Contents

PART ONE

WHAT COMPUTER VISION DOES

CHAPTER 1

Problems

PART TWO

ACTING ON IMAGES

CHAPTER 2

Simple Image Processing

2.1 IMAGES AS SAMPLED FUNCTIONS

Your first encounter with an image as something to compute with (rather than look at) is likely as an array for an intensity image, or set of three arrays for a color image. Knowing how the image ended up in this form is important if you want to interpret it. We will develop a quite detailed model of the geometry and physics underlying images later; a simple model will have to do for the moment.

The image you see as three arrays started as a spectral energy field – energy E as a function of position $\mathbf{x}$, direction ω and wavelength λ, so $E(\mathbf{x}, \omega, \lambda)$. This energy field is created by light leaving light sources, reflecting from surfaces, and eventually arriving at the entrance to the camera (Figure 2.1). This is usually but not always a lens. Various processes in lens and camera map some of the light that arrives to some photosensitve layer at the back of a camera. In turn, this photosensitive layer transduced the energy field into arrays of numbers.

The photosensor is divided into pixels (Figure 2.2). These pixels compute a weighted average of $E(\mathbf{x}, \omega, \lambda)$, averaged over a small range of positions (typically the size of the pixel) and a small range of directions (which is determined by the position of the pixel; the lens system takes care of this) and a large range of wavelengths. Assume the coordinate system is placed on the sensor, with x and y coordinates in the natural directions on the grid, and the sensor at $z = 0$. The grid separation between pixels is Δx in the x direction and Δy in the y direction. Then the pixel at the i, j'th location is at $(i\Delta x, j\Delta y, 0)$. The vast majority of photosensors are linear. This means for $\mathcal{P}$ a small range of positions centered at the pixel, $\mathcal{W}$ a small range of directions and Λ the wavelengths of interest, we can write the value reported by the pixel as

$$\begin{aligned} p_{ij} &= k \int_{\mathcal{P}} \int_{\mathcal{W}} \int_{\Lambda} E(\mathbf{u}, \omega, \lambda) w(\mathbf{u}, \omega, \lambda) d\mathbf{u} d\omega d\lambda \\ &\quad \text{here } w \text{ is the weight function or sensitivity of the sensor} \\ &= k \int_{\Lambda} E([i\Delta x, j\Delta y, 0], \omega, \lambda) w(\lambda) d\mathbf{u} d\lambda \\ &\quad \text{because the averages are over very small ranges} \\ &= k\Phi(i\Delta x, j\Delta y, 0) \end{aligned}$$

for an appropriate Φ (write this function out to be sure). The pixel at i, j on the grid is a *sample* of a function of position (Figure **??**).

Sampling a function can produce something that represents the function very poorly indeed. As Figure **??** illustrates, the key question is how many samples you draw compared to how much detail there is in the function. This has important consequences. Assume you wish *downsample* an image – reduce its size in each

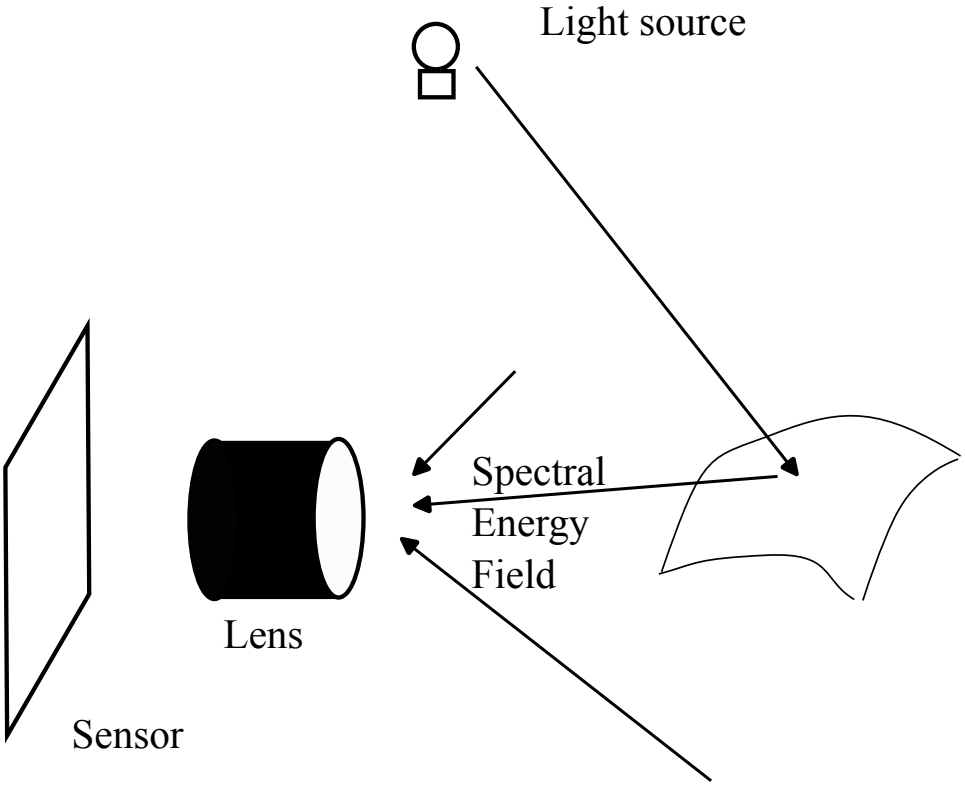


FIGURE 2.1: *A high-level model of imaging. Light leaves light sources and reflects from surfaces. Eventually, some light arrives at a camera and enters a lens system. Some of that light arrives at a photosensor inside the camera.*

dimension. The simplest strategy is to take every second pixel in each direction (downsampling by 2). This, as Figure **??** illustrates, is a bad idea. The result can seriously misrepresent the image. Notice how small structures can appear to be big ones, or disappear entirely. The general term for the kind of errors seen here is *aliasing*. In Chapter 33.2, we will be much more precise about these issues, and demonstrate a procedure to avoid aliasing.

Now imagine you wish to downsample by 3/2 (so a 150×150 image goes to a 100×100 image). There are two ways to approach this problem. You could scan the source (larger - $\mathcal{L}$) image and, for each pixel, determine where it goes. Alternatively, you could scan the target (smaller - $\mathcal{S}$) image and, for each pixel, determine what value it should receive. Scanning the source generally would lead to a problem. The rule $\mathcal{L}_{ij} \rightarrow \mathcal{S}_{(2/3)*(i-1),(2/3)*(j-1)}$ *when the source coordinates are integers* yields an image of the right size (check this), and yields: $\mathcal{L}_{11} \rightarrow \mathcal{S}_{11}$; $\mathcal{L}_{41} \rightarrow \mathcal{S}_{31}$; and so on. But there is a hole at $\mathcal{S}_{21}$, because we can only put values into the target image at integer locations.

Scanning the target yields another difficulty. The i, j'th location of $\mathcal{S}$ must get the value at $\mathcal{L}_{(3/2)*(i-1),(3/2)*(j-1)}$, but the coordinates are not integer values –

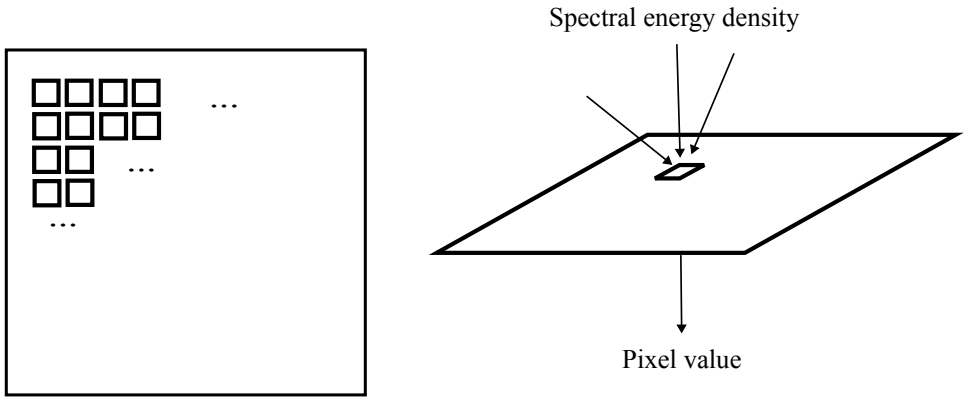


FIGURE 2.2: *The photosensor is divided into a grid of pixels, which are small sensitive locations. Each pixel receives an incoming spectral energy field, and turns it into a number. This number is typically a weighted average over a position in the sensor, a very small range of incoming directions and a large range of wavelengths. Each pixel is at a different position in the sensor, and the lens system and camera geometry ensure that each sees a different set of incoming directions, so that the averages produce a coherent image.*

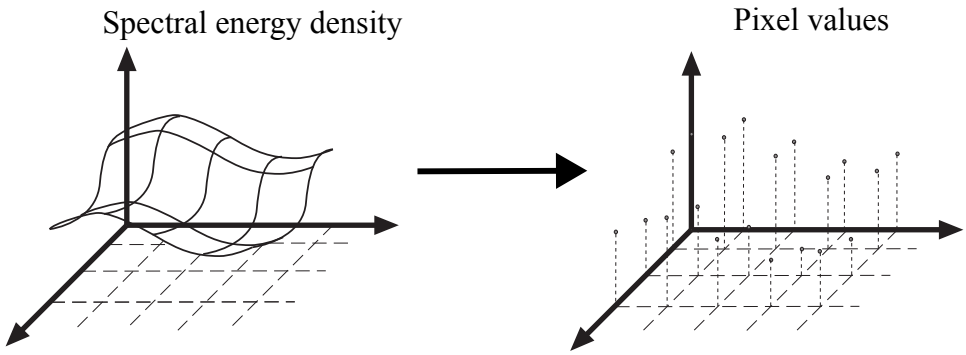


FIGURE 2.3: *Because each pixel in the sensor averages over a small range of directions and positions, the process mapping the input spectral energy distribution to pixel values can be thought of as sampling. On the* **left**, *is a representation of the energy distribution as a continuous function of position. The value reported at each pixel is the value of this function at the location of the pixel (***right***).*

we must obtain an approximate value. Two approximation procedures are common. One is *nearest neighbors* – you take the value at the integer point closest to location whose value you want (Figure 33.2). So for the running example, you would use the value at $2, 2$ if you wanted the value at $1.5, 1.5$. As Figure 33.2 shows, this strategy has problems.

A much better procedure is *interpolation*, where we fit a function to some pixel values then sample the fitted function. Most widely used is *bilinear interpolation*.

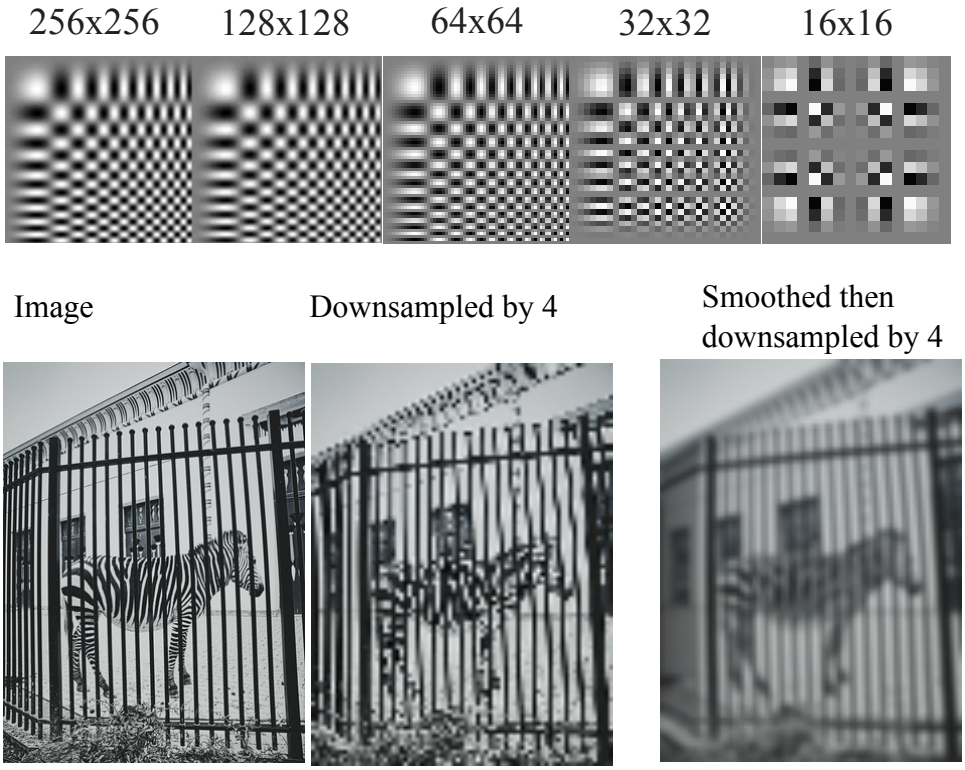


FIGURE 2.4: **Top row** *shows an image on the* **left**, *and then a set of versions resampled to different grids then expanded to the original size. Notice how detail is lost in the resampling process. Some small boxes disappear, and others turn into large boxes.* **Bottom left** *shows a picture of a zebra (copyright free, from Flickr user Fouquier);* **center left** *shows this image downsampled by four;* **bottom right** *shows what happens when the image is smoothed, then downsampled. The downsampled images are shown at the same size as the originals by simply printing larger pixels.*

We want a value at $i + \delta i, j + \delta j$, where i and j are integers; $0 < \delta i < 1$; and $0 < \delta j < 1$. Write v_{ij} for the value at i, j. Then use

$$v = v_{ij}(1 - \delta i)(1 - \delta j) + v_{i+1,j}(\delta i)(1 - \delta j) + v_{i,j+1}(1 - \delta i)(\delta j) + v_{i+1,j+1}(\delta i)(\delta j).$$

Notice that if δi and δj are both zero, then $v = v_{ij}$; if they are both one, $v = v_{i+1,j+1}$; and v will interpolate the value at the other two corners, too. By a little manipulation, you can show that this procedure boils down to: predict a value for $i + \delta i, j$ using a linear interpolate; predict a value for $i + \delta i, j + \delta j$ using a linear interpolate; now linearly interpolate between these two to get a value for $i + \delta i, j + \delta j$. Modern hardware is particularly efficient at bilinear interpolation, and any reasonable software environment will be able to do this for you.

More complicated interpolation procedures are possible. In *bicubic interpola-*

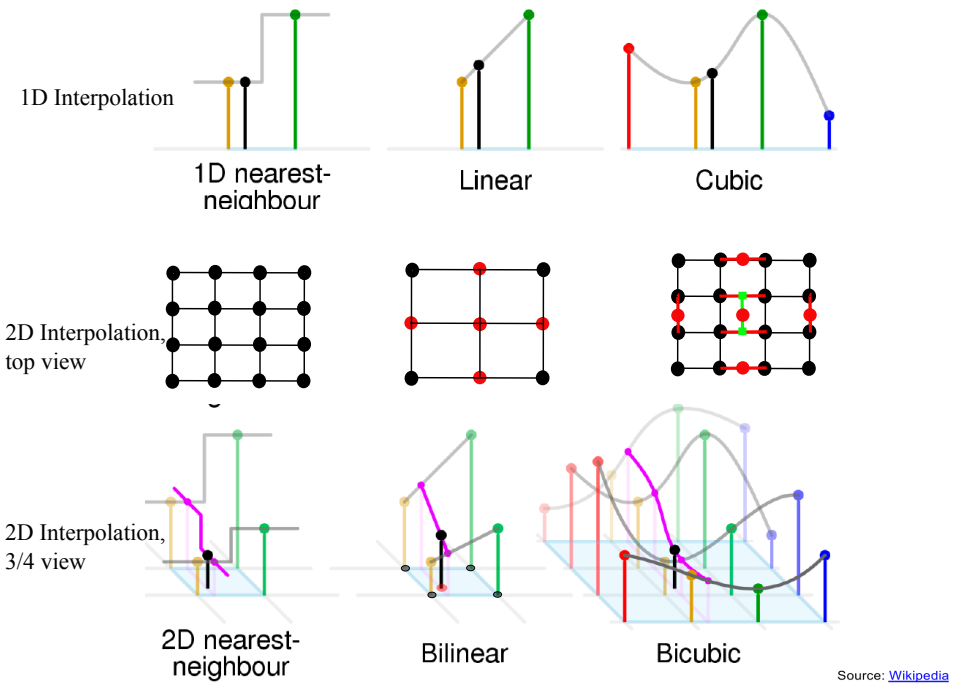


FIGURE 2.5: **Top row** *shows interpolation in 1D. To obtain a value for a location between samples, we could: choose the value of the closest sample (nearest neighbors,* **left***; connect the two closest samples with a line segment, then sample that (linear interpolation,* **center***); or construct a cubic curve between the four closest samples and sample that (cubic interpolation,* **right***).* **Center** *shows grids that capture what happens when you downsample an image. On the* **left***, a simple* 4×4 *image, as a grid of known (black) pixel values. To downsample this to a* 3×3 *image, we must evaluate the image at the red locations (***center***). On the* **right***, these locations placed on the original grid. For four of the red locations, we can evaluate using 1D linear interpolation, but the red point in the center must be evaluated, too. We can do so by (a) linearly interpolating horizontally to get values at the green squares then (b) linearly interpolating those vertically to get the value. This is bilinear interpolation.* **Bottom row** *shows a 3D view, with the grid on the bottom and function values represented by height above the grid.*

tion, the interpolate is cubic in δx and δy and depends on other neighboring pixels. Again, any reasonable software environment will be able to do this for you. While this procedure is more complicated and slower, in some applications the small improvements are justified. One occasionally important difference between bicubic interpolation is that for a bilinear interpolate, the local maxima are always at grid points, but for a bicubic interpolate, they may not be (exercises).

Interpolation is particularly important if you want to *upsample* an image, where you increase the number of pixels in a grid. To go from, say, a 100×100

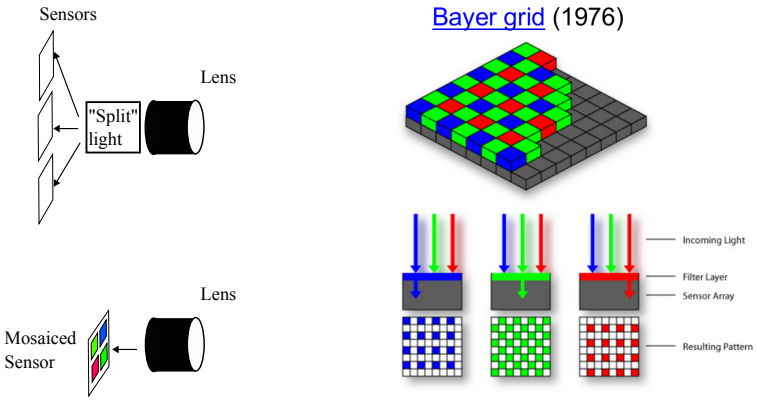


FIGURE 2.6: *There are two main ways to obtain color images. One can (as in* **top left***) build a multiccd camera with three imaging sensors. Each has a different response to wavelengths. The cheaper and lighter alternative is to use one imaging sensor (***bottom left***) but have a mosaic of pixels with different responses. This can be achieved by placing a small filter on each sensor location.* **Right** *shows one traditional such pattern of filters, a Bayer pattern.*

image to a 200×200 image, you could simply double each pixel (replace each pixel with a 2×2 block with the same value). But most upsampling requires finding intermediate values, which interpolation provides.

2.1.1 Color Images

Humans see color by comparing the response of different kinds of photoreceptor at nearby locations (Chapter 33.2). The main difference between these kinds of photoreceptor is in the sensitivity of the sensor with wavelength. Roughly, one type of sensor responds more strongly to longer wavelengths, another to medium wavelengths, and a third to shorter wavelengths (there are other kinds of sensor, and other differences).

Cameras parallel this process. The sensors used for the R (or red) layer of an RGB image respond more strongly to longer wavelengths; for the G (or green) layer, to medium wavelengths; and the B (or blue) to shorter wavelengths. Cameras must be engineered to produce the response of three different types of sensor *at the same place*. This can be done by using three imaging sensors and arranging for each sensor to receive the same light (lenses, mirrors, that sort of thing). Such *multiccd cameras* tend to be larger, heavier and more expensive. The more common strategy is to use one imaging sensor, and arrange that different pixels respond differently to wavelength. Typically, there are three types of pixel (R, G, and B), interleaved in a *mosaic* (Figure 33.2). This means that at many locations the camera does not measure R (or G, or B) response, and it obtains a value by interpolation.

Generally, mosaic patterns have more G pixels than R or B pixels. This is because G pixels are sensitive to a wider range of visible wavelengths than R and B pixels, and so the interpolation yields better results. Regular mosaic patterns

FIGURE 2.7: *A typical camera response function, mapping the response a linear sensor would compute to the output recorded by the camera. Notice that locations that would be quite dark for a linear sensor will be lighter; but as the linear sensor gets very bright, the output recorded by the camera grows slowly. This means that the range of outputs is smaller than the range of inputs, which is helpful for practical cameras. This response function is typically located deep in the camera's electronics. Typical consumer cameras apply a variety of transforms before reporting an image, though one can often persuade cameras to produce an untransformed, linear response image (a RAW file).*

can create effects in images, and there are *demosaicing* algorithms to remove these effects.

2.2 TRANSFORMING PIXEL VALUES

Linear image sensors present problems. The *dynamic range* (ratio of largest value to smallest value) of spectral energy fields can be startlingly large (1e6: 1 is often cited). Simple consumer cameras report 8 bits (256 levels) of intensity per channel. A picture from a linear camera that reports 8 bits per channel will look strange, because even relatively simple scenes have a higher dynamic range than 255. One can build cameras that can report significantly higher dynamic ranges, but this takes work (Chapter 33.2). If the camera has a linear response and a dynamic range of 255, either a lot of the image will be too dark to be resolved, or much of the image will be at the highest value, or both will happen. This is usually fixed by ensuring that the number digitized by the camera *isn't* linearly related to brightness. Internal electronics ensures that the *camera response function* mapping the intensity arriving at the sensor to the reported pixel value looks something like Figure 2.7. This increases the response to dark values, and reduces it to light values, so that the overall distribution of pixel values is familiar. Typically, the function used approximates the response of film (which isn't linear) because people are familiar with that. A camera response function is one example of a *pointwise image transformation*.

Most such transformations occur *after* the image has been digitized. You

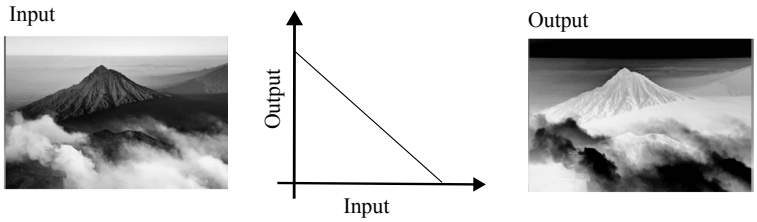


FIGURE 2.8: *Mapping individual pixel values using the mapping in the* **center** *will transform the image on the* **left** *to that on the* **right**. *This function maps light pixel values to dark ones, and vice versa, and is often called a negative.*

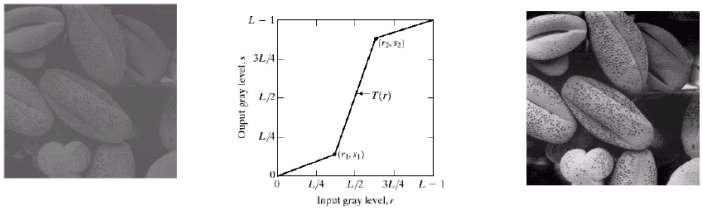


FIGURE 2.9: *Mapping individual pixel values using the mapping in the* **center** *will transform the image on the* **left** *to that on the* **right**. *Notice how the mapping compresses the range of dark and light pixels, and expands that of mid-range pixels, so adjusting the contrast of the image.*

take the array of pixels and apply some function to each pixel value. Simple, but useful, examples include: forming a negative (map x to $1 - x$, Figure 2.8); contrast adjustment (choose a function that makes dark pixels darker and light pixels lighter, Figure 2.9); and gamma correction (using a function that corrects for a quirk of image encoding, Figure 2.10).

2.3 TRANSFORMING AND WARPING IMAGES

Two usual conventions for image coordinate systems is shown in Figure 2.11. The inversion of the y-axis and of the order of coordinates in one is an annoying leftover from the way matrices are indexed. When I write $\mathcal{I}_{ij}$, I mean this coordinate system, so $\mathcal{I}_{00}$ is the top left corner of the image, and $0 \leq i \leq M$ and $0 \leq j \leq N$ – the image is $M \times N$ pixels. In this coordinate system, increasing i goes down the image, and increasing j moves to the left. When I write $\mathcal{I}(x, y)$, I mean a coordinate system in which the bottom left of the image is $(0, 0)$, the top right of the image is $(1, a)$ (where a would be 1 if the image was square, $a < 1$ if the image is short and wide, and $a > 1$ if the image is tall and narrow). In this view of an image, we will not worry about the number of pixels; if x, y refers to a point that isn't on the pixel grid, I assume it is reconstructed using bilinear interpolation.

Write $\mathcal{S}$ for a source image and $\mathcal{T}$ for a target image. Many important transformations have the form $\mathcal{T}(u(x, y), v(x, y)) = \mathcal{S}(x, y)$. Simpler examples include:

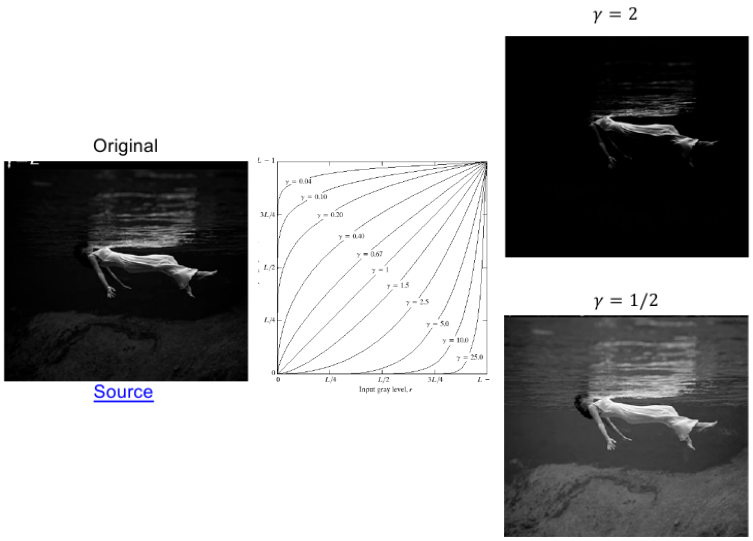


FIGURE 2.10: *Many imaging and rendering devices have a response that is a power of the input, so that output* $= C input^{\gamma}$*, where* γ *is a parameter of the device. One can simulate this effect by applying a transform like those shown in the* **center** *(curves for several values of* γ*). Note that you can remove the effect of such a transform* – gamma correct *the image – by applying another such transform with an appropriately chosen* γ*. The image on the* **left** *is transformed to the two examples on the* **right** *with different* γ *values.*

- **Translating an image** where $u(x, y) = x + t_x$, $v(x, y) = y + t_y$ (check that if $t_x > 0, t_y > 0$, the image moves up and to the right, as in the figure).
- **Rotating an image** where $u(x, y) = x \cos\theta - y \sin\theta$, $v(x, y) = x \sin\theta + y \cos\theta$ (check that if $\theta > 0$, the rotation is counterclockwise, as in the figure).
- **A Euclidean transformation** is a rotation and scale, so $u(x, y) = x \cos\theta - y \sin\theta + t_x$, $v(x, y) = x \sin\theta + y \cos\theta + t_y$.

These cases are simpler, because the source does not shrink or grow, so we need not worry too much about sampling error.

There is a general algorithm that works well. Scan the target image pixels in some order. For each pixel location s, t *in the target image* obtain x, y so that $s = u(x, y)$, $t = v(x, y)$. Obtain the value of the source image at x, y, and place it in the s, t location. There are two cases where x, y may not be on the pixel grid: in one, it lies between four pixel locations, so you use a bilinear interpolate; in the other, it is way outside the source image, so you use some other value (usually, either light, dark or mid-gray, depending). This procedure is known as *inverse warping*.

You might wish to *forward warp* – scan the source image, compute the coordinates of each pixel in the new image, then place the pixel value at that location.

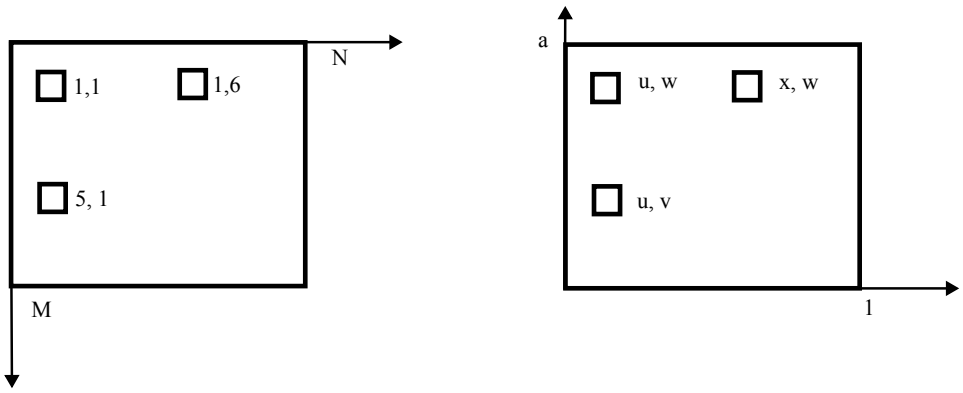


FIGURE 2.11: *Two common coordinate systems for images. On the* **left**, *the origin is at the top left corner, and we count in pixels. This is an $M \times N$ image. I will use the convention $\mathcal{I}_{ij}$ for points in this coordinate system, so the top right pixel is $\mathcal{I}_{16}$. On the right, the origin is at the bottom left, and the coordinate axes are more familiar. It is a good idea to use a range from $0-1$ (rather than $0-M$) in this coordinate system, but if the image is not square one direction will run from 0 to a. I will use the convention $\mathcal{I}(x,y)$ for points in this coordinate system, so that the bottom left pixel is $\mathcal{I}(u,v)$.*

If you do this, the target image will likely have holes in it, for reasons explained above (Section 2.1).

More complicated warps include

- **Scaling an image uniformly** where $u(x,y) = sx$, $v(x,y) = sy$ (check that if $s > 1$, the image gets bigger, and if $s < 1$, it gets smaller, as in the figure).
- **Scaling an image non-uniformly** where $u(x,y) = sx$, $v(x,y) = ty$.
- **Affine transformations** where $u(x,y) = ax + by + c$, $v(x,y) = dx + ey + f$.
- **Projective transformations** where $u(x,y) = \frac{ax+by+c}{gx+hy+i}$, $v(x,y) = \frac{dx+ey+f}{gx+hy+i}$.

Generally, these transformations are implemented by inverse warping, but you may need to take care to smooth the image first. This is because these transformations could cause the image to get smaller. In turn, there is a danger of aliasing (as in Section 2.1). This can be reduced by smoothing the source image before warping it.

2.4 IMAGES IN TERMS OF OTHER IMAGES

On occasion, it is useful to represent images in terms of other images. For example, imagine you have a large set of face images. It is natural to think about (a) what the "average" face looks like and (b) how a particular face is different from the average. A representation on principal components achieves this.

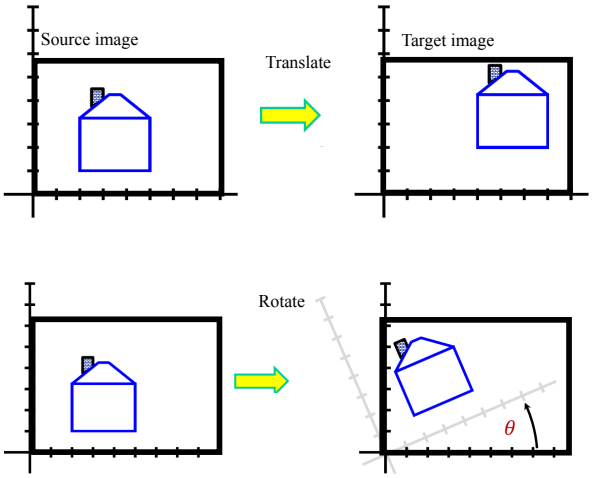


FIGURE 2.12: *Write $\mathcal{S}$ for a source image and $\mathcal{T}$ for a target image.* **Top row** *shows an image translation which maps the source pixel at (x, y) to the location $x + t_x$, $y + t_y$ in the target (so $\mathcal{T}(x + t_x, y + t_y) = \mathcal{S}(x, y)$). You should confirm that for this figure, $t_x > 0, t_y > 0$.* **Bottom row** *shows an image rotation, where where $\mathcal{T}(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) = \mathcal{S}(x, y)$. You should confirm that in this figure, $\theta > 0$. Generally, one implements these transformations by an inverse warp (see text). In each figure, the dark frames show the set of pixels in the source and target images. Notice that some target pixel values may be unknown, because their value comes from outside the source frame.*

2.4.1 Quick Principal Components Analysis

Assume we have a dataset of N d-dimensional vectors $\{\mathbf{x}\}$. This dataset has mean $\mathsf{mean}\,(\{\mathbf{x}\})$ and covariance $\mathsf{Covmat}\,(\{\mathbf{x}\})$. Principal components analysis yields a set of directions $\mathbf{p}_j$, which are eigenvectors of the covariance matrix. These directions are orthonormal (so that $\mathbf{p}_i^T\mathbf{p}_j$ is one if $i = j$ and zero otherwise.

Any data item $\mathbf{x}_i$ can be represented as

$$\mathbf{x}_i = \mathsf{mean}\,(\{\mathbf{x}\}) + \sum_{j=1}^{w} s_{i,j}\mathbf{p}_j.$$

Most datasets have the remarkable property that this representation has very low error even when w is considerably smaller than d (Chapter 33.2 if you haven't seen this before). The coefficents $s_{i,j}$ have strong properties, too. First, the mean over the dataset of each coefficient is zero, so

$$\frac{1}{N}\sum_i s_{i,j} = 0$$

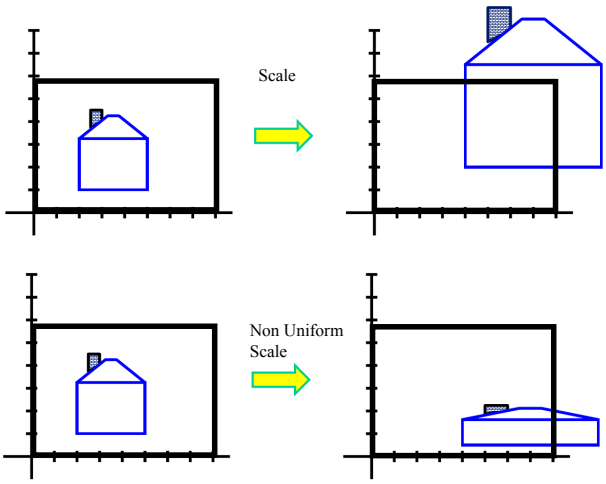


FIGURE 2.13: **Top row** *shows a uniform image scale which maps the source pixel at* (x, y) *to the location* sx, sy *in the target (so* $\mathcal{T}(sx, sy) = \mathcal{S}(x, y)$*). You should confirm that for this figure,* $s > 1$. **Bottom row** *shows a non-uniform image scale which maps the source pixel at* (x, y) *to the location* sx, ty *in the target (so* $\mathcal{T}(sx, ty) = \mathcal{S}(x, y)$*). You should confirm that for this figure,* $s > 1 > t$. *Generally, one implements these transformations by an inverse warp, but doing so requires care: because the image changes size, it may need smoothing before rescaling.*

and second, the directions can be ordered by the variance of the coefficients. Write

$$\mathsf{var}\left(\{s\}\right)_j = \frac{1}{N}\sum_i s_{i,j}^2$$

for the variance of the j'th coefficient; then if $k > j$, $\mathsf{var}\left(\{s\}\right)_k < \mathsf{var}\left(\{s\}\right)_j$. Note that $\mathsf{var}\left(\{s\}\right)_j$ is the j'th largest eigenvalue of the covariance. The $\mathbf{p}_j$ are known as *principal components* (sometimes *loadings*) of the dataset; the $s_{i,j}$ are sometimes known as *scores*, but are usually just called *coefficients*. Forming the representation is called *principal components analysis* or *PCA*.

2.4.2 Example: Representing Faces with Principal Components

An image is usually represented as an array of values. We will consider intensity images, so there is a single intensity value in each cell. You can turn the image into a vector by rearranging it, for example stacking the columns onto one another. This means you can take the principal components of a set of images. Doing so was something of a fashionable pastime in computer vision for a while, though there are some reasons that this is not a great representation of pictures. However, the representation yields pictures that can give great intuition into a dataset.

Figure 2.16 shows the mean of a set of face images encoding facial expressions of Japanese women (available at `http://www.kasrl.org/jaffe.html`; there are tons of face datasets at `http://www.face-rec.org/databases/`). I reduced the

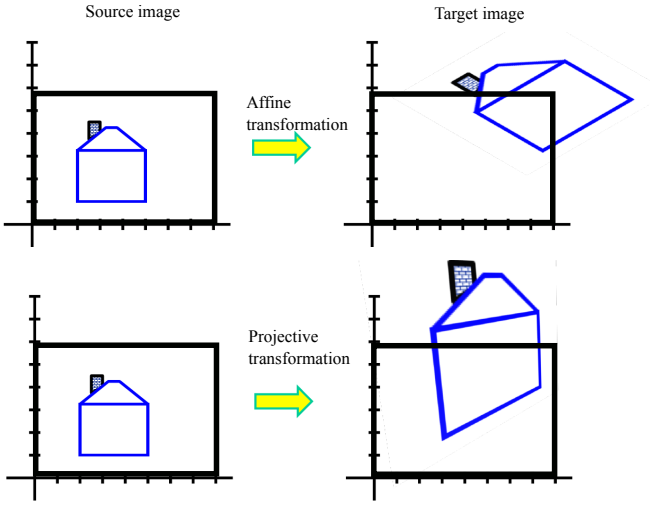


FIGURE 2.14: **Top row** *shows an affine transformation of the image which maps the source pixel at* (x, y) *to the location* $ax + by + c$, $dx + ey + f$ *in the target (so* $\mathcal{T}(ax + by + c, dx + ey + f) = \mathcal{S}(x, y)$*). You should confirm that, for this figure,* $(ae - bd) > 0$. **Bottom row** *shows a projective transformation of the image, which maps the source pixel at* (x, y) *to the location* $(\frac{ax+by+c}{gx+hy+i}, \frac{dx+ey+f}{gx+hy+i})$ *(so* $\mathcal{T}(\frac{ax+by+c}{gx+hy+i}, \frac{dx+ey+f}{gx+hy+i}) = \mathcal{S}(x, y)$*). Generally, one implements these transformations by an inverse warp, but doing so requires care: because the image changes size, it may need smoothing before rescaling.*

images to 64x64, which gives a 4096 dimensional vector. The eigenvalues of the covariance of this dataset are shown in figure 2.15; there are 4096 of them, so it's hard to see a trend, but the zoomed figure suggests that the first couple of hundred contain most of the variance. Once we have constructed the principal components, they can be rearranged into images; these images are shown in figure 2.16. Principal components give quite good approximations to real images (figure 2.17).

The principal components sketch out the main kinds of variation in facial expression. Notice how the mean face in Figure 2.16 looks like a relaxed face, but with fuzzy boundaries. This is because the faces can't be precisely aligned, because each face has a slightly different shape. The way to interpret the components is to remember one adjusts the mean towards a data point by adding (or subtracting) some scale times the component. So the first few principal components have to do with the shape of the haircut; by the fourth, we are dealing with taller/shorter faces; then several components have to do with the height of the eyebrows, the shape of the chin, and the position of the mouth; and so on. These are all images of women who are not wearing spectacles. In face pictures taken from a wider set of models, moustaches, beards and spectacles all typically appear in the first couple of dozen principal components.

A representation on enough principal components results in pixel values that are closer to the true values than the measurements (this is one sense of the word

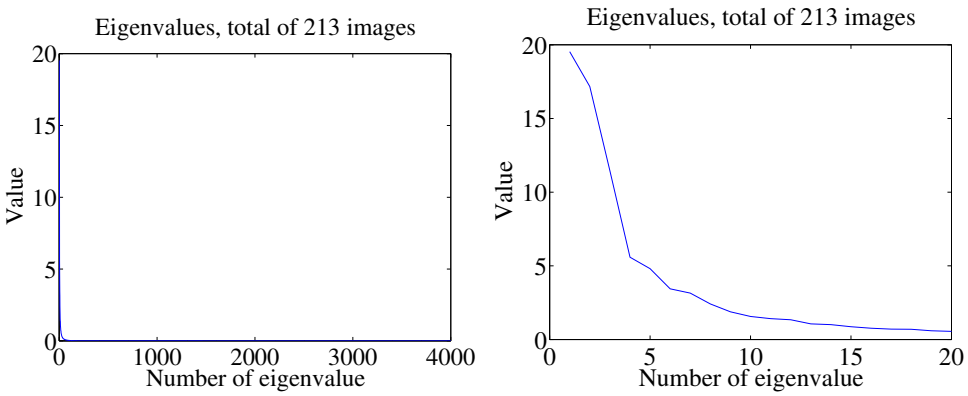


FIGURE 2.15: *On the* **left**, *the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right**, *a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

"smoothing"). Another sense of the word is blurring. Irritatingly, blurring reduces noise, and some methods for reducing noise, like principal components, also blur (figure 2.17). But this doesn't mean the resulting images are better *as images*. In fact, you don't have to blur an image to smooth it. Producing images that are both accurate estimates of the true values and look like sharp, realistic images requires quite substantial technology, beyond our current scope.

FIGURE 2.16: *The mean and first 16 principal components of the Japanese facial expression dataset.*

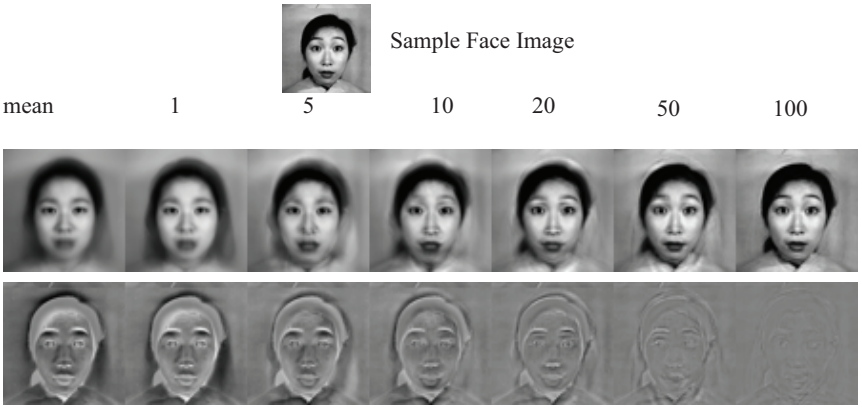


FIGURE 2.17: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*

CHAPTER 3

Patterns, Smoothing and Filters

TODO: convolution vs filtering **TODO:** associative **TODO:** linear **TODO:** shift invariant - equivariance **TODO:** useful non-linear filters **TODO:** padding **TODO:** stride

Pictures of zebras and of dalmatians have black and white pixels, and in about the same number, too. The difference between the two – stripes vs. spots – are obvious when you look at small groups of pixels. Pixels tend to look like their neighbors, so when a pixel disagrees with its neighbors, it might have been affected by noise. So looking at small groups of pixels is a simple and effective way to suppress noise, too. In this chapter, we introduce methods for obtaining descriptions of the appearance of a small group of pixels. The key process here is forming weighted sums of pixel values using different patterns of weights. to find different image patterns.

