— CM50248 —

Visual Understanding 1

Course Notes

Lecturer:
Dr Christian Richardt

Last updated: 2017-10-01

These notes were originally written by Prof. Peter Hall.
Updated by Dr Christian Richardt for 2017/2018.

Contents

Chapter 1

Basics

This section outlines the background material you are expected to know for this course.

1.1 Ways to think about images

It is natural to think that an image is a picture of something, but computer vision practitioners think of images in many different ways, depending on context. To begin with, the images we deal with come not just from standard cameras, but come from a vast array of different types: x-rays, depth images, infra-red; and from an even wider range of devices: MRI machines, submarines, planetary landers, cars, as well as ordinary cameras. Here we view consider images in a variety of ways.

1.1.1 Images as evidence

From a philosophical point of view, images are best regarded as *evidence* for something, rather than a picture of something. After all, any one image may be caused by many things: are we looking at a photograph of the Prime Minister, or a photograph of a waxwork? Fundamentally, this is (I think) the right way to look at images, because it allows for uncertainty. This view has lead to the rise of statistics in computer vision. Some examples of difficult images are shown in Figure 1.1.

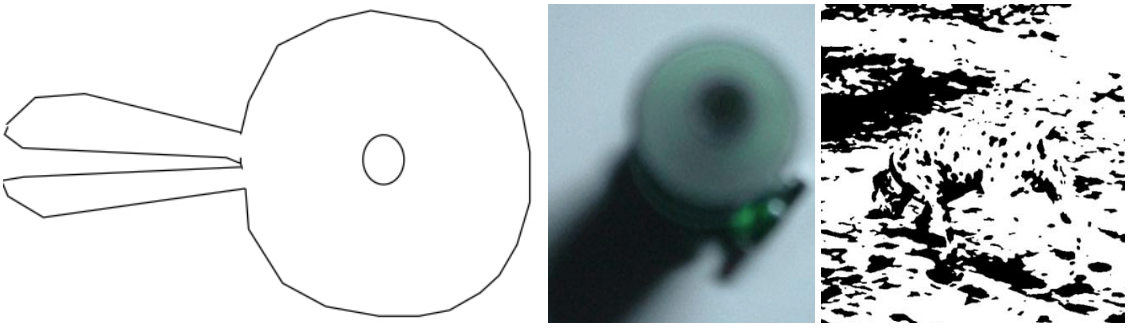

Figure 1.1: These pictures show that images can only ever be evidence for something, rather than a picture of something. Is the left image a rabbit or a duck? What every-day object is in the middle photograph — and would a different view be easier to understand? What is in the processed photograph on the right?

1.1.2 Images as pixels

Images are typically represented in a computer by a 2D array of colours or intensities. Intensity images (monochrome or greyscale images) usually have 8 bits per pixel. Full-colour images have three channels: for red, green and blue intensities. It is common to think of intensities ranging between zero and one. This view, seen in Figure 1.2, is needed when we want to store, input and output images.

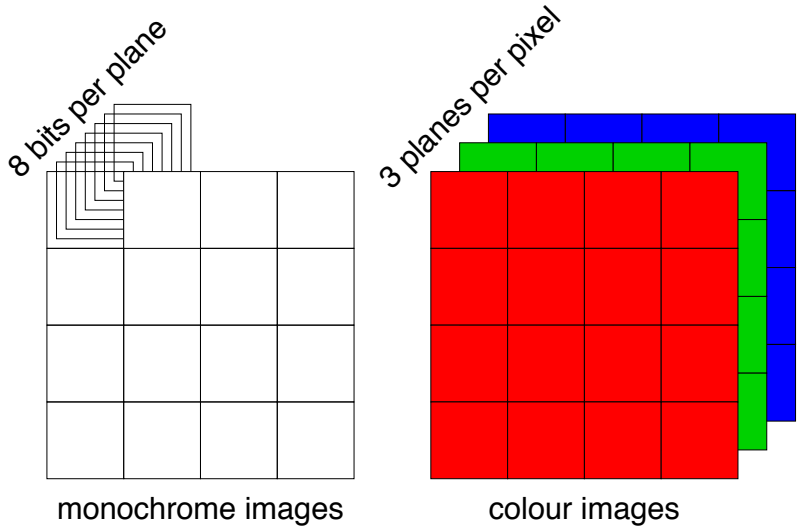

Figure 1.2: A schematic view of monochrome and colour images.

When we want to mathematically manipulate images (which is often), we think of them as functions. For monochrome images, this is intensity as a function of two spatial variables, $v = f(x, y)$, but for colour images, we need to think of a multi-valued function, $(r, g, b) = \mathbf{c}(x, y)$. It sometimes helps to think of an intensity image as a height field, that is as a surface; bright pixels being higher then darker ones, say.

1.1.3 Images as functions

1.1.4 Images as a sum of waves

Another way to think about images mathematically is as the sum of some other functions. The most common example of this is the *frequency domain* view, in which the image is built by summing sinusoids. This view of the image is rather abstract, but is very useful in understanding filtering processes, and is also used in applications such as compression.

1.1.5 Images as points

Yet another way to think of images is as points. To help see how this works think of an image of one pixel, whose intensity varies between 0 and 1. Given such a "picture", we can plot it on the $[0, 1]$ interval. Now consider a picture of two pixels. In this case, we have two axes: one for each pixel. Each point in the square $[0, 1]^2$ is a possible picture. Similarly, pictures with three pixels are points

in the cube $[0,1]^3$. Finally, pictures with N pixels are points in the *hyper-cube* $[0,1]^N$ – there is an axis for every pixel. This kind of representation can, perhaps surprisingly, be useful.

1.2 Windows, neighbourhoods, regions and patches

A **window** on an image is a small group of pixels contained within a geometric shape of some kind, usually square, sometimes circular, and occasionally other shapes.

A **neighbourhood** of a pixel comprises surrounding pixels – its neighbours. We can think of the 4-neighbours (north, east, south and west), or the 8-neighbours (also including north-east, north-west, south-east and south-west). More complex definitions are possible.

A **region** is an arbitrarily shaped collection of pixels. We can think of pixels in the region as "in", and all other pixels as "out". Two pixels A and B in a region are connected if we can move from one pixel by moving from pixel A to one of its "in" neighbours, from there to some other neighbour, and so on, to eventually arrive at the pixel B, having moved only through "in" pixels. Clearly, some paths may be allowing using 8-neighbours that are not possible using 4-neighbours, so we talk of *4-connectivity* and *8-connectivity*.

Chapter 2

Low-level vision

2.1 Linear filtering: convolution

Linear filters are a very important class of low level techniques. The basic idea behind all linear filters is very simple: form sums of weighted pixel values from a "source" image, and store the result in a "target" image. Linear filtering is simple, but surprisingly powerful. Linear filtering can be used to blur images, to "emboss" images, to detect edges, remove noise etc. Exactly the same weighted-sum process is used in each case, only the weights differ.

Consider the case where we have a $(2N+1)\times(2N+1)$ window; this is a square with an odd number of pixels on each side. The indices of each pixel will serve as its location. It is mathematically convenient to place the centre of the window at the origin, so we will allow negative indices. The weight at pixel (i,j) is $w(i,j)$. The window is placed over pixel (x,y) – that is, the central window pixel is aligned with the pixel at (x,y). The image value at this pixel is $f(x,y)$. The weighted sum is given by

$$g(x,y) = \sum_{j=y-N}^{y+N} \sum_{i=x-N}^{x+N} w(i,j) f(x-i, y-j), \tag{2.1}$$

where $g(x,y)$ is a pixel in the target image. Notice that this sum "flips" the window. This scheme is illustrated in Figure 2.1. Algorithm 2.1 gives pseudo-code; this pseudo-code is incomplete in several ways, in particular problems arise when the window "overlaps" the border of the image. How to handle border conditions is left as an exercise.

Algorithm 2.1 Pseudo-code for convolution.

```
INPUT: a source image and a window of weights

FOR each pixel in the source
    Locate window at source pixel, (x, y)
    sum = 0
    FOR each pixel (i, j) in window around (x, y)
        sum = sum + pixel(x-i, y-j) * weight(i, j)
    END
    target pixel(x,y) = sum
END

OUTPUT: a target image
```

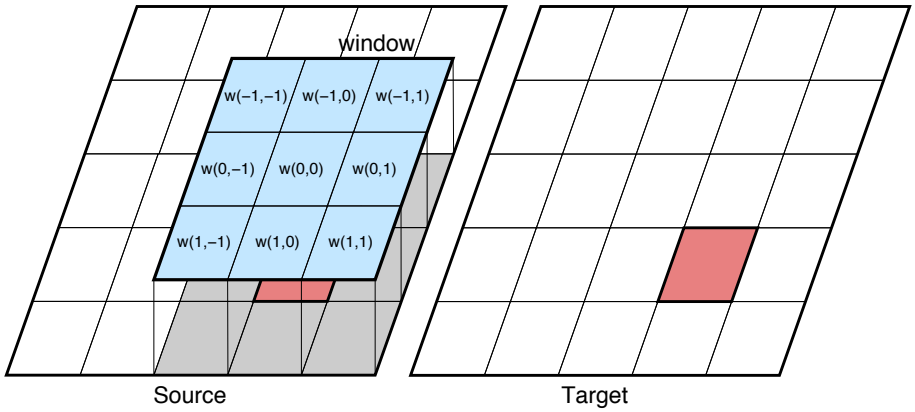

Figure 2.1: Linear filtering scheme. A window of weights is placed over a source image. A weighted sum of source pixels is stored in the target pixel.

This weighted-sum process just described is more formally called **convolution**. More properly, it is called *discrete* convolution, because it takes place at discrete locations. The continuous form of convolution uses integration rather than summation and is defined in 2D as

$$g(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(u,v)h(x-u,y-v)\,du\,dv, \tag{2.2}$$

in which $f(\cdot)$ is the image and $h(\cdot)$ the *convolution kernel*, which plays the role of the window of weights. This is often written as $f * h$, as a short-hand for the integration, so

$$g = f * h \tag{2.3}$$

means the same thing.

You may have spotted that the mathematical definition differs from the weighted-sum given above: the weighted sum uses kernel coordinates rather than image coordinates. A change of variables fixes this: set $u = x - i$, $v = y - j$, then

$$g(x,y) = \sum_{v=-N}^{N} \sum_{u=-N}^{N} f(u,v)w(x-u,y-v). \tag{2.4}$$

The reason for the first definition is purely pragmatic: if implemented as written, we would have to have a huge window, most of which was zero. It is sensible to take advantage of the fact that $f * g = g * f$, and so write code that loops only over the non-zero part of the window.

As mentioned, the effect of convolution depends on the kernel being used. Linear filters (also called "kernels", and "point-spread functions") can be specifically designed for particular tasks, but there are commonly used filters for common tasks such as blurring and edge detection. Simple filters for these tasks are shown in Figure 2.2. The blurring filter simply averages the pixel values inside a 3×3 window. The edge detectors subtract values to the left (above) from pixel values to the right (below); these values differ most where there is a discontinuity in brightness.

$$b = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad v = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad h = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.2: Three common linear filters (from left to right): average blurring, horizontal edges, and vertical edges.

The simple filters in Figure 2.2 are just a few of the many variants for blurring and edge detection. Better filters for the same tasks can be designed around the Gaussian function.

