achieved using Markov random fields (MRFs), total variation with regularizers (Pock 2008), or more efficient semi-global matching (SGM) algorithms (Hirschmüller 2008). The very popular library for efficient large-scale stereo matching (LIBELAS) by Geiger et al. (2010) uses an alternative to global optimization that provides fast and accurate results for a variety of indoor and outdoor scenes. Stereo vision involves a significant amount of computation but there is considerable scope for parallelization using multiple cores, MIMD instruction sets, GPUs, custom chips and FPGAs. The use of nonparametric local transforms is described by Zabih and Woodfill (1994) and Banks and Corke (2001).

An emerging alternative to stereo vision are cameras based on time-of-flight measurement which are dropping rapidly in cost. A pulse of infra-red light illuminates the scene and every pixel records the intensity and time delay of the reflected energy. Time-of-flight sensors include the REAL3 devices by Infineon (infineon.com) and PhotonICs from **pmd**technologies (pmdtec.com). Complete time-of-flight cameras include the Kinect for Xbox One (Kinect 2) and various from **pmd**technologies. This type of camera works well indoors and even in complete darkness, but outdoors under full sun the maximum range is limited just as it is for structured light.▸

In fact it is worse than structured light. The illumination energy is limited by eye-safety considerations and structured light concentrates that energy over a line whereas time-of-flight cameras spread it over an area.

The ICP algorithm (Besl and McKay 1992) is used for a wide range of applications from robotics to medical imaging. ICP is fast but determining the correspondences via nearest neighbors 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 orthonormal rotation matrices have been 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. A more recent extension by Newcombe et al. (2011) performs camera tracking and dense 3D reconstruction from a single moving RGB camera. 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.

Bundle adjustment or structure from motion (SfM) is a big field with a large literature that cover many variants of the problem, for example robustness to outliers, and specific applications and camera types. Classical introductions include Triggs et al. (2000) and Hartley and Zisserman (2003). Recent theses by Warren (2015), Sünderhauf (2012) and Strasdat (2012) are comprehensive and readable. Unfortunately every reference uses different notation. Estimating the camera matrix for each view, computing a projective reconstruction, and then upgrading it to a Euclidean reconstruction is described by Hartley and Zisserman (2003) and Ma et al. (2003).

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 readable two-part tutorial introduction to visual odometry (VO) is Scaramuzza and Fraundorfer (2011) and Fraundorfer and Scaramuzza (2012). Visual odometry is discussed by Nistér et al. (2006) using point features and monocular or stereo vision. Maimone et al. (2007) describe experience with stereo-camera VO on the Mars rover and Corke et al. (2004) describe monocular catadioptric VO for a prototype planetary rover.

Mosaicing is a process as old as photography. In the past it was highly skilled and labor 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. High-quality offline 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, and is available in OpenCV. An open source version (Glover et al. 2012) is available at https://github.com/arrenglover/openfabmap. Chatfield et al. (2011) discussed some recent improvements to the bag-of-words image retrieval problem.

Image sequence analysis is the core of many real-time robotic vision systems. Real-time feature tracking across frames 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 real-time performance.

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). 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 various NASA Mars rovers are available online as left+right pairs or encoded in anaglyphs. 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.

The popular LIBELAS library (http://www.cvlibs.net/software/libelas) for large-scale stereo matching supports parallel processing using OpenMP and has MATLAB and ROS interfaces. Various stereo vision algorithms are compared for speed and accuracy at the KITTI (www.cvlibs.net/datasets/kitti/eval_scene_flow.php) and Middlebury (vision.middlebury.edu/stereo/eval3) benchmark sites.

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, is available at http://cs.unc.edu/~ssinha/Research/GPU_KLT. The ViSP cross-platform library includes tracking capability and can be found at https://visp.inria.fr. Pointers to SIFT and SURF implementations are given on page 456. 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 and handles perspective and catadioptric cameras. Andrew Davison's monocular visual SLAM system (MonoSLAM) for C and MATLAB is available at http://www.doc.ic.ac.uk/~ajd/software.html.

The sparse bundle adjustment software by Lourakis (users.ics.forth.gr/~lourakis/sba) is an efficient C implementation that is widely used and has a MATLAB and OpenCV wrapper. One application is Bundler (www.cs.cornell.edu/~snavely/bundler) which can perform matching of points from thousands of cameras over city scales and has enabled reconstruction of cities such as Rome (Agarwal et al. 2014), Venice and Dubrovnik. Some of these large-scale datasets are available from grail.cs.washington.edu/projects/bal and www.robots.ox.ac.uk/~vgg/data/data-mview.html. A MATLAB interface to Bundler is available at www.mathworks.com/matlabcentral/fileexchange/46341. SFMedu, a Structure from Motion System for Education (http://vision.princeton.edu/courses/SFMedu) has learning resources and MATLAB source code. Other open source solvers that can be used for sparse bundle adjustment include g^{2}o, SSBA and CERES, all implemented in C++. g^{2}o by Kümmerle et al. (2011) (github.com/RainerKuemmerle/g2o) can also be used to solve SLAM problems. SSBA by Christopher Zach is available at https://github.com/chzach/SSBA. The CERES solver from Google (ceres-solver.org) is a library for modeling and solving large complex optimization problems on desktop and mobile platforms and also supports parallel processing using OpenMP. A MATLAB interface is available at github.com/tikroeger/BA_Matlab.