3.1 LINEAR FILTERS AND CONVOLUTION

3.1.1 Pattern Detection by Convolution

For the moment, think of an image as a two dimensional array of intensities. Write $\mathcal{I}_{ij}$ for the pixel at position i, j. We will construct a small array (a *mask* or *kernel*) $\mathcal{W}$, and compute a new image $\mathcal{N}$ from the image and the mask, using the rule

$$\mathcal{N}_{ij} = \sum_{uv} \mathcal{I}_{i-u,j-v} \mathcal{W}_{uv}$$

which we will write

$$\mathcal{N} = \mathcal{W} * \mathcal{I}.$$

In some sources, you might see $\mathcal{W} * * \mathcal{I}$ (to emphasize the fact that the image is 2D). We sum over all u and v that apply to $\mathcal{W}$; for the moment, do not worry about what happens when an index goes out of the range of $\mathcal{I}$. This operation is known as *convolution*, and $\mathcal{W}$ is often called the *kernel* of the convolution. You should look closely at the expression; the "direction" of the dummy variable u (resp. v) has been reversed compared with what you might expect (unless you have a signal processing background). What you might expect – sometimes called *correlation* or *filtering* – would compute

$$\mathcal{N}_{ij} = \sum_{uv} \mathcal{I}_{i+u,j+v} \mathcal{W}_{uv}$$

which we will write

$$\mathcal{N} = \texttt{filter}(\mathcal{I}, \mathcal{W}).$$

This difference isn't particularly significant, but if you forget that it is there, you compute the wrong answer.

We carefully avoid inserting the range of the sum; in effect, we assume that the sum is over a large enough range of u and v that all nonzero values are taken

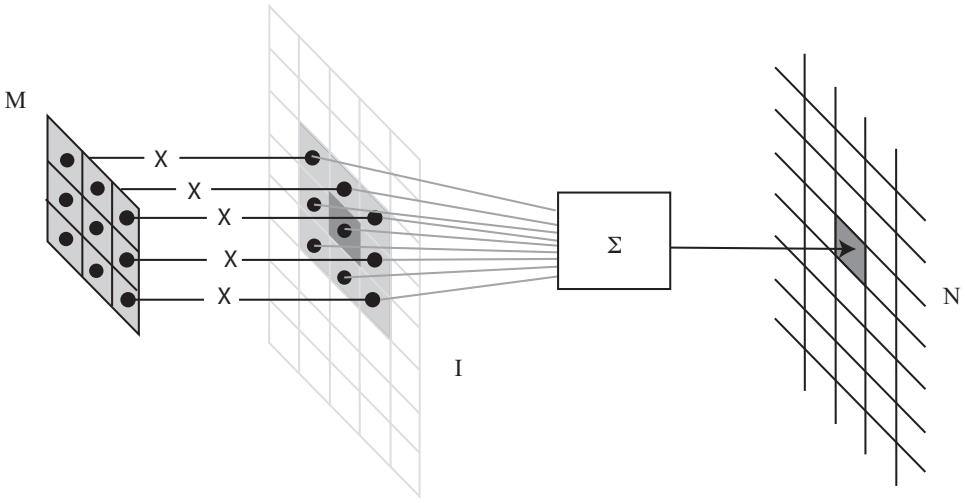


FIGURE 3.1: *To compute the value of $\mathcal{N}$ at some location, you shift a copy of $\mathcal{M}$ (the flipped version of $\mathcal{W}$) to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results.*

into account. Furthermore, we assume that any values that haven't been specified are zero; this means that we can model the kernel as a small block of nonzero values in a sea of zeros. We use this common convention regularly in what follows.

This operation is linear. You should check that:

- if $\mathcal{I}$ is zero, then $\mathcal{W} * \mathcal{I}$ is zero;
- $(k\mathcal{W}) * \mathcal{I}) = k(\mathcal{W} * \mathcal{I}) = \mathcal{W} * (k\mathcal{I})$;
- $(\mathcal{W}) * \mathcal{I}) = k(\mathcal{W} * \mathcal{I}) = \mathcal{W} * (k\mathcal{I})$;
- $(\mathcal{W} + \mathcal{V}) * \mathcal{I}) = \mathcal{W} * \mathcal{I} + \mathcal{V} * \mathcal{I}$;
- $\mathcal{W} * (\mathcal{I} + \mathcal{J}) = \mathcal{W} * \mathcal{I} + \mathcal{W} * \mathcal{J}$;
- $\mathcal{W} * (\mathcal{V} * \mathcal{I}) = (\mathcal{W} * \mathcal{V}) * \mathcal{I}$ (so convolution is associative).

3.1.2 Convolution as Pattern Detection

You should think of the value of $\mathcal{N}_{ij}$ as a dot-product. To see this, flip $\mathcal{W}$ in both directions to form $\mathcal{M}$. Then

$$\begin{aligned}\mathcal{N}_{ij} &= \sum_{uv} \mathcal{I}_{i-u,j-v}\mathcal{W}_{uv} \\ &= \sum_{uv} \mathcal{I}_{i+u,j+v}\mathcal{M}_{uv}\end{aligned}$$

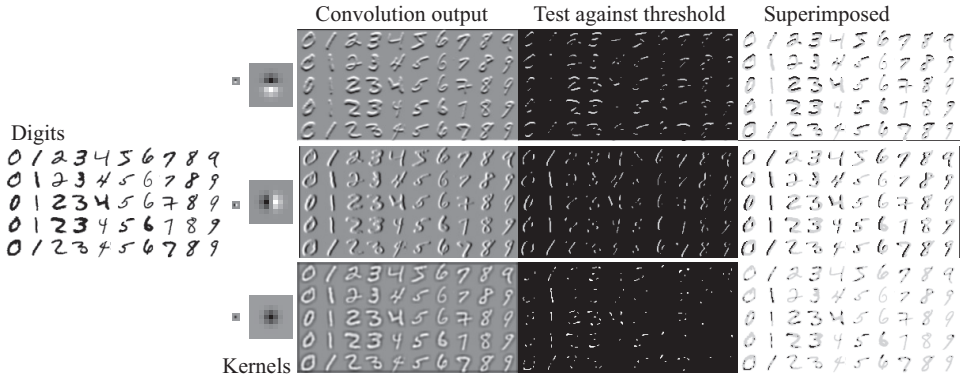


FIGURE 3.2: *On the* **far left**, *some images from the MNIST dataset. Three kernels appear on the* **center left**; *the small blocks show the kernels scaled to the size of the image, so you can see the size of the piece of image the kernel is applied to. The larger blocks show the kernels (mid-grey is zero; light is positive; dark is negative). The kernel in the top row responds most strongly to a dark bar above a light bar; that in the middle row responds most strongly to a dark bar to the left of a light bar; and the bottom kernel responds most strongly to a spot.* **Center** *shows the results of applying these kernels to the images. You will need to look closely to see the difference between a medium response and a strong response.* **Center right** *shows pixels where the response exceeds a threshold. You should notice that this gives (from top to bottom): a horizontal bar detector; a vertical bar detector; and a line ending detector. These detectors are moderately effective, but not perfect.* **Far right** *shows detector responses (in black) superimposed on the original image (grey) so you can see the alignment between detections and the image.*

equivalently

$$\begin{aligned}\mathcal{N} &= \mathcal{I} * \mathcal{W} \\ &= \texttt{filter}(\mathcal{I}, \mathcal{M})\end{aligned}$$

This means that you can think about convolution like this. To compute the value of $\mathcal{N}$ at some location, you place $\mathcal{M}$ (the flipped version of $\mathcal{W}$) at some location in the image; you multiply together the elements of $\mathcal{I}$ and $\mathcal{M}$ that lie on top of one another, ignoring everything in $\mathcal{I}$ outside $\mathcal{M}$; then you sum the results (Figure 3.1).

In turn, you can think about convolution as forming a dot product between $\mathcal{M}$ and the piece of image that lies under $\mathcal{M}$ (reindex the two windows to be vectors). This view explains why a convolution is interesting: it is a very simple pattern detector. Assume that $\mathbf{u}$ and $\mathbf{v}$ are unit vectors. Then $\mathbf{u} \cdot \mathbf{v}$ is largest when $\mathbf{u} = \mathbf{v}$, and smallest when $\mathbf{u} = -\mathbf{v}$. Using the dot-product analogy, for $\mathcal{N}_{ij}$ to have a large and positive value, the piece of image that lies under $\mathcal{M}$ must "look like" $\mathcal{M}$. Figure 3.2 give some examples.

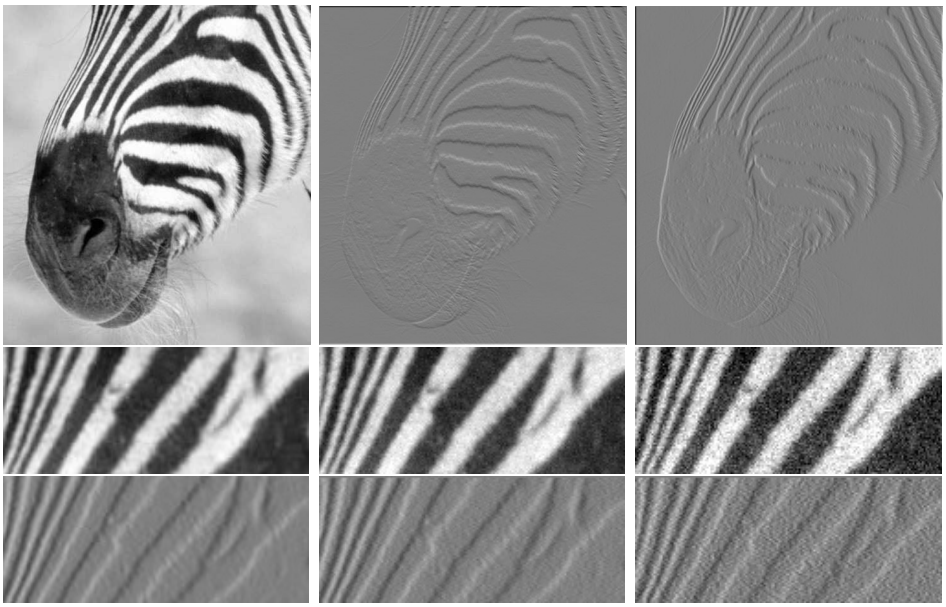


FIGURE 3.3: *The* **top row** *shows estimates of derivatives obtained by finite differences. The image at the* **left** *shows a detail from a picture of a zebra. The* **center** *image shows the partial derivative in the y-direction—which responds strongly to horizontal stripes and weakly to vertical stripes—and the* **right** *image shows the partial derivative in the x-direction—which responds strongly to vertical stripes and weakly to horizontal stripes. However, finite differences respond strongly to noise. The image at* **center left** *shows a detail from a picture of a zebra; the next image in the row is obtained by adding a random number with zero mean and normal distribution ($\sigma = 0.03$; the darkest value in the image is 0, and the lightest 1) to each pixel; and the third image is obtained by adding a random number with zero mean and normal distribution ($\sigma = 0.09$) to each pixel. The* **bottom row** *shows the partial derivative in the x-direction of the image at the head of the row. Notice how strongly the differentiation process emphasizes image noise; the derivative figures look increasingly grainy. In the derivative figures, a mid-gray level is a zero value, a dark gray level is a negative value, and a light gray level is a positive value.*

3.1.3 Convolution to Estimate Derivatives

Image derivatives can be approximated using another example of a convolution process. Because

$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon, y) - f(x,y)}{\epsilon},$$

we might estimate a partial derivative as a symmetric *finite difference*:

$$\frac{\partial h}{\partial x} \approx h_{i+1,j} - h_{i-1,j}.$$

This is the same as a convolution, where the convolution kernel is

$$\mathcal{H} = \begin{Bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{Bmatrix}.$$

Notice that this kernel could be interpreted as a template: it gives a large positive response to an image configuration that is positive on one side and negative on the other, and a large negative response to the mirror image.

As Figure 3.3 illustrates, finite differences give a most unsatisfactory estimate of the derivative. This is because finite differences respond strongly (i.e., have an output with large magnitude) at fast changes, and fast changes are characteristic of noise. Roughly, this is because image pixels tend to look like one another. For example, if we used a camera with some pixels that were stuck at either black or white, the output of the finite difference process would be large at those pixels because they are, in general, substantially different from their neighbors. We need some method to control image noise.

3.1.4 Smoothing with Convolution

The simplest model of image noise is the *additive stationary Gaussian noise* (or *Gaussian noise*) model, where each pixel has added to it a value chosen independently from the same Gaussian (normal) probability distribution. This distribution almost always has zero mean. The standard deviation is a parameter of the model. The model is intended to describe thermal noise in cameras and is illustrated in Figure 3.5.

Gaussian noise tends to result in pixels not looking like their neighbors. Other kinds of noise have this property too. Examples include: occasional pixels stuck at full dark or full bright; small random numbers with zero mean added to some pixel values; or some pixels multiplied by random numbers close to one. It is natural to attempt to reduce the effects of noise by replacing each pixel with a weighted average of its neighbors, a process often referred to as *smoothing*. When you smooth an image, it often looks as though it was taken by a defocussed camera, so smoothing is sometimes called *blurring*.

Replacing each pixel with an unweighted average computed over some fixed region centered at the pixel is the same as convolution with a kernel that is a block of ones multiplied by a constant (you should check this). This is a poor model of blurring; its output does not look like that of a defocused camera (Figure 3.6). The reason is clear. Assume that we have an image in which every point but the center point is zero, and the center point is one. If we blur this image by forming an unweighted average at each point, the result looks like a small, bright box, but this is not what defocused cameras do. We want a blurring process that takes a small bright dot to a circularly symmetric region of blur, brighter at the center than at the edges and fading slowly to darkness. As Figure 3.6 suggests, a set of weights of this form produces a much more convincing defocus model.

A good formal model for a fuzzy blob is the *symmetric Gaussian kernel*

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

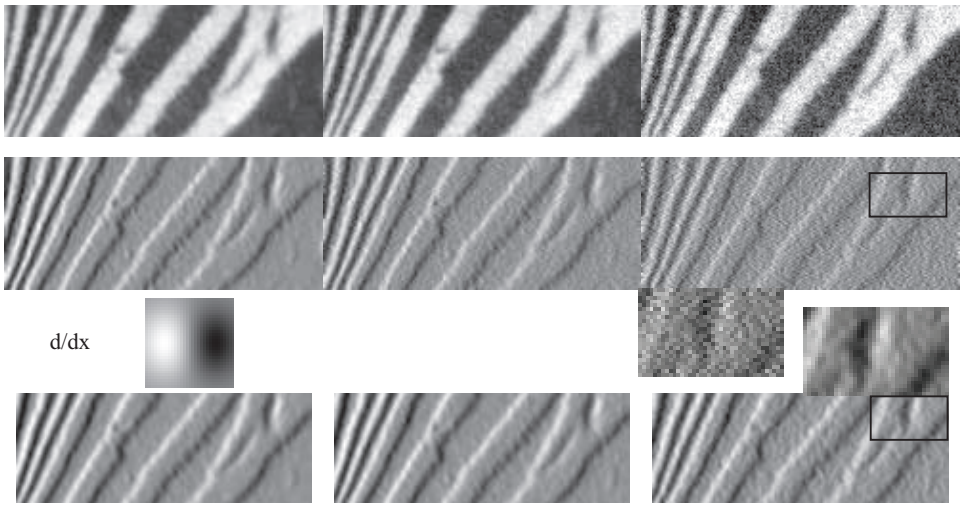


FIGURE 3.4: *Derivative of Gaussian filters are less extroverted in their response to noise than finite difference filters. The image at* **top left** *shows a detail from a picture of a zebra;* **top center** *shows the same image corrupted by zero mean stationary additive Gaussian noise, with* $\sigma = 0.03$ *(pixel values range from 0 to 1).* **Top right** *shows the same image corrupted by zero mean stationary additive Gaussian noise, with* $\sigma = 0.09$*. The second row shows the finite difference in the* x*-direction of each image. These images are scaled so that zero is mid-gray, the most negative pixel is dark, and the most positive pixel is light; we used a different scaling for each image. Notice how the noise results in occasional strong derivatives, shown by a graininess in the derivative maps for the noisy images. The final row shows the partial derivative in the* x*-direction of each image, in each case estimated by a derivative of Gaussian filter with* σ *one pixel. Again, these images are scaled so that zero is mid-gray, the most negative pixel is dark, and the most positive pixel is light; we used a different scaling for each image. The images are smaller than the input image, because we used a* 13×13 *pixel discrete kernel. This means that the six rows (resp. columns) on the top and bottom of the image (resp. left and right) cannot be evaluated exactly, because for these rows the kernel covers some points outside the image; we have omitted these values. Notice how the smoothing helps reduce the impact of the noise; this is emphasized by the detail images (between the second and final row), which are doubled in size. The details show patches that correspond from the finite difference image and the smoothed derivative estimate. We show a derivative of Gaussian filter kernel, which (as we expect) looks like the structure it is supposed to find. This is not to scale (it'd be extremely small if it were).*

illustrated in Figure 3.7. σ is referred to as the *standard deviation* of the Gaussian (or its "sigma"!); the units are interpixel spaces, usually referred to as *pixels*. This is a continuous fuzzy blob. You can get a discrete fuzzy blob by sampling it. The constant term makes the integral over the whole plane equal to one and is often

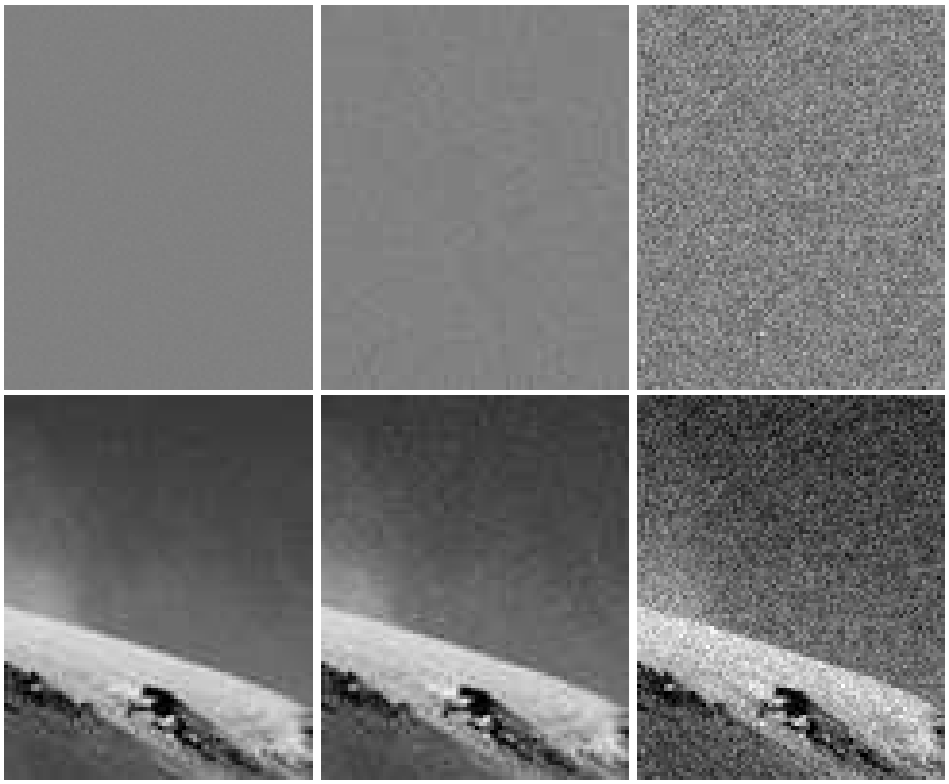


FIGURE 3.5: *The* **top row** *shows three realizations of a stationary additive Gaussian noise process. We have added half the range of brightnesses to these images to show both negative and positive values of noise. From left to right, the noise has standard deviation* $1/256$*,* $4/256$*, and* $16/256$ *of the full range of brightness, respectively. This corresponds roughly to bits zero, two, and five of a camera that has an output range of eight bits per pixel. The* **lower row** *shows this noise added to an image. In each case, values below zero or above the full range have been adjusted to zero or the maximum value accordingly.*

ignored in smoothing applications. The name comes from the fact that this kernel has the form of the probability density for a 2D normal (or Gaussian) random variable with a particular covariance.

This smoothing kernel forms a weighted average that weights pixels at its center much more strongly than at its boundaries. One can justify this approach qualitatively: Smoothing suppresses noise by enforcing the requirement that pixels should look like their neighbors. By downweighting distant neighbors in the average, we can ensure that the requirement that a pixel looks like its neighbors is less strongly imposed for distant neighbors. Choice of scale follows from the following considerations:

- If the standard deviation of the Gaussian is very small—say, smaller than one

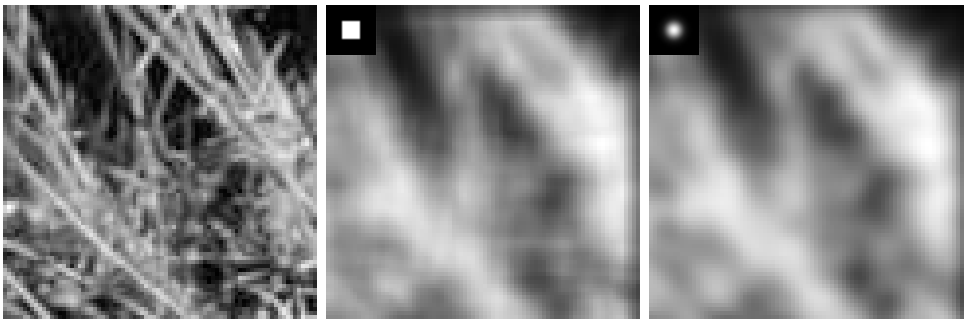


FIGURE 3.6: *Although a uniform local average may seem to give a good blurring model, it generates effects not usually seen in defocusing a lens. The images above compare the effects of a uniform local average with weighted average. The image on the* **left** *shows a view of grass; in the* **center**, *the result of blurring this image using a uniform local model; and on the* **right**, *the result of blurring this image using a set of Gaussian weights. The degree of blurring in each case is about the same, but the uniform average produces a set of narrow vertical and horizontal bars—an effect often known as* ringing. *The small insets show the weights used to blur the image, themselves rendered as an image; bright points represent large values and dark points represent small values (in this example, the smallest values are zero).*

pixel—the smoothing will have little effect because the weights for all pixels off the center will be very small.

- For a larger standard deviation, the neighboring pixels will have larger weights in the weighted average, which in turn means that the average will be strongly biased toward a consensus of the neighbors. This will be a good estimate of a pixel's value, and the noise will largely disappear at the cost of some blurring.

- Finally, a kernel that has a large standard deviation will cause much of the image detail to disappear, along with the noise.

Figure 3.8 illustrates these phenomena. You should notice that Gaussian smoothing can be effective at suppressing noise.

In applications, a discrete smoothing kernel is obtained by constructing a $2k + 1 \times 2k + 1$ array whose i, jth value is

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{((i-k-1)^2 + (j-k-1)^2)}{2\sigma^2}\right).$$

Notice that some care must be exercised with σ. If σ is too small, then only one element of the array will have a nonzero value. If σ is large, then k must be large, too; otherwise, we are ignoring contributions from pixels that should contribute with substantial weight.

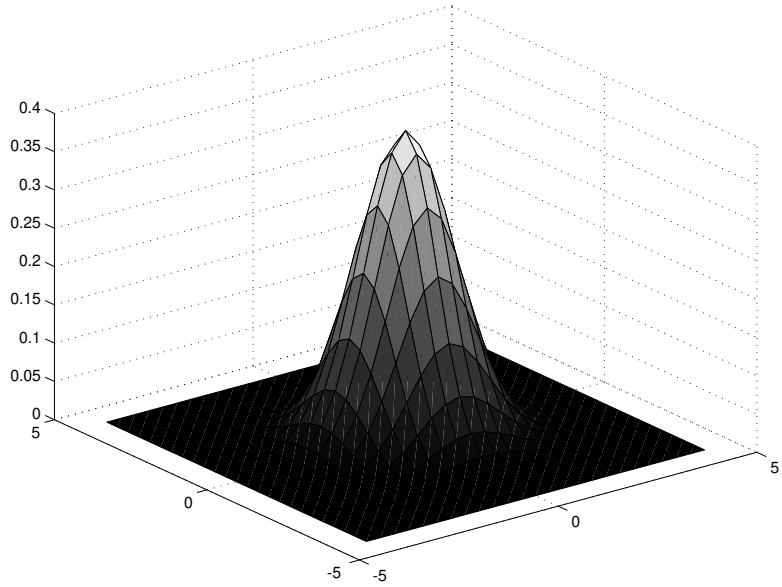


FIGURE 3.7: *The symmetric Gaussian kernel in 2D. This view shows a kernel scaled so that its sum is equal to one; this scaling is quite often omitted. The kernel shown has $\sigma = 1$. Convolution with this kernel forms a weighted average that stresses the point at the center of the convolution window and incorporates little contribution from those at the boundary. Notice how the Gaussian is qualitatively similar to our description of the point spread function of image blur: it is circularly symmetric, has strongest response in the center, and dies away near the boundaries.*

3.2 ESTIMATING GRADIENTS AND ORIENTATIONS

We have seen that finite differences give poor estimates of image gradients for noisy images. Any image gradient of significance to us has effects over a pool of pixels. For example, the contour of an object can result in a long chain of points where the image derivative is large. As another example, a corner typically involves many tens of pixels. If the noise at each pixel is independent and additive, then large image derivatives caused by noise are a local event. Smoothing the image before we differentiate will tend to suppress noise at the scale of individual pixels, because it will tend to make pixels look like their neighbors. However, gradients that are supported by evidence over multiple pixels will tend not to be smoothed out. This suggests differentiating a smoothed image (Figure 3.4).

3.2.1 Derivative of Gaussian Filters

The convolution I have described is a sampled analogue of a continuous operation. If $I(x, y)$ is a continuous image, and $W(x, y)$ is some continuous function, we have $(W * I)(x, y) = \int_{u,v} I(x-u, y-v)W(u, v)dudv$. Notice that, to make this definition meaningful, we need to make assumptions about the range of the integral and the

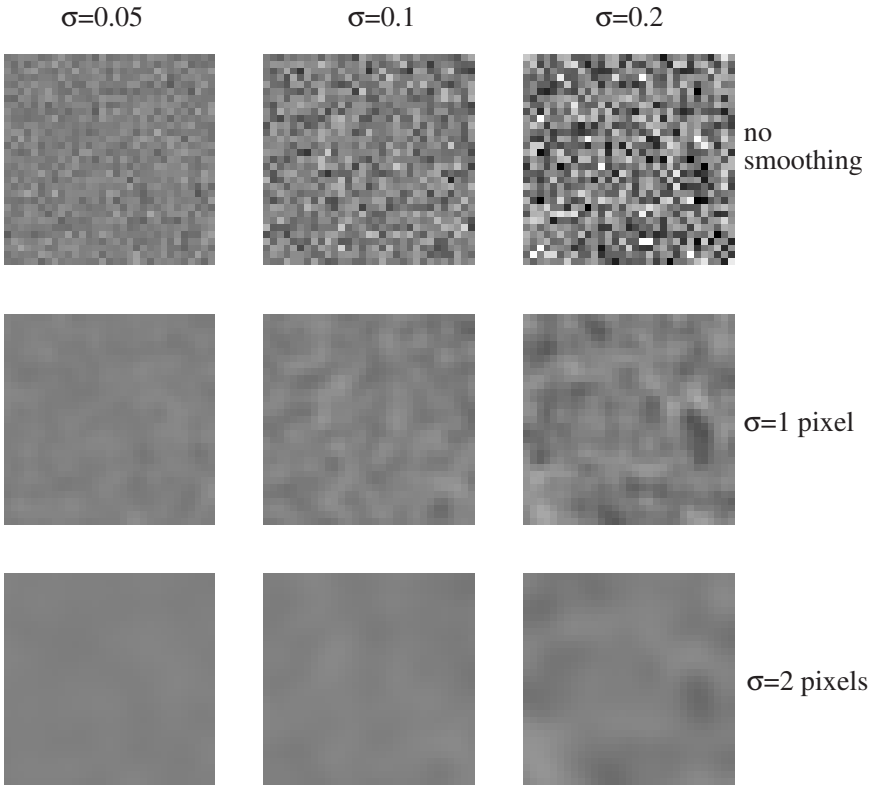


FIGURE 3.8: *The* **top row** *shows images of a constant mid-gray level corrupted by additive Gaussian noise. In this noise model, each pixel has a zero-mean normal random variable added to it. The range of pixel values is from zero to one, so that the standard deviation of the noise in the first column is about 1/20 of full range. The* **center row** *shows the effect of smoothing the corresponding image in the top row with a Gaussian filter of σ one pixel. Notice the annoying overloading of notation here; there is Gaussian noise and Gaussian filters, and both have σ's. One uses context to keep these two straight, although this is not always as helpful as it could be, because Gaussian filters are particularly good at suppressing Gaussian noise. This is because the noise values at each pixel are independent, meaning that the expected value of their average is going to be the noise mean. The* **bottom row** *shows the effect of smoothing the corresponding image in the top row with a Gaussian filter of σ two pixels.*

values of the image and kernel that are analogous to those we made for the sampled case. Context will tell whether I am referring to the sampled operation (commonly) or the continuous operation (seldom). Convolution has an important property. Write $\texttt{shift}(\mathcal{I})$ for an operation that shifts an image (so $\texttt{shift}(\mathcal{I})_{i+tx,j+t_y} = I_{ij}$, or in the continuous case, $\texttt{shift}(\mathcal{I})(\S + \sqcup_\S, \dagger + \sqcup_\dagger) = \mathcal{I}(\S, \dagger)$). Convolution is *shift invariant*, meaning that $\mathcal{W} * \texttt{shift}(\mathcal{I}) = \texttt{shift}(\mathcal{W} * \mathcal{I})$. This is true for both sampled and continuous operations. It can be proven that if an operation is (a)

linear and (b) shift invariant, then there is some convolution kernel that implements it. You should remember this result.

Smoothing an image and then differentiating it is the same as convolving it with the derivative of a smoothing kernel. This fact is most easily seen by thinking about continuous convolution. First, differentiation is linear and shift invariant. This means that there is some kernel—we dodge the question of what it looks like—that differentiates. That is, given a function $I(x, y)$,

$$\frac{\partial I}{\partial x} = K_{(\partial/\partial x)} * I.$$

Now we want the derivative of a smoothed function. We write the convolution kernel for the smoothing as S. Recalling that convolution is associative, we have

$$(K_{(\partial/\partial x)} * (S * I)) = (K_{(\partial/\partial x)} * S) * I = (\frac{\partial S}{\partial x}) * I.$$

This fact appears in its most commonly used form when the smoothing function is a Gaussian. We can then write

$$\frac{\partial (G_\sigma * I)}{\partial x} = (\frac{\partial G_\sigma}{\partial x}) * I,$$

that is, we need only convolve with the derivative of the Gaussian, rather than convolve and then differentiate. As discussed in Section 7.3, smoothed derivative filters look like the effects they are intended to detect. The x-derivative filters look like a vertical light blob next to a vertical dark blob (an arrangement where there is a large x-derivative), and so on (Figure 7.11). Smoothing results in much smaller noise responses from the derivative estimates (Figure 3.4).

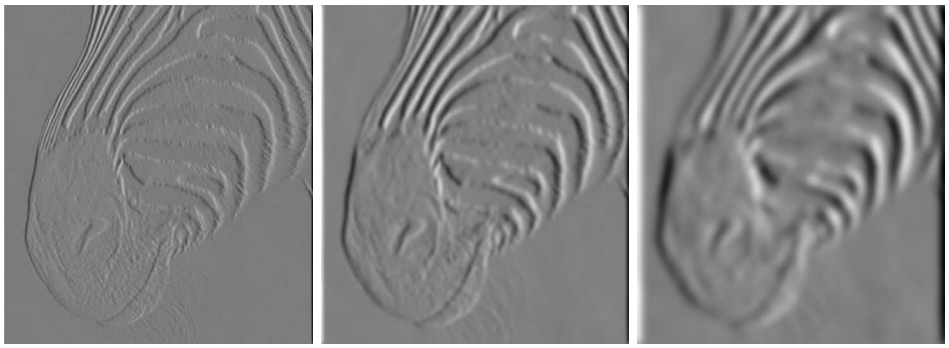


FIGURE 3.9: *The scale (i.e., σ) of the Gaussian used in a derivative of Gaussian filter has significant effects on the results. The three images show estimates of the derivative in the x direction of an image of the head of a zebra obtained using a derivative of Gaussian filter with σ one pixel, three pixels, and seven pixels (***left** *to* **right***). Note how images at a finer scale show some hair, the animal's whiskers disappear at a medium scale, and the fine stripes at the top of the muzzle disappear at the coarser scale.*

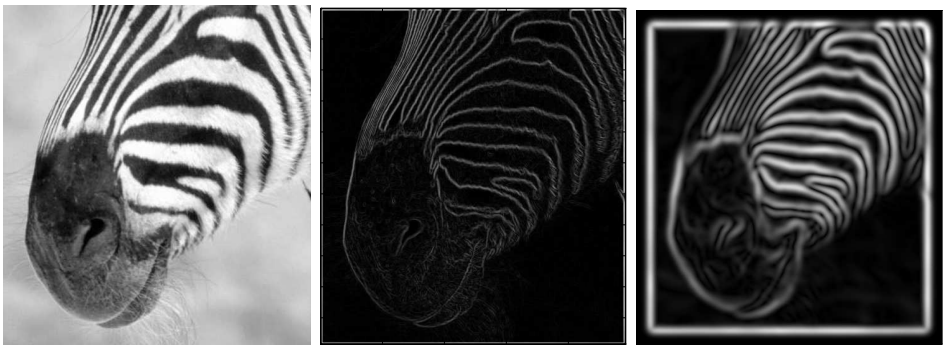


FIGURE 3.10: *The gradient magnitude can be estimated by smoothing an image and then differentiating it. This is equivalent to convolving with the derivative of a smoothing kernel. The extent of the smoothing affects the gradient magnitude; in this figure, we show the gradient magnitude for the figure of a zebra at different scales. At the* **center***, gradient magnitude estimated using the derivatives of a Gaussian with* $\sigma = 1$ *pixel; and on the* **right***, gradient magnitude estimated using the derivatives of a Gaussian with* $\sigma = 2$ *pixel. Notice that large values of the gradient magnitude form thick trails.*

The choice of σ used in estimating the derivative is often called the *scale* of the smoothing. Scale has a substantial effect on the response of a derivative filter. Assume we have a narrow bar on a constant background, rather like the zebra's whisker. Smoothing on a scale smaller than the width of the bar means that the filter responds on each side of the bar, and we are able to resolve the rising and falling edges of the bar. If the filter width is much greater, the bar is smoothed into the background and the bar generates little or no response (Figure 3.9).

3.2.2 Orientations

As the light gets brighter or darker (or as the camera aperture opens or closes), the image will get brighter or darker, which we can represent as a scaling of the image value. The image $\mathcal{I}$ will be replaced with $s\mathcal{I}$ for some value s. The magnitude of the gradient scales with the image, i.e., $\|\nabla\mathcal{I}\|$ will be replaced with $s|\nabla\mathcal{I}\|$. This creates problems for edge detectors, because edge points may appear and disappear as the image gradient values go above and below thresholds with the scaling. One solution is to represent the *orientation* of image gradient, which is unaffected by scaling (Figure 3.11). The gradient orientation field depends on the smoothing scale at which the gradient was computed. Orientation fields can be quite characteristic of particular textures (Figure **??**), and we will use this important property to come up with more complex features below.

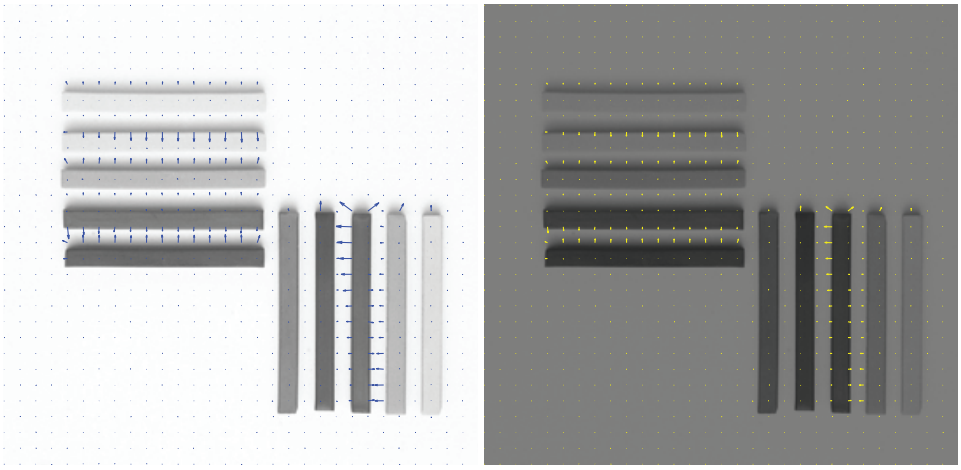


FIGURE 3.11: *The magnitude of the image gradient changes when one increases or decreases the intensity. The orientation of the image gradient does not change; we have plotted every 10th orientation arrow, to make the figure easier to read. Note how the directions of the gradient arrows are fixed, whereas the size changes.* Philip Gatward © Dorling Kindersley, used with permission.

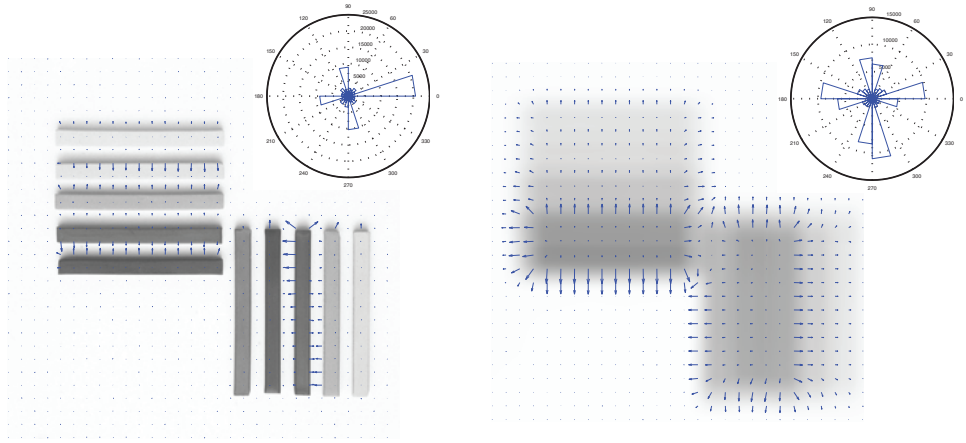


FIGURE 3.12: *The scale at which one takes the gradient affects the orientation field. We show the overall trend of the orientation field by plotting a rose plot, where the size of a wedge represents the relative frequency of that range of orientations.* **Left** *shows an image of artists pastels at a fairly fine scale; here the edges are sharp, and so only a small set of orientations occurs. In the heavily smoothed version on the* **right***, all edges are blurred and corners become smooth and blobby; as a result, more orientations appear in the rose plot.* Philip Gatward © Dorling Kindersley, used with permission.

CHAPTER 4

Simple Image Mosaics

Back in the days when photographs were printed on paper by special stores, one way to make a photograph of a large object was to take several different, overlapping pictures; print them; place one printed image down on a corkboard; then slide the other printed pictures around on a corkboard until they are registered to the first and one another; and then pin them down. This is a *mosaic* – a collection of pictures that have been registered (Figure 4.4). There are a number of reasons to build mosaics. You might simply not have the right camera, and so have to assemble a big picture out of small ones. In the overlapping portions, you have more than one picture, and can use this to make improved estimates of pixel values or to identify moving objects. You can show various interesting changes in the scene.

Methods for building mosaics out of digital images are now highly developed, and appear in a number of consumer applications. The key trick is registering the images. Procedures differ slightly depending on what class of geometric transformation is used to register the images. We will mostly discuss mosaics built by translating images; Chapter 33.2 describes methods relevant for more general transformations.

4.1 SIMPLE MOSAICS

For the moment, assume we have two images $\mathcal{A}$ and $\mathcal{B}$. These two images are overlapping views of a scene that can be aligned by a translation. These images are continuous functions of position in the image plane – they haven't been sampled yet. The fact that they can be aligned means that there is some t_x, t_y so that $\mathcal{A}(x, y) = \mathcal{B}(x - t_x, y - t_y)$ when both $x, y \in [0, 1] \times [0, 1]$ and $x + t_x, y + t_y \in [0, 1] \times [0, 1]$. Visualize this as placing $\mathcal{B}$ on top of $\mathcal{A}$, then sliding $\mathcal{B}$ by t_x, t_y; then the parts of $\mathcal{A}$ and $\mathcal{B}$ that overlap look the same. We are given $\mathcal{A}$ and $\mathcal{B}$ and must find t_x, t_y.

Least squares should spring to mind. We are dealing with sampled images, and so we will search for m, n that are integers, and minimize

$$C_{\text{reg}}(m, n) = \frac{1}{N_o} \sum_{\text{overlap}} (\mathcal{A}_{ij} - \mathcal{B}_{i-m,j-n})^2$$

where overlap is the rectangle of pixel locations with meaningful values for both $\mathcal{A}$ and $\mathcal{B}$ and N_o is the number of pixels in that rectangle. It is important that C_{reg} is an average, because we need to compare overlaps of different sizes (Figure 33.2)

Once we have a good estimate of the translation, we could refine it using bilinear interpolation (Section 33.2) to reconstruct values we don't have. But minimizing this cost function isn't just a piece of linear algebra. The obvious strategy for finding m, n is to apply each translation; compute the objective function; then

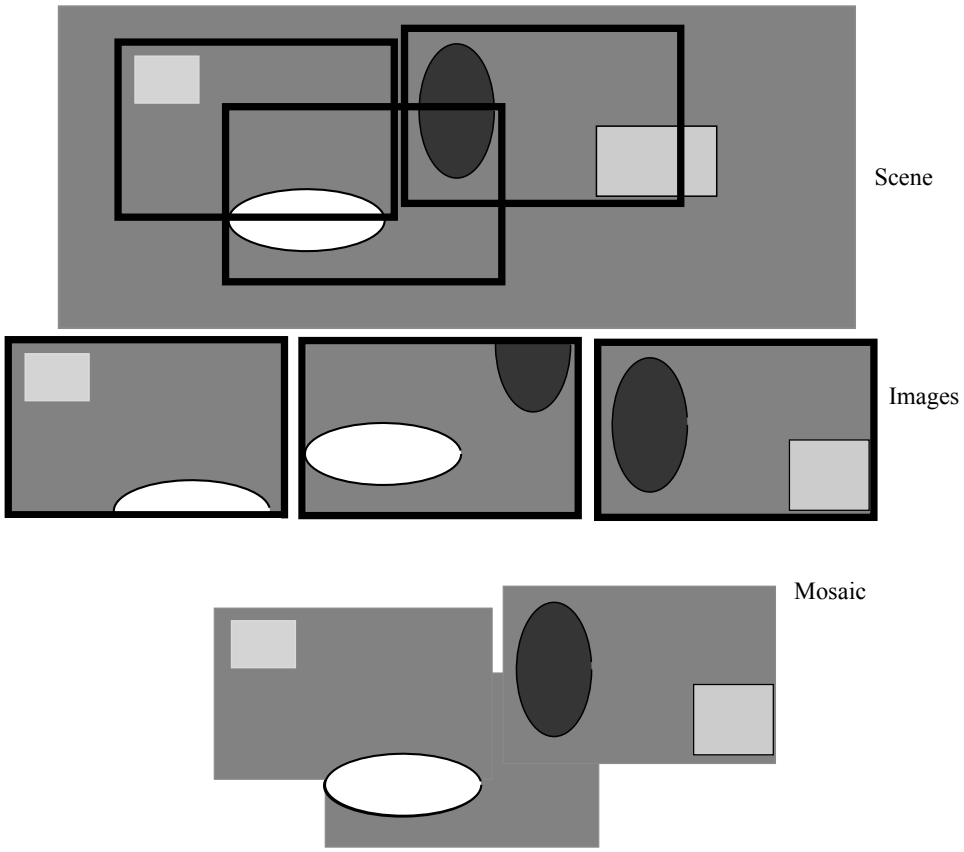


FIGURE 4.1: **Top** *shows a set of overhead images of a simple scene. The dark boundaries show each of three image frames.* **Center** *shows the actual images obtained in these frames.* **Bottom** *shows the mosaic that can be recovered by sliding images with respect to one another. This mosaic can't show features that haven't been imaged, but does show the relative configuration of scene components.*

take the translation with smallest value. This isn't a good strategy, because we must evaluate the objective function too often. The obvious modification – assume that m, n are in a small range, and search only those – doesn't help because we may not find the actual minimum. But notice that if we smoothed and subsampled $\mathcal{A}$ and $\mathcal{B}$, we could compute a coarse estimate of m, n from those, and then perhaps refine the estimate.