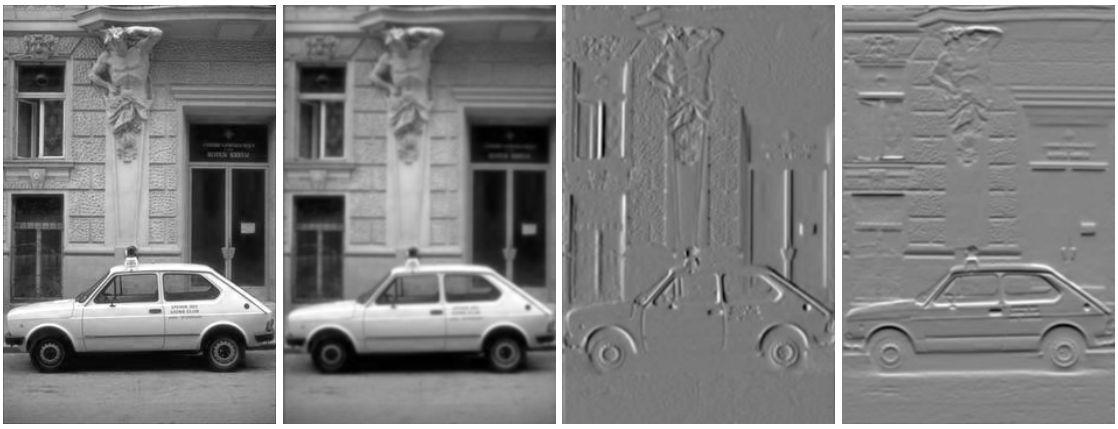

Figure 2.3: Some effects of linear filtering. From left to right: original photograph, blurred version, vertical edges, horizontal edges. These effects are made just by changing the weight values in the window (some weights need to be negative).

2.2 Gaussian filters and scale space

Gaussian-based filters are very common because:

1. they are easy to define,
2. they have useful analytic and practical properties,
3. they scale,
4. they steer.

Gaussians underpin *scale space*.

The two-dimensional radially symmetric Gaussian is defined by

$$g(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right). \tag{2.5}$$

The parameter σ governs the width of the Gaussian function. The normalising factor ensures the filter sums to unity.

The Gaussian can be used to blur images. In this case, more weight is placed on pixels near the centre of the filter. The amount of blur is controlled by σ; larger values yield wider Gaussians, and more blur. This is the basis of *scaling*, because as the degree of blur rises, only larger scale objects remain visible.

The first derivatives of the Gaussian can be used for edge detection, look for places with peak values of

$$\frac{\partial g(x,y)}{\partial x} = -\frac{x}{2\pi\sigma^4}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \tag{2.6}$$

$$\frac{\partial g(x,y)}{\partial y} = -\frac{y}{2\pi\sigma^4}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{2.7}$$

for vertical and horizontal edge detection, respectively. These detectors are tuned to pick up edges at the scale of σ. The total derivative in the direction θ is

$$\mathrm{d}g(x,y) = \frac{\partial g(x,y)}{\partial x}\cos(\theta) + \frac{\partial g(x,y)}{\partial y}\sin(\theta). \tag{2.8}$$

Rotating the edge detector to a given angle is called *steering*.

Higher-order derivatives can also be used as filter kernels.

2.2.1 Scale space

Objects in images have *scale*, which relates to their size in pixels. Gaussian blurring can be thought of as suppressing objects with a scale of less than σ. This leads directly to the idea of a *scale space*, which has an input image at 'the bottom' ($\sigma = 0$) and filtered images appear up the 'scale' axis, as in Figure 2.4.

Since small objects no longer appear in filtered images, it is common to reduce the size of filtered images, giving a 'pyramid', as in Figure 2.4.

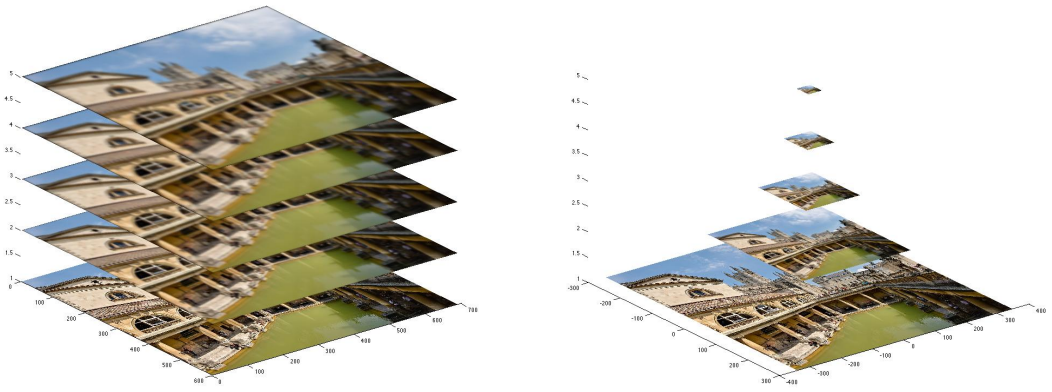

Figure 2.4: **Left:** Gaussian scale space. **Right:** A Gaussian pyramid.

2.3 The Fourier transform

To better understand the effects of linear filtering, we must consider the frequency domain representation of images. This will also help us understand aliasing (and anti-aliasing) in computer graphics, as well as give meaning to terms such as *low-pass* and *high-pass* filtering. Filtering in the frequency domain can often be faster than spatial-domain filtering. Finally, an understanding of the frequency domain serves as a starting point for understanding wavelets (an advanced topic we will not discuss).

The idea is simple. First think of a monochrome image as a surface. Next realise this surface can be made by summing many 'corrugated' surfaces, that is sinusoids with a given frequency and direction. If we choose one set of a corrugations, we get a particular greyscale picture. If we choose another set of corrugates, we get a different picture. In fact, each picture has a unique set of corrugates, and each set of corrugates gives a unique picture.

Here is an analogy. Think about the surface of a pond, which can be made to undulate in complicated ways by throwing in stones – the waves from each stone add together to make the pond's surface. This is called wave *superposition*. The gray-level surface of a picture can be though of a pond surface captured at an instant in time, and the corrugate surfaces are the waves made by different corrugates varying in frequency, amplitude, and phase. We can easily get a new picture by varying any of these properties. The first three of these properties are shown in Figure 2.5 for a one-dimensional corrugate, which is a sinusoid; we cannot show "angle" for one-dimensional corrugates.

We can express this mathematically using

$$f(x,y) = \sum_{i=1}^{N} a_i \sin(2\pi(xu_i + yv_i) + \phi_i), \tag{2.9}$$

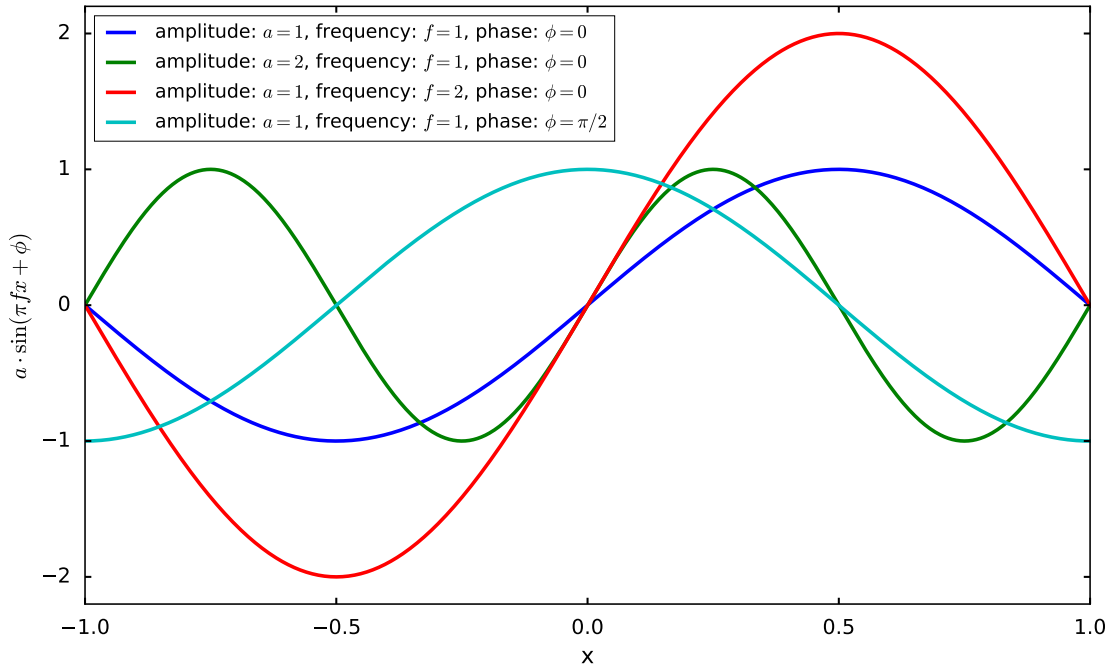

Figure 2.5: Properties of a one-dimensional sinusoid. The base wave is shown in blue with variations in other colours.

where $f(x,y)$ is the value of the image at the pixel (x,y), made by summing N corrugates. The i^{th} corrugate has amplitude a_i and phase ϕ_i. The frequency in the x-direction is u_i and in the y-direction is v_i; taken together, these give a corrugate of frequency (xu_i+yv_i) that travels at an angle $\arctan(v_i/u_i)$ to the x-axis (compare this to the vector dot product).

It turns out that each and every image has exactly one set of corrugates which, when added, make the image. One way to specify the corrugates we are using is to use a matrix. The position of each matrix element is considered as the vector $[u,v]^\top$, which fixes direction and frequency. So the matrix represents the uv-plane, as seen in Figure 2.6. We want two values in the matrix element, one to give the amplitude and the other the phase. This is done using complex numbers written as $a\exp(i\phi)$.

The **Fourier transform** decomposes an image, $f(x,y)$ into its corrugates. It gives a new function $F(u,v)$, which is the continuous analogue to the matrix we've just discussed. The $[u,v]^\top$ position specifies a corrugate, just as before, and the value at each point is complex.

$$F(u,v) = \int_{-\infty}^{\infty} f(x,y)\exp(-i2\pi(xu+yv))\,\mathrm{d}x\,\mathrm{d}y \tag{2.10}$$

The inverse Fourier transform reverses the process – it takes the corrugates and adds them to make a picture. In general, this picture will have complex values too, but in practice, pictures are real values (the imaginary component is zero).

$$f(x,y) = \int_{-\infty}^{\infty} F(u,v)\exp(i2\pi(xu+yv))\,\mathrm{d}u\,\mathrm{d}v \tag{2.11}$$

The Fourier transforms of several pictures are shown in Figure 2.7.