Pointcloud library (PCL) (pointclouds.org) is a large-scale, open and standalone package for 2D/3D image and point cloud processing with support for feature detectors and descriptors, 3D registration, kd-trees, shape segmentation, surface meshing, visualization, camera interfaces and includes g^{2}o. The Point Data Abstraction Library (PDAL) (www.pdal.io) is a library and set of Unix command line tools for manipulating point cloud data.

Point clouds can be stored in a number of common open formats. Point cloud data (PCD) files are defined by Pointcloud library (PCL) (pointclouds.org) and can be imported into MATLAB using www.mathworks.com/matlabcentral/fileexchange/40382.

Polygon file format (PLY) files are designed to describe meshes but can be used to represent an unmeshed point cloud, and there are a number of great visualizers such as MeshLab and potree. PCL and PDAL can read, write and convert many point cloud file formats.

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

Exercises

1. Corner features and matching (page 462). 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 grey values; *(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 the patch descriptor sizes of 3×3, 7×7 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 470)
   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) from N inliers and M outliers?
7. 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 example of Fig. 14.16 has epipolar lines that slope slightly upward. What does this indicate about the two camera views?
8. Essential matrix (page 469)
   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`.

9. Homography (page 477)
   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 courtyard image, to determine the plane of the wall with respect to the camera. You will need the camera intrinsic parameters.
10. Load a reference image of this book's cover from `rvc2_cover.png`. Next, capture an image that includes the book's front cover, compute SIFT or SURF features, match them and use RANSAC to estimate an homography between the two views of the book cover. Decompose the homography to estimate rotation and translation. Put all of this into a real-time loop and continually display the pose of the book relative to the camera.
11. Sparse stereo (page 482)
   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?
12. Bundle adjustment (page 497)
   a) Vary the initial condition for the second camera, for example, set it to the identity matrix.
   b) Set the initial camera translation to 3 m in the x-direction, and scale the landmark coordinates by $10\times$. What is the final value of the back-projection error and the second camera pose.
   c) Experiment with anchoring landmarks and cameras.
   d) Derive the two Jacobians $\boldsymbol{A}$ (hard) and $\boldsymbol{B}$.
13. 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.
14. Stereo vision. Using the rock piles example (page 483)
   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?
15. Using the rock piles example (page 483) 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.
16. Download an anaglyph image and convert it into a pair of greyscale images, then compute dense stereo.
17. Variations to stereo matching
   a) Try some other stereo images, either acquired with a stereo camera or from the Middlebury dataset.

b) Perform stereo matching using the SAD rather than NCC metric. Use the `'metric'` option to `istereo`.
c) Apply the census (`icensus`) or rank transforms (`irank`) to the left and right image prior to matching using the SAD measure and investigate the matching quality. More details in Banks and Corke (2001).

18. Stereo vision. For a pair of identical cameras with a focal length of 8 mm, 1 000 × 1 000 pixels that are 10 μm square on an 80 mm baseline and with parallel optical axes:
a) Sketch the fields of views of the camera in a plan view. If the cameras are viewing a plane surface normal to the principal axes how wide is the horizontal overlapping field of view in units of pixels?
b) Assuming that disparity error is normally distributed with $2\sigma = 0.2$ pixels compute and plot the distribution of error in the z-coordinate of the reconstructed 3D points which have a mean disparity of 0.5, 1, 2, 5, 10 and 20 pixels. Draw 1 000 random values of disparity, convert these to Z and plot a histogram (distribution) of their values.

19. Mona Lisa on your wall. Acquire an image of a room in your house and display it using MATLAB. Select four points, using `ginput`, to define the corners of the virtual frame on your wall. Perhaps use the corners of an existing rectangular feature in your room such as a window, poster or picture. Estimate the appropriate homography, warp the Mona Lisa image and insert it into the original image of your room.

20. Plane fitting (page 504)
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. 14.18. 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`.

21. ICP (page 505)
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.

22. Perspective correction (page 509)
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.

23. Mosaicing (page 512)
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.

24. Image stabilization can be used to virtually stabilize an unsteady camera, perhaps one that is handheld, on a drone or on a mobile robot traversing rough terrain. Capture a short image sequence $\boldsymbol{I}_1, \boldsymbol{I}_2 \cdots \boldsymbol{I}_N$ from an unsteady camera. For frame i, $i \geq 2$ estimate an homography with respect to frame 1, warp the image appropriately, and store it in an array. Display the stabilized image sequence using `ianimate`.

25. Bag of words (page 514)
    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.
    e) the RootSIFT trick described by Arandjelović and Zisserman (2012).
    f) SURF rather than SIFT features.
    g) SURF corner detector with BRISK or FREAK features.
26. Visual odometry, page 520. Modify the example script to
    a) use SIFT or SURF features instead of Harris. What happens to accuracy and execution time?
    b) ensure that features are more uniformly spread over the scene, investigate the `'suppress'` option of `icorner`.
    c) plot the fundamental matrix residuals at each time step (there are two of them). Is there a pattern here? Adjust the RANSAC parameters so as to reduce the number of times bundle adjustment fails.
    d) use a robust bundle adjuster, either find one or implement one (hard).
    e) use a Kalman filter with simple vehicle dynamics to smooth the velocity estimates.
27. Learn about kd-trees. What problems in this chapter could benefit from kd-trees?