4.1.1 Registration, RGB, and Prokudin-Gorskii

Color photography is usually dated to the 1930's when it first became available to the public. In fact, James Clerk Maxwell described a method to capture a color photograph in an 1855 paper. The procedure likely looks straightforward to you: obtain three color filters, and take a picture of the scene through each of

these filters. Capturing these *color separations* presented a number of technical challenges, and the first color photograph was taken by Thomas Sutton in 1861. Actually displaying pictures obtained like this was tricky. One had to pass red light through the red separation, green through the green, and blue through the blue, then ensure all three resulting images lay on top of one another on screen. Turning them into the image files we are familiar with is also tricky, because each layer of the separation is typically a bit offset from the others (the camera moved slightly between photographs), and each layer has aged and been damaged slightly differently.

A class assignment, now hallowed by tradition in computer vision, but likely to have originated with A. Efros in 2010, studies this problem. It uses the pictures of Sergei Mikhailovich Prokudin-Gorskii (1863-1944) traveled the Russian empire and took color photographs of many scenes. He left Russia in 1918. His negatives survived and ended up in the Library of Congress. A digitized version of the collection is available online. The assignment asks students to register the color separations for some of these images.

This is very like forming a mosaic (the separations overlap; they can be aligned by translation). It presents two important challenges. First, the separations do not agree exactly when they overlap – if they did, the image would be a monochrome image – and so the cost function needs to be adjusted. Second, the high resolution version of the scans are quite big, and there can be moderately large offsets. Looking at each offset in turn is hideously expensive. We will deal with that problem in the next section.

Notation for generalizing the cost function is easy. Write $\mathcal{A}_{O(m,n)}$ (etc.) for the image window of $\mathcal{A}$ that overlaps the other image when that image is translated by m, n. We can write

$$C_{\text{reg}}(m,n) = d(\mathcal{A}_{O(m,n)}, \mathcal{B}_{O(m,n)}).$$

The original cost function had

$$d(\mathcal{A}_{O(m,n)}, \mathcal{B}_{O(m,n)}) = \frac{1}{N_o} \sum_{\text{overlap}} \left(\mathcal{A}_{ij} - \mathcal{B}_{i-m,j-n}\right)^2.$$

Useful alternatives include:

- The *cosine distance*, given by:

$$\sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} * \mathcal{B}_{i-m,j-n})}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}.$$

  Annoyingly, this cost function is largest when best, even though it's called a distance. Some authors subtract this distance from one (its largest value) to fix this.

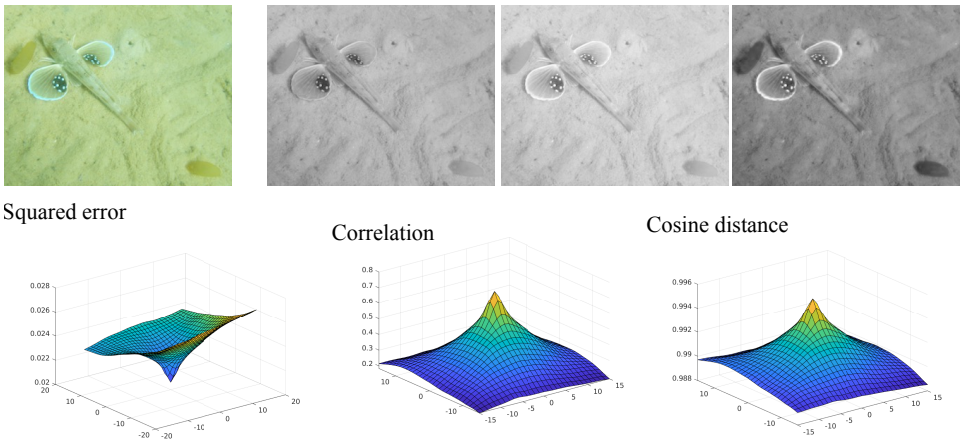


FIGURE 4.2: **Top left** *shows a Gurnard, flashing its pectoral fins in alarm, at Long Beach in Cape Town.* **Top rest** *shows the color separations of this image (in red, green, blue order). The image is slightly blue-green (taken at about 5 meters depth, where water absorbs red light), and this shows as a darker red separation.* **Bottom** *shows how various cost functions react to registering red to blue. The correct alignment is at 0, 0 and the images are 257 by 323. Notice that: all the extrema are in the right place, but the correlation and cosine distance must be maximized, and the squared error minimized; the squared error changes relatively little from the best to the worst, because the blue image is rather unlike the red; both cosine distance and correlation are much more sensitive.*

- The *correlation coefficient*, given by:

$$\sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} - \mu_A) * (\mathcal{B}_{i-m,j-n} - \mu_B)}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}$$

$$\text{where } \mu_A = \frac{1}{N_O} \sum_{overlap} \mathcal{A}_{ij} \text{ and}$$

$$\text{where } \mu_B = \frac{1}{N_O} \sum_{overlap} \mathcal{B}_{ij}.$$

  This is big for the best alignment. Notice how this corrects for the mean of the overlap in each window.

Each is in the range -1 to 1, and neither scales with the size of the overlap neighborhood. Terminology in this area is severely confused. The cosine distance isn't a distance; it is sometimes referred to as *normalized correlation*; and sometimes as *correlation*. Several functions similar to correlation are referred to as correlation.

4.2 TECHNIQUE: SCALE AND IMAGE PYRAMIDS

TODO: two zebra images?

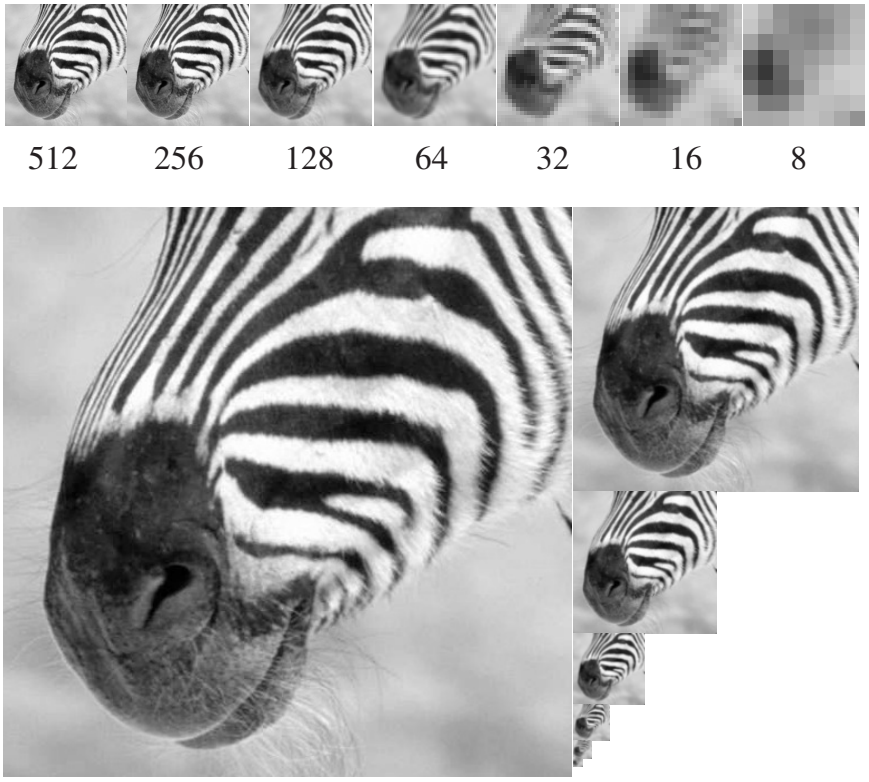


FIGURE 4.3: *A Gaussian pyramid of images running from 512x512 to 8x8. On the top row, we have shown each image at the same size (so that some have bigger pixels than others), and the lower part of the figure shows the images to scale. Notice that if we convolve each image with a fixed-size filter, it responds to quite different phenomena. An 8x8 pixel block at the finest scale might contain a few hairs; at a coarser scale, it might contain an entire stripe; and at the coarsest scale, it contains the animal's muzzle.*

Images look quite different at different scales. An *image pyramid* is a collection of smoothed and resampled representations of an image. The name comes from a visual analogy. Typically, each layer of the pyramid is half the width and half the height of the previous layer; if we were to stack the layers on top of each other, a pyramid would result. In a *Gaussian pyramid*, each layer is smoothed by a symmetric Gaussian kernel and resampled to get the next layer (Figure 4.3). These pyramids are most convenient if the image dimensions are a power of two or a multiple of a power of two. The smallest image is the most heavily smoothed; the layers are often referred to as *coarse scale* versions of the image.

Now look at the zebra's muzzle in Figure 4.3, and think about registering this image to itself. The 8×8 version has very few pixels, and looks like a medium dark bar, darker at the muzzle end. Finding a translation to register this image to itself should be fairly straightforward, and unambiguous. Assume we find m_8, n_8.

In the 16×16 version, some stripes are visible. Registering this image to itself might be more difficult, because the stripes will create local minima of the cost function (check you follow this remark; think about what happens if you have the images registered, and then shift the muzzle perpendicular to the stripes). But if we have an estimate of the translation from the 8×8 version, we do not need to search a large range of translations to register the 16×16 version. We need to look only at four translations: $2 * m_8, 2 * n_8$; $2 * m_8 + 1, 2 * n_8$; $2 * m_8, 2 * n_8 + 1$; and $2 * m_8 + 1, 2 * n_8 + 1$. The same reasoning applies when going from the 16×16 version to the 32×32 version, and so on. This strategy is known as *coarse-to-fine search*.

4.2.1 The Gaussian Pyramid

With a little notation, we can write simple expressions for the layers of a Gaussian pyramid. The operator $S^{\downarrow}$ downsamples an image; in particular, the j, kth element of $S^{\downarrow}(\mathcal{I})$ is the $2j$, $2k$th element of $\mathcal{I}$. The nth level of a pyramid $P(\mathcal{I})$ is denoted $P(\mathcal{I})_n$. With this notation, we have

$$\begin{aligned} P_{\text{Gaussian}}(\mathcal{I})_{n+1} &= S^{\downarrow}(G_\sigma * * P_{\text{Gaussian}}(\mathcal{I})_n) \\ &= (S^{\downarrow} G_\sigma) P_{\text{Gaussian}}(\mathcal{I})_n) \end{aligned}$$

(where we have written G_σ for the linear operator that takes an image to the convolution of that image with a Gaussian). The finest scale layer is the original image:

$$P_{\text{Gaussian}}(\mathcal{I})_1 = \mathcal{I}.$$

TODO: make this a procedure box

Set the finest scale layer to the image
For each layer, going from next to finest to coarsest
  Obtain this layer by smoothing the next finest
  layer with a Gaussian, and then subsampling it
end

Algorithm 4.1: *Forming a Gaussian Pyramid.*

TODO: procedure box for translation registering an image to another

4.3 BUILDING MOSAICS

Here is a simple procedure to build a mosaic from a set of images $\{\mathcal{I}_1, \ldots, \mathcal{I}_N\}$, sketched in Figure 4.4. Construct a large array of pixels value "unknown" to serve as the mosaic. Choose an image – say $\mathcal{I}_1$ – to place at the center, and transfer $\mathcal{I}_1$'s pixels to the corresponding locations in the mosaic. In Figure 4.4, step I shows this. Now find an image $\mathcal{I}_k$ which overlaps with an image in the mosaic and register it to that image, then update the mosaic using its pixels. For step II in Figure 4.4 this is $\mathcal{I}_2$ which is registered to $\mathcal{I}_1$. Proceed until there aren't any images that overlap

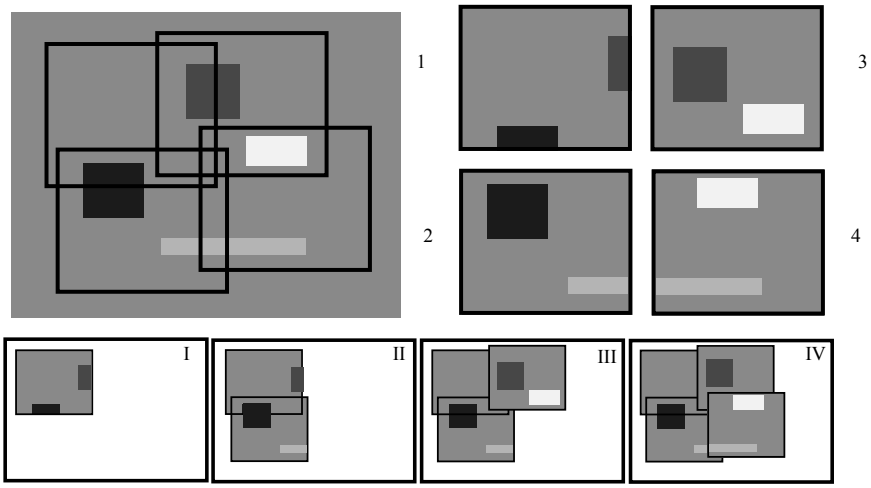


FIGURE 4.4: **Top left** shows a simple scene with the location of four images; these are shown on the **top right**. The steps of creating a mosaic are sketched in the Roman numeral panes on the **bottom**. In this case, the translation of the fourth image with respect to the first is poorly estimated; more detail in the text.

(step III, $\mathcal{I}_3$ to $\mathcal{I}_2$; step IV, $\mathcal{I}_4$ to $\mathcal{I}_3$). Now compute the pixel values in the mosaic using all the overlap information.

In some applications, the image with a good overlap will be obvious. For example, if you build a mosaic out of overhead aerial images, the images are going to be timestamped, and the next image will overlap rather well with the current image. In other cases, you can try to register coarse scale versions of the images with one another. Not all pairs will register, but those that do with a small enough translation will likely have a good overlap, and you can use this to obtain an overlapping image.

There are a variety of ways to compute a mosaic from registered images, depending on what one is trying to achieve. Many locations in the mosaic are overlapped by multiple images. At these locations, you have more than one estimate of the pixel value (one from each image that overlaps that location). One strategy is to simply average these values. There are more interesting possibilities than taking a mean. For example, imagine you want to build a mosaic that suppresses the movement of a moving object (Figure 33.2). Averaging will produce a mosaic with a blurred version of the object. Instead, collect all the values for each location, and take the median. As the figure illustrates, this will suppress the moving object. Alternatively, imagine you want to emphasize the movement; then collect all the values, and the value most unlike the median.

4.3.1 Bundle Adjustment

Our simple procedure does not produce the best possible mosaic. To see this, assume you have images $\mathcal{I}_1, \ldots, \mathcal{I}_4$, as in Figure 4.6, and you introduce them into the mosaic in that order. Write $T_{2\to 1}$ for the translation to align $\mathcal{I}_2$ with $\mathcal{I}_1$ by

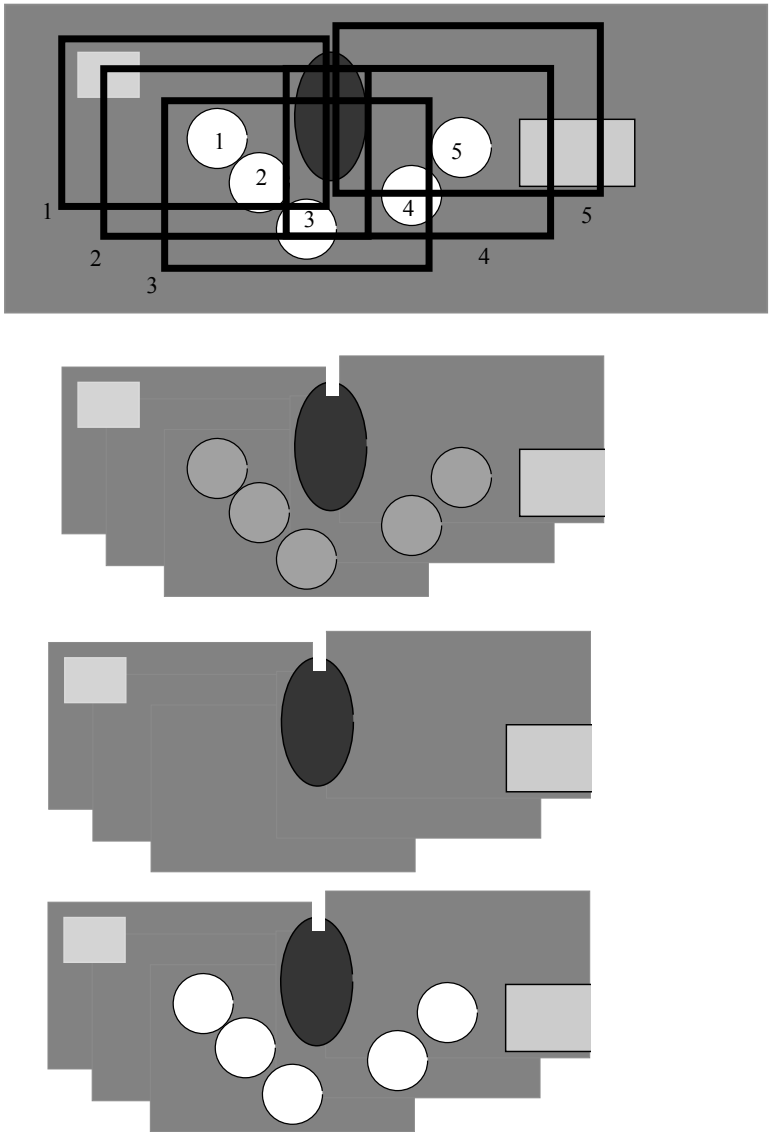


FIGURE 4.5: **Top** *shows a set of images of a simple scene with a moving object (the circle). This is at location 1 in frame 1, and so on. The background does not move. The frames are registered to one another to produce a mosaic. The value at a location in the mosaic is a summary of possibly many pixel values (one for each image that overlaps that location). Different procedures for summarization lead to quite different mosaics.* **Row 2** *shows what happens if one averages. The circles affect the average, and appear. But using a median will produce* **row 3** *– here the moving object has been suppressed, and we can see the background. Finally, using the pixel value most different from the median yields* **row 4**, *where the motion of the circle is now visible.*

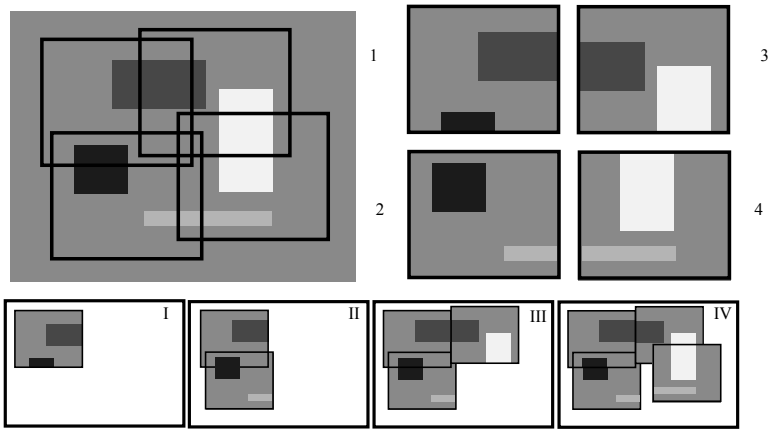


FIGURE 4.6: **Top left** *shows a simple scene with the location of four images; these are shown on the* **top right**. *The steps of creating a mosaic are sketched in the Roman numeral panes on the* **bottom**. *In this case, the translation of the fourth image with respect to the first is poorly estimated; more detail in the text.*

translating $\mathcal{I}_2$ to the right place in the mosaic coordinate system. The effect of this translation is shown in step II in Figure 4.6. Now you estimate $T_{3\rightarrow 1}$ to register $\mathcal{I}_3$ to $\mathcal{I}_1$ (Step III). Notice that the patterns in the scene have been chosen to show an exaggerated case where the registration error between $\mathcal{I}_3$ and $\mathcal{I}_1$ is likely to be large (many different horizontal translations of $\mathcal{I}_3$ will register with $\mathcal{I}_1$ well). Finally, you estimate $\mathcal{T}_{4\rightarrow 3}$ (Step IV). Notice how error has cascaded. $\mathcal{I}_3$ is poorly registered to $\mathcal{I}_1$, and $\mathcal{I}_4$ is registered to $\mathcal{I}_3$, meaning that $\mathcal{I}_4$ is poorly registered to $\mathcal{I}_1$.

Notice that this isn't necessary. Some pixels of $\mathcal{I}_4$ overlap $\mathcal{I}_2$. If these pixels contributed to the registration, $\mathcal{I}_4$ and $\mathcal{I}_3$ could be properly registered. This problem occurs quite generally – it is not just a result of an odd scene – and is often referred to as a failure of *loop closure*. This refers to the idea that if 2 is registered to 1, 3 is registered to 2, 4 is registered to 3, all the way up to N is registered to $N-1$, N may not be registered to 1 at all well – the loop does not close.

There are several procedures to prevent or control this kind of error propagation. Start by registering the images by pairs as described. The easiest strategy is now to repeat: fix all but one image, then register that image to all the others it overlaps with. This approach can work, but may be slow, because it may take a long time for improvements to propagate across all the images. The alternative, which is better but can be onerous, is to fix one image in place, then adjust all others to register in every overlap. Associate a translation $t_{i;x}, t_{j;y}$ with each image $\mathcal{I}_i$, and set $t_{1;x} = 0, t_{1;y} = 0$. Write $\mathbf{t}$ for a vector of all these translations except $t_{1;x}, t_{1;y}$, and $O_{ij}(\mathbf{t})$ for the area of the overlap between $\mathcal{I}_i$ and $\mathcal{I}_j$. Now minimize

$$\sum_{i,j \in \text{overlapping pairs of images}} \left\{ \frac{1}{O_{ij}(\mathbf{t})} \sum_{x,y \in \text{overlap}} \left[\mathcal{I}_{i;x+t_{i;x},y+t_{i;y}} - \mathcal{I}_{j;x+t_{j;x},y+t_{j;y}} \right]^2 \right\}$$

as a function of $\mathbf{t}$. This isn't a straightforward optimization problem. If you require

that $\mathbf{t}$ are integers – so that the sample points of overlapping images lie on top of one another – then searching for the right set of integer values is hard. If, instead, you treat this as a continuous optimization problem, the objective function is hard to evaluate because you will need to do a lot of bilinear interpolation. A coarse-to-fine search will work for this problem, too. While it is important to know that this difficulty is present, there is no need to resolve it in detail yet (but see Section 33.2 if you're concerned).

CHAPTER 5

Image Segmentation

Segmentation methods break images into groups of pixels, to obtain a more compact representation of what is interesting in the image. There is no correct segmentation of an image. Instead, the segmentation we want depends on what we are going to do with it. A *region* is a collection of pixels that belong together.

- **Shot boundary detection:** Typical movies are broken into *shots* — much shorter subsequences of frames that show largely the same objects (so the shot is the region here). It is helpful to represent a video as a collection of shots, where each is represented with a *key frame*. A key frame is a typical or representative frame. If the shot is coherent, this could even be chosen at random. A representation like this can be used to search for videos or to summarize videos to support browsing. Finding the boundaries of these shots automatically is the goal of *shot boundary detection*. Methods are typically straightforward but very useful.

- **Background subtraction:** Many images show an object (**foreground**) against a largely irrelevant **background**, and we want to separate the foreground from the background. There are two regions here – foreground and background. The simplest case occurs when we see multiple frames, with a possibly moving object on a stable background. For example, we might be detecting parts on a conveyor belt. Another example is counting motor cars in an overhead view of a road — the road is pretty stable in appearance. Another, less obvious, example is in human–computer interaction. Quite commonly, a camera is fixed (say, on top of a monitor) and views a room. Pretty much anything in the view that doesn't look like the room is interesting.

- **Forming superpixels:** The number of pixels in an image can be inconvenient for many applications. Smoothing and subsampling the image is one way to cope, but a smoothed and subsampled image loses detail because pixel values that are very different will be averaged together. An alternative is to find a set of *superpixels*. These are image regions that are about the same size, occur on a rough grid, and have reasonable boundaries (Figure 33.2; it is reasonable to think of them as somewhat distorted large pixels). They are not usually rectangular so their boundaries can respect edges in the image. The value associated with a superpixel is an average of pixel values that are quite similar, so the representation can be more compact than an image and also quite accurate.

- **Forming image segments:** We should like to decompose an image into regions that have roughly coherent color and texture, often called *segments*. Typically, the shape of these regions isn't particularly important, but the coherence is important. Superpixels need to be about the same size as one

another, but segments do not – some segments can be big (Figure 33.2). **TODO:** why these aren't superpixels

5.1 SHOT BOUNDARY DETECTION AND BACKGROUND SUBTRACTION

**** something ****

5.1.1 Background Subtraction

Background subtraction works as follows. Obtain an estimate of what the background looks like; optionally, obtain an estimate of what the foreground looks like. Now use this estimate to classify a pixel by determining whether it is more like the background or the foreground. Finally, keep the foreground pixels. There are a large number of variants of this recipe. Important sources of variation include: how to obtain the estimates of appearance; how to use these to classify pixels; and whether to use spatial models when classifying. Spatial models capture, for example, the tendency of a background pixel to have background neighbors. Such models can improve the foreground at the cost of increased complexity of classification.

One way to model the background is simply to take a picture. This approach works rather poorly because the background typically changes slowly over time. For example, the road may get more shiny as it rains and less when the weather dries up; people may move books and furniture around in the room, and so on. An alternative that usually works quite well is to estimate the value of background pixels using a **moving average**. In this approach, we estimate the value of a particular background pixel as a weighted average of the previous values. Typically, pixels in the distant past should be weighted at zero, and the weights increase smoothly. Ideally, the moving average should track the changes in the background, meaning that if the weather changes quickly (or the book mover is frenetic) relatively few pixels should have nonzero weights, and if changes are slow the number of past pixels with nonzero weights should increase. The approach can be quite successful, but needs to be used on quite coarse scale images as Figures 5.2 and 5.3 illustrate.

5.1.2 Shot Boundary Detection

A shot boundary detection algorithm must find frames in the video that are significantly different from the previous frame. Our test of significance must take account of the fact that, within a given shot, both objects and the background can move around in the field of view. Typically, this test takes the form of a distance; if the distance is larger than a threshold, a shot boundary is declared. There are a variety of standard techniques for computing a distance:

- **Frame differencing** algorithms take pixel-by-pixel differences between each two frames in a sequence and sum the squares of the differences. These algorithms are unpopular, because they are slow — there are many differences — and because they tend to find many shots when the camera is shaking.
- **Histogram-based** algorithms compute color histograms for each frame and compute a distance between the histograms. A difference in color histograms is a sensible measure to use because it is insensitive to the spatial arrangement of colors in the frame (e.g., small camera jitters will not affect the histogram).

FIGURE 5.1: *The figure shows every fifth frame from a sequence of 120 frames of a child playing on a patterned sofa. The frames are used at an 80 x 60 resolutio, for reasons we discuss in Figure 5.3. Notice that the child moves from one side of the frame to the other during the sequence.*

- **Block comparison** algorithms compare frames by cutting them into a grid of boxes and comparing the boxes. This is to avoid the difficulty with color histograms, where a red object disappearing off-screen in the bottom left corner is equivalent to a red object appearing on screen from the top edge. Typically, these block comparison algorithms compute an interframe distance that is a composite — taking the maximum is one natural strategy — of interblock distances, each computed using methods like those used for interframe distances.
- **Edge differencing** algorithms compute edge maps for each frame, and then compare these edge maps. Typically, the comparison is obtained by counting the number of potentially corresponding edges (nearby, similar orientation, etc.) in the next frame. If there are few potentially corresponding edges, there is a shot boundary. A distance can be obtained by transforming the number of corresponding edges.

5.2 IMAGE REGIONS AS PIXEL CLUSTERS

5.2.1 Clustering Generalities

We want to find groups of pixels (data items) that belong together, and so must *cluster* pixels. Clustering is a process whereby a data set is replaced by **clusters**, which are collections of data items that belong together. Generally, we can cluster in two ways:

- In **divisive clustering**, the entire data set is regarded as a cluster, and then clusters are recursively split to yield a good clustering. Assume each split

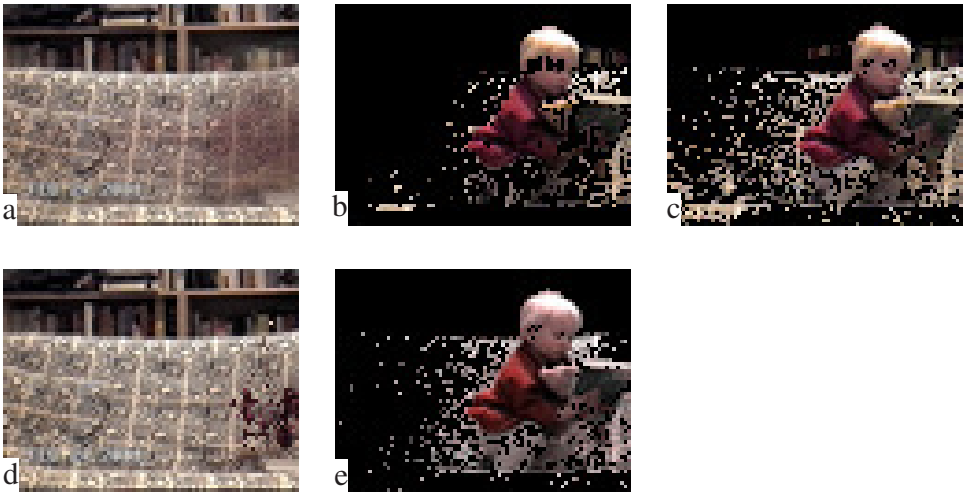


FIGURE 5.2: *Background subtraction results for the sequence of Figure 5.1 using 80 x 60 frames. We compare two methods of computing the background: (***a***) The average of all 120 frames — notice that the child spent more time on one side of the sofa than the other, leading to the faint blur in the average there. (***b***) Pixels whose difference from the average exceeds a small threshold. (***c***) Those whose difference from the average exceeds a somewhat larger threshold. Notice that, in each case, there are some excess pixels and some missing pixels. (***d***) A background computed using a somewhat more sophisticated method (described briefly in Section* **??***). (***e***) Pixels that this method believes are different from the background. Again, notice the missing pixels.*

breaks a cluster in two pieces (not compulsory, but convenient). The pieces are chosen so that each piece contains data items that are near one another and the pieces are far.

- In **agglomerative clustering**, each data item is regarded as a cluster, and nearby clusters are recursively merged to yield a good clustering.

In each case, we need to think about what makes a good inter-cluster distance. Even if a natural distance between data items is available (which may not be the case for vision problems), there is no canonical intercluster distance. Generally, one chooses a distance that seems appropriate for the data set. Three standard cases are:

- Choose the distance between the closest elements as the intercluster distance — this tends to yield extended clusters (statisticians call this method *single-link clustering*).

- Choose the maximum distance between an element of the first cluster and one of the second — this tends to yield rounded clusters (statisticians call this method *complete-link clustering*).

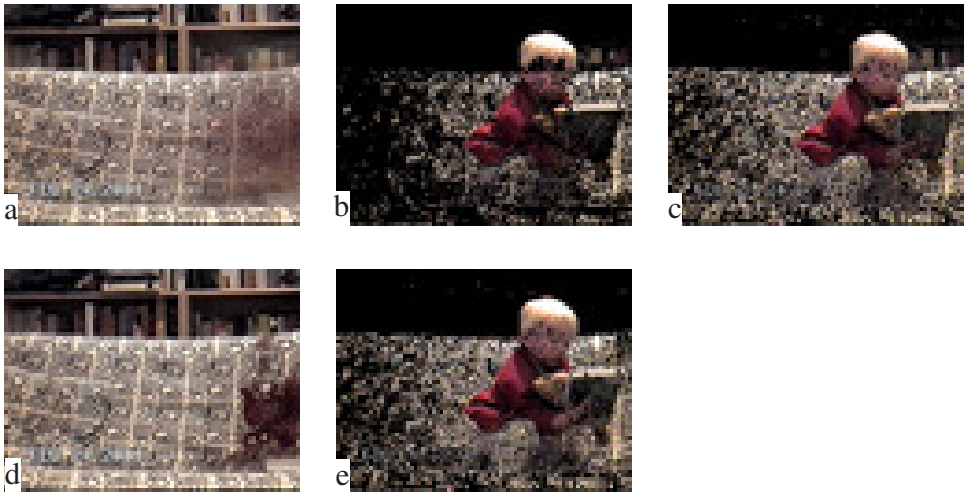


FIGURE 5.3: *Registration can be a significant nuisance in background subtraction, particularly for textures. These figures show results for the sequence of Figure 5.1, using 160 x 120 frames. We compare two methods of computing the background: (***a***) The average of all 120 frames — notice that the child spent more time on one side of the sofa than the other, leading the a faint blur in the average there. (***b***) Pixels whose difference from the average exceeds a small threshold. (***c***) Those whose difference from the average exceeds a somewhat larger threshold. (***d***) A background computed using a somewhat more sophisticated method (described briefly in Section* **??***). (***e***) Pixels that this method believes are different from the background. Notice that the number of problem pixels — where the pattern on the sofa has been mistaken for the child — has markedly increased. This is because small movements can cause the high spatial frequency pattern on the sofa to be misaligned, leading to large differences.*

- Choose an average of distances between elements in the clusters — this also tends to yield "rounded" clusters (statisticians call this method *group average clustering*).

In each case, we must also think about how many clusters there should be. This tells us when to stop splitting or merging. Usually, we cluster data to support some other application, and the best way to address this question is to find a solution that makes that other application work. Notice that we can avoid committing to a solution, because each algorithm naturally forms a tree. This tree represents many different clustering of the data, depending on how you prune it. For some applications, it is helpful to keep the tree (or much of it).

TODO: cluster tree figure

Computing Distances between Pixels

We know ways to compute distances between clusters from distances between data points. The recipe for computing distances between pixels is to compute some

feature vector describing each pixel, then compute a distance between the vectors. Natural vectors include:

- **Color:** Use a feature vector describing the color of the pixel. It's a good idea to use a color representation where distances reflect perceived change of color well, so Lab values or Luv values are a sensible choice. Regions are then groups of pixels with very similar color, and so may be spread out, or even not connected.
- **Color and position:** Stack the x, y position of the pixel together with the color. There is now a scaling problem – if the image is (say) 1024×1024 the difference in position could be very large compared to the difference in color. Choosing a scale is a balance between obtaining spatially compact clusters (high weight on difference in position) and obtaining few clusters (low weight on difference in position).
- **Color, position and texture:** Stack a texture descriptor together with position and color. This texture descriptor could come from many sources (some we haven't yet discussed). A traditional option is to choose a set of small pattern detectors (filters, Section 33.2). The texture descriptor is then an average of each filter's response in some local patch (filters, Section 33.2). Experience shows that a big descriptor is a good idea, but a big descriptor means more problems with relative scaling of the distances. Scaling these descriptors is largely a matter of experiment. A natural place to start is to decorrelate the descriptors (Section 33.2).

TODO: Decorrelation into notes

5.2.2 Clustering and Segmentation by K-means

We could phrase clustering as minimizing an objective function. Assume we know there are k clusters, where k is known. Each cluster is assumed to have a center; we write the center of the ith cluster as $\mathbf{c}_i$. The jth element to be clustered is described by a feature vector $\mathbf{x}_j$, constructed as in the previous section. Write δ for a table linking clusters to data items. The i, j'th entry, δ_{ij} is 1 if the j'th data item belongs to the i'th cluster, and $\delta_{ij} = 0$ otherwise. Neither the $\mathbf{c}$'s nor the δ_{ij}'s are known. We want to choose centers so that all the elements are close to the center of their cluster, yielding the objective function

$$\Phi(\delta, \mathbf{c}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in \text{elements of } i\text{'th cluster}} (\mathbf{x}_j - \mathbf{c}_i)^T (\mathbf{x}_j - \mathbf{c}_i) \right\} = \sum_{i,j} \delta_{ij} (\mathbf{x}_j - \mathbf{c}_i)^T (\mathbf{x}_j - \mathbf{c}_i).$$

We need to minimize this by choice of δ and $\mathbf{c}$. There are constraints: $\delta_{ij} \in \{0, 1\}$; and each point belongs to exactly one cluster, so $\sum_i \delta_{ij} = 1$ for each j. Solving this problem exactly is known to be NP-hard, but useful approximate solutions are quite easily obtained. Notice that if δ is known, it is easy to compute the best center for each cluster (average the points in the cluster; check if you're not sure). Similarly, if $\mathbf{c}$ is known, then δ is easy to get (allocate each point to the closest cluster center). This suggests an algorithm that iterates through two activities:

- Assume the cluster centers are known and allocate each point to the closest cluster center.

- Assume the allocation is known and choose a new set of cluster centers.

We then choose a start point by randomly choosing cluster centers and then iterate these stages alternately. This process eventually converges to a local minimum of the objective function (the value either goes down or is fixed at each step; it is bounded below; and we discount the prospect of symmetries in the objective function). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters unless we modify the allocation phase to ensure that each cluster has some nonzero number of points. This algorithm is usually referred to as *k-means*.

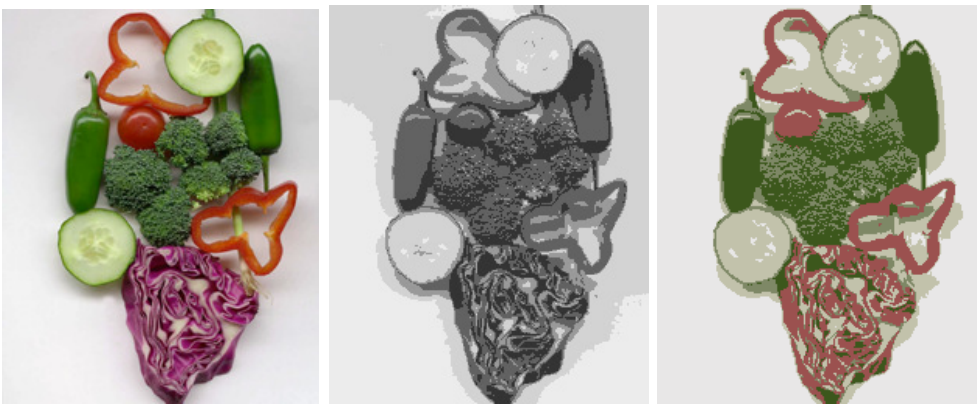


FIGURE 5.4: *On the* **left**, *an image of mixed vegetables, which is segmented using k-means to produce the images at* **center** *and on the* **right**. *We have replaced each pixel with the mean value of its cluster; the result is somewhat like an adaptive requantization as one would expect. In the center, a segmentation obtained using only the intensity information. At the right, a segmentation obtained using color information. Each segmentation assumes five clusters.*

One difficulty with using this approach for segmenting images is that segments are not connected and can be very widely scattered (Segmentation/Figures 5.4 and 5.5). This effect can be reduced by using pixel coordinates as features — an approach that results in large regions being broken up (Figure 5.6).

5.2.3 Graph Theoretic Clustering

The application of graphs to clustering is this: Take each element of the collection to be clustered and associate it with a vertex on a graph. Now construct an edge from every element to every other, and associate with this edge a weight representing the extent to which the elements are similar. Each weight is an *affinity* – if the two vertices are similar, it is large, and if they are different, it is small. Now cut edges in the graph to form a good set of connected components — ideally,

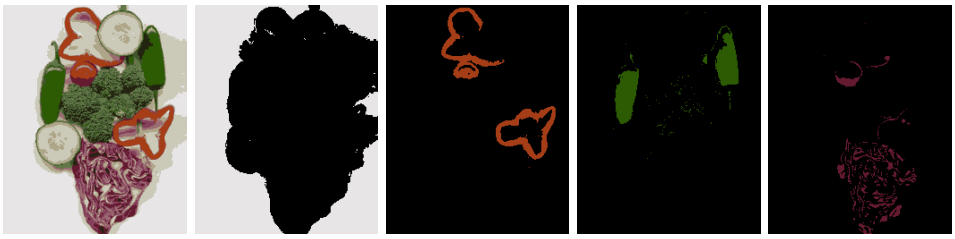


FIGURE 5.5: *Here we show the image of vegetables segmented with k-means, assuming a set of 11 components. The* **left** *figure shows all segments shown together, with the mean value in place of the original image values. The other figures show four of the segments. Note that this approach leads to a set of segments that are not necessarily connected. For this image, some segments are actually quite closely associated with objects, but one segment may represent many objects (the peppers); others are largely meaningless. The absence of a texture measure creates serious difficulties as the many different segments resulting from the slice of red cabbage indicate.*

FIGURE 5.6: *Five of the segments obtained by segmenting the image of vegetables with a k-means segmenter that uses position as part of the feature vector describing a pixel, now using 20 segments rather than 11. Note that the large background regions that should be coherent have been broken up because points got too far from the center. The individual peppers are now better separated, but the red cabbage is still broken up because there is no texture measure.*

the within-component edges are large compared with the across-component edges. Each component is a cluster.

It is straightforward to construct affinities out of distances. If $d(\mathbf{x}_i, \mathbf{x}_j)$ is the distance between two feature vectors, then

$$\exp\left(-\frac{d(\mathbf{x}_i, \mathbf{x}_j)^2{}^2}{2\sigma}\right)$$

is an affinity (big when similar, small when different). Here σ is a scale used to adjust the affinity. Now represent the affinities between each pair of points in a matrix $\mathcal{A}$, and recover clusters by analysis of that matrix.

Extracting a Single Good Cluster

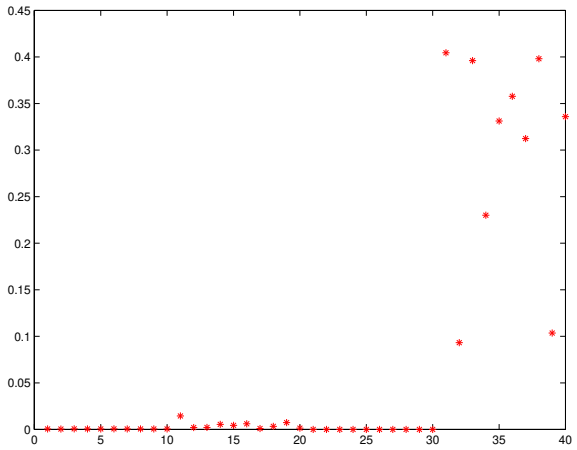


FIGURE 5.7: *The eigenvector corresponding to the largest eigenvalue of the affinity matrix for the dataset of Figure* **??** *using* $\sigma_d = 0.2$. *Notice that most values are small, but some — corresponding to the elements of the main cluster — are large. The sign of the association is not significant, because a scaled eigenvector is still an eigenvector.*

A good cluster is one where elements that are strongly associated with the cluster also have large values connecting one another in the affinity matrix. Write $\mathbf{w}$ for the vector of weights linking elements to the cluster. Now the function

$$\mathbf{w}^T \mathcal{A} \mathbf{w}$$

is a sum of terms of the form

$$\begin{aligned} &\{\text{association of element } i \text{ with the cluster}\} \\ &\quad \times \{\text{affinity between } i \text{ and } j\} \\ &\times \{\text{association of element } j \text{ with the cluster}\}. \end{aligned}$$

We can obtain a cluster by choosing a set of association weights that maximize this objective function. The objective function is useless on its own because scaling $\mathbf{w}$ by λ scales the total association by λ^2. However, we can normalise the weights by requiring that $\mathbf{w}^T\mathbf{w} = 1$, so it is natural to maximize $\mathbf{w}^T\mathcal{A}\mathbf{w}$ subject to $\mathbf{w}^T\mathbf{w} = 1$. This is an eigenvalue problem (check, or Section 33.2 if you haven't seen this pattern) and we must solve

$$\mathcal{A}\mathbf{w} = \lambda\mathbf{w}.$$

For problems where reasonable clusters are apparent, we expect that these cluster weights are large for some elements, which belong to the cluster, and nearly zero for others, which do not (Figure 5.7). In fact, we can get the weights for other clusters from other eigenvectors of $\mathcal{A}$ as well.