As it turns out, any linear filtering we can perform in the spatial domain using convolution, we can also perform in the frequency domain. In fact, convolution and the Fourier transform are related via the **convolution theorem** which is this: Suppose f and g are functions with Fourier transforms F

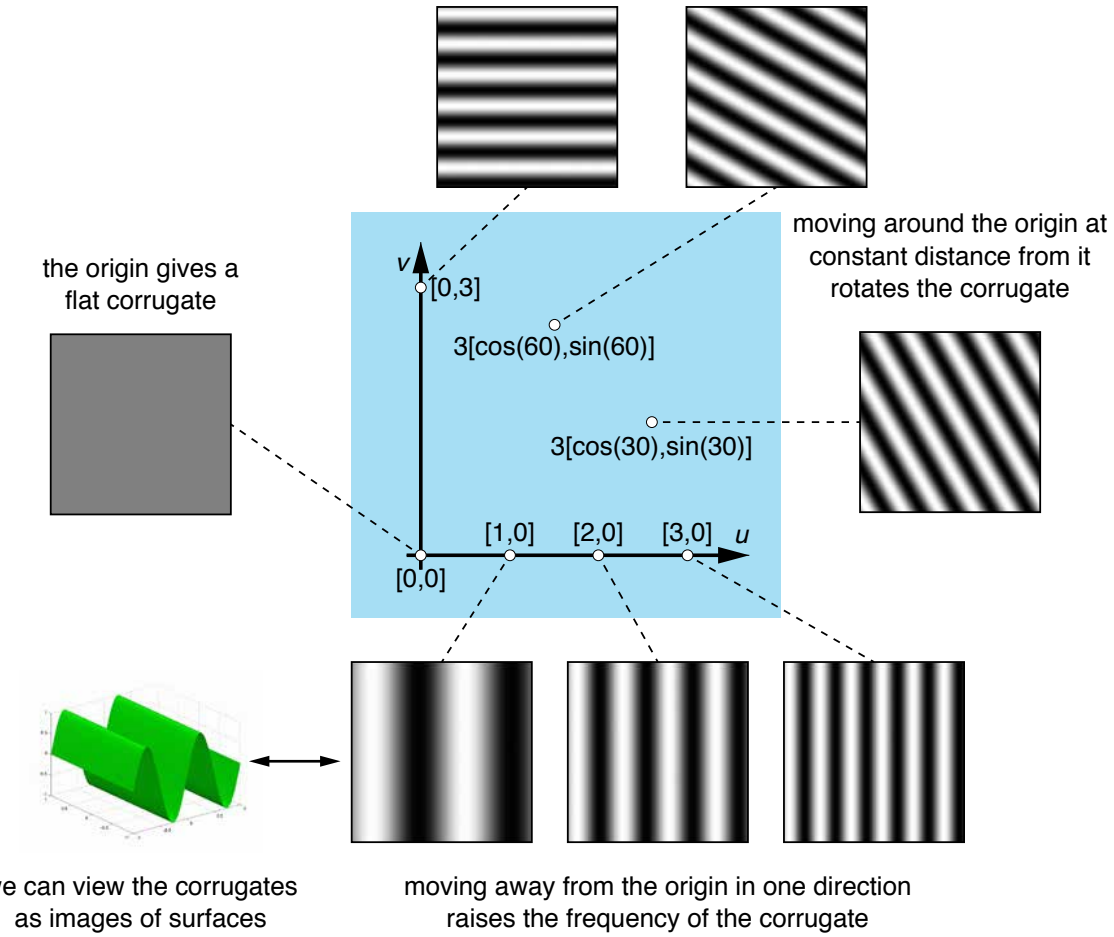

Figure 2.6: A diagram to show how different corrugates relate to the uv-plane. Here, all amplitudes are set to unity, and all phases to zero. Notice how the dots in the uv-plane make a picture.

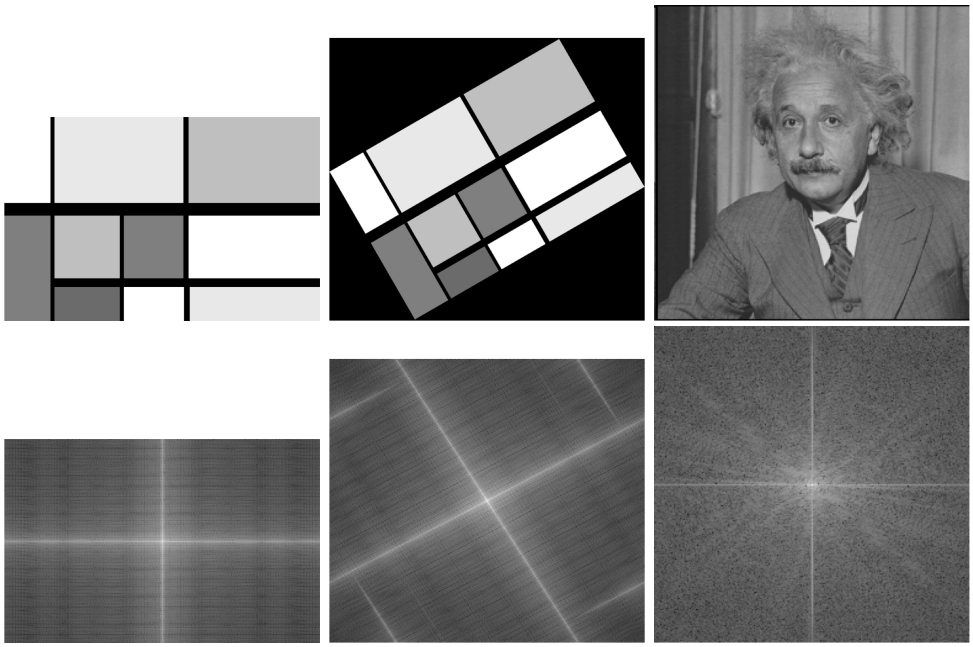

Figure 2.7: A set of pictures (top) and their Fourier transforms (below). Only the absolute part of shown.

and G, respectively, written as $f \leftrightarrow F$ and $g \leftrightarrow G$, then

$$\mathcal{F}[f * g] \leftrightarrow F \cdot G \tag{2.12}$$

Otherwise said: the Fourier transform of the convolution of f and g is the product of the Fourier transforms of the individual functions. Note the product of the functions is *not* matrix multiplication – rather corresponding (complex) values of the functions are multiplied.

So, if we want to convolve with some window we can, if we want, apply the Fourier transform to the window (kernel) and take the inverse Fourier transform of their product. This may seem a "long way around", and so it is, but is can often take *less* time than standard convolution.

Chapter 3

Features

3.1 Canny Edges

If I_x and I_y are image derivatives (typically, the output of an image and a derivative filter), then

$$r = \sqrt{I_x^2 + I_y^2} \tag{3.1}$$

$$\theta = \arctan(I_y / I_x) \tag{3.2}$$

are the 'magnitude' and 'orientation' of a gradient.

But, the 'edges' found are really only image gradients, they are not semantic in the sense of dividing one part of an image from another part. Canny edges [1986] are one step towards that. Canny edges are named after John Canny, who invented an algorithm to locate edges in images.

Canny's algorithm turns a greyscale gradient magnitude map into a binary image, in which bright pixels denote edge locations. It makes use of the gradient information by 'non-maximum suppression': pick a pixel, move outward into a window along the direction of the gradient. Set any pixels with strength value lower than the maximum to zero.

Following this, the algorithm traces along the remaining pixels. Some of these are noise (high-pass filters amplify noise), others are genuine edges; noise can be stronger than edges so a simple threshold in impossible. Instead, Canny uses two thresholds: the first is high and is used to identify the very strongest edges. Tracing along these, we find weaker edges, which we accept provided they are above the low threshold. The pixels visited are all set to 1, and the rest are set to 0. Hence the binary output, as in Figure 3.1.

3.2 Harris/Stephens corners

So called *Harris features* [Harris and Stephens, 1988] are commonly used to detect corners, as in Figure 3.2. The algorithm uses a matrix of derivatives:

$$\mathbf{A} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}. \tag{3.3}$$

Now any square matrix can be written as a product like this:

$$\mathbf{A} = \mathbf{U}\mathbf{L}\mathbf{U}^{-1}, \tag{3.4}$$

Figure 3.1: A picture and its Canny edge map.

in which the matrix $\mathbf{U}$ is orthonormal and $\mathbf{L}$ is diagonal. These are the eigenvectors and eigenvalues, respectively. But these are expensive to compute and need not be explicitly computed, because

$$M_c = L_1 L_2 - \kappa(L_1 + L_2)^2 \tag{3.5}$$

$$= \det(\mathbf{A}) - \kappa \operatorname{trace}^2(\mathbf{A}) \tag{3.6}$$

is a number that gives a numerical measure of the "cornerness" at the point. The determinant and trace are easy to compute, in the specific case above:

$$\det(\mathbf{A}) = Ix^2 I_y^2 - I_{xy}^2, \tag{3.7}$$

$$\operatorname{trace}(\mathbf{A}) = I_x^2 + I_y^2. \tag{3.8}$$

The κ is a parameter of the algorithm that is usually chosen between 0.04 and 0.15.

Harris corners are used in many applications, especially where *matching* is required. For example, the three dimensional shape of objects can be reconstructed from a pair of pictures, if points in them can be corresponded (matched). By using Harris corners, the matching process is made a little easier.

3.3 Texture

Within computer vision, the term 'texture' refers to a patch of image that depicts things like stones on beach, a wall, leaves, a patterned jumper, etc.

One way to characterise a texture is through its response to a *filter bank*. A filter bank comprises a collection of filters at different scales, and of different order of differentiation, angle, and so on. When a filter bank is applied to an image the output at a pixel is a vector of responses, one response per filter. This *feature vector* can be thought of as a point in *feature space*. Pixels of similar texture crowd together in this feature space, which defines a texture.

3.4 SIFT

The scale-invariant feature transform, short SIFT, by Lowe [2004], is a commonly used feature, often used in place of Harris corners, because they are more robust. SIFT uses a feature detector based

Figure 3.2: A photograph with detected Harris corners.

on a difference-of-Gaussians scale space (Section 2.2.1), and describes features using a histogram of oriented gradients.

Chapter 4

Cameras and 3D reconstruction

4.1 Projection

Projection in Computer Vision refers to the process by which a 2D picture of the 3D real world is formed. This inevitably implies a loss of information; in particular the distance of objects from the viewing system is lost. However, if a pair of images is used, then the location of points in 3D can be recovered; a process known as **3D reconstruction**. To do this requires a detailed understanding of cameras. Different applications require different information, and make different assumptions, so it is not surprising that many models of cameras have been developed. Forsyth and Ponce [2011] give details of some of these, and mention real-world effects such a lens distortion that are sometimes important.

The *perspective* (or *pinhole*) camera is a very simple camera that is easy to model, and often suffices as a good approximation to reality. In the perspective camera, rays of light travel in straight from points in the scene through a focal point (the pinhole). The light falls onto a flat surface behind the focus, and may be recorded on film or digital media. From a mathematical points of view, the flat surface could just as well be in front of the pin hole, as Figure 4.1 shows.

4.1.1 A perspective camera

Perspective projection is perhaps the most common model of projection. We begin modelling perspective projection using similar triangles. The underlying assumption is that a ray of light travels in a straight line. So a ray of light leaving some real-world point and travelling toward the focal point. At some point along its path, the light ray passes through an image plane, and in doing so makes an image of the real-world point on the image plane. Now consider the line from the focal point in a direction perpendicular to the image plane. Measure the distance from the focal point to the image plane along this line, call this distance w. Also measure the perpendicular distance of the real-world point to the line, call this H; and the distance from the focal point to the foot of the points' perpendicular, W. The distance from the image in the image plane to the reference line is $h = Hw/W$, obtained using similar triangles. This is clear to see in Figure 4.1, where the diagram has been drawn using a convenient "side on" view.