Extracting Weights for a Set of Clusters

In typical vision problems, there are strong association weights between relatively few pairs of elements. We can reasonably expect to be dealing with clusters that are quite tight and distinct. A natural procedure is to take the first eigenvector, use that to decide what belongs in the first cluster (large values in the eigenvector), then take the second eigenvector to decide what belongs in the second cluster (large values in the eigenvector for things that aren't in the first) and so on.

Normalized Cuts

An alternative approach is to cut the graph into two connected components such that the cost of the cut is a small fraction of the total affinity within each group. We can formalize this as decomposing a weighted graph V into two components A and B and scoring the decomposition with

$$\frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$$

(where $cut(A,B)$ is the sum of weights of all edges in V that have one end in A and the other in B, and $assoc(A,V)$ is the sum of weights of all edges that have one end in A). This score is small if the cut separates two components that have few edges of low weight between them and many internal edges of high weight. We would like to find the cut with the minimum value of this criterion, called a *normalized cut*. Actually finding this cut is algorithmically tricky, but sensible approximation procedures are known. The criterion is successful in practice (Segmentation/Figures 5.8 and 5.9).

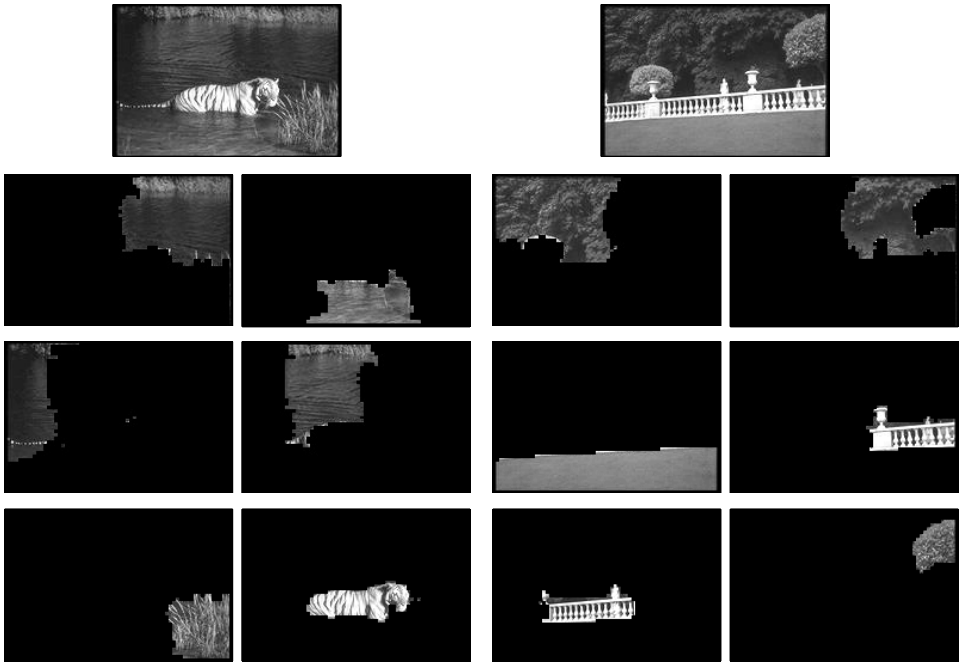


FIGURE 5.8: *The images on top are segmented using the normalized cuts framework, described in the text, into the components shown. The affinity measures used involved intensity and texture as in Section* **??**. *The image of the swimming tiger yields one segment that is essentially tiger, one that is grass, and four components corresponding to the lake. Similarly, the railing shows as three reasonably coherent segments. Note the improvement over k-means segmentation obtained by having a texture measure.*

FIGURE 5.9: **Top:** *two frames from a motion sequence, that shows a moving view of a person.* **Bottom:** *spatiotemporal segments established using normalized cuts and a spatiotemporal affinity function (Section* **??***).*

CHAPTER 6

Mapping Images to Image-Like Things

CHAPTER 7

Sampling and Aliasing

7.1 SPATIAL FREQUENCY AND FOURIER TRANSFORMS

We have used the trick of thinking of a signal $g(x, y)$ as a weighted sum of a large (or infinite) number of small (or infinitely small) box functions. This model emphasizes that a signal is an element of a vector space. The box functions form a convenient basis, and the weights are coefficients on this basis. We need a new technique to deal with two related problems so far left open:

- Although it is clear that a discrete image version cannot represent the full information in a signal, we have not yet indicated what is lost.
- It is clear that we cannot shrink an image simply by taking every kth pixel—this could turn a checkerboard image all white or all black—and we would like to know how to shrink an image safely.

All of these problems are related to the presence of fast changes in an image. For example, shrinking an image is most likely to miss fast effects because they could slip between samples; similarly, the derivative is large at fast changes.

These effects can be studied by *a change of basis*. We change the basis to be a set of sinusoids and represent the signal as an infinite weighted sum of an infinite number of sinusoids. This means that fast changes in the signal are obvious, because they correspond to large amounts of high-frequency sinusoids in the new basis.

7.1.1 Fourier Transforms

The change of basis is effected by a *Fourier transform*. We define the Fourier transform of a signal $g(x, y)$ to be

$$\mathcal{F}(g(x,y))(u,v) = \int_{-\infty}^{\infty}\int g(x,y)e^{-i2\pi(ux+vy)}dxdy.$$

Assume that appropriate technical conditions are true to make this integral exist. It is sufficient for all moments of g to be finite; a variety of other possible conditions are available [**?**]. The process takes a complex valued function of x, y and returns a complex valued function of u, v (images are complex valued functions with zero imaginary component).

For the moment, fix u and v, and let us consider the meaning of the value of the transform at that point. The exponential can be rewritten

$$e^{-i2\pi(ux+vy)} = \cos(2\pi(ux+vy)) + i\sin(2\pi(ux+vy)).$$

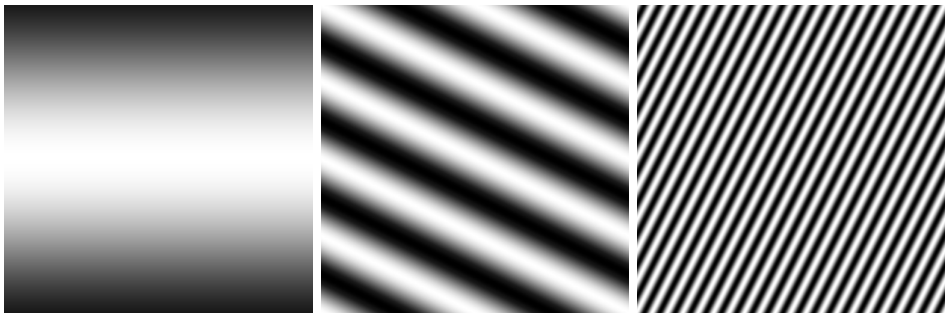


FIGURE 7.1: *The real component of Fourier basis elements shown as intensity images. The brightest point has value one, and the darkest point has value zero. The domain is* $[-1,1] \times [-1,1]$*, with the origin at the center of the image. On the* **left***,* $(u,v) = (0, 0.4)$*; in the* **center***,* $(u,v) = (1,2)$*; and on the* **right** $(u,v) = (10,-5)$*. These are sinusoids of various frequencies and orientations described in the text.*

These terms are sinusoids on the x, y plane, whose orientation and frequency are given by u, v. For example, consider the real term, which is constant when $ux+vy$ is constant (i.e., along a straight line in the x, y plane whose orientation is given by $\tan\theta = v/u$). The gradient of this term is perpendicular to lines where $ux + vy$ is constant, and the frequency of the sinusoid is $\sqrt{u^2+v^2}$. These sinusoids are often referred to as *spatial frequency components*; a variety are illustrated in Figure 7.1.

The integral should be seen as a dot product. If we fix u and v, the value of the integral is the dot product between a sinusoid in x and y and the original function. This is a useful analogy because dot products measure the amount of one vector in the direction of another.

In the same way, the value of the transform at a particular u and v can be seen as measuring the amount of the sinusoid with given frequency and orientation in the signal. The transform takes a function of x and y to the function of u and v whose value at any particular (u,v) is the amount of that particular sinusoid in the original function. This view justifies the model of a Fourier transform as a change of basis.

Linearity

The Fourier transform is linear:

$$\mathcal{F}(g(x,y) + h(x,y)) = \mathcal{F}(g(x,y)) + \mathcal{F}(h(x,y))$$

and

$$\mathcal{F}(kg(x,y)) = k\mathcal{F}(g(x,y)).$$

The Inverse Fourier Transform It is useful to recover a signal from its Fourier transform. This is another change of basis with the form

$$g(x,y) = \int_{-\infty}^{\infty}\int \mathcal{F}(g(x,y))(u,v)e^{i2\pi(ux+vy)}dudv.$$

Fourier Transform Pairs Fourier transforms are known in closed form

TABLE 7.1: *A variety of functions of two dimensions and their Fourier transforms. This table can be used in two directions (with appropriate substitutions for u, v and x, y) because the Fourier transform of the Fourier transform of a function is the function. Observant readers might suspect that the results on infinite sums of δ functions contradict the linearity of Fourier transforms. By careful inspection of limits, it is possible to show that they do not (see, for example, ?). Observant readers also might have noted that an expression for $\mathcal{F}(\frac{\partial f}{\partial y})$ can be obtained by combining two lines of this table.*

Function	Fourier transform
$g(x,y)$	$\int_{-\infty}^{\infty}\int g(x,y)e^{-i2\pi(ux+vy)}dxdy$
$\int_{-\infty}^{\infty}\int \mathcal{F}(g(x,y))(u,v)e^{i2\pi(ux+vy)}dudv$	$\mathcal{F}(g(x,y))(u,v)$
$\delta(x,y)$	1
$\frac{\partial f}{\partial x}(x,y)$	$u\mathcal{F}(f)(u,v)$
$0.5\delta(x+a,y)+0.5\delta(x-a,y)$	$\cos 2\pi au$
$e^{-\pi(x^2+y^2)}$	$e^{-\pi(u^2+v^2)}$
$box_1(x,y)$	$\frac{\sin u}{u}\frac{\sin v}{v}$
$f(ax,by)$	$\frac{\mathcal{F}(f)(u/a,v/b)}{ab}$
$\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)$	$\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(u-i,v-j)$
$(f**g)(x,y)$	$\mathcal{F}(f)\mathcal{F}(g)(u,v)$
$f(x-a,y-b)$	$e^{-i2\pi(au+bv)}\mathcal{F}(f)$
$f(x\cos\theta-y\sin\theta, x\sin\theta+y\cos\theta)$	$\mathcal{F}(f)(u\cos\theta-v\sin\theta, u\sin\theta+v\cos\theta)$

for a variety of useful cases; a large set of examples appears in **?**. We list a few in Table 7.1 for reference. The last line of Table 7.1 contains the *convolution theorem*; convolution in the signal domain is the same as multiplication in the Fourier domain.

Phase and Magnitude The Fourier transform consists of a real and a

complex component:

$$\begin{aligned}\mathcal{F}(g(x,y))(u,v) &= \int\int_{-\infty}^{\infty} g(x,y)\cos(2\pi(ux+vy))dxdy + \\ &\quad i\int\int_{-\infty}^{\infty} g(x,y)\sin(2\pi(ux+vy))dxdy \\ &= \Re(\mathcal{F}(g)) + i * \Im(\mathcal{F}(g)) \\ &= \mathcal{F}_R(g) + i * \mathcal{F}_I(g).\end{aligned}$$

It is usually inconvenient to draw complex functions of the plane. One solution is to plot $\mathcal{F}_R(g)$ and $\mathcal{F}_I(g)$ separately; another is to consider the *magnitude* and *phase* of the complex functions, and to plot these instead. These are then called the *magnitude spectrum* and *phase spectrum*, respectively.

The value of the Fourier transform of a function at a particular u, v point depends on the whole function. This is obvious from the definition because the domain of the integral is the whole domain of the function. It leads to some subtle properties, however. First, a local change in the function (e.g., zeroing out a block of points) is going to lead to a change *at every point* in the Fourier transform. This means that the Fourier transform is quite difficult to use as a representation (e.g., it might be very difficult to tell whether a pattern was present in an image just by looking at the Fourier transform). Second, the magnitude spectra of images tends to be similar. This appears to be a fact of nature, rather than something that can be proven axiomatically. As a result, the magnitude spectrum of an image is surprisingly uninformative (see Figure 7.2 for an example).

7.2 SAMPLING AND ALIASING

The crucial reason to discuss Fourier transforms is to get some insight into the difference between discrete and continuous images. In particular, it is clear that some information has been lost when we work on a discrete pixel grid, but what? A good, simple example comes from an image of a checkerboard, and is given in Figure 7.3. The problem has to do with the number of samples relative to the function; we can formalize this rather precisely given a sufficiently powerful model.

7.2.1 Sampling

Passing from a continuous function—like the irradiance at the back of a camera system—to a collection of values on a discrete grid —like the pixel values reported by a camera—is referred to as *sampling*. We construct a model that allows us to obtain a precise notion of what is lost in sampling.

Sampling in One Dimension

Sampling in one dimension takes a function and returns a discrete set of values. The most important case involves sampling on a uniform discrete grid, and we assume that the samples are defined at integer points. This means we have a process that takes some function and returns a vector of values:

$$\texttt{sample}_{1D}(f(x)) = \mathbf{f}.$$

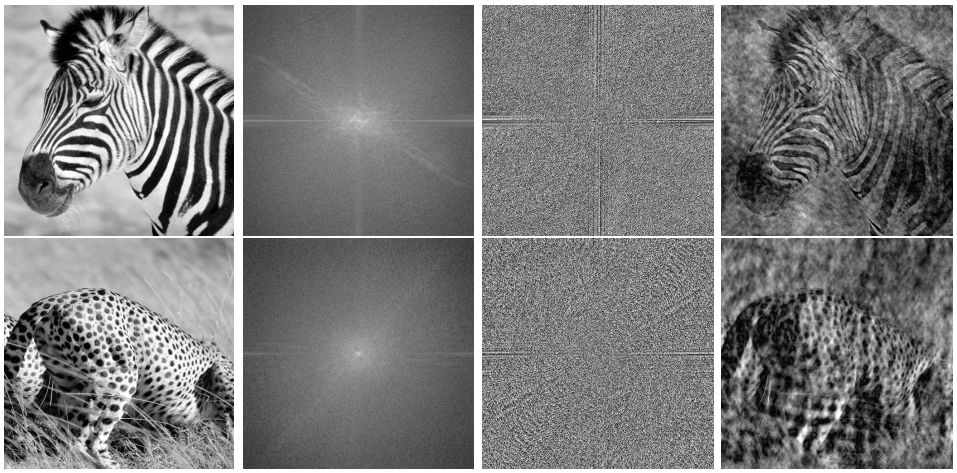


FIGURE 7.2: *The second image in each row shows the log of the magnitude spectrum for the first image in the row; the third image shows the phase spectrum scaled so that $-\pi$ is dark and π is light. The final images are obtained by swapping the magnitude spectra. Although this swap leads to substantial image noise, it doesn't substantially affect the interpretation of the image, suggesting that the phase spectrum is more important for perception than the magnitude spectrum.*

We model this sampling process by assuming that the elements of this vector are the values of the function $f(x)$ at the sample points and allowing negative indices to the vector (Figure 7.4). This means that the ith component of $\mathbf{f}$ is $f(x_i)$.

Sampling in Two Dimensions

Sampling in 2D is very similar to sampling in 1D. Although sampling can occur on nonregular grids (the best example being the human retina), we proceed on the assumption that samples are drawn at points with integer coordinates. This yields a uniform rectangular grid, which is a good model of most cameras. Our sampled images are then rectangular arrays of finite size (all values outside the grid being zero).

In the formal model, we sample a function of two dimensions, instead of one, yielding an array (Figure 7.5). We allow this array to have negative indices in both dimensions, and can then write

$$\texttt{sample}_{2D}(F(x,y)) = \mathcal{F},$$

where the i, jth element of the array $\mathcal{F}$ is $F(x_i, y_j) = F(i,j)$.

Samples are not always evenly spaced in practical systems. This is quite often due to the pervasive effect of television; television screens have an aspect ratio of 4:3 (width:height). Cameras quite often accommodate this effect by spacing sample points slightly farther apart horizontally than vertically (in jargon, they have *non-square pixels*).

A Continuous Model of a Sampled Signal

We need a continuous model of a sampled signal. Generally, this model is used to evaluate integrals; in particular, taking a Fourier transform involves integrating the product of our model with a complex exponential. It is clear how this integral should behave: the value of the integral should be obtained by adding up values at each integer point. This means we cannot model a sampled signal as a function that is zero everywhere except at integer points (where it takes the value of the signal), because this model has a zero integral.

An appropriate continuous model of a sampled signal relies on an important property of the δ function:

$$\begin{aligned}\int_{-\infty}^{\infty} a\delta(x)f(x)dx &= a\lim_{\epsilon\to 0}\int_{-\infty}^{\infty} d(x;\epsilon)f(x)dx \\ &= a\lim_{\epsilon\to 0}\int_{-\infty}^{\infty} \frac{bar(x;\epsilon)}{\epsilon}(f(x))dx \\ &= a\lim_{\epsilon\to 0}\sum_{i=-\infty}^{\infty} \frac{bar(x;\epsilon)}{\epsilon}(f(i\epsilon)bar(x-i\epsilon;\epsilon))\epsilon \\ &= af(0).\end{aligned}$$

Here we have used the idea of an integral as the limit of a sum of small strips.

An appropriate continuous model of a sampled signal consists of a δ-function at each sample point weighted by the value of the sample at that point. We can obtain this model by multiplying the sampled signal by a set of δ-functions, one at each sample point. In one dimension, a function of this form is called a *comb function* (because that's what the graph looks like). In two dimensions, a function of this form is called a *bed-of-nails function* (for the same reason).

Working in 2D and assuming that the samples are at integer points, this procedure gets

$$\begin{aligned}\texttt{sample}_{2D}(f) &= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f(i,j)\delta(x-i,y-j) \\ &= f(x,y)\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}.\end{aligned}$$

This function is zero except at integer points (because the δ-function is zero except at integer points), and its integral is the sum of the function values at the integer points.

7.2.2 Aliasing

Sampling involves a loss of information. As this section shows, a signal sampled too slowly is misrepresented by the samples; high spatial frequency components of the original signal appear as low spatial frequency components in the sampled signal—an effect known as *aliasing*.

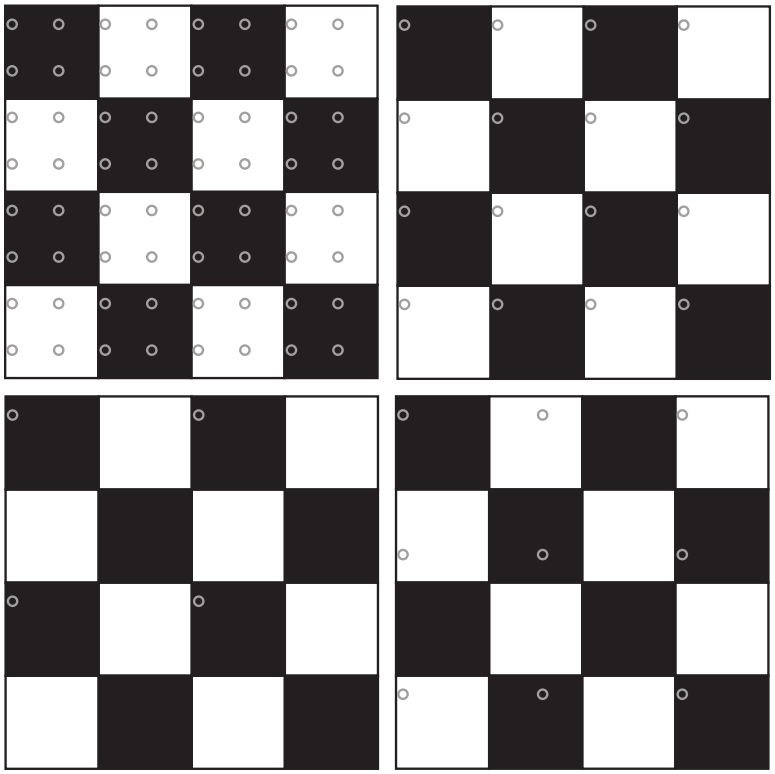


FIGURE 7.3: *The two checkerboards on the* **top** *illustrate a sampling procedure that appears to be successful (whether it is or not depends on some details that we will deal with later). The gray circles represent the samples; if there are sufficient samples, then the samples represent the detail in the underlying function. The sampling procedures shown on the* **bottom** *are unequivocally unsuccessful; the samples suggest that there are fewer checks than there are. This illustrates two important phenomena: first, successful sampling schemes sample data often enough; and second, unsuccessful sampling schemes cause high-frequency information to appear as lower-frequency information.*

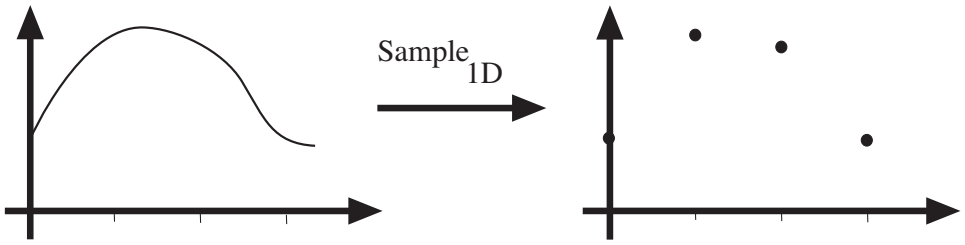


FIGURE 7.4: *Sampling in 1D takes a function and returns a vector whose elements are values of that function at the sample points. For our purposes, it is enough that the sample points be integer values of the argument. We allow the vector to be infinite dimensional and have negative as well as positive indices.*

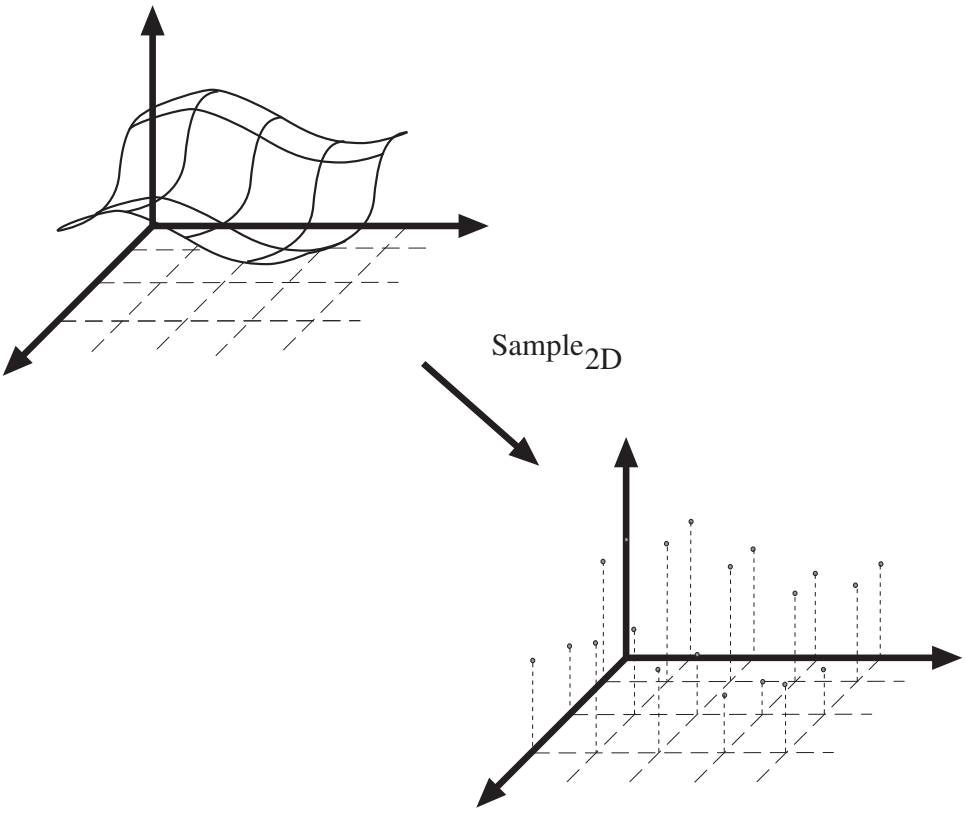


FIGURE 7.5: *Sampling in 2D takes a function and returns an array; again, we allow the array to be infinite dimensional and to have negative as well as positive indices.*

The Fourier Transform of a Sampled Signal

A sampled signal is given by a product of the original signal with a bed-of-nails function. By the convolution theorem, the Fourier transform of this product is the convolution of the Fourier transforms of the two functions. This means that the Fourier transform of a sampled signal is obtained by convolving the Fourier transform of the signal with another bed-of-nails function.

Now convolving a function with a shifted δ-function merely shifts the function (see exercises). This means that the Fourier transform of the sampled signal is the sum of a collection of shifted versions of the Fourier transforms of the signal, that is,

$$\begin{aligned}\mathcal{F}(\texttt{sample}_{2D}(f(x,y))) &= \mathcal{F}\left(f(x,y)\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}\right)\\ &= \mathcal{F}(f(x,y)) * *\mathcal{F}\left(\left\{\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\delta(x-i,y-j)\right\}\right)\\ &= \sum_{i=-\infty}^{\infty}F(u-i,v-j),\end{aligned}$$

where we have written the Fourier transform of $f(x,y)$ as $F(u,v)$.

If the support of these shifted versions of the Fourier transform of the signal does not intersect, we can easily reconstruct the signal from the sampled version. We take the sampled signal, Fourier transform it, and cut out one copy of the Fourier transform of the signal and Fourier transform this back (Figure 7.6).

However, if the support regions *do* overlap, we are not able to reconstruct the signal because we can't determine the Fourier transform of the signal in the regions of overlap, where different copies of the Fourier transform will add. This results in a characteristic effect, usually called *aliasing*, where high spatial frequencies appear to be low spatial frequencies (see Figure 7.8 and exercises). Our argument also yields *Nyquist's theorem*: the sampling frequency must be at least twice the highest frequency present for a signal to be reconstructed from a sampled version. By the same argument, if we happen to have a signal that has frequencies present only in the range $[2k-1\Omega, 2k+1\Omega]$, then we can represent that signal exactly if we sample at a frequency of at least 2Ω.

7.2.3 Smoothing and Resampling

Nyquist's theorem means it is dangerous to shrink an image by simply taking every kth pixel (as Figure 7.8 confirms). Instead, we need to filter the image so that spatial frequencies above the new sampling frequency are removed. We could do this exactly by multiplying the image Fourier transform by a scaled 2D bar function, which would act as a low-pass filter. Equivalently, we would convolve the image with a kernel of the form $(\sin x \sin y)/(xy)$. This is a difficult and expensive (a polite way of saying *impossible*) convolution because this function has infinite support.

The most interesting case occurs when we want to halve the width and height of the image. We assume that the sampled image has no aliasing (because if it

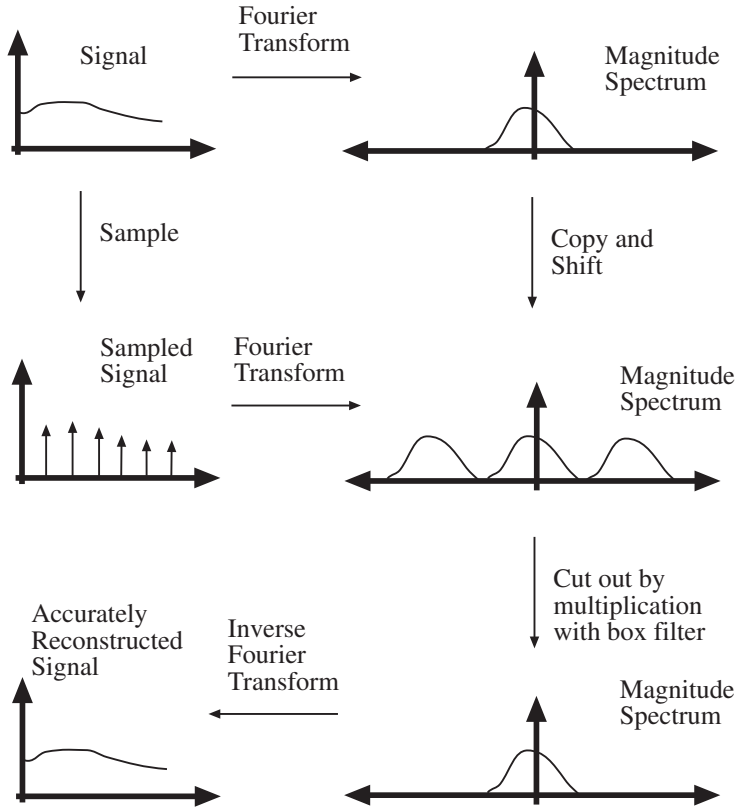


FIGURE 7.6: *The Fourier transform of the sampled signal consists of a sum of copies of the Fourier transform of the original signal, shifted with respect to each other by the sampling frequency. Two possibilities occur. If the shifted copies do not intersect with each other (as in this case), the original signal can be reconstructed from the sampled signal (we just cut out one copy of the Fourier transform and inverse transform it). If they do intersect (as in Figure 7.7), the intersection region is added, and so we cannot obtain a separate copy of the Fourier transform, and the signal has aliased.*

did, there would be nothing we could do about it anyway; once an image has been sampled, any aliasing that is going to occur has happened, and there's not much we can do about it without an image model). This means that the Fourier transform of the sampled image is going to consist of a set of copies of some Fourier transform, with centers shifted to integer points in u, v space.

If we resample this signal, the copies now have centers on the half-integer points in u, v space. This means that, to avoid aliasing, we need to apply a filter that strongly reduces the content of the original Fourier transform outside the range $|u| < 1/2$, $|v| < 1/2$. Of course, if we reduce the content of the signal *inside* this range, we might lose information, too. Now the Fourier transform of a Gaussian is a Gaussian, and Gaussians die away fairly quickly. Thus, if we were to convolve the

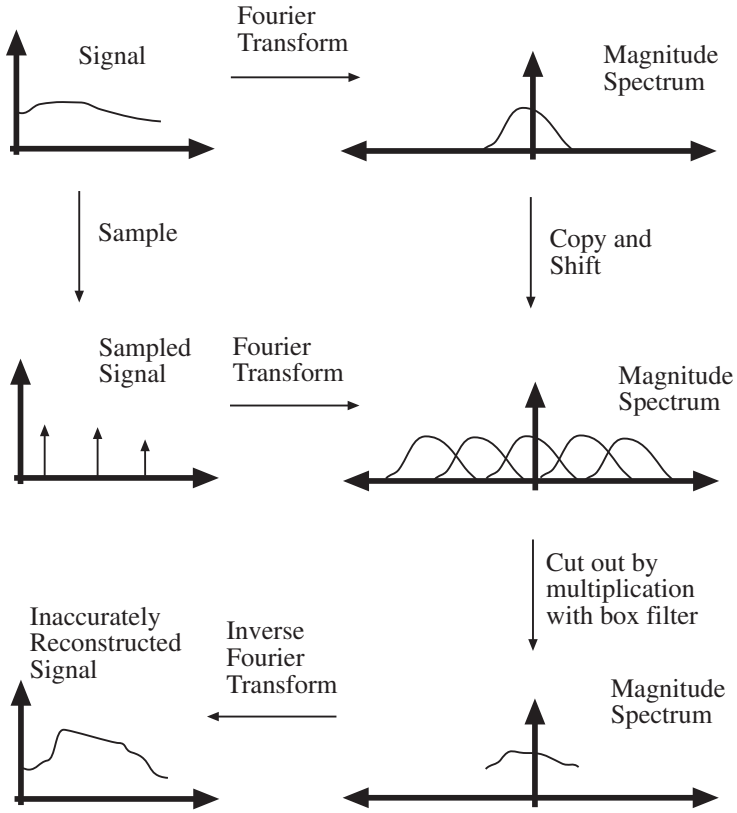


FIGURE 7.7: *The Fourier transform of the sampled signal consists of a sum of copies of the Fourier transform of the original signal, shifted with respect to each other by the sampling frequency. Two possibilities occur. If the shifted copies do not intersect with each other (as in Figure 7.6), the original signal can be reconstructed from the sampled signal (we just cut out one copy of the Fourier transform and inverse transform it). If they do intersect (as in this figure), the intersection region is added, and so we cannot obtain a separate copy of the Fourier transform, and the signal has aliased. This also explains the tendency of high spatial frequencies to alias to lower spatial frequencies.*

image with a Gaussian—or multiply its Fourier transform by a Gaussian, which is the same thing—we could achieve what we want.

The choice of Gaussian depends on the application. If σ is large, there is less aliasing (because the value of the kernel outside our range is very small), but information is lost because the kernel is not flat within our range; similarly, if σ is small, less information is lost within the range, but aliasing can be more substantial. Figures 7.9 and 7.10 illustrate the effects of different choices of σ.

We have been using a Gaussian as a low-pass filter because its response at high spatial frequencies is low and its response at low spatial frequencies is high. In fact, the Gaussian is not a particularly good low-pass filter. What one wants

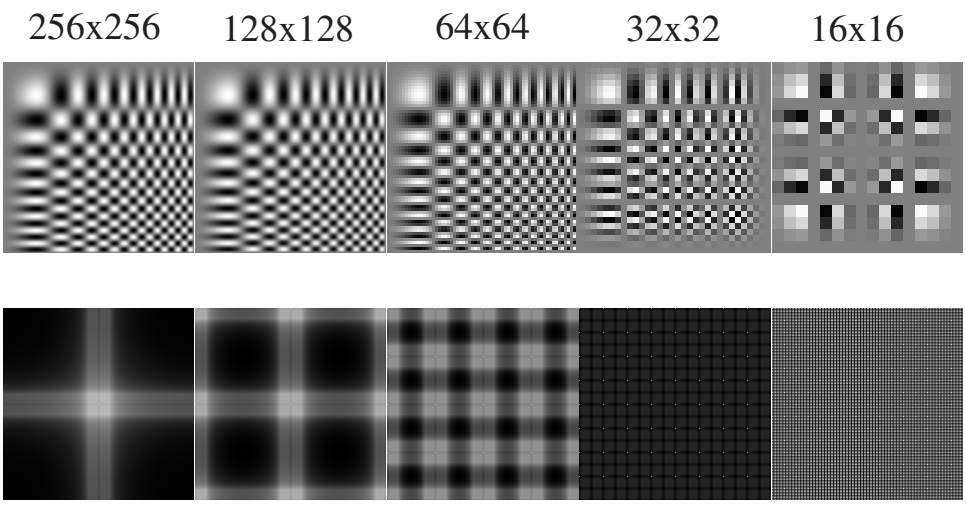


FIGURE 7.8: *The* **top row** *shows sampled versions of an image of a grid obtained by multiplying two sinusoids with linearly increasing frequency—one in x and one in y. The other images in the series are obtained by resampling by factors of two without smoothing (i.e., the next is a 128x128, then a 64x64, etc., all scaled to the same size). Note the substantial aliasing; high spatial frequencies alias down to low spatial frequencies, and the smallest image is an extremely poor representation of the large image. The* **bottom row** *shows the magnitude of the Fourier transform of each image displayed as a log to compress the intensity scale. The constant component is at the center. Notice that the Fourier transform of a resampled image is obtained by scaling the Fourier transform of the original image and then tiling the plane. Interference between copies of the original Fourier transform means that we cannot recover its value at some points; this is the mechanism underlying aliasing.*

is a filter whose response is pretty close to constant for some range of low spatial frequencies—the pass band—and whose response is also pretty close to zero—for higher spatial frequencies—the stop band. It is possible to design low-pass filters that are significantly better than Gaussians. The design process involves a detailed compromise between criteria of ripple—how flat is the response in the pass band and the stop band?—and roll-off—how quickly does the response fall to zero and stay there? The basic steps for resampling an image are given in Algorithm 7.1.

7.3 FILTERS AS TEMPLATES

It turns out that filters offer a natural mechanism for finding simple patterns because filters respond most strongly to pattern elements that look like the filter. For example, smoothed derivative filters are intended to give a strong response at a point where the derivative is large. At these points, the kernel of the filter looks like the effect it is intended to detect. The x-derivative filters look like a vertical light blob next to a vertical dark blob (an arrangement where there is a large x-derivative), and so on.

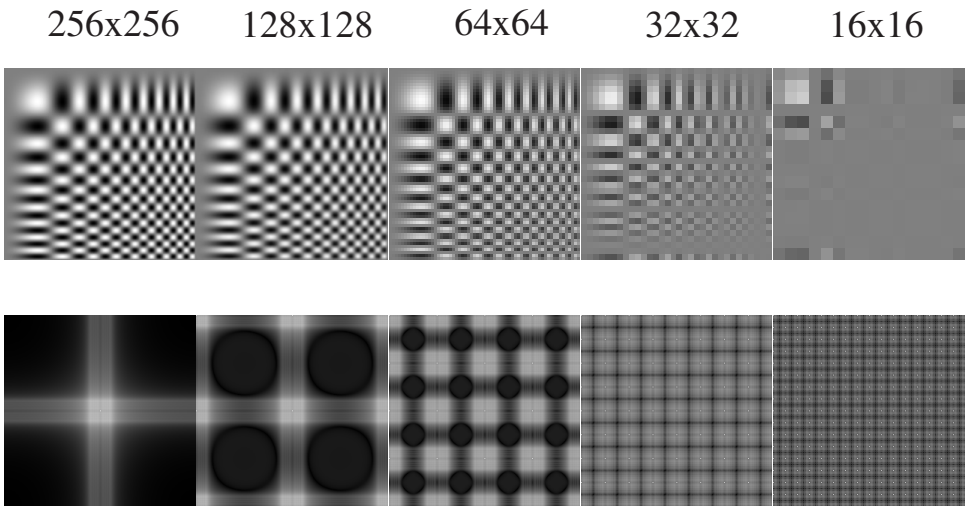


FIGURE 7.9: **Top:** *Resampled versions of the image of Figure 7.8, again by factors of two, but this time each image is smoothed with a Gaussian of σ one pixel before resampling. This filter is a low-pass filter, and so suppresses high spatial frequency components, reducing aliasing.* **Bottom:** *The effect of the low-pass filter is easily seen in these log-magnitude images; the low-pass filter suppresses the high spatial frequency components so that components interfere less, to reduce aliasing.*

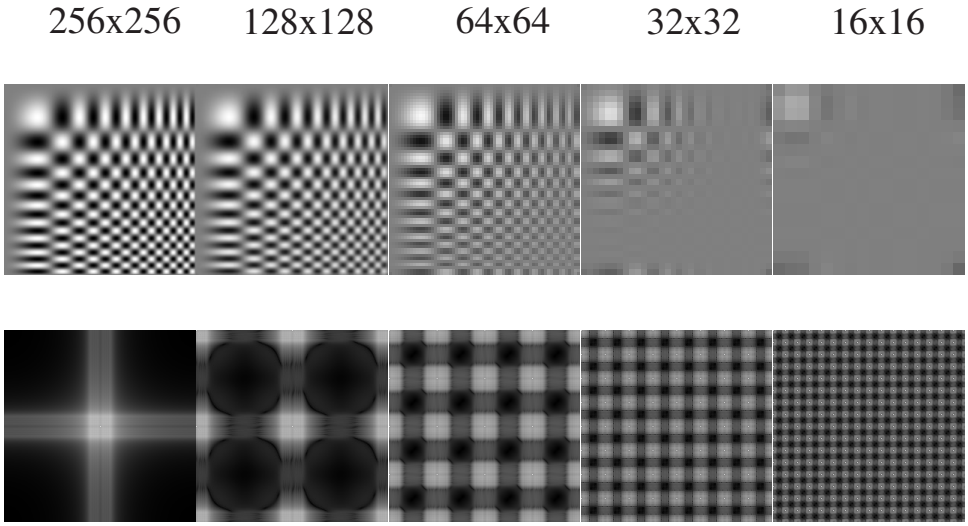


FIGURE 7.10: **Top:** *Resampled versions of the image of Figure 7.8, again by factors of two, but this time each image is smoothed with a Gaussian of σ two pixels before resampling. This filter suppresses high spatial frequency components more aggressively than that of Figure 7.9.* **Bottom:** *The effect of the low-pass filter is easily seen in these log-magnitude images; the low-pass filter suppresses the high spatial frequency components so that components interfere less, to reduce aliasing.*

Apply a low-pass filter to the original image
  (a Gaussian with a σ of between one
  and two pixels is usually an acceptable choice).
Create a new image whose dimensions on edge are half
  those of the old image
Set the value of the i, jth pixel of the new image to the value
  of the $2i$, $2j$th pixel of the filtered image

Algorithm 7.1: *Subsampling an Image by a Factor of Two.*

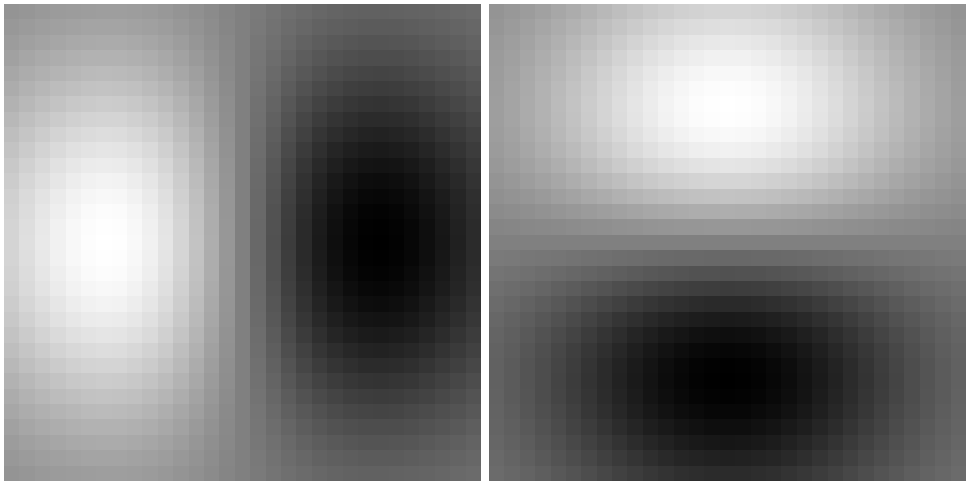


FIGURE 7.11: *Filter kernels look like the effects they are intended to detect. On the* **left**, *a smoothed derivative of Gaussian filter that looks for large changes in the x-direction (such as a dark blob next to a light blob); on the* **right**, *a smoothed derivative of Gaussian filter that looks for large changes in the y-direction.*

It is generally the case that filters intended to give a strong response to a pattern look like that pattern (Figure 7.11). This is a simple geometric result.

7.3.1 Convolution as a Dot Product

Recall from Section **??** that, for $\mathcal{G}$, the kernel of some linear filter, the response of this filter to an image $\mathcal{H}$ is given by

$$R_{ij} = \sum_{u,v} G_{i-u,j-v} H_{uv}.$$

Now consider the response of a filter at the point where i and j are zero. This is

$$R = \sum_{u,v} G_{-u,-v} H_{u,v}.$$

This response is obtained by associating image elements with filter kernel elements, multiplying the associated elements, and summing. We could scan the image into a vector and the filter kernel into another vector in such a way that associated elements are in the same component. By inserting zeros as needed, we can ensure that these two vectors have the same dimension. Once this is done, the process of multiplying associated elements and summing is precisely the same as taking a dot product.

This is a powerful analogy because this dot product, like any other, achieves its largest value when the vector representing the image is parallel to the vector representing the filter kernel. This means that a filter responds most strongly when it encounters an image pattern that looks like the filter. The response of a filter gets stronger as a region gets brighter, too.

Now consider the response of the image to a filter at some other point. Nothing significant about our model has changed. Again, we can scan the image into one vector and the filter kernel into another vector, such that associated elements lie in the same components. Again, the result of applying this filter is a dot product. There are two useful ways to think about this dot product.

7.3.2 Changing Basis

We can think of convolution as a dot product between the image and *a different vector* (because we have moved the filter kernel to lie over some other point in the image). The new vector is obtained by rearranging the old one so that the elements lie in the right components to make the sum work out. This means that, by convolving an image with a filter, we are representing the image on a new *basis* of the vector space of images—the basis given by the different shifted versions of the filter. The original basis elements were vectors with a zero in all slots except one. The new basis elements are shifted versions of a single pattern.

For many of the kernels discussed, we expect that this process will *lose* information—for the same reason that smoothing suppresses noise—so that the coefficients on this basis are redundant. This basis transformation is valuable in texture analysis. Typically, we choose a basis that consists of small, useful pattern components. Large values of the basis coefficients suggest that a pattern component is present, and texture can be represented by representing the relationships between these pattern components, usually with some form of probability model.

PART THREE

WORKING WITH POINTS

CHAPTER 8

Detecting Edges and Corners

CHAPTER 9

Interest Points

One strategy for registering an image to another is to find *interest points* and register those. Interest points have the following important properties:

- It must be possible to find them reasonably reliably, even when image brightness changes.
- It must be possible to *localize* the point (ie tell where the point is) by looking at an image window around the point. For example, a corner can be localized; but a point along a straight edge can't, because sliding a window around the point along the edge leads to a new window that looks like the original.
- The location of the point must be *covariant* under at least some natural image transformations. This means that, if the image is transformed, the point will be found in an appropriate spot in the transformed image. Equivalently, the points "stick to" objects in the image – if the camera moves, the point stays on the object where it was, and so moves in the image. So if, for example, if I_2 is obtained by rotating I_1, then there should be an interest point at each location in I_2 obtained by rotating the position of an interest point in I_1.
- It must be possible to compute a description of the image in the neighborhood of the point, so the point can be matched. Ideally, corresponding points in different images will have similar descriptions, and different points will have different images. To compute this description, we need to be able to construct a neighborhood of the interest point that is covariant. So, for example, if the image is zoomed in, the neighborhood in the image gets bigger; and if it is zoomed out, the neighborhood gets smaller. Using a fixed size neighborhood when the image zooms won't work, because the neighborhood in the zoomed in image will contain patterns that aren't in the neighborhood in the zoomed out image.

These properties are summarized in Figure **??**. The direct constructions for interest points are worth reviewing, because they expose how these properties are achieved. Learned constructions are now competitive with direct constructions, and I describe one in section 33.2.

9.1 DIRECT INTEREST POINT DETECTORS

9.1.1 Finding Corners

Interest points are usually constructed at corners, because they can be localized and are quite easy to find with a straightforward detector. At a corner, we expect two important effects. First, there should be large gradients. Second, in a small neighborhood, the gradient orientation should swing sharply. We can identify corners by looking at variations in orientation within a window. In particular, the

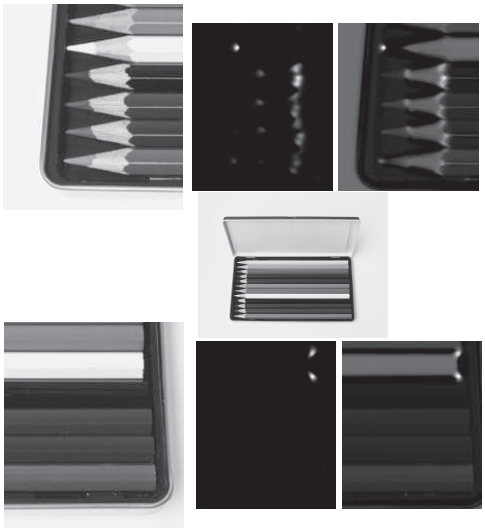


FIGURE 9.1: *The response of the Harris corner detector visualized for two detail regions of an image of a box of colored pencils (***center***).* **Top left***, a detail from the pencil points;* **top center***, the response of the Harris corner detector, where more positive values are lighter. The* **top right** *shows these overlaid on the original image. To overlay this map, we added the images, so that areas where the overlap is notably dark come from places where the Harris statistic is negative (which means that one eigenvalue of* $\mathcal{H}$ *is large, the other small). Note that the detector is affected by contrast, so that, for example, the point of the mid-gray pencil at the top of this figure generates a very strong corner response, but the points of the darker pencils do not, because they have little contrast with the tray. For the darker pencils, the strong, contrasty corners occur where the lead of the pencil meets the wood. The* **bottom** *sequence shows corners for a detail of pencil ends. Notice that responses are quite local, and there are a relatively small number of very strong corners.* Steve Gorton © Dorling Kindersley, used with permission.

matrix

$$\begin{aligned}\mathcal{H} &= \sum_{window} \left\{(\nabla I)(\nabla I)^T\right\} \\ &\approx \sum_{window} \left\{ \begin{array}{cc} (\frac{\partial G_\sigma}{\partial x} * *\mathcal{I})(\frac{\partial G_\sigma}{\partial x} * *\mathcal{I}) & (\frac{\partial G_\sigma}{\partial x} * *\mathcal{I})(\frac{\partial G_\sigma}{\partial y} * *\mathcal{I}) \\ (\frac{\partial G_\sigma}{\partial x} * *\mathcal{I})(\frac{\partial G_\sigma}{\partial y} * *\mathcal{I}) & (\frac{\partial G_\sigma}{\partial y} * *\mathcal{I})(\frac{\partial G_\sigma}{\partial y} * *\mathcal{I}) \end{array} \right\}\end{aligned}$$

gives a good idea of the behavior of the orientation in a window. In a window of constant gray level, both eigenvalues of this matrix are small because all the terms are small. In an edge window, we expect to see one large eigenvalue associated with gradients at the edge and one small eigenvalue because few gradients run in other directions. But in a corner window, both eigenvalues should be large.

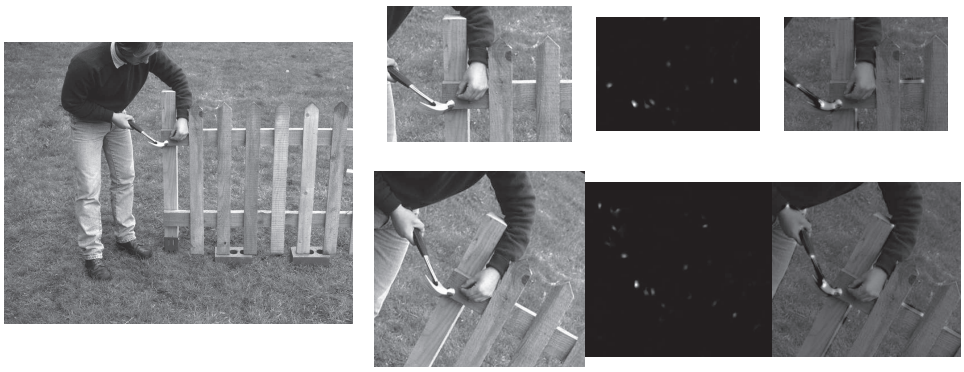


FIGURE 9.2: *The response of the Harris corner detector is unaffected by rotation and translation. The* **top row** *shows the response of the detector on a detail of the image on the* **far left**. *The* **bottom row** *shows the response of the detector on a corresponding detail from a rotated version of the image. For each row, we show the detail window (***left***); the response of the Harris corner detector, where more positive values are lighter (***center***); and the responses overlaid on the image (***right***). Notice that responses are quite local, and there are a relatively small number of very strong corners. To overlay this map, we added the images, so that areas where the overlap is notably dark come from places where the Harris statistic is negative (which means that one eigenvalue of $\mathcal{H}$ is large, the other small). The arm and hammer in the top row match those in the bottom row; notice how well the maps of Harris corner detector responses match, too.*

The *Harris corner detector* looks for local maxima of

$$\det(\mathcal{H}) - k(\frac{\text{trace}(\mathcal{H})}{2})^2$$

where k is some constant [**?**]; we used 0.5 for Figure 9.1. These local maxima are then tested against a threshold. This tests whether the product of the eigenvalues (which is $\det(\mathcal{H})$) is larger than the square of the average (which is $(\text{trace}(\mathcal{H})/2)^2$). Large, locally maximal values of this test function imply the eigenvalues are both big, which is what we want. Figure 9.1 illustrates corners found with the Harris detector. This detector is unaffected by translation and rotation (Figure 9.2).

9.1.2 Building Neighborhoods

There are many ways of representing a neighborhood around an interesting corner. Methods vary depending on what might happen to the neighborhood. In what follows, we will assume that neighborhoods are only translated, rotated, and scaled (rather than, say, subjected to an affine or projective transformation), and so without loss of generality we can assume that the patches are circular. We must estimate the radius of this circle. There is technical machinery available for the neighborhoods that result from more complex transformations, but it is more intricate; see [].

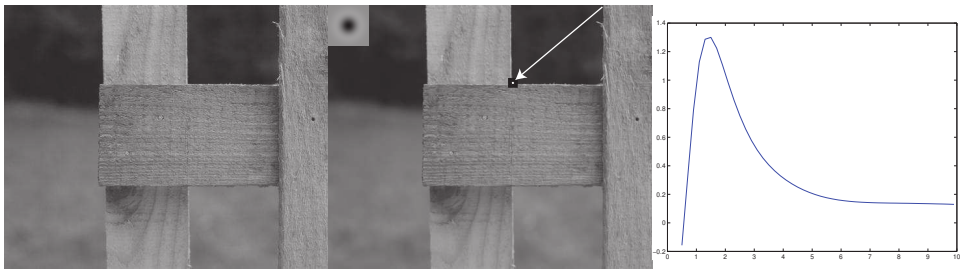


FIGURE 9.3: *The scale of a neighborhood around a corner can be estimated by finding a local extremum, in* scale *of the response at that point to a smoothed Laplacian of Gaussian kernel. On the* **left**, *a detail of a piece of fencing. In the center, a corner identified by an arrow (which points to the corner, given by a white spot surrounded by a black ring).* Overlaid *on this image is a Laplacian of Gaussian kernel, in the* **top right** *corner; dark values are negative, mid gray is zero, and light values are positive. Notice that, using the reasoning of Section 7.3, this filter will give a strong positive response for a dark blob on a light background, and a strong negative response for a light blob on a dark background, so by searching for the strongest response at this point as a function of scale, we are looking for the size of the best-fitting blob. On the* **right**, *the response of a Laplacian of Gaussian* at the location of the corner*, as a function of the smoothing parameter (which is plotted in pixels). There is one extremal scale, at approximately 2 pixels. This means that there is one scale at which the image neighborhood looks most like a blob (some corners have more than one scale).* © Dorling Kindersley, used with permission.

To turn a corner into an image neighborhood, we must estimate the radius of the circular patch (equivalently, its scale). The radius estimate should get larger proportionally when the image gets bigger. For example, in a 2x scaled version of the original image, our method should double its estimate of the patch radius. This property helps choose a method. We could center a blob of fixed appearance (say, dark on a light background) on the corner, and then choose the scale to be the radius of the best fitting blob. An efficient way to do this is to use a Laplacian of Gaussian filter.

The *Laplacian* of a function in 2D is defined as

$$(\nabla^2 f)(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

It is natural to smooth the image before applying a Laplacian. Notice that the Laplacian is a linear operator (if you're not sure about this, you should check), meaning that we could represent taking the Laplacian as convolving the image with some kernel (which we write as K_{∇^2}). Because convolution is associative, we have that

$$(K_{\nabla^2} * *(G_\sigma * *I)) = (K_{\nabla^2} * *G_\sigma) * *I = (\nabla^2 G_\sigma) * *I.$$

The reason this is important is that, just as for first derivatives, smoothing an image and then applying the Laplacian is the same as convolving the image with

the Laplacian of the kernel used for smoothing. Figure 9.3 shows the resulting kernel for Gaussian smoothing; notice that this looks like a dark blob on a light background.

Imagine applying a smoothed Laplacian operator to the image at the center of the patch. Write $\mathcal{I}$ for the image, ∇^2_σ for the smoothed Laplacian operator with smoothing constant σ, $\uparrow_k \mathcal{I}$ for the the image with size scaled by k, (x_c, y_c) for the coordinates of the patch center, and (x_{kc}, y_{kc}) for the coordinates of the patch center in the scaled image. Assume that upscaling is perfect, and there are no effects resulting from the image grid. This is fair because effects will be small for the scales of interest for us. Then, we have

$$(\nabla^2_{k\sigma} \uparrow_k \mathcal{I})(x_c, y_c) = (\nabla^2_\sigma \mathcal{I})(x_{kc}, y_{kc})$$

(this is most easily demonstrated by reasoning about the image as a continuous function, the operator as a convolution, and then using the change of variables formula for integrals). Now choose a radius r for the circular patch centered at (x_c, y_c), such that

$$r(x_c, y_c) = \underset{\sigma}{\text{argmax}} \ \nabla^2_\sigma \mathcal{I}(x_c, y_c)$$

(Figure 9.3). If the image is scaled by k, then this value of r will be scaled by k too, which is the property we wanted. This procedure looks for the scale of the best approximating blob. Notice that a Gaussian pyramid could be helpful here; we could apply the same smoothed Laplacian operator to different levels of a pyramid to get estimates of the scale.

As we have seen, orientation histograms are a natural representation of image patches. However, we cannot represent orientations in image coordinates (for example, using the angle to the horizontal image axis), because the patch we are matching to might have been rotated. We need a reference orientation so all angles can be measured with respect to that reference. A natural reference orientation is the most common orientation in the patch. We compute a histogram of the gradient orientations in this patch, and find the largest peak. This peak is the reference orientation for the patch. If there are two or more peaks of the same magnitude, we make multiple copies of the patch, one at each peak orientation.

9.1.3 Describing Neighborhoods with Orientations

We know the center, radius, and orientation of a set of an image patch, and must now represent it. Orientations should provide a good representation. They are unaffected by changes in image brightness, and different textures tend to have different orientation fields. The pattern of orientations in different parts of the patch is likely to be quite distinctive. Our representation should be robust to small errors in the center, radius, or orientation of the patch, because we are unlikely to estimate these exactly right.

We must build features that can make it obvious what orientations are present, and roughly where they are, but are robust to some rearrangement. One approach is to represent the neighborhood with a histogram of the elements that appear there. This will tell us what is present, but it confuses too many patterns with one another. For example, all neighborhoods with vertical stripes will get mixed

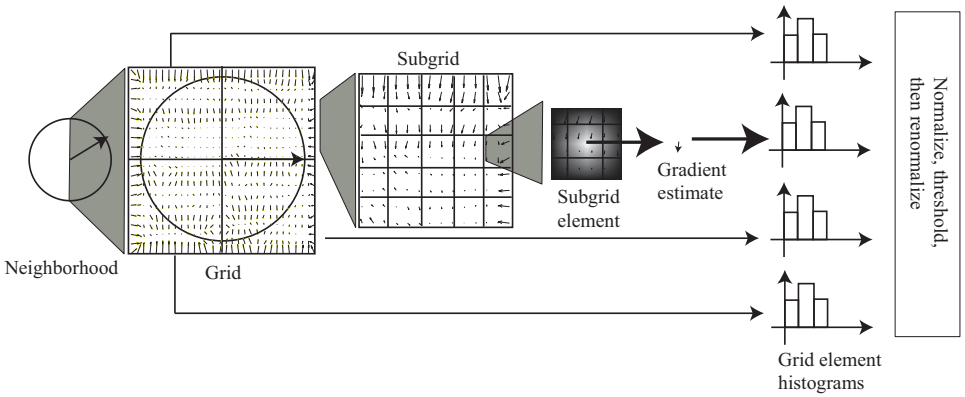


FIGURE 9.4: *To construct a SIFT descriptor for a neighborhood, we place a grid over the rectified neighborhood. Each grid is divided into a subgrid, and a gradient estimate is computed at the center of each subgrid element. This gradient estimate is a weighted average of nearby gradients, with weights chosen so that gradients outside the subgrid cell contribute. The gradient estimates in each subgrid element are accumulated into an orientation histogram. Each gradient votes for its orientation, with a vote weighted by its magnitude and by its distance to the center of the neighborhood. The resulting orientation histograms are stacked to give a single feature vector. This is normalized to have unit norm; then terms in the normalized feature vector are thresholded, and the vector is normalized again.*
TODO: Source, Credit, Permission: SIFTPIC

up, however wide the stripe. The natural approach is to take histograms locally, within subpatches of the neighborhood. This leads to a very important feature construction.

A *SIFT descriptor* (for Scale Invariant Feature Transform) is constructed out of image gradients, and uses both magnitude and orientation. The descriptor is normalized to suppress the effects of change in illumination intensity. The descriptor is a set of histograms of image gradients that are then normalized. These histograms expose general spatial trends in the image gradients in the patch but suppress detail. For example, if we estimate the center, scale, or orientation of the patch slightly wrong, then the rectified patch will shift slightly. As a result, simply recording the gradient at each point yields a representation that changes between instances of the patch. A histogram of gradients will be robust to these changes. Rather than histogramming the gradient at a set of sample points, we histogram local averages of image gradients; this helps avoid noise.

There is now extensive experimental evidence that image patches that match one another will have similar SIFT feature representations, and patches that do not will tend not to.

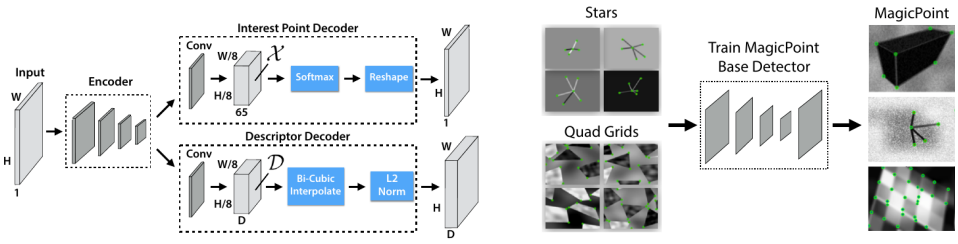


FIGURE 9.5: *SuperPoint uses an encoder with two heads (***left***), one of which predicts the locations of interest points and the other of which predicts a descriptor. The location finder assumes that there is at most one interest point per 8x8 image tile, and predicts which (if any) location is that point. A basic location finder is trained using a cross-entropy loss with a dataset of rendered images where interest point locations are known (***right***).*
TODO: Source, Credit, Permission

9.2 SUPERPOINT: A LEARNED INTEREST POINT DETECTOR

It turns out the list of properties of interest points is crisp enough that one can learn an interest point finder, and learned interest point finders now are dominant. SuperPoint uses a network architecture that is adapted to fast computation of points and descriptors, with a mixture of learned and non-learned components. This is trained in a series of steps. The first builds an elementary interest point finder. The second uses a clever trick with image transformations to significantly improve the interest point finder. The third refines point positions and descriptors with a matching loss.

9.2.1 Network Architecture

First, pass the image through an encoder, which encodes the image with series of convolutional layers, non-linear layers, and three 2×2 downsampling layers, so that it takes an $H \times W$ image and produces an $H/8 \times W/8 \times 256$ feature block. This block goes to two heads. One finds interest points, the other describes them. The interest point finder is trained discriminatively, by dividing the image into a grid of $H/8 \times W/8$ tiles (each tile is 8×8 pixels). Now assume there is at most one interest point in any tile. A 65 dimensional one-hot vector encodes where the interest point is if there is one (there are 64 locations for the point, and the last component is one if there isn't a point). The interest point finder maps the original block to an $H/8 \times W/8 \times 65$ block, which is passed through a softmax. Reshaping this with a fixed reshaping procedure gives the predicted location of the interest point. The interest point describer maps the original block to an $H/8 \times W/8 \times 256$ block. This is upsampled using a bicubic interpolation procedure (Section 33.2), and the predicted vector at each location is normalized to a unit vector.

TODO: Brief description of bicubic interpolation somewhere

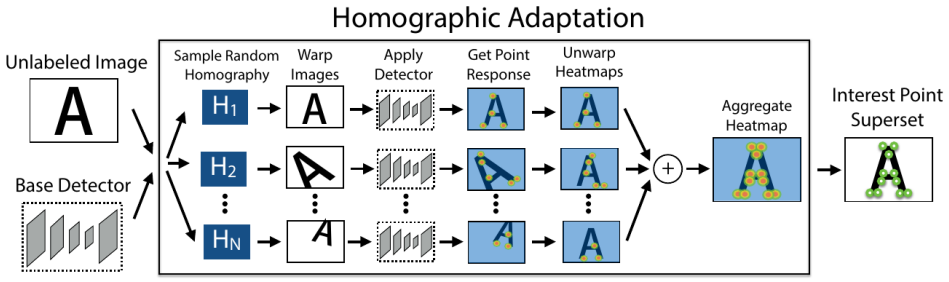


FIGURE 9.6: *The basic location finder of Figure 17 can be significantly improved by exploiting the constraint that interest point predictions should be covariant. The response of the finder to a transformed image, which is a heatmap, should be a transformed version of the response to the original image. Equivalently, apply the finder to a transformed image, and the inverse of the transformation to the resulting heatmap – that heatmap should be the same as the one the detector produces from the original image. This means that a composite finder can be built out of the the basic location finder by predicting heatmaps from images transformed with random (but carefully chosen) homographies, transforming the heatmaps back to the original image frame, then averaging them. Training this composite finder improves the original basic finder, without requiring real data.*
TODO: Source, Credit, Permission

9.2.2 Finding Interest Points

Generating a large number of relatively simple images with known interest point locations is easy. Use a simple computer graphics program to render collections of polygons; each vertex is an interest point. If any image has more than one interest points in one tile, discard all but one at random. We now have a labelled dataset of images (the labels are interest point locations), and a basic detector can be trained with this.

An interest point detector should be covariant under homographies – the interest points for a transformed image should be obtained by transforming the interest points of the original image. This likely won't be a property of the basic interest point detector, but it can be self-supervised very strongly using this idea. Write $f(\mathcal{I}, \theta)$ for the output of the interest point detector with parameters θ applied to the image $\mathcal{I}$ (this is a *heat map* – at every pixel location, there is a value giving the probability of an interest point at that location), and $\mathcal{H}(\mathcal{I})$ for the result of applying a homography to $\mathcal{I}$. The output of the detector can be thought of as an image, so a homography can be applied to it. Covariance means that the heat map $\mathcal{H}^{-1}(f(\mathcal{H}(\mathcal{I}), \theta))$ should be the same as $f(\mathcal{I}, \theta)$, at least for reasonable choices of $\mathcal{H}$. For each of the training images above, choose a collection of N homographies at random (taking care with cropping, etc. – details in []), and train the detector which produces the heat map

$$\frac{1}{N}\sum_i \mathcal{H}_i^{-1}(f(\mathcal{H}(\mathcal{I}), \theta)).$$

Training like this has quite strong effects on θ because the detector receives gradient if (say) an interest point is detected in the wrong place in a (say) rotated version of the original image. It is also an extremely efficient use of data.

9.2.3 Refining Detection and Learning to Describe

The refined detector can now be trained to improve interest point detections and to produce descriptions. Take a synthetic image with interest points known and apply a homography. The interest points in the result should be close to those predicted by applying the homography to the interest points in the original image. This property can be imposed with a cross-entropy loss between $\mathcal{H}f(\mathcal{I}, \theta)$ and $f(\mathcal{H}(\mathcal{I}), \theta)$.

Corresponding points in $\mathcal{I}$ and $\mathcal{H}(\mathcal{I})$ should have similar descriptors and pairs of points that don't correspond should have different descriptors. It is easier to impose this on tiles than points. For every pair of tiles, where one comes from $\mathcal{I}$ and the other from $\mathcal{H}(\mathcal{I})$, say the pair corresponds if there is some interest point in the first that maps to a point in the second. Otherwise, the pair does not correspond. Recall that the descriptors are computed on a coarse grid where each location corresponds to a tile (and are then upsampled). Write $\mathbf{d}(t)$ for the descriptor of a tile, and so on. The matching loss is a hinge loss that ensures that, if tiles t and t' correspond, then $\mathbf{d}^T(t)\mathbf{d}(t')$ is positive and greater than some margin, and if they do not, it is negative and less than some margin.

Resources: Interest Points *A pretrained version of SuperPoint can be found at* `https://github.com/magicleap/SuperPointPretrainedNetwork`. *There are implementations for TensorFlow () and PyTorch ().* *HPatches is an evaluation dataset (at* `https://github.com/hpatches/hpatches-dataset`*) which comes with evaluation protocols and benchmarks (at* `https://github.com/hpatches/hpatches-benchmark`*).* *OpenCV provides an implementation of the Harris corner detector, and procedures to compute SIFT descriptors.*

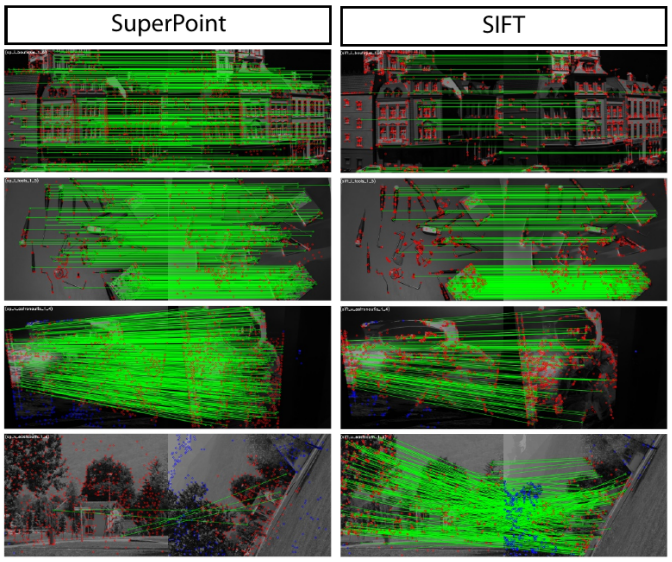


FIGURE 9.7: *SuperPoint produces many good interest point locations together with descriptors that are distinctive.* **Left** *shows SuperPoint detections and matches for four image pairs, and* **right** *for a SIFT based matcher. The images are transformed with a known homography (red dots are detected interest points; blue dots are detected interest points that are outside the field of view of the corresponding image, and so could not have a match; green lines indicate matches). Generally, SuperPoint produces large numbers of interest points that match well. The original SuperPoint is trained with relatively small image rotations, because big rotations are less common in practice, and so handles large image rotations poorly compared to a SIFT based matcher (fourth row).*
TODO: Source, Credit, Permission

CHAPTER 10

Fitting and Grouping

Sometimes data points (pixels; edge points; and so on) belong together because together they mostly conform to some explicit model. So, for example, many of the points in Figure 33.2 lie on a line (at least by eye). We must then find the model most points conform to. This activity is usually called *fitting*.

Typically, there are three problems in fitting a model to data points. First, given the points that belong to the model, what is the model? Second, which points belong to which model? Finally, how many models are there?

TODO: some more intro fitting text

10.1 LEAST SQUARES LINE FITTING

Line fitting is extremely useful. In many applications, objects are characterized by the presence of straight lines. Assume that all the points that belong to a particular line are known, and the parameters of the line must be found. We adopt the notation

$$\overline{u} = \frac{\sum u_i}{k}$$

to simplify the presentation.

10.1.1 Least Squares

Least squares is a fitting procedure with a long tradition (which is the only reason we describe it!). It yields a simple analysis but has a substantial bias. For this approach, we represent a line as $y = ax + b$. At each data point, we have (x_i, y_i); we decide to choose the line that best predicts the measured y coordinate for each measured x coordinate.

This means we want to choose the line that minimises

$$\sum_i (y_i - ax_i - b)^2.$$

By differentiation, the line is given by the solution to the problem

$$\begin{pmatrix} \overline{y^2} \\ \overline{y} \end{pmatrix} = \begin{pmatrix} \overline{x^2} & \overline{x} \\ \overline{x} & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}.$$

Although this is a standard linear solution to a classical problem, it's actually not much help in vision applications because the model is an extremely poor model. The difficulty is that the measurement error is dependent on coordinate frame — we are counting vertical offsets from the line as errors, which means that near vertical lines lead to quite large values of the error and quite funny fits (Figure 10.1). In fact, the process is so dependent on coordinate frame that it doesn't represent vertical lines at all.

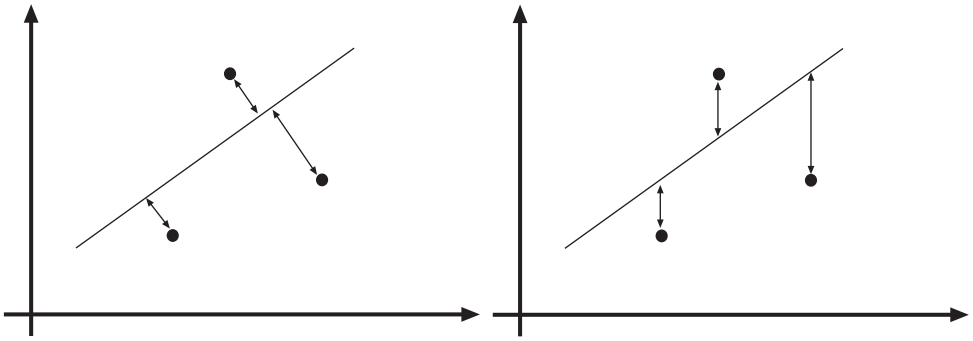


FIGURE 10.1: **Left:** *Total least-squares models data points as being generated by an abstract point along the line to which is added a vector perpendicular to the line. We wish to choose a line that minimizes the sum of distances to tokens measured (as distance usually is!) perpendicular to the line.* **Right:** *Least squares follows the same general outline, but assumes that the error appears only in the y coordinate. This yields a (very slightly) simpler mathematical problem at the cost of a poor fit.*

10.1.2 Total Least Squares

We could work with the actual distance between the point and the line (rather than the vertical distance). This leads to a problem known as *total least squares*. We can represent a line as the collection of points where $ax + by + c = 0$. Every line can be represented in this way, and we can think of a line as a triple of values (a, b, c). Notice that for $\lambda \neq 0$, the line given by $\lambda(a, b, c)$ is the same as the line represented by (a, b, c). In the exercises, you are asked to prove the simple, but extremely useful, result that the perpendicular distance from a point (u, v) to a line (a, b, c) is given by

$$\text{abs}(au + bv + c) \text{ if } a^2 + b^2 = 1.$$

In our experience, this fact is useful enough to be worth memorizing. To minimize the sum of perpendicular distances between points and lines, we need to minimize

$$\sum_i (ax_i + by_i + c)^2,$$

where $a^2 + b^2 = 1$ and C is some normalizing constant of no interest. Thus, a maximum-likelihood solution is obtained by maximizing this expression. Now using a Lagrange multiplier λ, we have a solution if

$$\begin{pmatrix} \overline{x^2} & \overline{xy} & \overline{x} \\ \overline{xy} & \overline{y^2} & \overline{y} \\ \overline{x} & \overline{y} & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \lambda \begin{pmatrix} 2a \\ 2b \\ 0 \end{pmatrix}$$

This means that

$$c = -a\overline{x} - b\overline{y}$$

and we can substitute this back to get the eigenvalue problem

$$\begin{pmatrix} \overline{x^2} - \overline{x}\,\overline{x} & \overline{xy} - \overline{x}\,\overline{y} \\ \overline{xy} - \overline{x}\,\overline{y} & \overline{y^2} - \overline{y}\,\overline{y} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \mu \begin{pmatrix} a \\ b \end{pmatrix}$$

Because this is a 2D eigenvalue problem, two solutions up to scale can be obtained in closed form (for those who care - it's usually done numerically!). The scale is obtained from the constraint that $a^2 + b^2 = 1$. The two solutions to this problem are lines at right angles, and one maximises the sum of squared distances and the other minimises it.

10.2 THE HOUGH TRANSFORM

The Hough transform is a method that promises a solution to all three (although in practice rarely delivers it). It is something worth understanding because the underlying method is quite general and appears in a number of applications.

One way to cluster points that could lie on the same structure is to record all the structures on which each point lies and then look for structures that get many votes. This (quite general) technique is known as the *Hough transform*. We take each image token and determine all structures *that could pass through that token*. We make a record of this set — you should think of this as voting — and repeat the process for each token. We decide on what is present by looking at the votes. For example, if we are grouping points that lie on lines, we take each point and vote for all lines that could go through it; we now do this for each point. The line (or lines) that are present should make themselves obvious because they pass through many points and so have many votes.

10.2.1 Fitting Lines with the Hough Transform

Hough transforms tend to be most successfully applied to line finding. We do this example to illustrate the method and its drawbacks. A line is easily parametrized as a collection of points (x, y) such that

$$x \cos\theta + y \sin\theta + r = 0.$$

Now any pair of (θ, r) represents a unique line, where $r \geq 0$ is the perpendicular distance from the line to the origin and $0 \leq \theta < 2\pi$. We call the set of pairs (θ, r) *line space*; the space can be visualized as a half-infinite cylinder. There is a family of lines that passes through any point token. In particular, the lines that lie on the curve *in line space* given by $r = -x_0 \cos\theta + y_0 \sin\theta$ all pass through the point token at (x_0, y_0).

Because the image has a known size, there is some R such that we are not interested in lines for $r > R$ — these lines are too far away from the origin for us to see them. This means that the lines we are interested in form a bounded subset of the plane, and we discretize this with some convenient grid (which we'll discuss later). The grid elements can be thought of as buckets into which we place votes. This grid of buckets is referred to as the *accumulator array*. For each point token, we add a vote to the total formed for every grid element on the curve corresponding to the point token. If there are many point tokens that are collinear, we expect there to be many votes in the grid element corresponding to that line.

10.2.2 Practical Problems with the Hough Transform

Unfortunately, the Hough transform comes with a number of important practical problems:

- **Quantization errors:** An appropriate grid size is difficult to pick. Too coarse a grid can lead to large values of the vote being obtained falsely because many quite different lines correspond to a bucket. Too fine a grid can lead to lines not being found because votes resulting from tokens that are not exactly collinear end up in different buckets, and no bucket has a large vote (Figure 10.2).
- **Difficulties with noise:** The attraction of the Hough transform is that it connects widely separated tokens that lie close to some form of parametric curve. This is also a weakness; it is usually possible to find many quite good phantom lines in a large set of reasonably uniformly distributed tokens (Figure 10.3). This means that regions of texture can generate peaks in the voting array that are larger than those associated with the lines sought (Figures 10.4 and 10.5).

The Hough transform is worth talking about because, despite these difficulties, it can often be implemented in a way that is quite useful for well-adapted problems. In practice, it is almost always used to find lines in sets of edge points. The following are useful implementation guidelines:

- **Ensure the minimum of irrelevant tokens:** This can often be done by tuning the edge detector to smooth out texture, setting the illumination to produce high-contrast edges, and so on.
- **Choose the grid carefully:** This is usually done by trial and error. It can be helpful to vote for all neighbors of a grid element at the same time one votes for the element.

10.3 FITTING CURVES

In principle, fitting curves is similar to fitting lines. We minimize the sum of squared distances between the points and the curve. However, it is usually very hard to tell the distance between a point and a curve. We can either solve this problem or apply various approximations (which are usually chosen because they are computationally simple, not because they result from clean models). We sketch some solutions for the distance problem for the two main representations of curves.

10.3.1 Implicit Curves

The coordinates of *implicit curves* satisfy some parametric equation; if this equation is a polynomial, then the curve is said to be *algebraic*, and this case is by far the most common. Some common cases are given in Table 10.1.

The Distance from a Point to an Implicit Curve Now we would like to know the distance from a data point to the closest point on the implicit curve. Assume that the curve has the form $\phi(x, y) = 0$. The vector from the closest point

TABLE 10.1: *Some implicit curves used in vision applications. Note that not all of these curves are guaranteed to have any real points on them — e.g.,* $x^2+y^2+1=0$ *doesn't. Higher degree curves are seldom used because it can be difficult to get stable fits to these curves.*

Curve	Equation
Line	$ax+by+c=0$
Circle, center (a, b), and radius r	$x^2+y^2-2ax-2by+a^2+b^2-r^2=0$
Ellipses (including circles)	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac<0$
Hyperbolae	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac>0$
Parabolae	$ax^2+bxy+cy^2+dx+ey+f=0$ where $b^2-4ac=0$
General conic sections	$ax^2+bxy+cy^2+dx+ey+f=0$

on the implicit curve to the data point is normal to the curve, so the closest point is given by finding all the (u,v) with the following properties:

1. (u,v) is a point on the curve — this means that $\phi(u,v)=0$;
2. $\mathbf{s}=(d_x,d_y)-(u,v)$ is normal to the curve.

Given all such $\mathbf{s}$, the length of the shortest is the distance from the data point to the curve. The second criterion requires a little work to determine the normal. The normal to an implicit curve is the direction in which we leave the curve fastest; along this direction, the value of ϕ must change fastest, too. This means that the normal at a point (u,v) is

$$(\frac{\partial\phi}{\partial x},\frac{\partial\phi}{\partial y}),$$

evaluated at (u,v). If the tangent to the curve is $\mathbf{T}$, then we must have $\mathbf{T}.\mathbf{s}=0$. Because we are working in 2D, we can determine the tangent from the normal, so that we must have

$$\psi(u,v;d_x,d_y)=\frac{\partial\phi}{\partial y}(u,v)\left\{d_x-u\right\}-\frac{\partial\phi}{\partial x}(u,v)\left\{d_y-v\right\}=0$$

at the point (u,v). We now have two equations in two unknowns and, *in principle* can solve them. However, this is very seldom as easy as it looks, as Example **??** indicates.

The distance between a point and a conic

A conic section is given by $ax^2+bxy+cy^2+dx+ey+f=0$. Given a data

point (d_x, d_y), the nearest point on the conic satisfies two equations:

$$au^2 + buv + cv^2 + du + ev + f = 0$$

and

$$2(a-c)uv - (2ad_y + e)u + (2cd_x + d)v + (ed_x - dd_y) = 0.$$

There can be up to four real solutions of this pair of equations (in the exercises, you are asked to demonstrate this, given an algorithm for obtaining the solutions, and asked to sketch various cases). As an example, choose the ellipse $2x^2 + y^2 - 1 = 0$, which yields the equations

$$2u^2 + v^2 - 1 = 0 \text{ and } 2uv - 4d_y u + 2d_x v = 0.$$

Let us consider a family of data points $(d_x, d_y) = (0, \lambda)$; then we can rearrange these equations to get

$$2u^2 + v^2 - 1 = 0 \text{ and } 2uv - 4\lambda u = 2u(v - 2\lambda) = 0.$$

The second equation helps: Either $u = 0$ or $v = 2\lambda$. Two of our solutions will be $(0, 1)$, $(0, -1)$. The other two are obtained by solving $2u^2 + 4\lambda^2 - 1 = 0$, which has solutions only if $-1/2 \leq \lambda \leq 1/2$. The situation is illustrated in Figure 10.6.

Approximations to the Distance Notice that for a relatively simple curve, we already have an unpleasant problem to solve. A curve with a slightly more complicated geometry — obtained by choosing ϕ to be a polynomial of higher degree, say d — leads to openly nasty problems. This is because the closest point on the curve would be obtained by solving two simultaneous polynomial equations, *both* of degree d. It can be shown that this can lead to as many as d^2 solutions, which are usually hard to obtain in practice. Various approximations to the distance between a point and an implicit algebraic curve have come into practice.

The best known is *algebraic distance*: In this case, we measure the distance between a curve and a point by evaluating the polynomial equation at that point, that is, we make the approximation

$$\text{distance between } (d_x, d_y) \text{ and } \phi(x, y) = 0 = \phi(d_x, d_y).$$

This approximation can be (rather roughly!) justified when the data points are quite close to the curve. For a point sufficiently close to the curve *and to first order*, $\phi(d_x, d_y)$ increases as (d_x, d_y) moves normal to the curve — because the normal to the curve is given by the gradient of ϕ — and does not increase as (d_x, d_y) moves tangent to the curve. One significant difficulty is that, as it stands, algebraic distance is ill defined because many polynomials correspond to the same curve. In particular, the curve given by $\mu\phi(x, y) = 0$ is the same as the curve given by $\phi(x, y) = 0$. This problem can be solved by normalizing the coefficients of the polynomial in some way.

We have already seen one example of this process in Section 10.1, where we fitted a line ($\phi(x, y) = ax + by + c = 0$) to a set of points by minimizing the

algebraic distance subject to the constraint that $a^2 + b^2 = 1$. In this case, the algebraic distance is the same as the actual distance. The choice of normalization is important. For example, if we try to fit conics ($ax^2+bxy+cy^2+dx+ey+f = 0$) using the constraint $b = 1$, we cannot fit circles. An alternative approximation is to use

$$\frac{\phi(d_x, d_y)}{|\nabla\phi(d_x, d_y)|},$$

which has the advantage of not requiring a normalizing constant; in the case of a line, this approximation is exact. Notice that this approximation has the same properties as algebraic distance — it goes up as one moves along the normal, and so on. The advantage of the approximation is that it is somewhat more accurate than algebraic distance because it is normalised by the length of the normal. This means that it can be read — roughly! — as giving the percentage distance along the normal from the curve to the point. In practice, this approximation is seldom used mainly because the use of algebraic distance yields simpler numerical problems.

Both of these approximations are dangerous because their behavior for data points that are far from the curve is strange and not well understood. As a result, the relationship between a fitted curve and a set of data points becomes a bit mysterious if the data points don't lie close to a curve of that class. Algebraic distance is used quite widely in practice because it yields easy numerical problems and can be used for higher dimensional problems like approximating the distance between points and implicit surfaces. The exact distance is often difficult to compute for such problems.

10.3.2 Parametric Curves

The coordinates of a *parametric curve* are given as parametric functions of a parameter that varies along the curve. Parametric curves have the form

$$(x(t), y(t)) = (x(t;\theta), y(t;\theta)) \quad t \in [t_{\min}, t_{\max}].$$

Table 10.2 shows the form of a variety of useful parametric curves.

The Distance from a Point to a Parametric Curve Assume we have a data point (d_x, d_y). The closest point on a parametric curve can be identified by its parameter value, which we shall write as τ. This point could lie at one or other end of the curve. Otherwise, the vector from our data point to the closest point is normal to the curve. This means that $\mathbf{s}(\tau) = (d_x, d_y) - (x(\tau), y(\tau))$ is normal to the tangent vector, so that $\mathbf{s}(\tau).\mathbf{T} = 0$. The tangent vector is

$$(\frac{dx}{dt}(\tau), \frac{dy}{dt}(\tau)),$$

which means that τ must satisfy the equation

$$\frac{dx}{dt}(\tau)\{d_x - x(\tau)\} + \frac{dy}{dt}(\tau)\{d_y - y(\tau)\} = 0.$$

Now this is only one equation, rather than two, but the situation is not much better than that for parametric curves. It is almost always the case that $x(t)$ and $y(t)$

TABLE 10.2: *A selection of parametric curves often used in vision applications. It is quite common to put together a set of cubic curves, with constraints on their coefficients such that they form a single continuous differentiable curve; the result is known as a* cubic spline.