We can place the image plane anywhere in space, except that the plane should not contain the focal point. Any ray that is not parallel to the plane will pierce it somewhere, and any point that lies on a particular ray appears on the plane at that location.

The above is an informal description of a *projective space*. It is clear that any 3D point can be located on the plane just by scaling the ray that passes through it. If fact, if the plane is at unit distance from

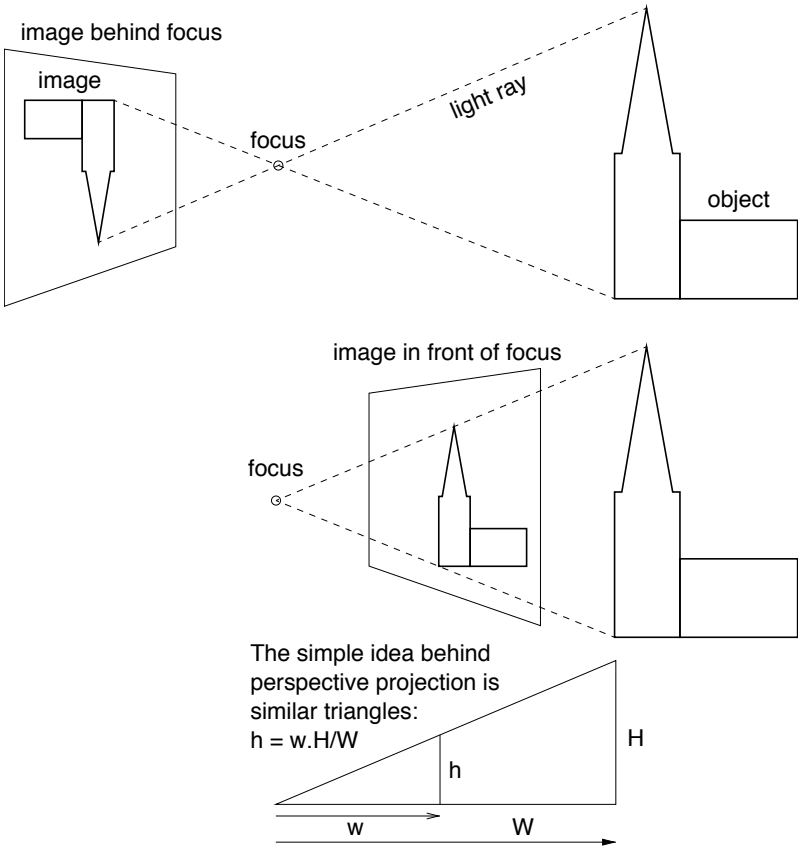

Figure 4.1: A perspective camera.

the focal point, then it is enough to scale the ray by the distance from the focus to the point; more exactly by the perpendicular distance to the plane. If the plane is conveniently set to to be parallel to the xy-plane, projection becomes no more than division by the z distance of the point. That is if $[x, y, z]^\top$ is the point, then under the canonical camera just described, it appears to be at

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \end{bmatrix}. \tag{4.1}$$

This canonical projection can be written using *homogeneous coordinates*. The homogeneous coordinates of a 3D point $[x, y, z]$ are any 4-vector $[wx, wy, wz, w]$, provided $w \neq 0$. The w value just scales the point along the ray to the focal point. Likewise, the homogeneous coordinates for a 2D point $[u, v]$ are all points $[uw, vw, w]$. If $w = 0$, the corresponding non-homogeneous point is $[x/0, y/0] = [\infty, \infty]$.

Given homogeneous coordinates (i.e. points in projective space), we can write perspective projection as

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{4.2}$$

Note that the input and output points are in projective space, and to map the output into a Euclidean space, we scale by $1/w$. Notice also that the projection matrix is singular – it has no inverse. It is a rank 3 matrix (the rank of a matrix is the number of linearly independent rows or columns). This feature is characteristic of all projection matrices, and reflects the fact that three dimensions are being "compressed" into two – depth information is lost.

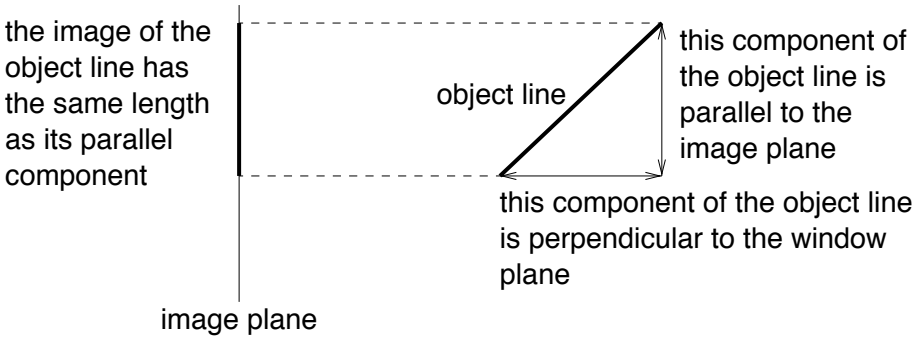

Figure 4.2: A side view of a basic affine camera.

This form of writing projection is especially convenient. It allows the camera to be translated by by a vector $-\mathbf{t}$, and rotated by a matrix $\mathbf{R}^\top$ in 3D space. Equivalently, the points are all translated by $\mathbf{t}$ and rotated by $\mathbf{R}$. What is more, once projected, the image points can be moved, scaled etc., which allows for characteristics of the camera such as non-square pixels; a 3×3 matrix $\mathbf{K}$ is used for this. The (pinhole) perspective camera is therefore modelled as a 3×4 matrix

$$\mathbf{P} = \mathbf{K}\left[\mathbf{R} \mid \mathbf{t}\right]. \tag{4.3}$$

If we set $\mathbf{R}=\mathbf{I}$, $\mathbf{t}=\mathbf{0}$, $\mathbf{K}=\mathbf{I}$, we have the canonical projection already given. Then

$$\mathbf{y} = \mathbf{P}\mathbf{x} \tag{4.4}$$

is projection, up to a scale.

The terms $\mathbf{R}$ and $\mathbf{t}$ are called the *extrinsic* parameters, whereas $\mathbf{K}$ houses the *intrinsic* parameters. To reconstruct (more exactly, to estimate a Euclidean reconstruction), we need to be know the extrinsic and intrinsic parameters.

4.1.2 An Affine Camera

Affine cameras assume parallel light rays that pierce the image plane. Now the canonical camera is

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{4.5}$$

In some texts (typically computer graphics texts), this is called *orthogonal* or *orthographic* projection. More generally,

$$\mathbf{P} = \mathbf{A}\mathbf{K}\left[\mathbf{R} \mid \mathbf{t}\right], \tag{4.6}$$

with

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{4.7}$$

Affine transforms preserve length. Affine cameras hence preserve the length of vector components that are parallel to the image plane, because the ray of light are all assumed to be orthogonal to it (hence the alternative names). A simple affine camera is shown in Figure 4.2. Affine cameras have rank two.

4.1.3 Camera Calibration

For many computer vision tasks, it is important to calibrate the camera, to determine its internal parameters. This requires an estimate of the projection matrix $\mathbf{P}$ using real-world data. To do this, it is very convenient to use a *calibration rig*, which is just a prop of known size and shape, such as a checkerboard or the inner corners of a cube.

We first write the general perspective transform (Equation 4.4) as three rows (each row $\mathbf{p}_i^\top$ is written as the transpose of a column vector $\mathbf{p}_i$):

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1^\top \\ \mathbf{p}_2^\top \\ \mathbf{p}_3^\top \end{bmatrix}, \tag{4.8}$$

in which case an object point $\mathbf{x}$ is projected to

$$u = \frac{\mathbf{p}_1^\top \mathbf{x}}{\mathbf{p}_3^\top \mathbf{x}}, \tag{4.9}$$

$$v = \frac{\mathbf{p}_2^\top \mathbf{x}}{\mathbf{p}_3^\top \mathbf{x}}, \tag{4.10}$$

which can be re-arranged to read

$$\mathbf{p}_1^\top \mathbf{x} - u\mathbf{p}_3^\top \mathbf{x} = 0, \tag{4.11}$$

$$\mathbf{p}_2^\top \mathbf{x} - v\mathbf{p}_3^\top \mathbf{x} = 0. \tag{4.12}$$

This can be written in matrix form as

$$\begin{bmatrix} \mathbf{x}^\top & \mathbf{0}^\top & -u\mathbf{x}^\top \\ \mathbf{0}^\top & \mathbf{x}^\top & -v\mathbf{x}^\top \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix}, \tag{4.13}$$

which usefully separates the unknowns into a column vector of 12 elements. The above is for a single object point, $\mathbf{x}$ and its image $[u, v]$, which clearly is not sufficient to determine the twelve unknowns of the perspective transform matrix (now written as a column vector). If we use at least six object points, we can estimate $\mathbf{P}$. Each new object point and its image generate a new pair of rows in the matrix above, giving

$$\begin{bmatrix} \mathbf{x}_1^\top & \mathbf{0}^\top & -u_1\mathbf{x}_1^\top \\ \mathbf{0}^\top & \mathbf{x}_1^\top & -v_1\mathbf{x}_1^\top \\ \mathbf{x}_2^\top & \mathbf{0}^\top & -u_2\mathbf{x}_2^\top \\ \mathbf{0}^\top & \mathbf{x}_2^\top & -v_2\mathbf{x}_2^\top \\ \vdots & \vdots & \vdots \\ \mathbf{x}_n^\top & \mathbf{0}^\top & -u_n\mathbf{x}_n^\top \\ \mathbf{0}^\top & \mathbf{x}_n^\top & -v_n\mathbf{x}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{4.14}$$

where n is the number of object point and image point pairs we have used. We write the final system as

$$\mathbf{XP} = \mathbf{0}. \tag{4.15}$$

This is a *homogeneous* linear system. Provided $n \geq 6$, we can find a solution in the *least-squares sense*. The easy way to do this is to use *singular value decomposition* (SVD). Any matrix $\mathbf{X}$ can be written as a product:

$$\mathbf{X} = \mathbf{USV}^\top, \tag{4.16}$$

in which $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices, where the columns are called the left and right singular vectors, and $\mathbf{S}$ is diagonal, comprising the singular values. The i^{th} singular value s_{ii} is associated with the i^{th} left singular vector, and the i^{th} right singular vector. The size of s_{ii} indicates the magnitude of its corresponding vector, and in particular

$$\mathbf{XV} = \mathbf{US}, \tag{4.17}$$

$$\mathbf{X}\begin{bmatrix}\mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n\end{bmatrix} = \begin{bmatrix}s_{11}\mathbf{u}_1 & s_{22}\mathbf{u}_2 & \dots & s_{nn}\mathbf{u}_n\end{bmatrix}. \tag{4.18}$$

Which means $\mathbf{X}\mathbf{v}_i = s_{ii}\mathbf{u}_i$. Clearly, the singular value s_{ii} denotes the magnitude of the product, and if $s_{ii} = 0$, we will have found a (non trivial) solution to $\mathbf{XP} = \mathbf{0}$. Usually, though, no singular value is zero – in which case we find the smallest s_{ii} and use the corresponding $\mathbf{v}_i$ as a solution. This is, in fact, the optimal solution. It corresponds to fitting a hyper-plane through the multidimensional points that are the columns of $\mathbf{X}$. If these points lay exactly on a hyper plane, we obtain an exact solution, so $s_{ii} = 0$, otherwise we minimise the sum of squared distance from each point to the fitted plane – we optimise in the least-squares sense.

Forsyth and Ponce [2002] discuss least-squares fitting (pages 39–42). Press et al. [2007] discuss SVD, and some of its uses, including but not limited to least-squares fitting. Golub and Van Loan [2013] also discuss SVD, and is well worth reading.

Having obtained a projection matrix $\mathbf{P}$, we need to separate out the intrinsic and extrinsic parameters. In doing so, we remember that because we have solved a homogeneous system, we can know $\mathbf{P}$ only up to a scale factor, say ρ. Now, writing each row as $[\mathbf{q}^\top z]$, we have

$$\rho\begin{bmatrix}\mathbf{p}_1^\top \\ \mathbf{p}_2^\top \\ \mathbf{p}_3^\top\end{bmatrix} = \rho\begin{bmatrix}\mathbf{q}_1^\top z_1 \\ \mathbf{q}_2^\top z_2 \\ \mathbf{q}_3^\top z_3\end{bmatrix} = \begin{bmatrix}(\alpha\mathbf{r}_1^\top - \alpha\cot(\theta)\mathbf{r}_2^\top + a_1\mathbf{r}_3^\top) & (\alpha t_1 - \alpha\cot(\theta)t_2 + a_1 t_3) \\ \left(\frac{\beta}{\sin(\theta)}\mathbf{r}_2^\top + a_2\mathbf{r}_3^\top\right) & \left(\frac{\beta}{\sin(\theta)}t_2 + a_2 t_3\right) \\ \mathbf{r}_3^\top & t_3\end{bmatrix}. \tag{4.19}$$

And we see

$$\rho = \frac{\pm 1}{\|\mathbf{q}_3\|}. \tag{4.20}$$

We obtain the intrinsic parameters as

$$a_1 = \rho^2\mathbf{q}_1^\top\mathbf{q}_3 \tag{4.21}$$

$$a_2 = \rho^2\mathbf{q}_2^\top\mathbf{q}_3 \tag{4.22}$$

$$\cos(\theta) = -\frac{(\mathbf{q}_1\times\mathbf{q}_3)^\top(\mathbf{q}_2\times\mathbf{q}_3)}{\|\mathbf{q}_1\times\mathbf{q}_3\|\,\|\mathbf{q}_2\times\mathbf{q}_3\|} \tag{4.23}$$

$$\alpha = \rho^2\|\mathbf{q}_1\times\mathbf{q}_3\|\sin(\theta) \tag{4.24}$$

$$\beta = \rho^2\|\mathbf{q}_2\times\mathbf{q}_3\|\sin(\theta) \tag{4.25}$$

The extrinsic parameters are:

$$\mathbf{r}_3 = \rho\mathbf{q}_3 \tag{4.26}$$

$$\mathbf{r}_1 = \frac{\mathbf{q}_2\times\mathbf{q}_3}{\|\mathbf{q}_2\times\mathbf{q}_3\|} \tag{4.27}$$

$$\mathbf{r}_2 = \mathbf{r}_3\times\mathbf{r}_1 \tag{4.28}$$

$$\mathbf{t} = \rho\mathbf{P}_{\text{int}}^{-1}\mathbf{z} \tag{4.29}$$

where $\mathbf{z} = [z_1, z_2, z_3]^\top$ and $\mathbf{P}_{\text{int}}$ is the newly acquired projection matrix that characterises the intrinsic parametrisation of the camera.

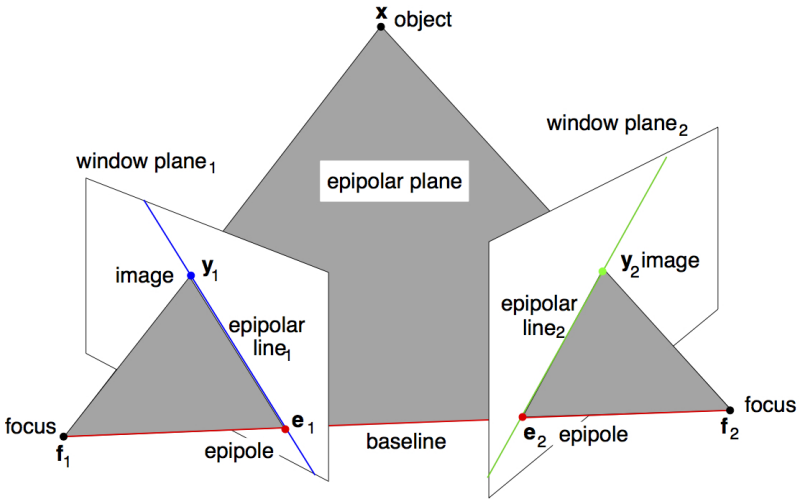

Figure 4.3: Epipolar geometry: the two foci, the object, and its two images **must** all lie in the same plane.

4.2 Epipolar geometry

Epipolar geometry is the name given to the necessary geometric arrangements that exist in the two-camera case, as illustrated in Figure 4.3. The *epipolar plane* is defined by the two foci of the cameras and the object point $\mathbf{x}$, and is of infinite extent. Each window plane[1] (that are of infinite extent, the window being a small region within the plane) intersects the epipolar plane to create an *epipolar line*. The image $\mathbf{y}_i$ of the point in the window must lie on the epipolar line. The line joining the foci is the *baseline*. This line intersects each window plane to create an *epipole* $\mathbf{e}_i$.

The *epipolar constraints* imposed by the geometry mean this: given the focal point of each camera and the image $\mathbf{y}_1$ of some object point $\mathbf{x}$ in camera one, then the corresponding image $\mathbf{y}_2$ in camera two must lie on the epipolar line in the other camera's window. We may not be able to say exactly where on the line – this depends on the distance to the object from the "known" camera, but we can say it definitely does lie on the corresponding epipolar line. This is shown in Figure 4.4 (left).

Note that a different object point that is not on the "current" epipolar plane generates a whole new epipolar plane – and that these planes intersect at in a line which is the baseline. In fact, we can think of the baseline as an axis about which the epipolar planes rotate. Furthermore, the set of epipolar lines in a window will rotate about that window's epipole. This too is shown in Figure 4.4 (right).

The epipolar geometry imposes constraints: the baseline vector $(\mathbf{f_2}-\mathbf{f_1})$, and the lines to each of the images $(\mathbf{y_1}-\mathbf{f_1})$, and $(\mathbf{y_2}-\mathbf{f_2})$ are coplanar (they lie in the epipolar plane, in fact). The normal to the epipolar plane is $(\mathbf{f_2}-\mathbf{f_1}) \times (\mathbf{y_2}-\mathbf{f_2})$, and this must be orthogonal to $(\mathbf{y_1}-\mathbf{f_1})$, so

$$(\mathbf{y_1}-\mathbf{f_1})^\top \big((\mathbf{f_2}-\mathbf{f_1}) \times (\mathbf{y_2}-\mathbf{f_2})\big) = 0, \tag{4.30}$$

which is independent of any coordinate system. This relation is put to use in constructing both the essential matrix and the fundamental matrix.

4.2.1 Essential matrix and fundamental matrix

The *essential matrix* is the name given to a matrix that enforces the epipolar constraint in the case of calibrated cameras.

[1] The 'window plane' is more commonly known as the 'image plane'.

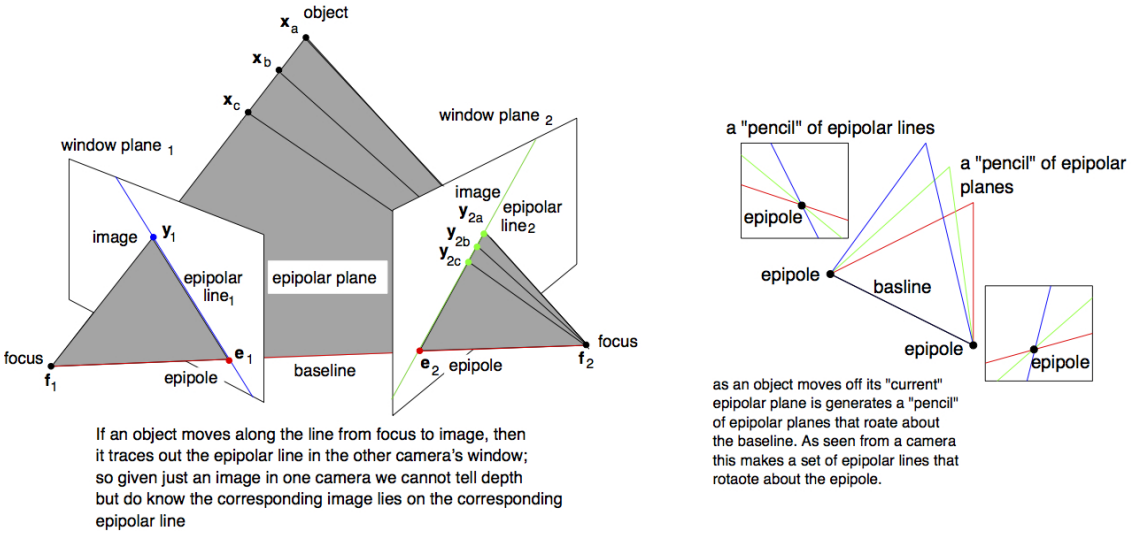

Figure 4.4: The effect of different points in the same line of view **(left)**, and different points at arbitrary locations in 3D **(right)**.

In practice, we observe points in images: that is, we can measure the coordinates of a point in an image, which are the $\mathbf{y}$ of Equation 4.4, for example. We would like the coordinates of this point in the 'real' world. We remind ourselves that (using $\mathbf{z} = [\mathbf{R} \mid \mathbf{t}]\,\mathbf{x}$)

$$\mathbf{y} = \frac{1}{s}\mathbf{K}\mathbf{z}, \tag{4.31}$$

where $\mathbf{K}$ is the 3×3 intrinsic calibration matrix, and $\mathbf{z}$ is where the light from an object $\mathbf{x}$ to the focus pierces the window (image) plane, at a distance s from the camera, and $\mathbf{y}$ is the image point as seen by the camera. So, if we are to place this image point on the window plane, we need the calibration parameters of the camera, that is we need $\mathbf{K}$ and the focal length s. How to determine the calibration matrix for a real camera has already been discussed in Section 4.1.3, but this does not yield the depth s. Fortunately, this does not matter, because we can always scale the camera to an equivalent one (takes exactly the same photo's) which has unit focal length. Hence we can write

$$\mathbf{z}' = \mathbf{K}^{-1}\mathbf{y} \tag{4.32}$$

to obtain a point in 3D that is equivalent to the image point in the camera.