Curves	Parametric Form	Parameters
Circles centered at the origin	$(r\sin(t), r\cos(t))$	$\theta = r$ $t \in [0, 2\pi)$
Circles	$(r\sin(t) + a, r\cos(t) + b)$	$\theta = (r, a, b)$ $t \in [0, 2\pi)$
Axis aligned ellipses	$(r_1\sin(t) + a, r_2\cos(t) + b)$	$\theta = (r_1, r_2, a, b)$ $t \in [0, 2\pi)$
Ellipses	$(\cos\phi\,(r_1\sin(t) + a) - \sin\phi\,(r_2\cos(t) + b)\,,$ $\sin\phi\,(r_1\sin(t) + a) + \cos\phi\,(r_2\cos(t) + b))$	$\theta = (r_1, r_2, a, b, \phi)$ $t \in [0, 2\pi)$
cubic segments	$(at^3 + bt^2 + ct + d, et^3 + ft^2 + gt + h)$	$\theta = (a, b, c, d, e, f, g, h)$ $t \in [0, 1]$

are polynomials because it is usually easier to do root finding for polynomials. At worst, $x(t)$ and $y(t)$ are ratios of polynomials because we can rearrange the left-hand side of our equation to come up with a polynomial in this case, too. However, we are still faced with a possibly large number of roots.

There is a second difficulty that makes fitting to parametric curves unpopular. Parametric curves with different coefficients may represent the same curve — for example, the curve $(x(t), y(t))$ for $t \in [0, 1]$ is the same as the curve $(x(2t), y(2t))$ for $t \in [0, 1/2]$. This situation can be very bad depending on the class of parametric curves that we use.

10.4 ROBUSTNESS

All of the line fitting methods described involve squared error terms. This can lead to poor fits in practice because a single wildly inappropriate data point can give errors that dominate those due to many good data points; these errors can result in a substantial bias in the fitting process (Figure 10.7). It is difficult to avoid such data points — usually called *outliers* — in practice. Errors in collecting or transcribing data points is one important source of outliers. Another common source is a problem with the model — perhaps some rare but important effect has been ignored or the magnitude of an effect has been badly underestimated. Finally, errors in correspondence are particularly prone to generating outliers. Practical vision problems usually involve outliers.

One approach to this problem puts the model at fault: The model predicts these outliers occurring perhaps once in the lifetime in the universe, and they clearly occur much more often than that. The natural response is to improve the model either by giving the noise "heavier tails" (Section 10.4.1) or by allowing an explicit outlier model. The second strategy requires a study of missing data problems — we don't know which point is an outlier and which isn't — and we defer discussion until Section **??** in the following chapter. An alternative approach is to search for points that appear to be good (Section 23.2.2).

10.4.1 M-estimators

The difficulty with modeling the source of outliers is that the model might be wrong. Generally, the best we can hope for from a probabilistic model of a process is that it is quite close to the right model. Assume that we are guaranteed that our model of a process is close to the right model — say, the distance between the density functions in some appropriate sense is less than ϵ. We can use this guarantee to reason about the design of estimation procedures for the parameters of the model. In particular, we can choose an estimation procedure by assuming that nature is malicious and well informed about statistics. These are generally sound assumptions for any enterprise; the world is full of opportunities for painful and expensive lessons in practical statistics. In this line of reasoning, we assess the goodness of an estimator by assuming that somewhere in the collection of processes close to our model is the real process, and it just happens to be the one that makes the estimator produce the worst possible estimates. The best estimator is the one that behaves best on the worst distribution close to the parametric model. This is a criterion that can be used to produce a wide variety of estimators.

An *M-estimator* estimates parameters by minimizing an expression of the form

$$\sum_i \rho(r_i(\mathbf{x}_i, \theta); \sigma),$$

where θ are the parameters of the model being fitted and $r_i(\mathbf{x}_i, \theta)$ is the residual error of the model on the ith data point. Generally, $\rho(u; \sigma)$ looks like u^2 for part of its range and then flattens out. A common choice is

$$\rho(u; \sigma) = \frac{u^2}{\sigma^2 + u^2}.$$

The parameter σ controls the point at which the function flattens out; we have plotted a variety of examples in Figure 10.8. There are many other M-estimators available. Typically, they are discussed in terms of their *influence function*, which is defined as

$$\frac{\partial \rho}{\partial \theta}.$$

This is natural because our criterion is

$$\sum_i \rho(r_i(\mathbf{x}_i, \theta); \sigma) \frac{\partial \rho}{\partial \theta} = 0.$$

For the kind of problems we consider, we would expect a good influence function to be antisymmetric — there is no difference between a slight overprediction and a slight underprediction — and to tail off with large values — because we want to limit the influence of the outliers.

There are two tricky issues with using M-estimators. First, the extremization problem is non-linear and must be solved iteratively. The standard difficulties apply: There may be more than one local minimum, the method may diverge, and the behavior of the method is likely to be quite dependent on the start point. A common strategy for dealing with this problem is to draw a subsample of the data

set, fit to that subsample using least squares, and use this as a start point for the fitting process. We do this for a large number of different subsamples — enough to ensure that there is a high probability that in that set there is at least one that consists entirely of good data points.

Second, as Figures 10.9 and 10.10 indicate, the estimators require a sensible estimate of σ, which is often referred to as *scale*. Typically, the scale estimate is supplied at each iteration of the solution method; a popular estimate of scale is

$$\sigma^{(n)} = 1.4826 \text{ median}_i \, |r_i^{(n)}(x_i; \theta^{(n-1)})| \, .$$

An M-estimator can be thought of as a trick for ensuring that there is more probability in the tails than would otherwise occur with a quadratic error. The function that is minimized looks like distance for small values of $\mathbf{x}$ — thus, for valid data points, the behavior of the M-estimator should be rather like maximum likelihood — and like a constant for large values of $\mathbf{x}$ — meaning that a component of probability is given to the tails of the distribution. The strategy of the previous section can be seen as an M-estimator, but with the difficulty that the influence function is discontinuous, meaning that obtaining a minimum is tricky.

10.4.2 RANSAC

An alternative to modifying the generative model to have heavier tails is to search the collection of data points for good points. This is quite easily done by an iterative process: First, we choose a small subset of points and fit to that subset, then we see how many other points fit to the resulting object. We continue this process until we have a high probability of finding the structure we are looking for.

For example, assume that we are fitting a line to a data set that consists of about 50% outliers. If we draw pairs of points uniformly and at random, then about a quarter of these pairs will consist entirely of good data points. We can identify these good pairs by noticing that a large collection of other points lie close to the line fitted to such a pair. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

This approach leads to an algorithm — search for a random sample that leads to a fit on which many of the data points agree. The algorithm is usually called `RANSAC`, for RANdom SAmple Consensus, and is displayed in Algorithm 23.1. To make this algorithm practical, we need to choose three parameters.

The Number of Samples Required Our samples consist of sets of points drawn uniformly and at random from the data set. Each sample contains the minimum number of points required to fit the abstraction we wish to fit. For example, if we wish to fit lines, we draw pairs of points; if we wish to fit circles, we draw triples of points, and so on. We assume that we need to draw n data points, and that w is the fraction of these points that are good (we need only a reasonable estimate of this number). Now the expected value of the number of

draws k required to get one point is given by

$$\begin{aligned}\mathrm{E}[k] &= 1P(\text{one good sample in one draw}) + 2P(\text{one good sample in two draws}) + \dots \\ &= w^n + 2(1-w^n)w^n + 3(1-w^n)^2 w^n + \dots \\ &= w^{-n}\end{aligned}$$

(where the last step takes a little manipulation of algebraic series). We would like to be fairly confident that we have seen a good sample, so we wish to draw more than w^{-n} samples; a natural thing to do is to add a few standard deviations to this number. The standard deviation of k can be obtained as

$$SD(k) = \frac{\sqrt{1-w^n}}{w^n}.$$

An alternative approach to this problem is to look at a number of samples that guarantees a low probability z of seeing only bad samples. In this case, we have

$$(1-w^n)^k = z,$$

which means that

$$k = \frac{\log(z)}{\log(1-w^n)}.$$

It is common to have to deal with data where w is unknown. However, each fitting attempt contains information about w. In particular, if n data points are required, then we can assume that the probability of a successful fit is w^n. If we observe a long sequence of fitting attempts, we can estimate w from this sequence. This suggests that we start with a relatively low estimate of w, generate a sequence of attempted fits, and then improve our estimate of w. If we have more fitting attempts than we need for the new, the process can stop. The problem of updating the estimate of w reduces to estimating the probability that a coin comes up heads or tails given a sequence of fits.

Telling Whether a Point Is Close We need to determine whether a point lies close to a line fitted to a sample. We do this by determining the distance between the point and the fitted line, and testing that distance against a threshold d; if the distance is below the threshold, the point lies close. In general, specifying this parameter is part of the modeling process. For example, when we fitted lines using maximum likelihood, there was a term σ in the model (which disappeared in the manipulations to find an maximum). This term gives the average size of deviations from the model being fitted.

In general, obtaining a value for this parameter is relatively simple. We generally need only an order of magnitude estimate, and the same value applies to many different experiments. The parameter is often determined by trying a few values and seeing what happens; another approach is to look at a few characteristic data sets, fitting a line by eye, and estimating the average size of the deviations.

The Number of Points That Must Agree Assume that we have fitted a line to some random sample of two data points. We need to know whether that line

is good. We do this by counting the number of points that lie within some distance of the line (the distance was determined in the previous section). In particular, assume that we know the probability that an outlier lies in this collection of points; write this probability as y. We should like to choose some number of points t such that y^t is small (say less than 0.05).

There are two ways to proceed. One is to notice that $y \leq (1 - w)$ and to choose t such that $(1 - w)^t$ is small. Another is to get an estimate of y from some model of outliers — for example, if the points lie in a unit square, the outliers are uniform, and the distance threshold is d, then $y \leq 2\sqrt{2}d$.

10.4.3 Example: Fitting a Line with RANSAC

10.5 HUMAN VISION: GROUPING AND GESTALT

Remarkably, the human vision system appears to have strong opinions about when data points belong together, which is why it is easy to tell that the points in Figure 33.2 mostly lie on a line. In turn, context affects how things are perceived (e.g., see the illusion of Figure 10.11). This observation led the Gestalt school of psychologists to reject the study of responses to stimuli and to emphasize grouping as the key to understanding visual perception. To them, grouping meant the tendency of the visual system to assemble some components of a picture together and to perceive them together (this supplies a rather rough meaning to the word context used above). Grouping, for example, is what causes the Müller–Lyer illusion of Figure 10.11 — the vision system assembles the components of the two arrows, and the horizontal lines look different from one another because they are peceived as components of a whole, rather than as lines. Furthermore, many grouping effects can't be disrupted by cognitive input; for example, you can't make the lines in Figure 10.11 look equal in length by deciding not to group the arrows.

A common experience of segmentation is the way that an image can resolve itself into a **figure** — typically, the significant, important object — and a **ground** — the background on which the figure lies. However, as Figure 10.12 illustrates, what is figure and what is ground can be profoundly ambiguous, meaning that a richer theory is required.

The Gestalt school used the notion of a *gestalt* — a whole or a group — and of its **gestaltqualität** — the set of internal relationships that makes it a whole (e.g., Figure 10.11) as central components in their ideas. Their work was characterized by attempts to write down a series of rules by which image elements would be associated together and interpreted as a group. There were also attempts to construct algorithms, which are of purely historical interest (see **?** for an introductory account that places their work in a broad context).

The Gestalt psychologists identified a series of factors, which they felt predisposed a set of elements to be grouped. These factors are important because it is quite clear that the human vision system uses them in some way. Furthermore, it is reasonable to expect that they represent a set of preferences about when tokens belong together that lead to a useful intermediate representation.

There are a variety of factors, some of which postdate the main Gestalt movement:

- **Proximity:** Tokens that are nearby tend to be grouped.

- **Similarity:** Similar tokens tend to be grouped together.
- **Common fate:** Tokens that have coherent motion tend to be grouped together.
- **Common region:** Tokens that lie inside the same closed region tend to be grouped together.
- **Parallelism:** Parallel curves or tokens tend to be grouped together.
- **Closure:** Tokens or curves that tend to lead to closed curves tend to be grouped together.
- **Symmetry:** Curves that lead to symmetric groups are grouped together.
- **Continuity:** Tokens that lead to continuous — as in joining up nicely, rather than in the formal sense — curves tend to be grouped.
- **Familiar configuration:** Tokens that, when grouped, lead to a familiar object tend to be grouped together.

These laws are illustrated in Segmentation/Figures 10.13, 10.14, 10.16, and **??**.

These rules can function fairly well as explanations, but they are insufficiently crisp to be regarded as forming an algorithm. The Gestalt psychologists had serious difficulty with the details, such as when one rule applied and when another. It is difficult to supply a satisfactory algorithm for using these rules — the Gestalt movement attempted to use an extremality principle.

Familiar configuration is a particular problem. The key issue is to understand just *what* familiar configuration applies in a problem and how it is selected. For example, look at Figure **??**. One might argue that the blobs are grouped because they yield a sphere. The difficulty with this view is explaining how this occurred — where did the hypothesis that a sphere is present come from? A search through all views of all objects is one explanation, but one must then explain how this search is organized. Do we check *every view* of *every* sphere with *every* pattern of spots? How can this be done efficiently?

The Gestalt rules do offer some insight because they explain what happens in various examples. These explanations seem to be sensible because they suggest that the rules help solve problems posed by visual effects that arise commonly in the real world — that is, they are *ecologically valid*. For example, continuity may represent a solution to problems posed by occlusion — sections of the contour of an occluded object could be joined up by continuity (see Figure 10.15).

This tendency to prefer interpretations that are explained by occlusion leads to interesting effects. One is the *illusory contour*, illustrated in Figure 10.17. Here a set of tokens suggests the presence of an object most of whose contour has no contrast. The tokens appear to be grouped together because they provide a cue to the presence of an occluding object, which is so strongly suggested by these tokens that one could fill in the no-contrast regions of contour.

This ecological argument has some force because it is possible to interpret most grouping factors using it. Common fate can be seen as a consequence of the fact that components of objects tend to move together. Equally, symmetry is a

useful grouping cue because there are a lot of real objects that have symmetric or close to symmetric contours. Essentially, the ecological argument says that tokens are grouped because doing so produces representations that are helpful for the visual world that people encounter. The ecological argument has an appealing, although vague, statistical flavor. From our perspective, Gestalt factors provide interesting hints, but should be seen as the *consequences* of a larger grouping process, rather than the process itself.

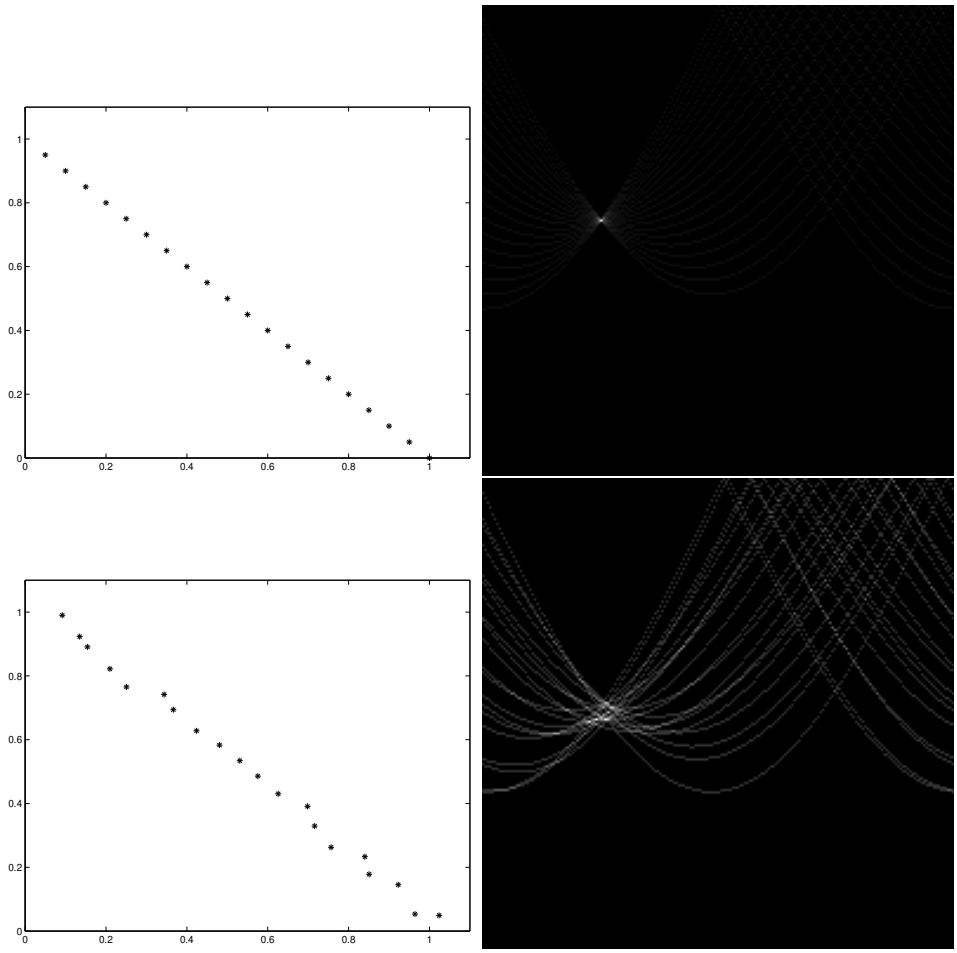


FIGURE 10.2: *The Hough transform maps each point like token to a curve of possible lines (or other parametric curves) through that point. These figures illustrate the Hough transform for lines. The* **left-hand column** *shows points, and the* **right-hand column** *shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). The* **top** *row shows what happens using a set of 20 points drawn from a line. On the* **top right**, *the accumulator array for the Hough transform of these points. Corresponding to each point is a curve of votes in the accumulator array; the largest set of votes is 20 (which corresponds to the brightest point). The horizontal variable in the accumulator array is θ and the vertical variable is r; there are 200 steps in each direction, and r lies in the range $[0, 1.55]$. On the* **bottom**, *these points have been offset by a random vector each element of which is uniform in the range $[0, 0.05]$; note that this offsets the curves in the accumulator array shown next to the points; the maximum vote is now 6 (which corresponds to the brightest value in this image — this value would be difficult to see on the same scale as the top image).*

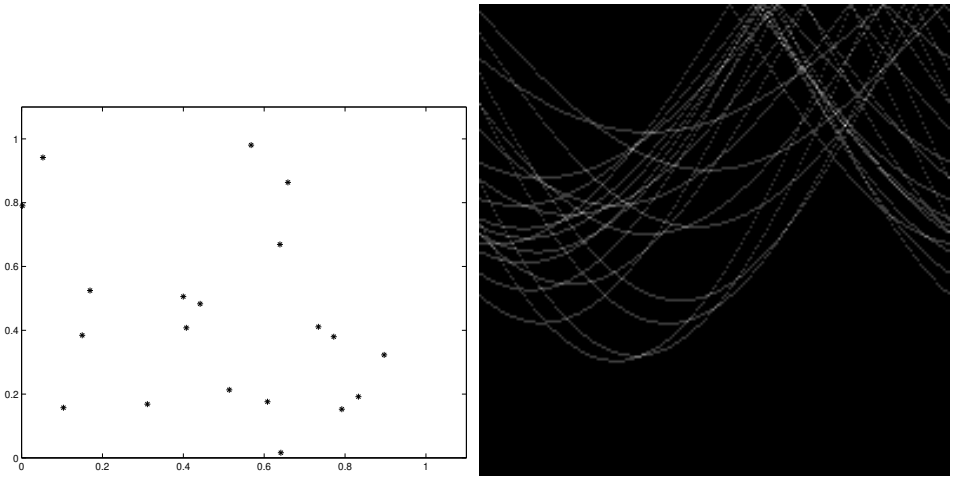


FIGURE 10.3: *The Hough transform for a set of random points can lead to quite large sets of votes in the accumulator array. As in Figure 10.2, the* **left-hand column** *shows points, and the* **right-hand column** *shows the corresponding accumulator arrays (the number of votes is indicated by the grey level, with a large number of votes being indicated by bright points). In this case, the data points are noise points (both coordinates are uniform random numbers in the range* $[0, 1]$*); the accumulator array in this case contains many points of overlap, and the maximum vote is now 4 (compared with 6 in Figure 10.2). Figures 10.4 and 10.5 explore noise issues somewhat further.*

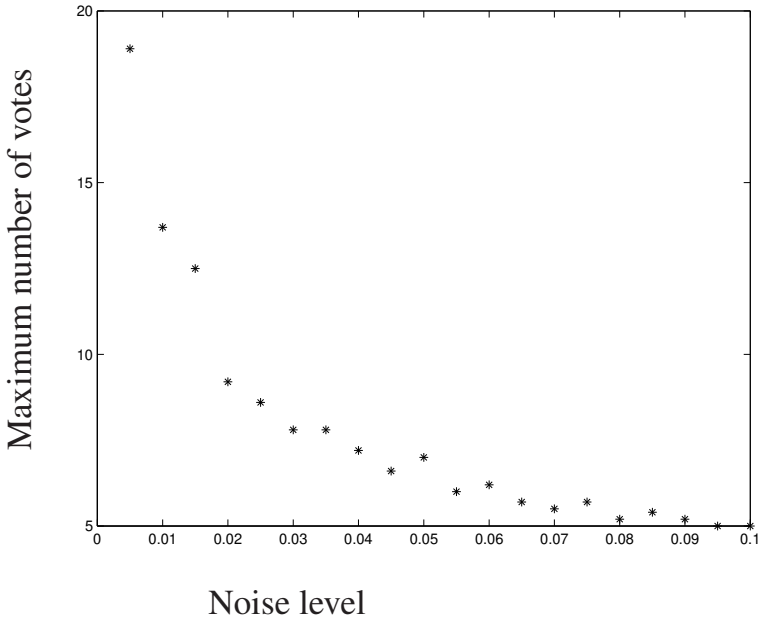


FIGURE 10.4: *The effects of noise make it difficult to use a Hough transform robustly. The plot shows the maximum number of votes in the accumulator array for a Hough transform of 20 points on a line perturbed by uniform noise plotted against the magnitude of the noise. The noise displaces the curves from each other and quickly leads to a collapse in the number of votes. The plot has been averaged over 10 trials. The accumulator array had the same quantization for each case shown here.*

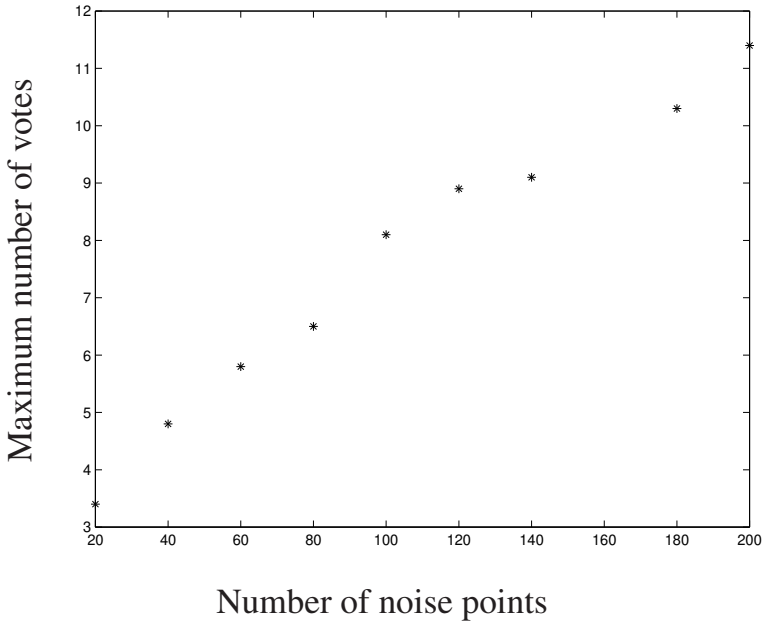


FIGURE 10.5: *A plot of the maximum number of votes in the accumulator array for a Hough transform of a set of points whose coordinates are uniform random numbers in the range* $[0, 1]$ *plotted against the number of points. As the level of noise goes up, the number of votes in the right bucket goes down, and the prospect of obtaining a large spurious vote in the accumulator array goes up. The plots have again been averaged over 10 trials. Compare this figure with Figure 10.4, but notice the slightly different scales; the comparison suggests that it can be quite difficult to pull a line out of noise with a Hough transform (because the number of votes for the line might be comparable with the number of votes for a line due to noise). These figures illustrate the importance of ruling out as many noise tokens as possible before performing a Hough transform.*

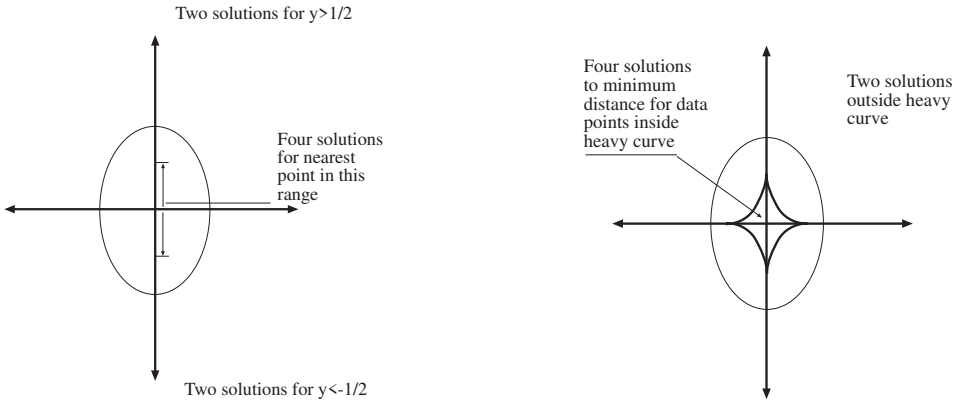


FIGURE 10.6: *On the* **left**, *the example worked in the text, where we study the number of possible solutions for the distance between a point and an ellipse for data points lying on the vertical axis. The figure on the* **right** *indicates the general case for this ellipse.*

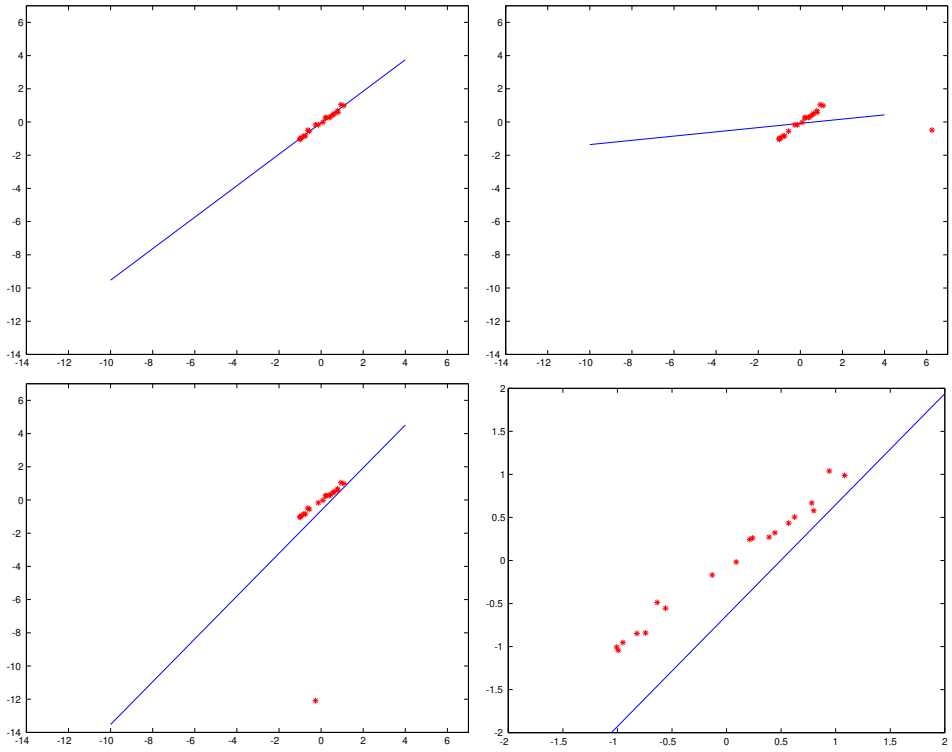


FIGURE 10.7: *Least-squares line fitting is extremely sensitive to outliers, both in x and y coordinates. At the* **top left**, *a good least-squares fit of a line to a set of points.* **Top right** *shows the same set of points, but with the x coordinate of one point corrupted. In this case, the slope of the fitted line has swung wildly.* **Bottom left** *shows the same set of points, but with the y-coordinate of one point corrupted. In this particular case, the x intercept has changed. These three figures are on the same set of axes for comparison, but this choice of axes does not clearly show how bad the fit is for the third case;* **bottom right** *shows a detail of this case — the line is clearly a bad fit.*

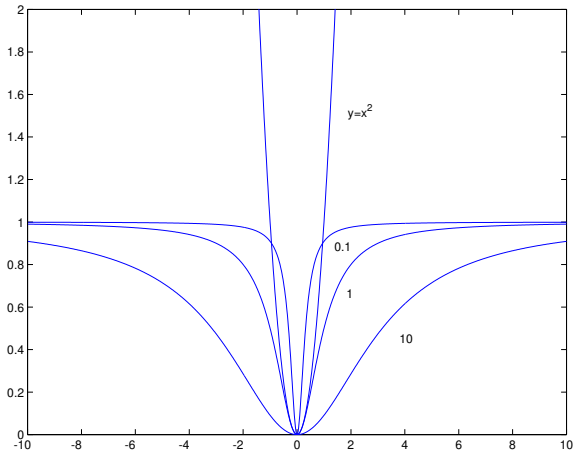


FIGURE 10.8: *The function* $\rho(x;\sigma) = x^2/(\sigma^2 + x^2)$*, plotted for* $\sigma^2 = 0.1$*,* 1*, and* 10*, with a plot of* $y = x^2$ *for comparison. Replacing quadratic terms with* ρ *reduces the influence of outliers on a fit — a point that is several multiples of* σ *away from the fitted curve is going to have almost no effect on the coefficients of the fitted curve because the value of* ρ *will be close to* 1 *and will change extremely slowly with the distance from the fitted curve.*

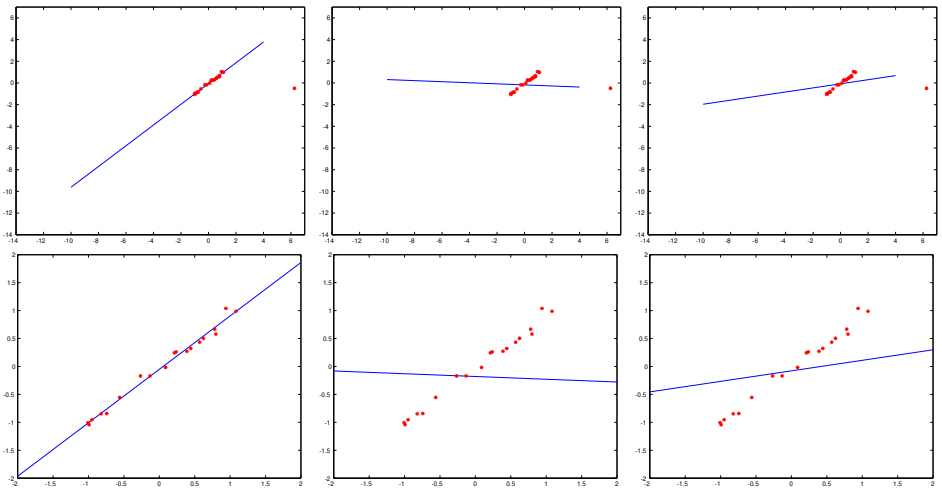


FIGURE 10.9: *The* **top row** *shows lines fitted to the second dataset of Figure 10.7 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.8). On the* **left**, *μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the* **center**, *the value of μ is too small so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the* **right**, *the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The* **bottom row** *shows closeups of the fitted line and the non-outlying data points for the same cases.*

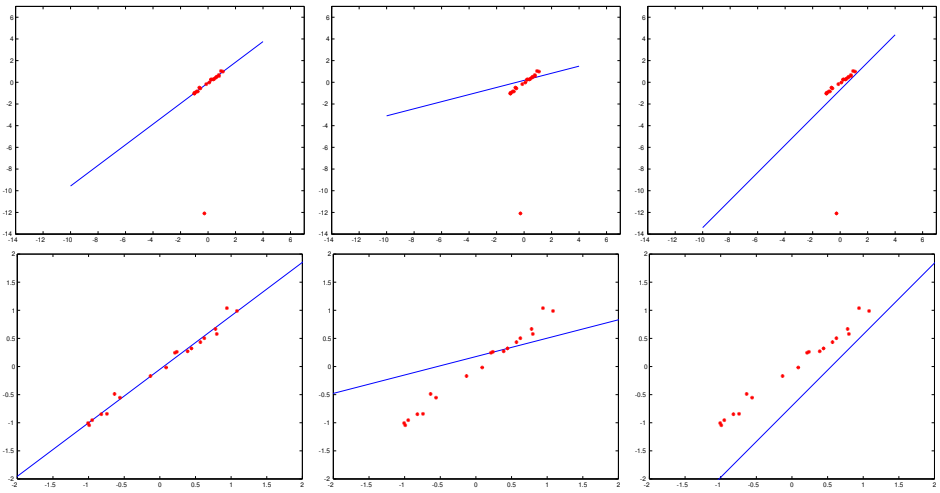


FIGURE 10.10: *The* **top row** *shows lines fitted to the third dataset of Figure 10.7 using a weighting function that deemphasizes the contribution of distant points (the function ϕ of Figure 10.8). On the* **left**, *μ has about the right value; the contribution of the outlier has been down-weighted, and the fit is good. In the* **center**, *the value of μ is too small, so that the fit is insensitive to the position of all the data points, meaning that its relationship to the data is obscure. On the* **right**, *the value of μ is too large, meaning that the outlier makes about the same contribution that it does in least-squares. The* **bottom row** *shows closeups of the fitted line and the non-outlying data points, for the same cases.*

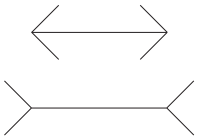


FIGURE 10.11: *The famous Muller–Lyer illusion; the horizontal lines are in fact the same length, although that belonging to the upper figure looks longer. Clearly, this effect arises from some property of the relationships that form the whole (the* gestaltqualität*), rather than from properties of each separate segment.*

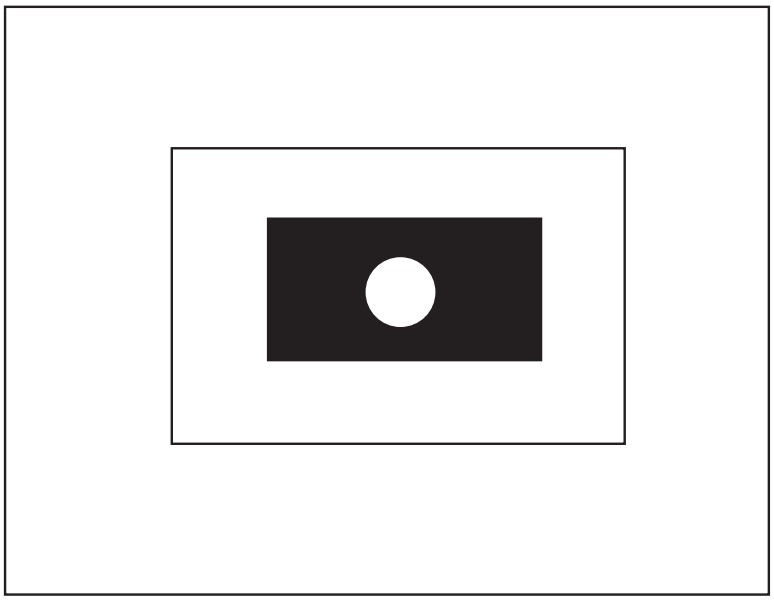


FIGURE 10.12: *One view of segmentation is that it determines which component of the image forms the figure and which the ground. The figure illustrates one form of ambiguity that results from this view; the white circle can be seen as figure on the black rectangular ground, or as ground where the figure is a black rectangle with a circular hole in it — the ground is then a white square.*

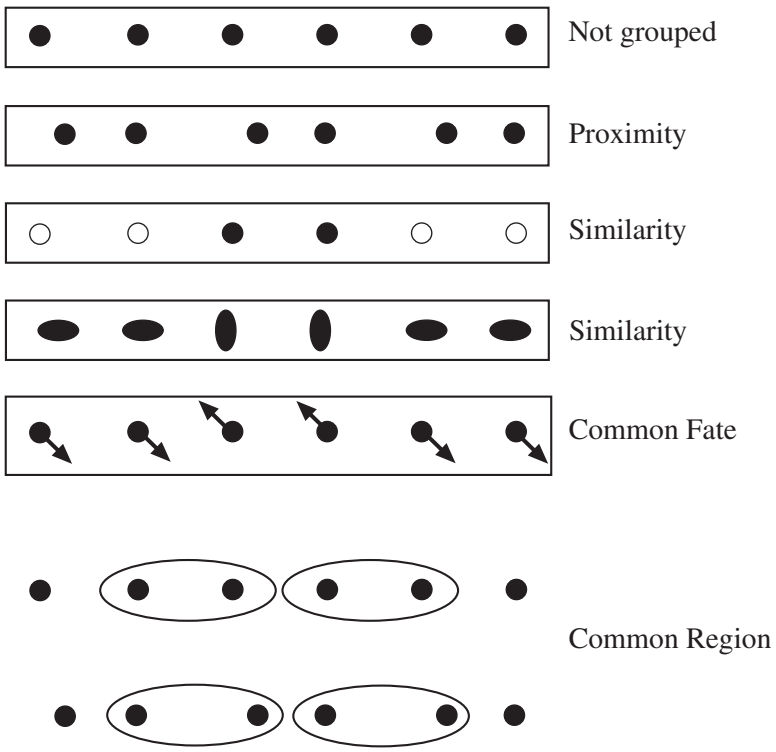


FIGURE 10.13: *Examples of Gestalt factors that lead to grouping (which are described in greater detail in the text).*

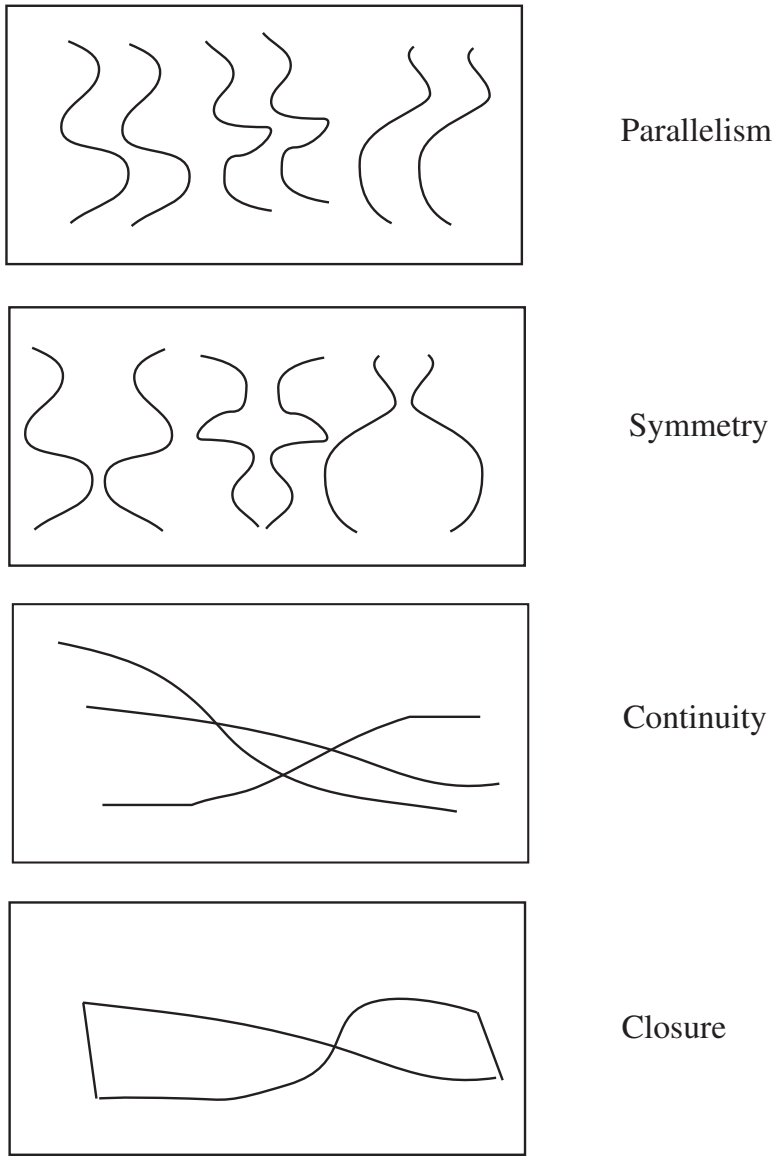


FIGURE 10.14: *Examples of Gestalt factors that lead to grouping (which are described in greater detail in the text).*

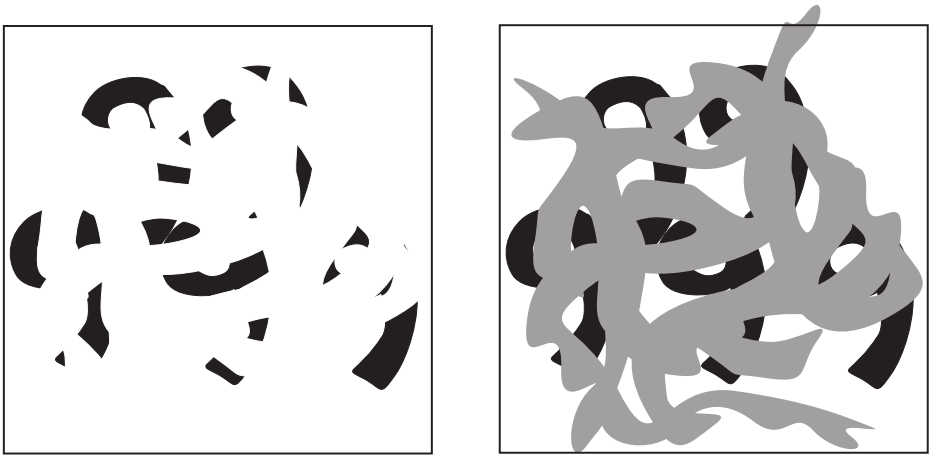


FIGURE 10.15: *Occlusion appears to be an important cue in grouping. It may be possible to see the pattern on the* **left** *as a collection of digits; that on the* **right** *is quite clearly some occluded digits. The black regions on the left and right are the same. The visual system appears to be helped by evidence that separated tokens are separated for a reason, rather than just scattered.*

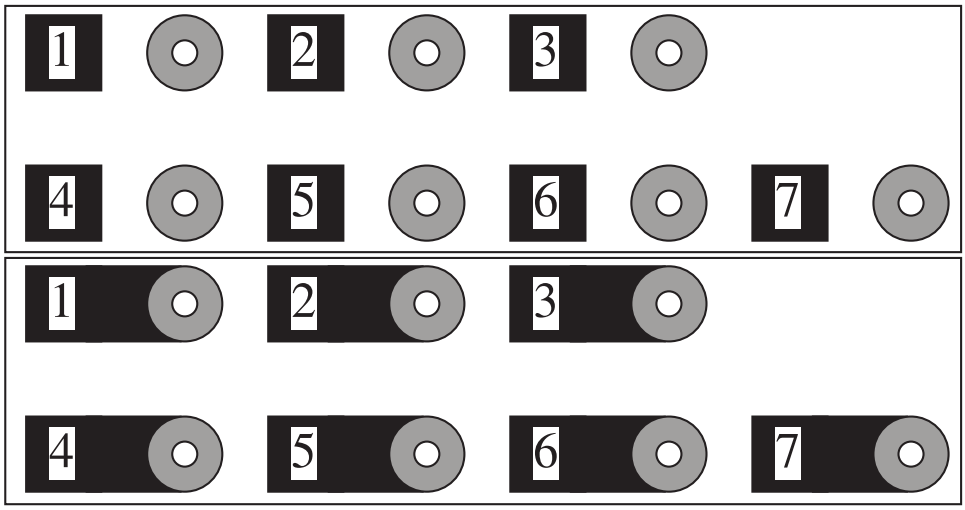


FIGURE 10.16: *An example of grouping phenomena in real life. The buttons on an elevator in the computer science building at U.C. Berkeley used to be laid out as in the* **top** *figure. It was common to arrive at the wrong floor and discover that this was because you'd pressed the wrong button — the buttons are difficult to group unambiguously with the correct label, and it is easy to get the wrong grouping at a quick glance. A public-spirited individual filled in the gap between the numbers and the buttons, as in the* **bottom** *figure, and the confusion stopped because the proximity cue had been disambiguated.*

FIGURE 10.17: *The tokens in these images suggest the presence of occluding objects whose boundaries don't contrast with much of the image. Notice that one has a clear impression of the position of the entire contour of the occluding figures. These contours are known as* illusory contours.