We must bear in mind that this newly computed $\mathbf{z}'$ is in a reference frame rigidly attached to the camera. This means that the $\mathbf{z}_1'$ is as seen from camera 1 – it is just scaled, sheared and shifted when compared to $\mathbf{y}$. Similar remarks apply to $\mathbf{z}_2'$. We must transform $\mathbf{z}_1'$ and $\mathbf{z}_2'$ into a common coordinate system, before we can use the epipolar constraint. We can choose any coordinate system we like – that attached to camera 1 is convenient. In this case, we set $\mathbf{z}_1 = \mathbf{z}_1'$ and $\mathbf{z}_2 = \mathbf{R}\mathbf{z}_2 - \mathbf{t}$. The translation is just the distance between the foci – which defines the baseline, so $\mathbf{t} = \mathbf{f}_2 - \mathbf{f}_1$. The rotation $\mathbf{R}$ will bring the viewing direction of camera 2 into line with that of camera 1: it maps from the camera 2 system into the camera 1 system.

Now, putting all this together, the epipolar constraint can be written:

$$\mathbf{z}_1^{\top}(\mathbf{t} \times \mathbf{z}_2) = 0 \tag{4.33}$$

$$\mathbf{z}_1'^{T}(\mathbf{t} \times (\mathbf{R}(\mathbf{z}_2' - \mathbf{t})) = 0 \quad \text{after substitution to get local coordinates} \tag{4.34}$$

$$\mathbf{z}_1'^{\top}(\mathbf{t} \times \mathbf{R}(\mathbf{z}_2') = 0 \quad \text{because } \mathbf{t} \times \mathbf{t} = \mathbf{0} \tag{4.35}$$

$$\mathbf{y}_1^{\top}\mathbf{K}_1^{-T}(\mathbf{t} \times \mathbf{R}(\mathbf{K}_2^{-1}\mathbf{y}_2)) = 0. \tag{4.36}$$

We would like this in matrix form. We can achieve our aim by noticing that any cross-product can be written as a skew-symmetric matrix; for example

$$\mathbf{x} \times \mathbf{z} = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}, \tag{4.37}$$

the matrix used in the cross-product, is often denoted $[\mathbf{x}_\times]$. We take advantage of this to write the epipolar constraint as

$$\mathbf{y}_1^\top \mathbf{K}_1'^\top \mathbf{E} \mathbf{K}_2^{-1} \mathbf{y}_2 = 0, \tag{4.38}$$

in which $\mathbf{E} = [\mathbf{t}_\times]\mathbf{R}$ is the **essential matrix**. If the calibration matrices are unknown, we write

$$\mathbf{y}_1^\top \mathbf{F} \mathbf{y}_2 = 0, \tag{4.39}$$

where $\mathbf{F} = \mathbf{K}_1^{-\top} \mathbf{E} \mathbf{K}_2^{-1}$ is the **fundamental matrix**.

Both equations can be given the same geometrical interpretation, we use the fundamental matrix to illustrate. The equation of a line in the plane in $ax + by + c = 0$, which can be written as $[x, y, 1][a, b, c]^\top = 0$. If we set $[a, b, c]^\top = \mathbf{F}\mathbf{p}_2$, then we see $\mathbf{F}\mathbf{y}_2$ contain line parameters a, b and c. The line is, in fact, the epipolar line corresponding to $\mathbf{y}_2$. Similarly, $\mathbf{F}^\top \mathbf{y}_1$ is the epipolar line in the window of the second camera.

Notice that each of the epipoles makes a degenerate line in the other camera (i.e. they appear to be a point). To see this, we observe

$$\mathbf{F}^\top \mathbf{e}_2 = \mathbf{R}[\mathbf{t}_\times]\mathbf{e}_2 = 0 \tag{4.40}$$

since $\mathbf{t}$ and $\mathbf{e}_2$ are parallel. Similarly, $\mathbf{F}\mathbf{e}_1 = 0$. Since $\mathbf{e} \neq \mathbf{0}$, it follow that $\mathbf{e}$ is in the null space of $\mathbf{F}$, and hence $\mathbf{F}$ must be singular.

4.2.2 Determining scene geometry: geometric calibration

It is possible to determine the fundamental matrix from a pair of images, from at least eight point correspondences. This is a process known as *weak calibration*. Let the i^{th} pair of corresponding points be $[u_i, v_i, 1]^\top$ and $[u_i', v_i', 1]^\top$. The epipolar constraint is expressed through the fundamental matrix as:

$$\begin{bmatrix} u_i & v_i & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_i' \\ v_i' \\ 1 \end{bmatrix} = 0. \tag{4.41}$$

We want to determine the f_{jk}, and so are motivated to write the above with these components as a vector:

$$\begin{bmatrix} u_i u_i' & u_i v_i' & u_i & v_i u_i' & v_i v_i' & v_i & u_i' & v_i' & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0. \tag{4.42}$$

Now, this is a homogeneous equation, and as a consequence the values of f_{jk} can be scaled arbitrarily. Therefore, choosing a scale in which $f_{33} = 1$ is a perfectly acceptable solution. This gives

$$\begin{bmatrix} u_i u'_i & u_i v'_i & u_i & v_i u'_i & v_i v'_i & v_i & u'_i & v'_i \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \end{bmatrix} = -1. \tag{4.43}$$

This allows us to express the epipolar constraint for exactly eight point correspondences as

$$\begin{bmatrix} u_1 u'_1 & u_1 v'_1 & u_1 & v_1 u'_1 & v_1 v'_1 & v_1 & u'_1 & v'_1 \\ u_2 u'_2 & u_2 v'_2 & u_2 & v_2 u'_2 & v_2 v'_2 & v_2 & u'_2 & v'_2 \\ u_3 u'_3 & u_3 v'_3 & u_3 & v_3 u'_3 & v_3 v'_3 & v_3 & u'_3 & v'_3 \\ u_4 u'_4 & u_4 v'_4 & u_4 & v_4 u'_4 & v_4 v'_4 & v_4 & u'_4 & v'_4 \\ u_5 u'_5 & u_5 v'_5 & u_5 & v_5 u'_5 & v_5 v'_5 & v_5 & u'_5 & v'_5 \\ u_6 u'_6 & u_6 v'_6 & u_6 & v_6 u'_6 & v_6 v'_6 & v_6 & u'_6 & v'_6 \\ u_7 u'_7 & u_7 v'_7 & u_7 & v_7 u'_7 & v_7 v'_7 & v_7 & u'_7 & v'_7 \\ u_8 u'_8 & u_8 v'_8 & u_8 & v_8 u'_8 & v_8 v'_8 & v_8 & u'_8 & v'_8 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \end{bmatrix} = - \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \tag{4.44}$$

which is not homogeneous and can be written succinctly as $\mathbf{U}\mathbf{f} = -\mathbf{1}$, hence $\mathbf{f} = -\mathbf{U}^{-1}\mathbf{1}$, provided $\mathbf{U}$ is invertable (which is the case unless the eight points have a particular geometric relation). This is a non-homogeneous set of equations, and when it is used to estimate the fundamental matrix, the algorithm is called the **eight-point algorithm**.

The eight-point algorithm has been criticised because it tends to be not very accurate. Forsyth and Ponce [2002] (pages 219–221) discuss two common alternatives. A third, very useful alternative is provided by Torr and Fitzgibbon [2004].

4.3 3D reconstruction from stereo

Reconstruction from 2D to 3D is possible using a variety of different information: motion, shading and stereo. We'll look only at stereo.

Suppose we somehow knew everything about a pair of stereo cameras: where they are in space as well as their intrinsic parameters. 3D reconstruction is now possible, provided corresponding points in images can be found using matching, which allows 'triangulation' back to 3D.

In practice, matching is very hard, and errors mean that the 3D reconstruction is far from perfect. We'll see how to approach these problems.

4.3.1 Matching: RANSAC

The method we will study relies on being able to detect *features* in images. The features are assumed to exist at points (so corners are useful example features), and we will try to match features in one image with those in another. This method belongs to the *landmark* class of matching methods (sparse correspondences), which contrast with the *correlation* class (dense correspondences), where one image is treated as a template that is to be transformed into the other. Bear in mind that no matter what method is used, the objective is always to produce a mapping between the images

so that given a point in one image, we can use the mapping to locate its corresponding point in the other.

Typically Harris corners (Section 3.2; Harris and Stephens, 1988) or – more recently – SIFT features (Section 3.4; Lowe, 2004) are used for matching. Whatever features are used, the detector may detect features where there are none (false positives), and it may also fail to detect features where they exist (false negatives). The consequence of this for matching features across images is that a feature in one image may not exist in the other. Furthermore, it is easy to make false matches, especially when there are multiple copies of a similar object, such as bricks in a wall.

We can make progress by assuming the cameras that acquired the images are separated by a small baseline and look in more-or-less the same direction: that is, we assume a stereo image pair. This means that a real feature in one image will not have moved very far in the second image. Another way to say this that the pictures look pretty much alike. The advantage of this is that if we pick a feature in image A, then we need only look in a small region of image B to find the corresponding point. Suppose a features is at $[x, y]^\top$ in image A, we may want to limit the search to an 11×11 pixel window centred around $[x, y]^\top$ in image B. The distance between the matching points is called *disparity* and can be used as a basis for 3D reconstruction.

The above matching process can and will produce false matches, so we need a way to decide which matches are good and which are bad. We use an iterative process [Fischler and Bolles, 1981], called *RANSAC* (random sample consensus), which has the following algorithmic form:

1. for a pre-specified number of iterations:
   (a) randomly choose four matches assumed to be good (aka inliers)
   (b) use the assumed inlier matches to compute a mapping between the images (aka model)
   (c) measure the quality of the mapping (e.g. percentage of inlier matches)
2. use the highest-quality mapping
3. optionally refit model to all inliers to improve model

We already know how to pick out features in images. Step 1a says just choose 4 feature pairs, at random. In practice, this is not quite random – the stereo assumption limits the number of possible choices to a manageable number. Four matches of the form $(\mathbf{a}_i, \mathbf{b}_i)$, for $i=1,\ldots,4$, are enough to compute a *homography* between the images. A homography is a map (transform) that allows one image to be translated, scaled, rotated, skewed and subjected to perspective-like distortions. We need homogeneous coordinates to apply a homography, and so write

$$\begin{bmatrix} b_{1i} \\ b_{2i} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} a_{1i} \\ a_{2i} \\ 1 \end{bmatrix}. \tag{4.45}$$

The upper-left 2×2 matrix will shear, rotate and scale. The right-most column vector $[h_{13}, h_{23}]^\top$ is a translation. The bottom-most row vector $[h_{31}, h_{32}]$ introduces a perspective-like distortion *after* the homogeneous point is converted to an ordinary point. If this vector is zero, then h_{33} will scale (in inverse fashion).

Under this homography, the point $\mathbf{b}_i$ can be computed from the $\mathbf{a}_i$ as

$$b_{1i} = \frac{h_{11}a_{1i} + h_{12}a_{2i} + h_{13}}{h_{31}a_{1i} + h_{32}a_{2i} + 1} \tag{4.46}$$

$$b_{2i} = \frac{h_{21}a_{1i} + h_{22}a_{2i} + h_{23}}{h_{31}a_{1i} + h_{32}a_{2i} + 1} \tag{4.47}$$

and these can be re-arranged to give

$$(h_{11}a_{1i} + h_{12}a_{2i} + h_{13}) - b_{1i}(h_{31}a_{1i} + h_{32}a_{2i} + 1) = 0 \tag{4.48}$$

$$(h_{21}a_{1i} + h_{22}a_{2i} + h_{23}) - b_{2i}(h_{31}a_{1i} + h_{32}a_{2i} + 1) = 0. \tag{4.49}$$

From here, we can separate out the unknowns, the h_{jk}, into a column vector and so write

$$\begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}a_{1i} & a_{2i} & 1 & 0 & 0 & 0 & -b_{1i}a_{1i} & -b_{1i}a_{2i} & -1\\ 0 & 0 & 0 & a_{1i} & a_{2i} & 1 & -b_{1i}a_{1i} & -b_{1i}a_{2i} & -1\end{bmatrix} \begin{bmatrix}h_{11}\\h_{12}\\h_{13}\\h_{21}\\h_{22}\\h_{23}\\h_{31}\\h_{32}\\h_{33}\end{bmatrix}. \tag{4.50}$$

Given we have $i = 1, \ldots, 4$, we stack the equations of each case to construct a larger system:

$$\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}a_{11} & a_{21} & 1 & 0 & 0 & 0 & -b_{11}a_{11} & -b_{11}a_{21} & -1\\ 0 & 0 & 0 & a_{11} & a_{21} & 1 & -b_{11}a_{11} & -b_{11}a_{21} & -1\\ a_{12} & a_{22} & 1 & 0 & 0 & 0 & -b_{12}a_{12} & -b_{12}a_{22} & -1\\ 0 & 0 & 0 & a_{12} & a_{22} & 1 & -b_{12}a_{12} & -b_{12}a_{22} & -1\\ a_{13} & a_{23} & 1 & 0 & 0 & 0 & -b_{13}a_{13} & -b_{13}a_{23} & -1\\ 0 & 0 & 0 & a_{13} & a_{23} & 1 & -b_{13}a_{13} & -b_{13}a_{23} & -1\\ a_{14} & a_{24} & 1 & 0 & 0 & 0 & -b_{14}a_{14} & -b_{14}a_{24} & -1\\ 0 & 0 & 0 & a_{14} & a_{24} & 1 & -b_{14}a_{14} & -b_{14}a_{24} & -1\end{bmatrix} \begin{bmatrix}h_{11}\\h_{12}\\h_{13}\\h_{21}\\h_{22}\\h_{23}\\h_{31}\\h_{32}\\h_{33}\end{bmatrix}, \tag{4.51}$$

or, more succinctly, $\mathbf{Ah} = \mathbf{0}$. The smallest singular vector of $\mathbf{A}$ is the best solution in the least-squares sense (see Section 4.1.3), and this can be easily scaled so that $h_{33} = 1$.

Now we have a mapping, we have to measure its quality. For this, we apply the mapping to all features in the image A and find which feature in image B it ends up closest to. That is, for each point $\mathbf{a}$, we measure

$$d(\mathbf{a}; \mathbf{H}) = \min_{\mathbf{b} \in B} |\mathbf{b} - \mathbf{Ha}|. \tag{4.52}$$

A quick way to do this is to use a *distance transform*. The quality of the mapping is then taken to be

$$D(\mathbf{H}) = \sum_{\mathbf{a} \in A} d(\mathbf{a}). \tag{4.53}$$

Clearly, we choose the $\mathbf{H}$ with the smallest measure.

In practice, a better set of matchings can be produced by using the inverse homography to compare points in B with those in A.

4.3.2 Reconstruction

Let $\mathbf{y}_1$ and $\mathbf{y}_2$ be matching points in a stereo pair, that is they are both projections of point $\mathbf{x}$; all points are homogeneous. We have

$$y_{1,i} = \frac{\mathbf{p}_{1,i}^\top \mathbf{x}}{\mathbf{p}_{1,3}^\top \mathbf{x}}, \tag{4.54}$$

where $y_{1,i}$ is the i^{th} component of image point $\mathbf{y}_1$ in camera 1, and $\mathbf{p}_{kj}^\top$ the j^{th} row of projection matrix $\mathbf{P}_k$. This can be rearranged to

$$(y_{1,i}\mathbf{p}_{1,3}^\top - \mathbf{p}_{1,i}^\top)\,\mathbf{x} = 0, \tag{4.55}$$

and similarly

$$(y_{2,i}\mathbf{p}_{2,3}^{\top} - \mathbf{p}_{2,i}^{\top})\,\mathbf{x} = 0. \tag{4.56}$$

Note P_{ji} is the ith row of the jth projection; a 1×4 row vector.

Putting these into a matrix, we have

$$\mathbf{Q} = \begin{bmatrix} y_{1,1}\mathbf{p}_{1,3} - \mathbf{p}_{1,1} \\ y_{1,2}\mathbf{p}_{1,3} - \mathbf{p}_{1,2} \\ y_{2,1}\mathbf{p}_{2,3} - \mathbf{p}_{2,1} \\ y_{2,2}\mathbf{p}_{2,3} - \mathbf{p}_{2,2} \end{bmatrix}. \tag{4.57}$$

The solution $\mathbf{x}$ to $\mathbf{Q}\mathbf{x} = \mathbf{0}$ is therefore the smallest right singular vector of Q, scaled to have homogeneous depth of 1.

4.3.3 Bundle Adjustment

Errors are inevitable, in calibration and in matching. Hence the outputs are noisy. Bundle adjustment is used to reduce the errors.

The basic idea is to reduce the error in an objective function by adjusting the camera parameters and 3D point positions. Let $\mathbf{y}(\mathbf{x}_j, C_i)$ be the predicted 2D position of a 3D point $\mathbf{x}_j$ under camera C_i, and let $\mathbf{y}_{ij}$ be the corresponding observation; then bundle adjustment is

$$\underset{\{\mathbf{x}_j\},\{C_i\}}{\operatorname{argmin}} \sum_{i=1}^{N}\sum_{j=1}^{M} w_{ij} d(\mathbf{y}(\mathbf{x}_j, C_i), \mathbf{y}_{ij}). \tag{4.58}$$

The function $d(\cdot,\cdot)$ is a distance, typically the *reprojection error*

$$d(\mathbf{a},\mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_2. \tag{4.59}$$

The weight $w_{ij} \in \{0,1\}$ is an indicator function for visiblity, being 1 only when point j is visible in image i.

Chapter 5

Motion

5.1 Optical flow

A classic algorithm for optical flow is due to Horn and Schunck [1981]. The algorithm makes two assumptions:

- the brightness of a point is invariant as it moves,
- the velocity field varies smoothly.

Let $I(x, y, t)$ be the brightness of the pixel at (x, y) at time t. The first assumption says this is constant as the brightness pattern under the pixel grid moves. Therefore

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0, \tag{5.1}$$

which follows from the chain rule for differentiation. We set

$$u = \frac{dx}{dt}, \tag{5.2}$$

$$v = \frac{dy}{dt}. \tag{5.3}$$

The problem for optical flow is to compute a velocity vector (u, v) at every point in the image.

The second assumption – smoothness of motion – is a bid to address the *aperture problem* in which direction of motion tangential to a contrast edge cannot be directly measured. The assumption is that the velocity does not change much in a local region. This is expressed by minimising the Laplacian over the velocity field. That is

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad \text{and} \tag{5.4}$$

$$\nabla^2 v = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \tag{5.5}$$

are small.

The Horn–Schunck algorithm find the velocity field $[u(x, y), v(x, y)]$ by minimising the energy

$$E = \iint \left[(I_x u + I_y v + I_t)^2 + \alpha^2 \left(\|\nabla u\|^2 + \|\nabla v\|^2 \right) \right] \mathrm{d}x\, \mathrm{d}y, \tag{5.6}$$

in which I_x, I_y, I_t are the partial derivatives of intensity over x, y, t.

The regularisation term $\left(\|\nabla u\|^2 + \|\nabla v\|^2\right)$ imposes smoothness, and its impact on the solution is controlled by parameter α.

A numerical solution comes down to finding the velocity field

$$I_x(I_x u + I_y v + I_t) - \alpha^2 \Delta u = 0 \tag{5.7}$$

$$I_y(I_x u + I_y v + I_t) - \alpha^2 \Delta v = 0 \tag{5.8}$$

via finite-difference approximations in which Δu is computed as the difference between local average velocities, $\bar{u}$, and the centre point velocity.

An iterative approach is used, in which the velocity field is updated

$$u^{i+1} = \bar{u}^k - \frac{I_x(I_x \bar{u}^i + I_y \bar{v}^i + I_t)}{k} \tag{5.9}$$

$$v^{i+1} = \bar{v}^k - \frac{I_y(I_x \bar{u}^i + I_y \bar{v}^i + I_t)}{k} \tag{5.10}$$

in which $k = \alpha^2 + I_x^2 + I_y^2$.

5.2 Tracking

Consider a simple *template tracker*. A template tracker has a template of the object it is tracking. The template will typically be a small window that has been masked out from a video frame to show the object. The template will *correlate* windows that look like itself: correlation being defined here as

$$c(u,v) = \sqrt{\sum_x \sum_y |I(x-u, y-v) - T(x,y)|^2}, \tag{5.11}$$

which assumes the template has been normalised so that $[0,0]^\top$ is its origin. We can imagine using correlation to locate the pixel $[u,v]^\top$ with the smallest correlation score. Further, we can imagine doing this for successive frames in a video, and so find the trajectory of the object.

If only things were that simple. The object may get bigger bigger (smaller) as it moves toward (away) from us. It may change appearance as it spins. It may become occluded, or partially occluded, as it passes behind foreground objects. There may be other objects just like it in the video, or at least sufficiently similar objects. All of these problems make tracking very difficult indeed. The first step to a better tracker in to use a motion model.

5.2.1 Motion models

Physics provides us with very strong models of motion. Newton's second law is that the rate of change of momentum is equal to the force applied. This is expressed in the differential equation:

$$\mathbf{F} = m\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}, \tag{5.12}$$

assuming that mass m (the constant of inertia) does not change. Acceleration is the rate of change of velocity, and velocity is the rate of change of displacement:

$$\mathbf{a} = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} \tag{5.13}$$

$$\mathbf{v} = \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \tag{5.14}$$

hence we can also write $\mathbf{a} = \mathrm{d}^2\mathbf{x}/\mathrm{d}t^2$. These equations are integrated to give the well-known equations of motion

$$\mathbf{x} = \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2, \tag{5.15}$$

in which $\mathbf{x}$ is the distance travelled in time t. To get the actual location we must, of course, add on the initial location:

$$\mathbf{x} = \mathbf{p} + \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2. \tag{5.16}$$

This is a very common motion model.

Perhaps a better starting point is the Taylor expansion of a function:

$$f(t+\delta) = f(t) + \frac{\mathrm{d}f(t)}{\mathrm{d}t}\delta + \frac{1}{2}\frac{\mathrm{d}^2 f(t)}{\mathrm{d}t^2}\delta^2 + \frac{1}{6}\frac{\mathrm{d}^3 f(t)}{\mathrm{d}t^3}\delta^3 + E[\delta^4] \tag{5.17}$$

$$= \sum_{k=0}^{N} \frac{1}{k!}\frac{\mathrm{d}^k f(t)}{\mathrm{d}t^k} + E[\delta^{N+1}], \tag{5.18}$$

which says that the value of a function, $f(\cdot)$ a distance δ from a known point t can be approximated to zeroth order by $f(t)$, to first order by adding the sloping line $\mathrm{d}f/\mathrm{d}t$, to second order by adding a quadratic term $\mathrm{d}^2 f/\mathrm{d}t^2$, and so on with the correction terms of increasing order and decreasing magnitude until an error $E[\cdot]$ remains; which is of order δ^{N+1}. You should compare the Taylor expansion with Newton's equations.