CHAPTER 11

Registration

Computing a transformation that aligns an image or a depth map or a set of images with another such is generally known as *registration*. One approach to registration is to abstract the image (etc.) as a set of points, often referred to as *point clouds*. Generally, we will write $\mathcal{P}$ for a point cloud whose i'th point is $\mathbf{p}_i$ and so on. Now assume we have two point clouds, $\mathcal{X}$ and $\mathcal{Y}$. Each is obtained by starting with a set of reference points, dropping some of them at random, transforming the remaining points, then adding noise to the points and also including some pure noise points. This is a reasonable model of what a depth camera or a LIDAR sensor might produce when it views an object. The model means that there is transformation that maps from one to the other, though it may not place every point in one on top of some point in the other. We want to determine the transformation between the two point clouds.

This problem occurs in a wide range of practical applications. As we shall see, calibrating a camera involves solving a version of this problem (Section 33.2). Determining where you are in a known map very often involves solving a version of this problem. Imagine, for example, a camera looking directly downwards from an aircraft flying at fixed height. The image in the camera translates and rotates as the aircraft moves. If we can compute the transformation from image i to image $i + 1$, we can tell how the aircraft has moved. Another useful case occurs when we have a depth map of a known object and want to compute the *pose* of the object (its position and orientation in the frame of the depth sensor). We could do so by reducing the object model to a point cloud, then computing the transformation from the object model to the point cloud from the depth sensor.

How one approaches this class of problem depends on three important factors.

- **Correspondence:** if it is known *which* observation corresponds to *which* reference point, the problem is relatively straightforward to solve (unless there are unusual noise effects). This case is uncommon, but does occur. In robotics, *beacons* are objects that identify themselves (perhaps by wearing a barcode; by transmitting some code; by a characteristic pattern) and can be localized. They are useful, precisely because they yield correspondence and so simplify computing the transformation. If correspondence is not known, which is the usual case, computing the transformation becomes rather harder.

- **Transformation:** there are closed form solutions for known correspondence and Euclidean or affine transformations. Homographies (and higher dimensional analogs) do not admit closed form transformations.

- **Noise:** computing a transformation can become very hard if many of the observations do not come from reference points, if many of the reference points are dropped, or if some observations are subject to very large noise effects.

11.1 REGISTRATION WITH KNOWN CORRESPONDENCE AND GAUSSIAN NOISE

11.1.1 Affine Transformations and Gaussian Noise

In the simplest case, the correspondence is known – perhaps $\mathcal{Y}$ consists of beacons and $\mathcal{X}$ of observations – and the only noise is Gaussian (so $N = M$). We will assume the noise is isotropic, which is by far the most usual case. Once you have followed this derivation, you will find it easy to incorporate a known covariance matrix. We have

$$\mathbf{x}_i = \mathcal{M}\mathbf{y}_i + \mathbf{t} + \xi_i \tag{11.1}$$

where ξ_i is the value of a normal random variable with mean $\mathbf{0}$ and covariance matrix $\Sigma = \sigma^2\mathcal{I}$. A natural procedure to estimate $\mathcal{M}$ and $\mathbf{t}$ is to maximize the likelihood of the noise. Because it will be useful later, we assume that there is a weight w_i for each pair, so the negative log-likelihood we must minimize is proportional to

$$\sum_i w_i \left(\mathbf{x}_i - \mathcal{M}\mathbf{y}_i - \mathbf{t}\right)^T \left(\mathbf{x}_i - \mathcal{M}\mathbf{y}_i - \mathbf{t}\right) \tag{11.2}$$

(the constant of proportionality is σ^2, which doesn't affect the optimization problem). The gradient of this cost with respect to $\mathbf{t}$ is

$$-2\sum_i w_i \left(\mathbf{x}_i - \mathcal{M}\mathbf{y}_i - \mathbf{t}\right) \tag{11.3}$$

which vanishes at the solution. In turn, if $\sum_i w_i\mathbf{x}_i = \sum_i w_i\mathcal{M}\mathbf{y}_i$, $\mathbf{t} = \mathbf{0}$. One straightforward way to achieve this is to ensure that both the observations and the reference points have a center of gravity at the origins. Write

$$\mathbf{c}_x = \frac{\sum_i w_i\mathbf{x}_i}{\sum_i w_i} \tag{11.4}$$

for the center of gravity of the observations (etc.) Now form

$$\mathbf{u}_i = \mathbf{x}_i - \mathbf{c}_x \text{ and } \mathbf{v}_i = \mathbf{y}_i - \mathbf{c}_y \tag{11.5}$$

and if we use $\mathcal{U}$ and $\mathcal{V}$, then the translation will be zero. In turn, the translation from the original reference points to the original observations is $\mathbf{c}_x - \mathbf{c}_y$.

We obtain $\mathcal{M}$ by minimizing

$$\sum_i w_i \left(\mathbf{u}_i - \mathcal{M}\mathbf{v}_i\right)^T \left(\mathbf{u}_i - \mathcal{M}\mathbf{v}_i\right). \tag{11.6}$$

Now write $\mathcal{W} = \mathsf{diag}\left([w_1, \ldots, w_N]\right)$, $\mathcal{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_N]$ (and so on). You should check that the objective can be rewritten as

$$\mathsf{Tr}\left(\mathcal{W}(\mathcal{U} - \mathcal{M}\mathcal{V})^T(\mathcal{U} - \mathcal{M}\mathcal{V})\right). \tag{11.7}$$

Now the trace is linear; $\mathcal{U}^T\mathcal{U}$ is constant; and we can rotate matrices through the trace (Section 33.2). This means the cost is equivalent to

$$\mathsf{Tr}\left(-2\mathcal{M}\mathcal{V}\mathcal{U}^T + \mathcal{M}^T\mathcal{M}\mathcal{V}\mathcal{V}^T\right) \tag{11.8}$$

which will be minimized when

$$\mathcal{M}\mathcal{V}\mathcal{W}\mathcal{V}^T = \mathcal{V}\mathcal{W}\mathcal{U}^T \tag{11.9}$$

(which you should check). Many readers will recognize a least squares solution here. The trace isn't necessary here, but it's helpful to see an example using the trace, because it will be important in the next case.

11.1.2 Euclidean Motion and Gaussian Noise

One encounters affine transformations relatively seldom in practice, though they do occur. Much more interesting is the case where the transformation is Euclidean. The least squares solution above isn't good enough, because the $\mathcal{M}$ obtained that way won't be a rotation matrix. But we can obtain a least squares solution with a rotation matrix, using a neat trick. We adopt the notation of the previous section, and change coordinates from $\mathbf{x}_i$ to $\mathbf{u}_i$ as above to remove the need to estimate translation.

We must choose $\mathcal{R}$ to minimize

$$\sum_i w_i(\mathbf{u}_i - \mathcal{R}\mathbf{v}_i)^T(\mathbf{u}_i - \mathcal{R}\mathbf{v}_i). \tag{11.10}$$

This can be done in closed form (a fact you should memorize). Equivalently, we must minimize

$$\begin{aligned}\sum_i w_i(\mathbf{u}_i - \mathcal{R}\mathbf{v}_i)^T(\mathbf{u}_i - \mathcal{R}\mathbf{v}_i) &= \mathsf{Tr}\left(\mathcal{W}(\mathcal{U} - \mathcal{R}\mathcal{V})(\mathcal{U} - \mathcal{R}\mathcal{V})^T\right) \\ &= \mathsf{Tr}\left(-2\mathcal{U}\mathcal{W}\mathcal{V}^T\mathcal{R}^T\right) + K \\ &\quad \text{(because } \mathcal{R}^T\mathcal{R} = \mathcal{I}\text{)} \\ &= -2\mathsf{Tr}\left(\mathcal{R}\mathcal{U}\mathcal{W}\mathcal{V}^T\right)\end{aligned}$$

Now we compute an SVD of $\mathcal{U}\mathcal{V}^T$ to obtain $\mathcal{U}\mathcal{W}\mathcal{V}^T = \mathcal{A}\mathcal{S}\mathcal{B}^T$ (where $\mathcal{A}$, $\mathcal{B}$ are orthonormal, and $\mathcal{S}$ is diagonal – Section 33.2 if you're not sure). Now $\mathcal{B}^T\mathcal{R}\mathcal{A}$ is orthonormal, and we must maximize $\mathsf{Tr}\left(\mathcal{B}^T\mathcal{R}\mathcal{A}\mathcal{S}\right)$, meaning $\mathcal{B}^T\mathcal{R}\mathcal{A} = \mathcal{I}$ (check this if you're not certain), and so $\mathcal{R} = \mathcal{B}\mathcal{A}^T$.

Procedure: 11.1 *Weighted Least Squares for Euclidean Transformations*

We have N reference points $\mathbf{x}_i$ whose location is measured in the agent's coordinate system. Each corresponds to a point in the world coordinate system with known coordinates $\mathbf{y}_i$, and the change of coordinates is a Euclidean transformation (rotation $\mathcal{R}$, translation $\mathbf{t}$). For each $(\mathbf{x}_i, \mathbf{y}_i)$ pair, we have a weight w_i. We wish to minimize

$$\sum_i w_i(\mathbf{x}_i - \mathcal{R}\mathbf{y}_i - \mathbf{t})^T(\mathbf{x}_i - \mathcal{R}\mathbf{y}_i - \mathbf{t}) \tag{11.11}$$

Write

$$\begin{aligned}
\mathbf{c}_x &= \frac{\sum_i w_i \mathbf{x}_i}{\sum_i w_i} \\
\mathbf{c}_y &= \frac{\sum_i w_i \mathbf{y}_i}{\sum_i w_i} \\
\mathbf{u}_i &= \mathbf{x}_i - \mathbf{c}_x \\
\mathbf{v}_i &= \mathbf{y}_i - \mathbf{c}_y
\end{aligned}$$

Then the least squares estimate $\hat{\mathbf{t}}$ of $\mathbf{t}$ is

$$\hat{\mathbf{t}} = \mathbf{c}_x - \mathbf{c}_y \tag{11.12}$$

Write $\mathcal{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N]$ (etc); $\mathcal{W} = \text{diag}(w_1, \ldots, w_N)$; and $\text{SVD}(\mathcal{U}\mathcal{S}\mathcal{V}) = \mathcal{A}\Sigma\mathcal{B}^T$. The least squares estimate $\hat{\mathcal{R}}$ is

$$\hat{\mathcal{R}} = \mathcal{B}\mathcal{A}^T \tag{11.13}$$

11.1.3 Homographies and Gaussian Noise

We now work with $d = 2$, and allow the transformation to be a homography. Solving for a homography requires solving an optimization problem, but estimating a homography from data is useful, and relatively easy to do. Furthermore, we can't recover the translation component from centers of gravity (exercises **TODO:** homography exercise). In all cases of interest, the points $\mathbf{x}_i$ and $\mathbf{y}_i$ will be supplied in affine coordinates, rather than homogeneous coordinates, and we convert to homogeneous coordinates by attaching a 1, as before. Write m_{ij} for the i, j'th element of matrix $\mathcal{M}$. In affine coordinates, a homography $\mathcal{M}$ will map $\mathbf{y}_i = (y_{i,x}, y_{i,y})$ to $\mathbf{x}_i = (x_{i,x}, x_{i,y})$ where

$$x_{i,x} = \frac{m_{11}y_{i,x} + m_{12}y_{i,y} + m_{13}}{m_{31}x_{i,x} + m_{32}x_{i,y} + m_{33}} \text{ and } x_{i,y} = \frac{m_{21}y_{i,x} + m_{22}y_{i,y} + m_{23}}{m_{31}x_{i,x} + m_{32}x_{i,y} + m_{33}} \tag{11.14}$$

Write $\mathcal{M}(\mathbf{y})$ for the result of applying the homography to $\mathbf{y}$, *in affine coordinates*. In most cases of interest, the coordinates of the points are not measured precisely, so

we observe $\mathbf{x}_i = \mathcal{M}(\mathbf{y}_i) + \xi_i$, where ξ_i is some noise vector drawn from an isotropic normal distribution with mean $\mathbf{0}$ and covariance Σ.

The error will be in affine coordinates – for example, in the image plane – which justifies working in affine rather than homogeneous coordinates. Again, we assume that the noise is isotropic, and so that $\Sigma = \sigma^2\mathcal{I}$. The homography can be estimated by minimizing the negative log-likelihood of the noise, so we must minimize

$$\sum_i w_i \xi_i^T \xi_i \tag{11.15}$$

where

$$\xi_i = \begin{bmatrix} x_{i,x} - \frac{m_{11}y_{i,x}+m_{12}y_{i,y}+m_{13}}{m_{31}y_{i,x}+m_{32}y_{i,y}+m_{33}} \\ x_{i,y} - \frac{m_{21}y_{i,x}+m_{22}y_{i,y}+m_{23}}{m_{31}y_{i,x}+m_{32}y_{i,y}+m_{33}} \end{bmatrix} \tag{11.16}$$

using standard methods (Levenberg-Marquardt is favored; Chapter 33.2). This approach is sometimes known as *maximum likelihood* . Experience teaches that this optimization is not well behaved without a strong start point.

There is an easy construction for a good start point. Notice that the equations for the homography mean that

$$x_{i,x}(m_{31}y_{i,x} + m_{32}y_{i,y} + m_{33}) - m_{11}y_{i,x} + m_{12}y_{i,y} + m_{13} = 0 \tag{11.17}$$

and

$$x_{i,y}(m_{31}y_{i,x} + m_{32}y_{i,y} + m_{33}) - m_{21}y_{i,x} + m_{22}y_{i,y} + m_{23} = 0 \tag{11.18}$$

so each corresponding pair of points $\mathbf{x}_i$, $\mathbf{y}_i$ yields two *homogeneous* linear equations in the coefficients of the homography. They are homogeneous because scaling $\mathcal{M}$ doesn't change what it does to points (check this if you're uncertain). If we obtain sufficient points, we can solve the resulting system of homogeneous linear equations. Four point correspondences yields an unambiguous solution; more than four – which is bteter – can be dealt with by least squares (exercises **TODO:** fourpoint homography). The resulting estimate of $\mathcal{M}$ has a good reputation as a start point for a full optimization.

Procedure: 11.2 *Estimating a Homography from Data*

Given N known source points $\mathbf{y}_i = (y_{i,x}, y_{i,y})$ in affine coordinates and N corresponding target points $\mathbf{x}_i$ with measured locations $(x_{i,x}, x_{i,y})$ and where measurement noise has zero mean and covariance $\Sigma = \sigma^2\mathcal{I}$, estimate the homography $\mathcal{M}$ with i, j'th element m_{ij} by minimizing:

$$\sum_i \xi_i^T \xi_i \tag{11.19}$$

where

$$\xi = \begin{bmatrix} x_{i,x} - \frac{m_{11}y_{i,x}+m_{12}y_{i,y}+m_{13}}{m_{31}y_{i,x}+m_{32}y_{i,y}+m_{33}} \\ x_{i,y} - \frac{m_{21}y_{i,x}+m_{22}y_{i,y}+m_{23}}{m_{31}y_{i,x}+m_{32}y_{i,y}+m_{33}} \end{bmatrix} \tag{11.20}$$

Obtain a start point by as a least-squares solution to the set of homogeneous linear equations

$$x_{i,x}(m_{31}y_{i,x} + m_{32}y_{i,y} + m_{33}) - m_{11}y_{i,x} + m_{12}y_{i,y} + m_{13} = 0 \tag{11.21}$$

and

$$x_{i,y}(m_{31}y_{i,x} + m_{32}y_{i,y} + m_{33}) - m_{21}y_{i,x} + m_{22}y_{i,y} + m_{23} = 0. \tag{11.22}$$

11.1.4 Projective Transformations and Gaussian Noise

A *projective transformation* is the analogue of a homography for higher dimensions. In affine coordinates, a projective transformation $\mathcal{M}$ will map $\mathbf{y}_i = (y_{i,1}, \ldots, y_{i,d})$ to $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$ where

$$x_{i,1} = \frac{m_{11}y_{i,1} + \ldots + m_{1d}y_{i,d} + m_{1(d+1)}}{m_{(d+1)1}y_{i,1} + \ldots + m_{(d+1)d}y_{i,d} + m_{(d+1)(d+1)}} \tag{11.23}$$

and

$$x_{i,d} = \frac{m_{d1}y_{i,1} + \ldots + m_{dd}y_{i,d} + m_{d(d+1)}}{m_{(d+1)1}y_{i,1} + \ldots + m_{(d+1)d}y_{i,d} + m_{(d+1)(d+1)}} \tag{11.24}$$

Estimating this transformation follows the recipe for a homography, but there are now more parameters. I have put the result in a box, below.

Procedure: 11.3 *Estimating a Projective Transformation from Data*

Given N known source points $\mathbf{y}_i = (y_{i,1}, \ldots, y_{i,d})$ in affine coordinates and N corresponding target points $\mathbf{x}_i$ with measured locations $(x_{i,1}, \ldots, x_{i,d})$ and where measurement noise has zero mean and is isotropic, the homography $\mathcal{M}$ with i, j'th element m_{ij} by minimizing:

$$\sum_i \xi_i^T \Sigma^{-1} \xi_i \tag{11.25}$$

where

$$\xi_i = \begin{bmatrix} x_{i,1} - \frac{m_{11}y_{i,1}+\ldots+m_{1d}y_{i,d}+m_{1(d+1)}}{m_{(d+1)1}x_{i,1}+\ldots+m_{(d+1)d}x_{i,d}+m_{(d+1)(d+1)}} \\ \ldots \\ x_{i,d} - \frac{m_{d1}y_{i,1}+\ldots+m_{dd}y_{i,d}+m_{d(d+1)}}{m_{(d+1)1}x_{i,1}+\ldots+m_{(d+1)d}x_{i,d}+m_{(d+1)(d+1)}} \end{bmatrix} \tag{11.26}$$

Obtain a start point by as a least squares solution to the set of homogeneous linear equations

$$\begin{aligned} 0 &= x_{i,1}(m_{(d+1)1}y_{i,1} + \ldots + m_{(d+1)d}y_{i,d} + m_{(d+1)(d+1)}) - \\ &\quad m_{11}y_{i,1} + \ldots + m_{1d}y_{i,d} + m_{1(d+1)} \\ &\ldots \\ 0 &= x_{i,d}(m_{(d+1)1}y_{i,1} + \ldots + m_{(d+1)d}y_{i,d} + m_{(d+1)(d+1)}) - \\ &\quad m_{(d+1)1}x_{i,1} + \ldots + m_{(d+1)d}x_{i,d} + m_{(d+1)(d+1)} \end{aligned}$$

11.2 UNKNOWN CORRESPONDENCE

11.2.1 Unknown Correspondence and ICP

Now assume correspondences are not known, and some reference (resp. observed) points may not even have corresponding observed (resp. reference) points. We have N reference points $\mathbf{y}_i$ and M observed points $\mathbf{x}_i$. For the moment, we will assume that all weights w_i are 1. A straightforward, and very effective, recipe for registering the points is *iterative closest points* or *ICP*. The key insight here is that, if the transformation is very close to the identity, then the $\mathbf{y}_{c(i)}$ that corresponds to $\mathbf{x}_i$ should be the closest reference point to $\mathbf{x}_i$. This finding the closest reference point to each measurement and computing the transformation using that correspondence. But the transformation might not be close to the identity, and so the correspondences might change. We could repeat the process until they stop changing.

Formally, start with a transformation estimate $\mathcal{T}_1$, a set of $\mathbf{m}_i^{(1)} = \mathcal{T}^{(1)}(\mathbf{y}_i)$ and then repeat two steps:

- **Estimate correspondences** using the transformation estimate. Then, for each $\mathbf{x}_i$, we find the closest $\mathbf{m}^{(n)}$ (say $\mathbf{m}_c^{(n)}$); then $\mathbf{x}_i$ corresponds to $\mathbf{m}_{c(i)}^{(n)}$.

- **Estimate a transformation** $\mathcal{T}^{(n+1)}$ using the corresponding pairs. This maps each $\mathbf{m}^{(n)}_{c(i)}$ to its corresponding $\mathbf{x}_i$.

These steps are repeated until convergence, which can be tested by checking if the correspondences don't change or if $\mathcal{T}^{(n+1)}$ is very similar to the identity. The require transformation is then

$$\mathcal{T}^{(n+1)} \circ \mathcal{T}^{(n)} \circ \ldots \mathcal{T}^{(1)} \tag{11.27}$$

There are a number of ways in which this very useful and very general recipe can be adapted. First, if there is any description of the points available, it can be used to cut down on correspondences (so, for example, we match only red points to red points, green points to green points, and so on). Second, finding an exact nearest neighbor in a large point cloud is hard and slow, and we might need to subsample the point clouds or pass to approximate nearest neighbors (more details below). Third, points that are very far from the nearest neighbor might cause problems, and we might omit them (again, more details below).

11.2.2 ICP and Sampling

One particularly useful application of ICP occurs when one wishes to register a mesh to a set of points. For example, you might want to register a cloud of measured points to a mesh model of an object built using a CAD modelling system. A natural procedure is to sample points on the mesh model to get a point cloud, then treat the problem using ICP. Another useful application is when one has two mesh models, where the triangulation of the meshes might not be the same. In this case, you could sample both meshes to end up with two point clouds, then register the point clouds. How one samples the mesh or meshes is important.

The ICP recipe becomes difficult to apply to point clouds when M or N are very large. One obvious strategy to control this problem applies when something else – say, a color measurement – is known about each point. For example, we might get such data by using a range camera aligned with a conventional camera, so that every point in the depth map comes with a color. When extra information is available, one searches only compatible pairs for correspondences.

Large point clouds are fairly common in autonomous vehicle applications. For example, the measurements might be LIDAR measurements of some geometry. It is quite usual now to represent that geometry with another, perhaps enormous, point cloud, which you could think of as a map. Registration would then tell the vehicle where it was in the map. Notice that in this application, there is unlikely to be a measurement that exactly corresponds to each reference point. Instead, when the registration is correct, every $\mathbf{x}_i$ is very close to some transformed $\mathbf{y}_i$, so a least squares estimate is entirely justified. In cases like this, one can subsample the reference point cloud, the measurement point cloud, or both.

The sampling procedure depends on the application, and can have significant effects. For example, imagine we are working with LIDAR on a vehicle which is currently in an open space next to a wall (Figure **??**). There will be many returns from the wall, and likely few from the open space. Uniformly sampled measurements would still have many returns from the wall, and few from the open space. This

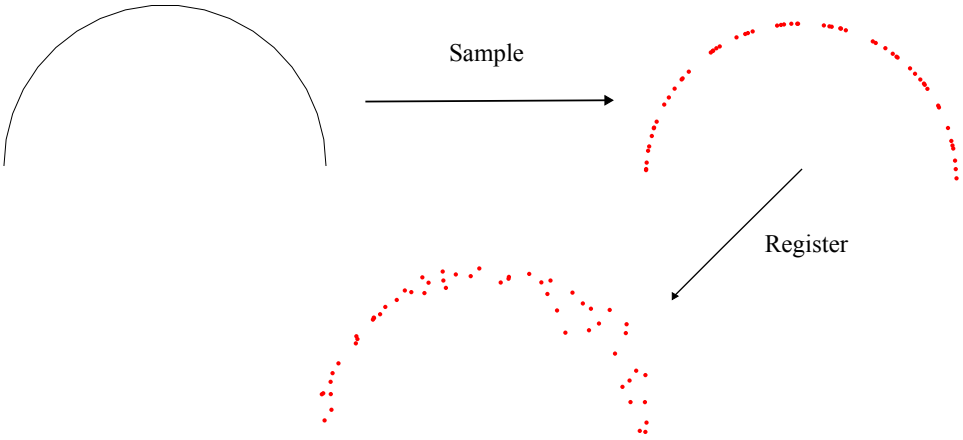


FIGURE 11.1: *In many problems, one has to register a mesh – which might come from a CAD model – to a set of measurements – which might come from LIDAR or from a range camera.* **Top left:** *shows a view of a very simple 1D mesh, in 2D. Registering this mesh to a set of measurements* **bottom** *is a straightforward application of ICP. One samples the mesh* **top left**, *then registers this set of points to the measurements.*

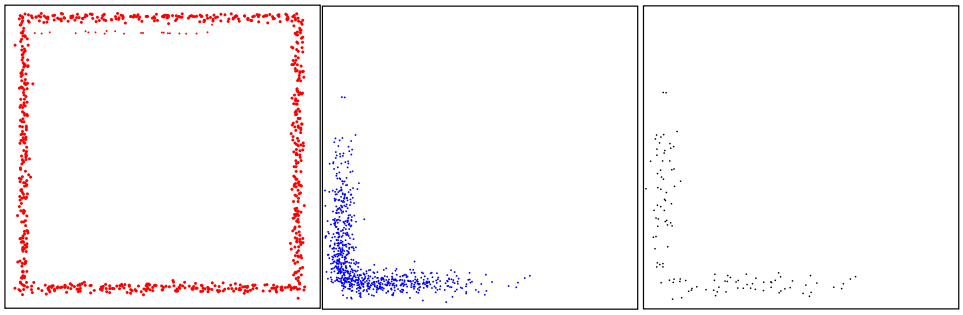


FIGURE 11.2: *On the* **left** *a map of a simple arena, represented as a point cloud. Such a map could be obtained by registering LIDAR measurements to one another. A LIDAR or depth sensor produces measurements in the sensor's coordinate system, and registering these measurements to the map will reveal where the sensor is. However, the sensor may measure points more densely at some positions than at others.* **Left** *shows such a measurement; note the heavy sampling of points near the corner and the light sampling on the edges. This can bias the registration, because the large number of points near the corner mean that the registration error consists mostly of errors from these points. It can also create significant computational problems, because finding the closest points will become slower as the number of points increases. A stratified sample of the measurements (***right***) is obtained by dividing the plane (in this case) into cells of equal area (usually a grid), then resampling the measurements at random* so there are no more than a fixed number of samples in each box. *Such a sample can both reduce bias and improve the speed of registration.* **TODO:** Source, Credit, Permission

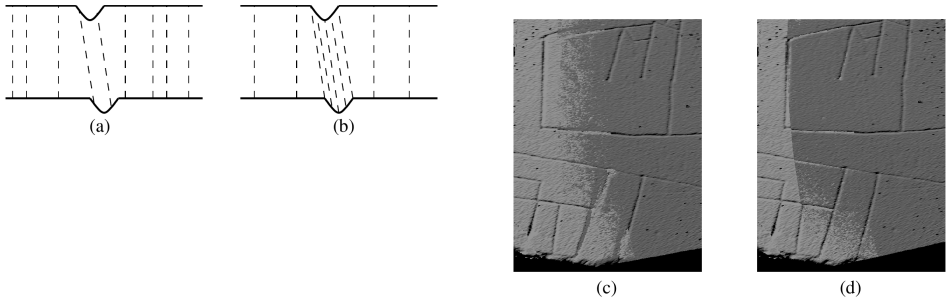


FIGURE 11.3: *The sample of points used in registration can be biased in useful ways. For example, (a) shows a cross section of a flat surface with a small groove (***above***) which needs to be registered to a similar surface (***below***). If point samples are drawn on the surface at random, then there will be few samples in the groove; the dashed lines indicate correspondences. In turn, the registration will be poor, because the surfaces can slide on one another. In (b), the samples have been drawn so that normal directions are evenly represented in the samples. Notice this means more samples concentrated in the groove, and fewer on the flat part. As a result, the surface is less free to slide, and the registration improves.*
TODO: what do c and d show? **TODO:** Source, Credit, Permission

could bias the estimate of the vehicle's pose. A better alternative would be to build a *stratified sample* by breaking the space around the vehicle into blocks of fixed size, then choosing uniformly at random a fixed number of samples in each block. In this scheme, the wall would be undersampled, and the open space would be oversampled, somewhat resolving the bias.

Another stratified sampling strategy is to ensure that surface normal directions are evenly represented in the samples. Make an estimate of a surface normal at each point (for example, by fitting a plane to the point and some of its nearest neighbors). Now break the unit sphere, which encodes the surface normals, into even cells, and sample the points so that each cell has the same number of samples. This approach is particularly useful when we are trying to register flat surfaces with small relief details on them (Figure **??**).

11.2.3 Beyond ICP

ICP minimizes a cost function

$$\sum_i \left[\min_j \| \mathbf{x}_j - \mathcal{T}(\mathbf{y}_i) \|_2^2 \right] = \sum_i E_i(\mathbf{T}) \tag{11.28}$$

by finding the corresponding pairs (the $\mathbf{x}_j$ that corresponds to $\mathbf{y}_i$), then minimizing, then repeating. This is an easy way to exploit the closed form solution for $\mathcal{T}$ when correspondence is known, but it isn't the only way. The min means the objective function isn't differentiable everywhere (exercises), but it is continuous, and it is differentiable at most locations. This is usually a sign that straightforward optimization methods can be applied successfully, which is true here. The

Levenberg-Marquardt algorithm (Section **??**) works particularly well here, because for a particular correspondence, the cost is a least squares cost, and because it doesn't require second derivatives. Notice that, to obtain the gradient of $E_i(\mathbf{T})$ with respect to $\mathbf{T}$, you need to know *which* $\mathbf{x}_j$ is closest to $\mathcal{T}(\mathbf{y}_i)$, so you still need to find the nearest neighbor.

11.2.4 Finding Nearest Neighbors

Finding the exact nearest neighbor of a query point in a large collection of reference points is more difficult than most people realize (one can beat linear search, but by only a very small factor []). However, finding a point that has high probability of being almost as close as the nearest neighbor (an *approximate nearest neighbor*) can be done rather fast using a variety of approximation schemes []. It is usual to substitute an approximate nearest neighbor, found using a k-d tree (eg []).

Resources: ICP **TODO:** *ICP Resources*

11.3 NOISE THAT ISN'T GAUSSIAN: ROBUSTNESS AND IRLS

In our examples, if we assume the noise is normal and isotropic, the squared error is reasonably described as negative log-likelihood. But in some cases, even when the measurements and the reference points are properly aligned, some measurement points may lie quite far from the closest reference point. One reason is pure error. Effects like scattering from rain or translucency can cause LIDAR or depth sensors to report measurements that are quite different from the actual geometry. Another is overhangs, which occur when either the reference or measured set contains points representing geometry that isn't in the other set. In this case, some points from one set should be far away from the closest point in the other set. Each of these effects (Figure 33.2) means that modelling noise as Gaussian may not be justified.

Large distances between some point pairs could have a significant effect on the estimate of the transformation. The square of a large number is very large indeed, so that reducing a large distance somewhat can justify incurring small to medium error on many other pairs (Figure **??**). A simple procedure to manage this effect is to ignore corresponding pairs if the distance between them is too large. One estimates the transformation using only pairs where distances are small. If points were omitted in one step of the iteration, they may return in another. This strategy can be helpful, but there is a danger that too many pairs are omitted and the iteration does not converge. Corresponding pairs with large distances between them are likely *outliers* – measurements or data that will not conform to a model, but can have significant impact on estimating the model. Well established procedures for handling outliers are easily adapted to registration problems.

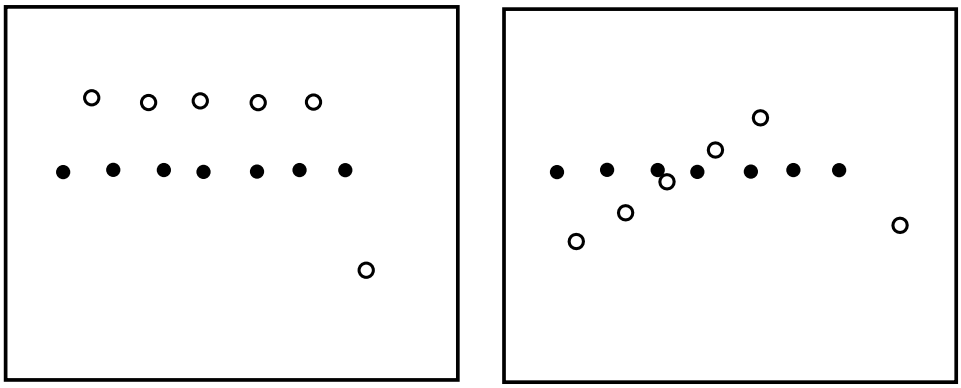


FIGURE 11.4: *Significant registration errors can be caused by just one point that is in the wrong place. On the* **left***, a set of empty points must be registered to a set of filled points. Notice that one empty point is badly out of place. An ideal registration would ignore this, and put the approximate line of empty points on the line of filled points. On the* **right***, the registration that actually results. The square of a large number is very large, meaning the minimum of the squared error isn't where you might think; reducing the large offset entirely justifies a set of medium sized errors. Points that lie significantly far from their "natural" positions are often known as* outliers.

11.3.1 IRLS: Weighting Down Outliers

Rather than just ignoring big distances, one might weight down correspondences that seem implausible. Doing so requires some way to estimate an appropriate set of weights. A large weight for errors at points that are "trustworthy" and a low weight for errors at "suspicious" points should result in a registration that is robust to outliers. We can obtain such weights using a *robust loss*, which will reduce the cost of large errors. This can be seen as modifying the probability model. Gaussian noise tends to produce few large values (which so have very large negative log-likelihood), and we want a model that has higher probability of large errors (equivalently, penalizes them less severely than a normal model would). Write θ for the parameters of the transformation, $\mathcal{T}_\theta$ for the transformation, and $r_i(\mathbf{x}_i, \mathbf{y}_{c(i)}, \theta)$ for the residual error of the model on the ith measurement and its corresponding reference point. For us, r_i will always be $l2norm\mathbf{x}_i - \mathcal{T}_\theta(\mathbf{y}_{c(i)})$. So rather than minimizing

$$\sum_i (r_i(\mathbf{x}_i, \mathbf{y}_{c(i)}, \theta))^2 \tag{11.29}$$

as a function of θ, we will minimize an expression of the form

$$\sum_i \rho(r_i(\mathbf{x}_i, \mathbf{y}_{c(i)}, \theta); \sigma), \tag{11.30}$$

for some appropriately chosen function ρ. Clearly, our negative log-likelihood is one such estimator (use $\rho(u; \sigma) = u^2$). The trick is to make $\rho(u; \sigma)$ look like u^2 for

FIGURE 11.5:
TODO: Figure showing a bunch of robust loss functions **TODO:** Source, Credit, Permission

smaller values of u, but ensure that it grows more slowly than u^2 for larger values of u.

The *Huber loss* uses

$$\rho(u;\sigma) = \begin{cases} \frac{u^2}{2} & |u| < \sigma \\ \sigma|u| - \frac{\sigma^2}{2} & \end{cases} \tag{11.31}$$

which is the same as u^2 for $-\sigma \leq u \leq \sigma$, switches to $|u|$ for larger (or smaller) σ, and has continuous derivative at the switch. The Huber loss is convex (meaning that there will be a unique minimum for our models) and differentiable, but is not smooth. The choice of the parameter σ (which is known as *scale*) has an effect on the estimate. You should interpret this parameter as the distance that a point can lie from the fitted function while still being seen as an *inlier* (anything that isn't even partially an outlier).

The *Pseudo Huber loss* uses

$$\rho(u;\sigma) = \sigma^2 \left(\sqrt{1 + \left(\frac{u}{\sigma}\right)} - 1 \right). \tag{11.32}$$

A little fiddling with Taylor series reveals this is approximately u^2 for $|u|/\sigma$ small, and linear for $|u|/\sigma$ big. This has the advantage of being differentiable.

The **** **TODO:** what is this loss called uses

$$\rho(u;\sigma) = \frac{\sigma^2 u^2}{u^2 + \sigma^2} \tag{11.33}$$

which is approximately u^2 for $|u|$ much smaller than σ, and close to σ^2 for $|u|$ much larger than σ.

Each of these losses increases monotonically in $|u|$ (the absolute value is important here!), so it is always better to reduce the residual. For the Huber loss and the Pseudo-Huber loss, the penalty grows with $|u|$, but grows more slowly with big $|u|$ than with small $|u|$. This implies that the underlying probability model will produce very large distances less often than large distances, but more often than a Gaussian model would. For the **** loss, the penalty eventually increases extremely slowly with increasing $|u|$, implying the underlying probability model is willing to produce arbitrarily large distances on occasion, and that the probability of large distances declines very slowly.

Our minimization criterion is

$$\begin{aligned} \nabla_\theta \left(\sum_i \rho(r(\mathbf{x}_i, \mathbf{y}_i, \theta); \sigma) \right) &= \sum_i \left[\frac{\partial \rho}{\partial u} \right] \nabla_\theta r(\mathbf{x}_i, \mathbf{y}_i, \theta) \\ &= 0. \end{aligned}$$

Here the derivative $\frac{\partial \rho}{\partial u}$ is evaluated at $r(\mathbf{x}_i, \mathbf{y}_i, \theta$, so it is a function of θ. Now notice that

$$\begin{aligned}\sum_i \left[\frac{\partial \rho}{\partial u}\right] \nabla_\theta r(\mathbf{x}_i, \mathbf{y}_i, \theta) &= \sum_i \left[\left(\frac{\frac{\partial \rho}{\partial u}}{r(\mathbf{x}_i, \mathbf{y}_i, \theta)}\right)\right] r(\mathbf{x}_i, \mathbf{y}_i, \theta) \nabla_\theta r(\mathbf{x}_i, \mathbf{y}_i, \theta) \\ &= \sum_i \left[\left(\frac{\frac{\partial \rho}{\partial u}}{r(\mathbf{x}_i, \mathbf{y}_i, \theta)}\right)\right] \nabla_\theta \left[r(\mathbf{x}_i, \mathbf{y}_i, \theta)\right]^2 \\ &= 0.\end{aligned}$$

Now $\left[r(\mathbf{x}_i, \mathbf{y}_i, \theta)\right]^2$ is the squared error. If we happened to know the true minimum $\hat{\theta}$ and wrote

$$w_i = \left(\frac{\frac{\partial \rho}{\partial u}}{r(\mathbf{x}_i, \mathbf{y}_i, \theta)}\right) \tag{11.34}$$

(evaluated at that minimum), then

$$\sum_i w_i \nabla_\theta \left[r(\mathbf{x}_i, \mathbf{y}_i, \theta)\right]^2 = 0 \tag{11.35}$$

at $\theta = \hat{\theta}$. We do not know w_i, but if we did, we already have a recipe to solve this problem for a variety of different transformations (Sections 33.2, 33.2 and 33.2). A natural strategy to adopt is to start with some transformation estimate and unit weights, then repeat:

- **Estimate correspondences** using the estimated transformation. Because all the robust losses are monotonic in $|u|$, finding the closest reference point to each measurement will do.

- **Re-estimate weights** using the new correspondences and the transformation.

- **Re-estimate transformation** using the new correspondences and the new weights, and the closed form algorithms from Sections 33.2, 33.2 and 33.2.

This procedure is known as *iteratively reweighted least squares*

Procedure: 11.4 *Estimating a Transformation from Data with a Robust Loss: Iteration*

Start with N known reference points $\mathbf{y}_i = (y_{i,1}, \ldots, y_{i,d})$ in affine coordinates and M measurements $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$, and a transformation estimate $\mathcal{T}_{\theta^{(1)}}$ with parameters θ^1. Form $\mathbf{m}_i^{(1)} = \mathcal{T}_{\theta^{(1)}}(\mathbf{y}_i)$, then iterate:

- for each $\mathbf{x}_i$, find $\mathbf{m}_{c(i)}^{(n+1)}$ that is closest;
- for each pair $(\mathbf{x}_i, \mathbf{m}_{c(i)}^{(n+1)})$, form $u_i = \| \mathbf{x}_i - \mathbf{m}_{c(i)}^{(n+1)} \|_2$ and

$$w_i = \left(\frac{\frac{\partial \rho}{\partial u}}{\mathbf{u}_i} \right); \tag{11.36}$$

- estimate $\mathcal{T}_{\theta^{(n+1)}}$ using the set of pairs $(\mathbf{x}_i, \mathbf{m}_{c(i)})$ and the weights w_i;
- form $\mathbf{m}_i = \mathcal{T}_{\theta^{(n+1)}}(\mathbf{m}_i)$.

Test for convergence by testing either that the correspondences did not change in a round, or by checking that $\mathcal{T}_{\theta^{(n+1)}}$ is close to the identity. The required transformation is $\mathcal{T}_{\theta^{(n+1)}} \circ \mathcal{T}_{\theta^{(n)}} \circ \ldots \circ \mathcal{T}_{\theta^{(1)}}$.

11.3.2 Starting IRLS

There are many local minima for the IRLS cost function (Figure 33.2 has one example). This means that IRLS likely won't work unless the $\mathbf{y}$ and the $\mathbf{x}$ are reasonably well aligned. One strong recipe for obtaining a start point involves repeating: obtain a small set of distinctive $\mathbf{x}$ which are spread out (call these $\mathbf{u}$); for each $\mathbf{u}_i$, choose at random from the s most likely corresponding $\mathbf{y}$ to get $\mathbf{v}_i$; compute a transformation $\mathcal{T}$ that takes $\mathbf{v}_i$ to $\mathbf{u}_i$ using least squares; apply this $\mathcal{T}$ and compute an error metric. We then use the $\mathcal{T}$ that obtains the best error metric seen in this randomized search to map $\mathbf{y}$ to $\mathbf{y}'$, and apply IRLS to the sets $\mathcal{X}$ and $\mathcal{Y}'$.

Making this recipe concrete requires identifying distinctive points and their likely correspondences as well as specifying the error metric.

Procedure: 11.5 *Estimating a Transformation from Data with a Robust Loss: Initialization*

Given N known reference points $\mathbf{y}_i = (y_{i,1}, \ldots, y_{i,d})$ in affine coordinates and M measurements $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$, initialize by: **TODO:** What is best? likely translation from cogs, affine / euclidean from second moments, but how do you compute second moments robustly?