If a computer program is given an initial position, initial velocity, acceleration, and any other terms, we can compute the path a particle takes using numerical integration. Many methods exist but the easiest (yet least stable, numerically) is called Euler integration (Algorithm 5.1). This simply updates the position and velocity at each time instant.

Algorithm 5.1 Pseudo-code for tracking with Euler integration.

```
Initialise position, velocity, accleration
decide a time interval
FOR each time instant
    position = position + velocity * interval
    velocity = velocity + acceleration * interval
END
```

If we choose to do so (and we will choose to do so), we can express this integration using matrix methods. First, we define the "state" of the particle as being a combination of its position, velocity, and acceleration – which is all that is needed to predict its future position. Next, we define a "system matrix" that maps the state a onto time instant to the state in the very next time instant.

Algorithm 5.2 Pseudo-code for tracking with matrix-based Euler integration.

```
Initialise state
decide a time interval
FOR each time instant
    state = A * state
    position = H * position
END
```

In Algorithm 5.2, `A` is the system matrix (state transition matrix), and `H` is a projection matrix that grabs the data we are interested in from the state. In the specific case of motion under Newton's

laws, we define a state vector by concatenating location, velocity and acceleration

$$\mathbf{s} = \begin{bmatrix} \mathbf{x}(0) \\ \mathbf{v}(0) \\ \mathbf{a} \end{bmatrix} \tag{5.19}$$

and define a system matrix for the interval δ

$$\mathbf{A} = \begin{bmatrix} \mathbf{1}^\top & \delta\mathbf{1}^\top & (\delta^2/2)\mathbf{1}^\top \\ \mathbf{0}^\top & \mathbf{1}^\top & \delta\mathbf{1}^\top \\ \mathbf{0}^\top & \mathbf{0}^\top & \mathbf{1}^\top \end{bmatrix}, \tag{5.20}$$

in which $\mathbf{1}$ is a vector which is 1 in every location, and $\mathbf{0}$ is a vector with 0 in every location; each of these being the same size as the position vector (so if the position is 2D, the vectors are 2-dimensional, and so on). The observation matrix is

$$\mathbf{H} = \begin{bmatrix} \mathbf{1}^\top & \mathbf{0}^\top & \mathbf{0}^\top \end{bmatrix}, \tag{5.21}$$

that is, a row vector whose length is the same as that of the state vector, and which in this case picks out the location of the particle.

In practice, we would not, of course, have direct access to the velocity and acceleration of a particle. However, we might hope to be able to observe the particle in the first three frames, and this is enough to compute the remaining positions. This requires no alteration at all to the general scheme just proposed, but does require new contents for the state, and new contents for the system matrices (in general the observation matrix must change too, but here it is the same). The state in the third frame (time instant) is

$$\mathbf{s} = \begin{bmatrix} \mathbf{x}(2\delta) \\ \mathbf{x}(1\delta) \\ \mathbf{x}(0\delta) \end{bmatrix} \tag{5.22}$$

and the system matrix is

$$\mathbf{A} = \begin{bmatrix} 3\mathbf{1}^\top & -3\mathbf{1}^\top & \mathbf{1}^\top \\ \mathbf{1}^\top & \mathbf{0}^\top & \delta\mathbf{0}^\top \\ \mathbf{0}^\top & \mathbf{1}^\top & \mathbf{0}^\top \end{bmatrix}. \tag{5.23}$$

Notice how the first line predicts the new location as a linear combination of the previous three locations, and that this "history" window of three points rolls along in time.

All of this would work very well, except it does not account for noise at all. Unfortunately, the presence of noise in the observations can mess things up considerably. Fortunately we have a remedy – the Kalman filter.

5.3 Kalman Filtering

The Kalman filter [Kálmán, 1960] is a *predictor corrector*: the Kalman filter "watches" an object as it moves – that is to say it gathers information about the object being tracked, at least over a short period of time. It then uses the information it has gathered to predict where the object will be in the next frame; this prediction requires a model of motion of the kind just discussed. Next, the Kalman filter looks for the object in the next frame – and corrects the model of motion using the difference between the observation and prediction.

But the Kalman filter does more. It recognises two kinds of noise:

1. The particle may not move under conventional laws of physics, for example the flight of a bird can be rather erratic. This is called *process noise*.
2. The particle has to be observed with a measuring instrument that will always introduce errors of its own, so called *measurement noise*.

The presence of noise means we have to correct our predictions, somehow. What is more, we should not only make a prediction, but in addition be able to state a level on confidence in our prediction. The Kalman filter does these things for us, too.

Let us consider noise, for the moment. One way to think about noise is that it "jitters" a true point to a new point, the one we end up seeing. Suppose for the moment these points are two-dimensional. We allow the true point to move under a motion model, and at each instant, jitter it into a new point. What is of interest is the error $\mathbf{e} = \mathbf{x}_{\text{seen}} - \mathbf{x}_{\text{true}}$. We can imagine plotting these errors as points in the plane. The way the points get spread out – their distribution – indicates any biasing in the error measurement, and also indicated the level of confidence we can have. If the points are widely spread, we would be inclined to have less confidence than if they were tightly clustered. We would hope for, and in fact assume, that the points are Gaussian distributed. This just means that the number of points in any patch of the plane is directly proportional to the volume of a 2D Gaussian under that patch. A Gaussian has a single peak at

$$\overline{\mathbf{e}} = \frac{1}{N} \sum_{i=1} N \mathbf{e}_i, \tag{5.24}$$

where N is the number of points being considered. This is the mean, and it is the most likely value for the error. The rate at which the distribution decays, and whether this rate is faster in one direction than another, is captured by the *covariance* matrix

$$\mathbf{C} = \frac{1}{N} \sum_{i=1} N (\mathbf{e}_i - \overline{\mathbf{e}})(\mathbf{e}_i - \overline{\mathbf{e}})^\top, \tag{5.25}$$

which assumes a Gaussian distribution of points.

The Kalman filter maintains a prediction for the most likely next location, and also maintains a covariance matrix that is used to indicate a confidence in the predicted value. The equations that underpin Kalman filtering are

$$\mathbf{s}_{k+1} = \mathbf{A}\mathbf{s}_k + \mathbf{n}_k, \tag{5.26}$$

$$\mathbf{x}_k = \mathbf{H}\mathbf{s}_k + \mathbf{m}_k, \tag{5.27}$$

in which $\mathbf{s}$ is a state vector (the subscripts indicate a given time step); $\mathbf{x}$ is the observable; $\mathbf{A}$ is the system update matrix, and $\mathbf{H}$ is the observation matrix. In fact, these equations are exactly those we used in modelling motion, except for the presence of noise vectors $\mathbf{n}$ and $\mathbf{m}$ to model process noise and measurement noise, respectively.

It is important to recognise that the vector $\mathbf{x}$ in the above is a real measurement (observation), and that $\mathbf{H}\mathbf{s}_k$ is the prediction of this observation. In the case of no noise, these two should be identical, but where there is noise, the error is exactly $\mathbf{m}$:

$$\mathbf{m}_k = \mathbf{x}_k - \mathbf{H}\mathbf{s}_k. \tag{5.28}$$

We assume the expected value of this error is zero, and compute its covariance as

$$\mathbf{P} = \frac{1}{N} \sum_{k=1}^{N} \mathbf{m}\mathbf{m}^\top. \tag{5.29}$$

What the Kalman filter tries to do, is to make this covariance as small as possible, which in practical terms means keeping the spread of errors as tight as possible. (The "volume" of the covariance –

the spread of points – can be measured either by the trace of the matrix or as the product of its singular values).

The Kalman filter corrects the state using

$$\mathbf{s}_{k+1} = \mathbf{A}\mathbf{s}_k + \mathbf{K}\mathbf{m}, \tag{5.30}$$

where $\mathbf{K}$ is the *gain*:

$$\mathbf{K} = \mathbf{P}\mathbf{H}^\top \left(\mathbf{H}\mathbf{P}\mathbf{H}^\top + \mathbf{R}\right)^{-1}, \tag{5.31}$$

where $\mathbf{R}$ is the covariance of the measurement (the measurement noise). This measures the confidence in the measurement, it is not covariance of the prediction error $\mathbf{m}$ (that covariance we have already called $\mathbf{P}$). For example, reconsider the template-based tracker. We might use the values from such a tracker to generate an $\mathbf{R}$ matrix. This gain, $\mathbf{K}$ minimises the error covariance $\mathbf{P}$, which is also corrected:

$$\mathbf{P}_{k+1} = (\mathbf{I} - \mathbf{K}\mathbf{H})\left(\mathbf{A}\mathbf{P}_k\mathbf{A}^\top + \mathbf{Q}\right), \tag{5.32}$$

where now $\mathbf{Q}$ is the *process noise* covariance; that is the noise intrinsic to the system.

Putting all this together, the Kalman filter is shown in Algorithm 5.3. Well, not quite. This is a simplified version of the Kalman filter; it does not allow for any *control* to be exerted over the system. Even so, it is sufficient for many tracking purposes. Bear in mind that the state can be anything we want it to be, and can include terms to dictate shape, say, so that objects and not just points can be tracked.

Algorithm 5.3 Pseudo-code for Kalman filtering.

```
Initialise:
  * state
  * the system matrix A
  * the observation matrix H
  * process covariance Q
  * measurment covariance R
  * error covariance P

FOR each time instant, k

    /* predict */
    newstate = A * state;  // advance predicted state
    newP = A * P * A' + Q; // advance predicted state covariance

    /* correct */
    K = newP * H' * inv( H * newP * H' + R);     // Kalman gain
    state = newstate + K*( x[k] - H * newstate); // correct state
    P = (I - K * H) * newP;                      // correct covaraince

    /* record observation
    z[k+1] = H*state;
END
```

The major limitation to the Kalman filter is that it assumes a Gaussian distribution over the prediction error; this is implicit in using a covariance matrix. But consider the case where someone's face is being tracked, and they happen to be walking; walking is of course not an easy motion to model – especially by a polynomial, and if the person happens to be in a crowd, things get much worse. It is very easy to imagine that there are many possible places for the next position, and the Kalman filter just breaks down.

Bibliography

John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986. ISSN 0162-8828. doi: 10.1109/TPAMI.1986.4767851.

Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24 (6):381–395, 1981. ISSN 0001-0782. doi: 10.1145/358669.358692.

David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 1st edition, 2002. ISBN 0130851981.

David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Pearson, 2nd edition, 2011. ISBN 013608592X.

Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2013. ISBN 1421407949. URL https://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm.

Chris Harris and Mike Stephens. A combined corner and edge detector. In *Proceedings of the Alvey Vision Conference*, pages 147–151, 1988. doi: 10.5244/C.2.23.

Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17: 185–203, August 1981. ISSN 0004-3702. doi: 10.1016/0004-3702(81)90024-2.

Rudolf E. Kálmán. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, March 1960. doi: 10.1115/1.3662552.

David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 0521880688. URL http://numerical.recipes/.

Philip H. S. Torr and Andrew W. Fitzgibbon. Invariant fitting of two view geometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):648–650, May 2004. ISSN 0162-8828. doi: 10.1109/TPAMI.2004.1273967.