11.3.3 Beyond IRLS: Line Processes

11.3.4 Registering Multiple Point Sets

PART FOUR

CLASSIFICATION, DETECTION, REGRESSION AND GENERATION

CHAPTER 12

Convolutional Image Classifiers

12.1 CONVOLUTIONAL LAYERS

12.1.1 Images as Patterns of Patterns of Patterns...

Images have important, quite general, properties (Figure 12.1). Images of "the same thing" — in this case, the same handwritten digit — can look fairly different. Small shifts and small rotations do not change the class of an image. Making the image somewhat brighter of somewhat darker does not change the class of the image either. Making the image somewhat larger, or making it somewhat smaller (then cropping or filling in pixels as required) does not change the class either. This means that individual pixel values are not particularly informative – you can't tell whether a digit image is, for example, a zero by looking at a given pixel, because the ink might slide to the left or to the right of the pixel without changing the digit.

I will use the MNIST dataset of handwritten digits as a source of examples in this chapter. This dataset is very widely used to check simple methods. It was originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. You can find this dataset in several places. The original dataset is at `http://yann.lecun.com/exdb/mnist/`. The version I used was prepared for a Kaggle competition (so I didn't have to decompress Lecun's original format). I found it at `http://www.kaggle.com/c/digit-recognizer`.

If you look at the images in Figure 12.2, you will notice another important property of images. Local patterns can be quite informative. Digits like 0 and 8 have loops. Digits like 4 and 8 have crossings. Digits like 1, 2, 3, 5 and 7 have line endings, but no loops or crossings. Digits like 6 and 9 have loops and line endings. Furthermore, spatial relations between local patterns are informative. A 1 has two line endings above one another; a 3 has three line endings above one another. These observations suggest a strategy that is a central tenet of modern computer vision: you construct features that respond to patterns in small, localized neighborhoods; then other features look at patterns of *those* features; then others look at patterns of those, and so on.

12.1.2 Stride and Padding

In the original operation, we used a window at every location in $\mathcal{I}$, but we may prefer to look at (say) a window at every second location. The centers of the windows we wish to look at lie on a grid of locations in $\mathcal{I}$. The number of pixels skipped between points on the grid is known as its *stride*. A grid with stride 1 consists of each spatial location. A grid with stride 2 consists of every second spatial location in $\mathcal{I}$, and so on. You can interpret a stride of 2 as either performing `conv` then keeping the value at every second pixel in each direction. Better is to think of the kernel striding across the image — perform the `conv` operation as above, but now

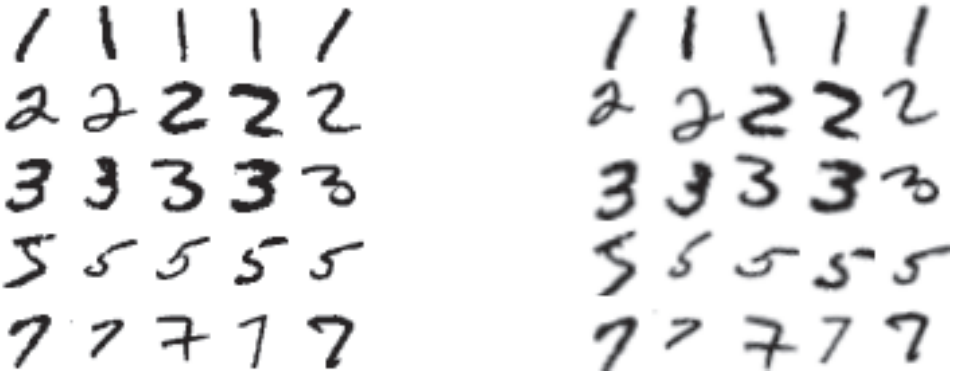


FIGURE 12.1: *On the* **left**, *a selection of digits from the MNIST dataset. Notice how images of the same digit can vary, which makes classifying the image demanding. It is quite usual that pictures of "the same thing" look quite different. On the* **right**, *digit images from MNIST that have been somewhat rotated and somewhat scaled, then cropped fit the standard size. Small rotations, small scales, and cropping really doesn't affect the identity of the digit.*

move the window by two pixels before multiplying and adding.

The description of the original operation avoided saying what would happen if the window at a location went outside $\mathcal{I}$. We adopt the convention that $\mathcal{N}$ contains entries only for windows that lie inside $\mathcal{I}$. But we can apply *padding* to $\mathcal{I}$ to ensure that $\mathcal{N}$ has the size we want. Padding attaches a set of rows (resp. columns) to the top and bottom (resp. left and right) of $\mathcal{I}$ to make it a convenient size. Usually, but not always, the new rows or columns contain zeros. By far the most common case uses $\mathcal{M}$ that are square with odd dimension (making it much easier to talk about the center). Assume $\mathcal{I}$ is $n_x \times n_y$ and $\mathcal{M}$ is $(2k+1) \times (2k+1)$; if we pad $\mathcal{I}$ with k rows on top and bottom and k columns on each side, $\texttt{conv}(\mathcal{I}, \mathcal{M})$ will be $n_x \times n_y$.

12.1.3 Convolutional Layers to Make Feature Maps

Images are naturally 3D objects with two spatial dimensions (up-down, left-right) and a third dimension that chooses a slice (R, G or B for a color image). This structure is natural for representations of image patterns, too — two dimensions that tell you where the pattern is and one that tells you what it is. The results in Figure 3.2 show a block consisting of three such slices. These slices are the response of a pattern detector *for a fixed pattern*, where there is one response for each spatial location in the block, and so are often called *feature maps*.

We will generalize `conv` and apply it to 3D blocks of data (which I will call *blocks*). Write $\mathcal{I}$ for an input block of data, which is now $x \times y \times d$. Two dimensions – usually the first two, but this can depend on your software environment – are spatial and the third chooses a slice. Write $\mathcal{M}$ for a 3D kernel, which is $k_x \times k_y \times d$. Now choose padding and a stride. This determines a grid of locations in the spatial dimensions of $\mathcal{I}$. At each location, we must compute the value of $\mathcal{N}$. To do so, take

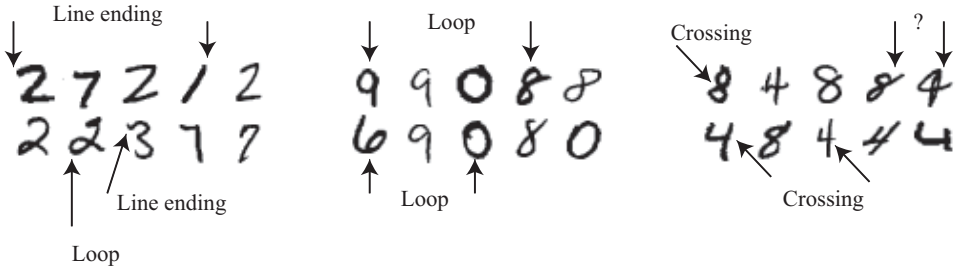


FIGURE 12.2: *Local patterns in images are quite informative. MNIST images, shown here, are simple images, so a small set of patterns is quite helpful. The relative location of patterns is also informative. So, for example, an eight has two loops, one above the other. All this suggests a key strategy: construct features that respond to patterns in small, localized neighborhoods; then other features that look at patterns of* those *features; then others that look at patterns of those, and so on. Each pattern (here line-endings, crossings and loops) has a range of appearances. For example, a line ending sometimes has a little wiggle as in the three. Loops can be big and open, or quite squashed. The list of patterns isn't comprehensive. The "?" shows patterns that I haven't named, but which appear to be useful. In turn, this suggests learning the patterns (and patterns of patterns; and so on) that are most useful for classification.*

the 3D window $\mathcal{W}$ of $\mathcal{I}$ at that location that is the same size as $\mathcal{N}$; you multiply together the elements of $\mathcal{M}$ and $\mathcal{W}$ that lie on top of one another; and you sum the results (Figure 3.1). This sum now goes over the third dimension as well. This produces a two dimensional $\mathcal{N}$.

TODO: Notation for data blocks; also, work in batches

To make this operation produce a block of data, use a 4D block of kernels. This *kernel block* consists of D kernels, each of which is a $k_x \times k_y \times d$ dimensional kernel. If you apply each kernel as in the previous paragraph to an $x \times y \times d$ dimensional $\mathcal{I}$, you obtain an $X \times Y \times D$ dimensional block $\mathcal{N}$, as in Figure 12.4. What X and Y are depends on k_x, k_y, the stride and the padding. A *convolutional layer* takes a kernel block and a bias vector of D bias terms. The layer applies the kernel block to an input block (as above), then adds the corresponding bias value to each slice.

Definition: 12.1 *Convolutional Layer*

A convolutional layer makes 3D blocks of data from 3D blocks of data, using a stride, padding, a block of kernels and a vector of bias terms. The details are in the text.

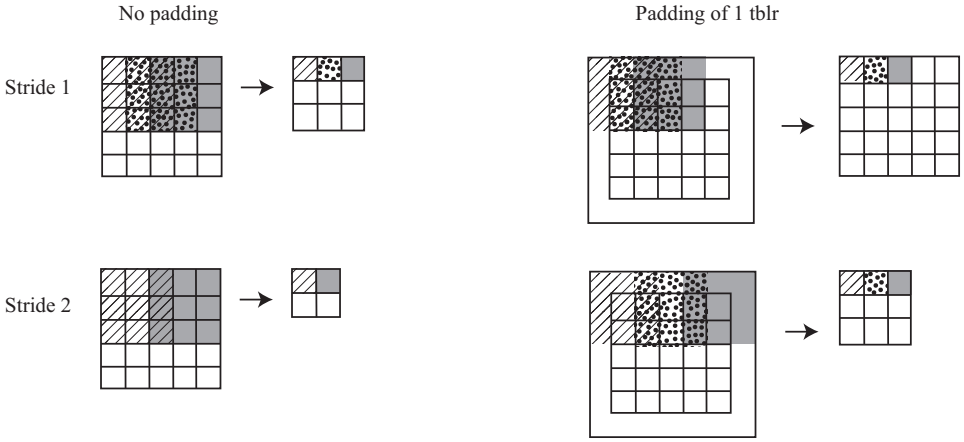


FIGURE 12.3: *The effects of stride and padding on* `conv`. *On the* **left**, `conv` *without padding accepts an* $\mathcal{I}$, *places a* 3×3 $\mathcal{M}$ *on grid locations determined by the stride, then reports values for valid windows. When the stride is* 1, *a* 5×5 $\mathcal{I}$ *becomes a* 3×3 $\mathcal{N}$. *When the stride is* 2, *a* 5×5 $\mathcal{I}$ *becomes a* 2×2 $\mathcal{N}$. *The hatching and shading show the window used to compute the corresponding value in* $\mathcal{N}$. *On the* **right**, `conv` *with padding accepts an* $\mathcal{I}$, *pads it (in this case, by one row top and bottom, and one column left and right), places a* 3×3 $\mathcal{M}$ *on grid locations in the padded result determined by the stride, then reports values for valid windows. When the stride is* 1, *a* 5×5 $\mathcal{I}$ *becomes a* 5×5 $\mathcal{N}$. *When the stride is* 2, *a* 5×5 $\mathcal{I}$ *becomes a* 3×3 $\mathcal{N}$. *The hatching and shading show the window used to compute the corresponding value in* $\mathcal{N}$.

Remember this: *A fully connected layer can be thought of as a convolutional layer followed by a ReLU layer. Assume you have an* $x \times y \times d$ *block of data. Reshape this to be a* $(xyd) \times 1 \times 1$ *block. Apply a convolutional layer whose kernel block has size* $(xyd) \times 1 \times D$, *and then a ReLU. This pair of layers produces the same result as a fully-connected layer of* D *units.*

.

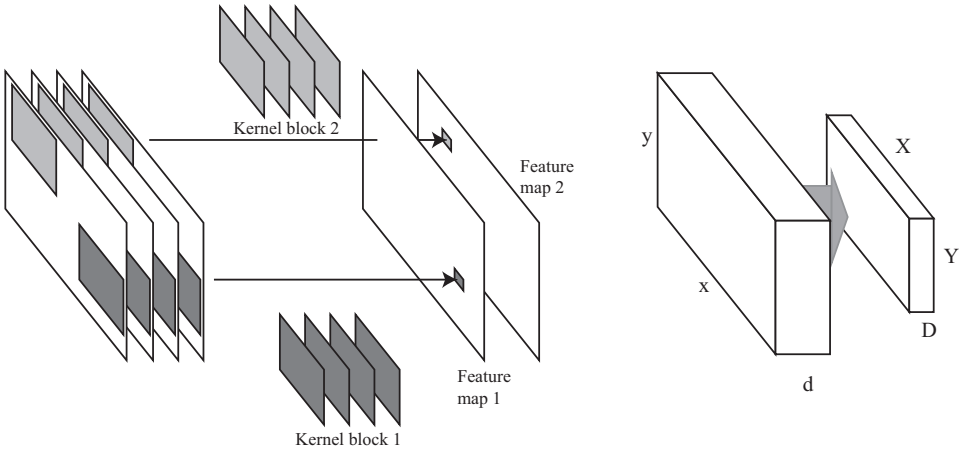


FIGURE 12.4: *On the* **left**, *two kernels (now 3D, as in the text) applied to a set of feature maps produce one new feature map per kernel, using the procedure of the text (the bias term isn't shown). Abstract this as a process that takes an* $x \times y \times d$ *block to an* $X \times Y \times D$ *block (as on the* **right**).

Remember this: *Take the output of a convolutional layer and apply a ReLU. First, think about what happens to one particular piece of image the size of one particular kernel. If that piece is "sufficiently similar" to the kernel, we will see a positive response at the relevant location. If the piece is too different, we will see a zero. This is a pattern detector as in Figure 3.2. What "sufficiently similar" means is tuned by changing the bias for that kernel. For example, a bias term that is negative with large magnitude means the image block will need to be very like the kernel to get a non-zero response. This pattern detector is (basically) a unit – apply a ReLU to a linear function of the piece of image, plus a constant. Now it should be clear what happens when all kernels are applied to the whole image. Each pixel in a slice represents the result of a pattern detector applied to the piece of image corresponding to the pixel. Each slice of the resulting block represents the result of a different pattern detector. The elements of the output block are often thought of as* features.

Remember this: *There isn't a standard meaning for the term convolutional layer. I'm using one of the two that are widely used. Software implementations tend to use my definition. Very often, research papers use the alternative, which is my definition followed by a non-linearity (almost always a ReLU). This is because convolutional layers mostly are followed by ReLU's in research papers, but it is more efficient in software implementations to separate the two.*

Different software packages use different defaults about padding. One default assumes that no padding is applied. This means that a kernel block of size $k_x \times k_y \times d \times D$ applied to a block of size $x \times y \times d$ with stride 1 yields a block of size $(n_x - k_x + 1) \times (n_y - k_y + 1) \times D$ (check this with a pencil and paper). Another assumes that the input block is padded with zeros so that the output block is $n_x \times n_y \times D$.

Remember this: *In Figure 3.2, most values in the output block are zero (black pixels in that figure). This is typical of pattern detectors produced in this way. This is an experimental fact that seems to be related to deep properties of images.*

Remember this: *A kernel block that is $1 \times 1 \times n_z \times D$ is known as a 1×1* convolution. *This is a linear map in an interesting way. Think of the input and output blocks as sets of column vectors. So the input block is a set of $n_x \times n_y$ column vectors, each of which has dimension $n_z \times 1$ (i.e. there is a column vector at each location of the input block). Write $\mathbf{i}_{uv}$ for the vector at location u, v in the input block, and $\mathbf{o}_{uv}$ for the vector at location u, v in the output block. Then there is a $D \times n_z$ matrix $\mathcal{M}$ so that the 1×1 convolution maps $\mathbf{i}_{uv}$ to*

$$\mathbf{o}_{uv} = \mathcal{M}\mathbf{i}_{uv}.$$

This can be extremely useful when the input has very high dimension, because $\mathcal{M}$ can be used to reduce dimension and *is learned from data.*

12.2 SIMPLE IMAGE CLASSIFICATION WITH MULTI-LAYER NETWORKS

TODO: We have a logistic regression on a feature stack

12.2.1 Convolutional Layer upon Convolutional Layer

Convolutional layers take blocks of data and make blocks of data, as do ReLU layers. This suggests the output of a convolutional layer could be passed through a ReLU, then connected to another convolutional layer, and so on. Doing this turns out to be an excellent idea.

Think about the output of the first convolutional layer. Each location receives inputs from pixels in a window about that location. The output of the ReLU, as we have seen, forms a simple pattern detector. Now if we put a second layer on top of this, each location in the second layer receives inputs from first layer values in a window about that location. This means that locations in the second layer are affected by a larger window of pixels than those in the first layer. You should think of these as representing "patterns of patterns". If we place a third layer on top of the second layer, locations in that third layer will depend on an even larger window of pixels. A fourth layer will depend on a yet larger window, and so on. The key point here is that we can choose the patterns that are detected by *learning* what kernels will be applied at each layer.

The window of pixels in the original image that is used to compute the value at some location in a data block is referred to as its *receptive field*. Usually, all that matters is the size of the receptive field, which will be the same for every location in a given block if we ignore the boundary of the input image. The receptive field of a location in the first convolutional layer will be given by the kernel of that layer. Determining the receptive field for later layers requires some bookkeeping (among other things, you must account for any stride or pooling effects).

12.2.2 Pooling

Now imagine we pass a data block through a set of layers as in Figure 33.2; its spatial dimension will not change. This tends to be a problem, because the receptive field of a location in the top layer will tend to have a large overlap with the receptive field for a location next to it. In turn, the values that the units take will be similar, and so there will be redundant information in the output block. It is usual to try and deal with this by making blocks get smaller. One strategy is to occasionally have a layer that has stride 2.

An alternative strategy is to use *pooling*. A pooling unit reports a summary of its inputs. In the most usual arrangement, a pooling layer halves each spatial dimension of a block. For the moment, ignore the entirely minor problems presented by a fractional dimension. The new block is obtained by pooling units that pool a window at each feature map of the input block to form each feature map of the output block. If these units pool a 2×2 window with stride 2 (ie they don't overlap), the output block is half the size of the input block. We adopt the convention that the output reports only valid input windows, so that this takes an $x \times y \times d$ block to an $\text{floor}(x/2) \times \text{floor}(y/2) \times d$ block. So, as Figure 12.5 shows, a $5 \times 5 \times 1$ block becomes a $2 \times 2 \times 1$ block, but one row and one column are ignored. A common alternative is pooling a 3×3 window with a stride of 2; in this case, a $5 \times 5 \times 1$ block becomes a $2 \times 2 \times 1$ block without ignoring rows or columns. Each unit reports either the largest of the inputs (yielding a *max pooling* layer) or the average of its inputs (yielding an *average pooling* layer).

FIGURE 12.5: *In a pooling layer, pooling units compute a summary of their inputs, then pass it on. The most common case is 2x2, illustrated here on the* **left**. *We tile each feature map with 2x2 windows that do not overlap (so have stride 2). Pooling units compute a summary of the inputs (usually either the max or the average), then pass that on to the corresponding location in the corresponding feature map of the output block. As a result, the spatial dimensions of the output block will be about half those of the input block. On the* **right**, *the common alternative of pooling in overlapping 3x3 windows with stride 2.*

12.2.3 CIFAR-10: an Example Dataset

CIFAR-10 is a dataset of 32×32 color images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is often used to evaluate image classification algorithms. There are 50, 000 training images and 10, 000 test images, and the test-train split is standard. Images are evenly split between the classes. Figure 12.6 shows the categories, and examples from each category. There is no overlap between the categories (so "automobile" consists of sedans, etc. and "truck" consists of big trucks). You can download this dataset from `https://www.cs.toronto.edu/kriz/cifar.html`.

12.2.4 Simple Convolutional Image Classifiers

We now know constraints a simple image classifier should meet, and can sketch a structure. A set of layers accepts an image and produces a feature vector that is passed to multiclass logistic regression (equivalently, a fully connected layer followed by a softmax layer). Experience teaches the feature vector should be high dimensional, but not unmanageable (dimension 1024 is quite commonly used). The image which goes into the set of layers is large spatially, but has few feature dimensions – one for a grey level image, three for a color image. As the data block passes up the network, we expect each location will have a larger receptive field. In turn, because there are more different interesting big patterns than there are small ones, we expect the feature dimension to grow. Finally, we expect that the spatial dimension of the data block should shrink, because if it does not, it will contain a great deal of redundant information. Figure 12.10 shows three different representations of one simple (and not very good) classification architecture. Here the feature vector is 64 dimensional. Notice how the data blocks shrink spatially and grow in feature dimension as they move toward the final classifier. Notice also

FIGURE 12.6: *The CIFAR-10 image classification dataset consists of 60, 000 images, in a total of 10 categories. The images are all 32x32 color images. This figure shows 20 images from each of the 10 categories and the labels of each category. On the* **far right**, *the mean of the images in each category. I have doubled the brightness of the means, so you can resolve color differences. The per-category means are different, and suggest that some classes look like a blob on a background, and others (eg ship, truck) more like an outdoor scene.*

how the size of the receptive field of the blocks grows – this network is finding patterns of patterns of patterns (etc.), then classifying using the presence of particular patterns.

The architecture of Figure 12.10 is not particularly effective (Figure **??** shows what happens when it is trained on CIFAR-10). Part of the difficulty is that the features are not sufficiently distinctive. Equivalently, the patterns the network is trained to look for are not complicated enough. Seeing a convolutional layer followed by a ReLU as a pattern detector suggests a solution: have more layers (in jargon, a deeper network). But stacks of convolutional and pooling layers become difficult to train as they get deeper.

The problem is that the features are now extremely complicated functions of the original image and the network weights. When training starts, the weights will mostly be wrong. Now look at the expressions for the gradient in Section 27. All those jacobians mean that the gradient of the loss with respect to parameters in an early layer depends very strongly on the parameter values in later layers – but in the early stages of training, these will be wrong, meaning the gradient may not be particularly helpful. As the number of layers increases, this effect becomes more pronounced, and deeper networks become very hard to train. Another way to think about this problem is to consider some layer – after each training step, the distribution of inputs to that layer will change because the earlier layer weights have changed. This can make it difficult to choose a useful set of weights for that layer. The effect will become more pronounced as there are more earlier layers, making a deeper network very hard to train.

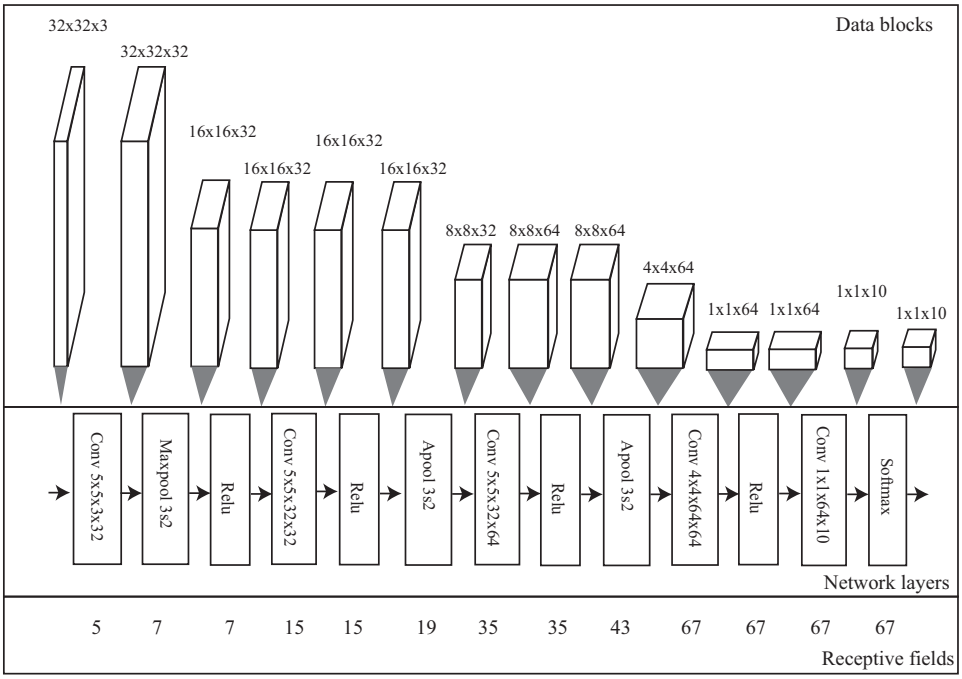


FIGURE 12.7: *Three different representations of a simple (and not very good) classification architecture. The top row shows the size of each data block leaving each layer; the center row shows what the layers are; and the bottom row shows the size of the receptive field. In this network, the final feature vector is 64 dimensional. Notice how the data blocks shrink spatially and grow in feature dimension as they move toward the final classifier. Notice also how the size of the receptive field of the blocks grows – this network is finding patterns of patterns of patterns (etc.), then classifying using the presence of particular patterns. Although the input is only 32x32, padding means that a location can have a receptive field considerably bigger than 32x32.*

12.2.5 Batch Normalization

One important trick for training deeper networks is *batch normalization*. A batch normalization layer attempts to control the distribution of its outputs. There are two stages. First, the normalizing stage adjusts the feature vector at each location in the data block by subtracting an offset and then scaling it. The offset and scales are chosen so that all feature vectors produced by the normalization block in the data set should have zero mean and unit standard deviation (which takes some work, below). The offset and the scale are properties of the data set, not learned parameters. But this scale and offset could mean that the network does not perform as well as it should. The correction stage compensates for the offset and scale by offsetting and rescaling the features with learned parameters, so that – in principle – the layer could just pass on its inputs.

Write μ_i for the offset and σ_i for the scale applied to the i'th feature map

by the normalizing stage (and assume for the moment we know them). Then the normalizing stage forms $\mathcal{N}$ from $\mathcal{X}$, where

$$n_{b,u,v,i} = (x_{b,u,v,i} - \mu_i)/\sigma_i.$$

Now the correction stage computes $\mathcal{O}$, where

$$o_{b,u,v,i} = (n_{b,u,v,i} - m_i)/s_i$$

and m_i, s_i are learned parameters. We would like the mean over the dataset of $n_{b,u,v,i}$ to be zero, and the standard deviation to be one. In principle, we could obtain the μ and σ by stopping training; passing the entire dataset through the network; and computing means and standard deviations. Doing so would be very slow and likely not very helpful. Instead, we could obtain a reasonable estimate by observing the mean and standard deviations of the features for the previous batch. Although the network will have changed because of the descent step, the change will be small. Furthermore, an average over a batch should be a good estimate of the average over the whole dataset. These parameters are often referred to as *running means*.

Batch normalization comes with software overheads, but current environments will deal with the details for you. There is one important issue. At test time, the network should not obtain running means from the previous batch, but use a fixed value obtained during training. This is usually handled with an instruction that tells the environment that the net is being evaluated rather than trained.

Batch normalization is widely believed to be helpful in training very deep networks, but it is less certain *why* it is helpful. There is some evidence that the explanation I used to introduce the idea doesn't actually explain why it batch normalization is helpful [], and a range of other explanations have been offered []. For our purposes, it is enough to know that it's helpful.

12.2.6 Residual Layers

A *residual layer* is a straightforward and very effective way to control this problem. Figure 33.2 illustrates a residual layer. Write $\mathbf{x}_i$ for the input block. We then form

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathcal{F}(\mathbf{x}_i; \theta_i)$$

where $\mathcal{F}(\cdot; \theta_i)$ is a *residual function*, with parameters θ_i. Current best practice uses the sequence of batch normalization, convolution and ReLU shown in Figure 33.2. For the moment, assume that $\mathcal{F}$ always produces an output block the same size as the input block. Now consider a stack of two residual layers applied to an input $\mathbf{x}_1$: we will obtain

$$\begin{aligned} \mathbf{x}_3 &= \mathbf{x}_2 + \mathcal{F}(\mathbf{x}_2; \theta_2) \\ &= \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_1; \theta_1) + \mathcal{F}(\mathbf{x}_1 + \mathcal{F}(\mathbf{x}_1; \theta_1); \theta_2), \end{aligned}$$

so that $\mathbf{x}_3$ depends in part on the input, in part directly on θ_1, and in part on θ_1 through a residual layer with parameters θ_2. In turn, this means that

$$\mathcal{J}_{\mathbf{x}_3;\theta_1} = \mathcal{J}_{\mathcal{F};\theta_1} + \mathcal{J}_{\mathcal{F};\theta_2}\mathcal{J}_{\mathcal{F};\theta_1}$$

so that having a poor estimate of θ_2 will not completely mangle the gradient of loss with respect to θ_1. Another way to think about this effect is to notice that some signal goes directly from the input to $\mathbf{x}_2$; some goes from input to $\mathbf{x}_2$ via the first residual layer ($\mathcal{F}(\cdot;\theta_1)$); and some goes through two residual layers.

Some minor difficulties occur if the convolutional layers in $\mathcal{F}$ change the number of features or the spatial dimension of the data block $\mathbf{x}$. Typically, if $\mathcal{F}$ has more features than $\mathbf{x}$, we project off the extra features. If $\mathcal{F}$ has fewer features than $\mathbf{x}$, we add the residual block to a subblock of $\mathbf{x}$ of the same size. Finally, if one is smaller than the other spatially, we can up or downsample as appropriate.

12.3 IMPROVING TRAINING

12.3.1 Augmentation and Ensembles

Three important practical issues that need to be addressed to build very strong image classifiers.

- **Data sparsity:** Datasets of images are never big enough to show all effects accurately. This is because an image of a horse is still an image of a horse even if it has been through a small rotation, or has been resized to be a bit bigger or smaller, or has been cropped differently, and so on. There is no way to take account of these effects in the architecture of the network.

- **Data compliance:** We want each image fed into the network to be the same size.

- **Network variance:** The network we have is never the best network; training started at a random set of parameters, and has a strong component of randomness in it. For example, most minibatch selection algorithms select random minibatches. Training the same architecture on the same dataset twice will not yield the same network.

All three can be addressed by some care with training and test data.

Generally, the way to address data sparsity is *data augmentation*, by expanding the training dataset to include different rotations, scalings, and crops of images. Doing so is relatively straightforward. You take each training image, and generate a collection of extra training images from it. You can obtain this collection by: resizing and then cropping the training image; using different crops of the same training image (assuming that training images are a little bigger than the size of image you will work with); rotating the training image by a small amount, resizing and cropping; and so on.

There are some cautions. When you rotate then crop, you need to be sure that no "unknown" pixels find their way into the final crop. You can't crop too much, because you need to ensure that the modified images are still of the relevant class, and too aggressive a crop might cut out the horse (or whatever) entirely. This somewhat depends on the dataset. If each image consists of a tiny object on a large background, and the objects are widely scattered, crops need to be cautious; but if the object covers a large fraction of the image, the cropping can be quite aggressive.

Cropping is usually the right way to ensure that each image has the same size. Resizing images might cause some to stretch or squash, if they have the wrong aspect ratio. This likely isn't a great idea, because it will cause objects to stretch or squash, making them harder to recognize. It is usual to resize images to a convenient size without changing the aspect ratio, then crop to a fixed size.

There are two ways to think about network variance (at least!). If the network you train isn't the best network (because it can't be), then it's very likely that training multiple networks and combining the results in some way will improve classification. You could combine results by, for example, voting. Small improvements can be obtained reliably like this, but the strategy is often deprecated because it isn't particularly elegant or efficient. A more usual approach is to realize that the network might very well handle one crop of a test image rather better than others (because it isn't the best network, etc.). Small improvements in performance can be obtained very reliably by presenting multiple crops of a test image to a given network, and combining the results for those crops.

12.3.2 Advanced Tricks: Gradient Scaling

Everyone is surprised the first time they learn that the best direction to travel in when you want to minimize a function is not, in fact, backwards down the gradient. The gradient *is* uphill, but repeated downhill steps are often not particularly efficient. An example can help, and we will look at this point several ways because different people have different ways of understanding this point.

We can look at the problem with algebra. Consider $f(x, y) = (1/2)(\epsilon x^2 + y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $(\epsilon x, y)$. For simplicity, use a fixed learning rate η, so we have

$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1 - \epsilon\eta)x^{(r-1)} \\ (1 - \eta)y^{(r-1)} \end{bmatrix}.$$

If you start at, say, $(x^{(0)}, y^{(0)})$ and repeatedly go downhill along the gradient, you will travel very slowly to your destination. You can show that

$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1 - \epsilon\eta)^r x^{(0)} \\ (1 - \eta)^r y^{(0)} \end{bmatrix}.$$

The problem is that the gradient in y is quite large (so y must change quickly) and the gradient in x is small (so x changes slowly). In turn, for steps in y to converge we must have $|1 - \eta| < 1$; but for steps in x to converge, we require only the much weaker constraint $|1 - \epsilon\eta| < 1$. Imagine we choose the largest η we dare for the y constraint. The y value will very quickly have small magnitude, though its sign will change with each step. But the x steps will move you closer to the right spot only extremely slowly.

Another way to see this problem is to reason geometrically. Figure 12.8 shows this effect for this function. The gradient is at right angles to the level curves of the function. But when the level curves form a narrow valley, the gradient points across the valley rather than down it. The effect isn't changed by rotating and translating the function (Figure 12.9).

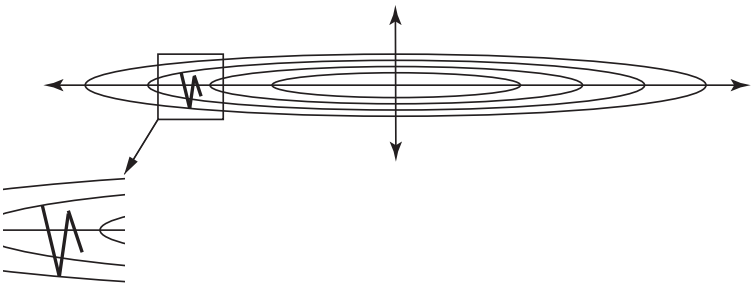


FIGURE 12.8: *A plot of the level curves (curves of constant value) of the function* $f(x, y) = (1/2)(\epsilon x^2 + y^2)$. *Notice that the value changes slowly with large changes in* x, *and quickly with small changes in* y. *The gradient points mostly toward the* x*-axis; this means that gradient descent is a slow zig-zag across the "valley" of the function, as illustrated. We might be able to fix this problem by changing coordinates,* if *we knew what change of coordinates to use.*

You may have learned that Newton's method resolves this problem. This is all very well, but to apply Newton's method we would need to know the matrix of second partial derivatives. A network can easily have thousands to millions of parameters, and we simply can't form, store, or work with matrices of these dimensions. Instead, we will need to think more qualitatively about what is causing trouble.

One useful insight into the problem is that fast changes in the gradient vector are worrying. For example, consider $f(x) = (1/2)(x^2 + y^2)$. Imagine you start far away from the origin. The gradient won't change much along reasonably sized steps. But now imagine yourself on one side of a valley like the function $f(x) = (1/2)(x^2 + \epsilon y^2)$; as you move along the gradient, the gradient in the x direction gets smaller very quickly, then points back in the direction you came from. You are not justified in taking a large step in this direction, because if you do you will end up at a point with a very different gradient. Similarly, the gradient in the y direction is small, and stays small for quite large changes in y value. You would like to take a small step in the x direction and a large step in the y direction.

You can see that this is the impact of the second derivative of the function (which is what Newton's method is all about). But we can't do Newton's method. We would like to travel further in directions where the gradient doesn't change much, and less far in directions where it changes a lot. There are several methods for doing so.

Momentum: We should like to discourage parameters from "zig-zagging" as in the example above. In these examples, the problem is caused by components of the gradient changing sign from step to step. It is natural to try and smooth the gradient. We could do so by forming a moving average of the gradient. Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive

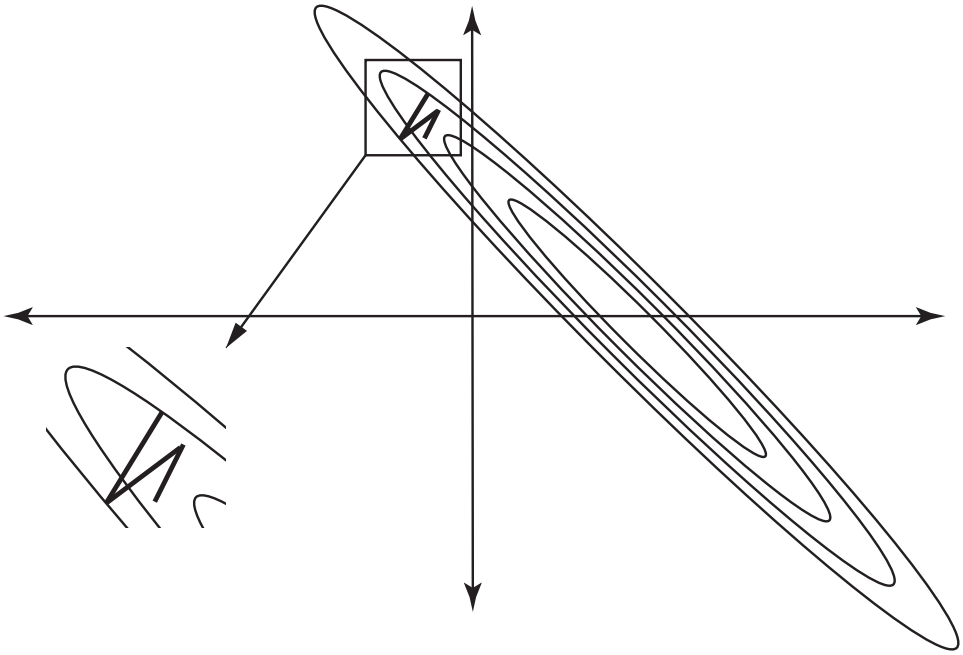


FIGURE 12.9: *Rotating and translating a function rotates and translates the gradient; this is a picture of the function of figure 12.8, but now rotated and translated. The problem of zig-zagging remains. This is important, because it means that we may have serious difficulty choosing a good change of coordinates.*

number $\mu < 1$. Then we iterate

$$\begin{aligned} \mathbf{v}^{(r+1)} &= \mu\mathbf{v}^{(r)} + \eta\nabla_\theta E \\ \theta^{(r+1)} &= \theta^{(r)} - \mathbf{v}^{(r+1)} \end{aligned}$$

Notice that, in this case, the update is an average of all past gradients, each weighted by a power of μ. If μ is small, then only relatively recent gradients will participate in the average, and there will be less smoothing. Larger μ lead to more smoothing. A typical value is $\mu = 0.9$. It is reasonable to make the learning rate go down with epoch when you use momentum, but keep in mind that a very large μ will mean you need to take several steps before the effect of a change in learning rate shows.

Adagrad: We will keep track of the size of each component of the gradient. In particular, we have a running cache $\mathbf{c}$ which is initialized at zero. We choose a small number α (typically 1e-6), and a fixed η. Write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration.Then we iterate

$$\begin{aligned} c_i^{(r+1)} &= c_i^{(r)} + (g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta\frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Notice that each component of the gradient has its own learning rate, set by the history of previous gradients.

RMSprop: This is a modification of Adagrad, to allow it to "forget" large gradients that occurred far in the past. Again, write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. We choose another number, Δ, (the *decay rate*; typical values might be 0.9, 0.99 or 0.999), and iterate

$$\begin{aligned} c_i^{(r+1)} &= \Delta c_i^{(r)} + (1-\Delta)(g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Adam: This is a modification of momentum that rescales gradients, tries to forget large gradients, and adjusts early gradient estimates to correct for bias. Again, write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. We choose three numbers β_1, β_2 and ϵ (typical values are 0.9, 0.999 and 1e-8, respectively), and some stepsize or learning rate η. We then iterate

$$\begin{aligned} \mathbf{v}^{(r+1)} &= \beta_1 * \mathbf{v}^{(r)} + (1-\beta_1) * \nabla_\theta E \\ c_i^{(r+1)} &= \beta_2 * c_i^{(r)} + (1-\beta_2) * (g_i^r)^2 \\ \hat{\mathbf{v}} &= \frac{\mathbf{v}^{(r+1)}}{1-\beta_1^t} \\ \hat{c}_i &= \frac{\hat{c}_i^{(r+1)}}{1-\beta_2^t} \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{\hat{v}_i}{\sqrt{\hat{c}_i} + \epsilon} \end{aligned}$$

Remember this: *If you are not getting improvements during training, use a gradient scaling trick.*

12.4 A PRACTICAL IMAGE CLASSIFIER FOR CIFAR-10

TODO: Do this with a 101 layer resnet? and rebore

We can now put together image classifiers using the following rough architecture. A convolutional layer receives image pixel values as input. The output is fed to a stack of convolutional layers, each feeding the next, possibly with ReLU layers intervening. There are occasional max-pooling layers, or convolutional layers with stride 2, to ensure that the data block gets smaller and the receptive field gets bigger as the data moves through the network. The output of the final layer is fed to one or more fully connected layers, with one output per class. Softmax takes these outputs and turns them into class-probabilities. The whole is trained by batch gradient descent, or a variant, as above, using a log-loss.

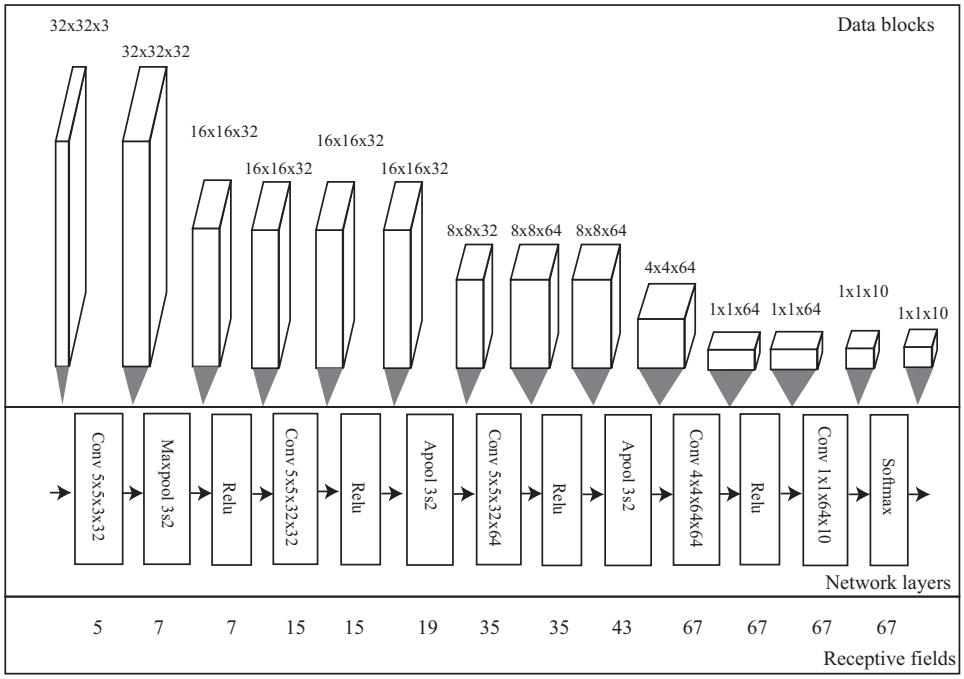


FIGURE 12.10: *Three different representations of the simple network used to classify CIFAR-10 images for this example. Details in the text.*

Notice that different image classification networks differ by relatively straightforward changes in architectural parameters. Mostly, the same thing will happen to these networks (variants of batch gradient descent on a variety of costs; dropout; evaluation). In turn, this means that we should use some form of specification language to put together a description of the architecture of interest. Ideally, in such an environment, we describe the network architecture, choose an optimization algorithm, and choose some parameters (dropout probability, etc.). Then the environment assembles the net, trains it (ideally, producing log files we can look at) and runs an evaluation. The tutorials mentioned in section **??** each contain examples of image classifiers for the relevant environments. In this example, I used Matconvnet, because I am most familiar with Matlab.

Figure 12.10 shows the network used to classify CIFAR-10 images. This network is again a standard classification network for CIFAR-10, distributed with Matconvnet. Again, I have shown the network in three different representations. The network layer representation, in the center of the figure, records the type of each layer and the size of the relevant convolution kernels. The first layer accepts the image which is a $32 \times 32 \times 3$ block of data (the data block representation), and applies a convolution.

In this network, the convolution *was* padded so that the resulting data block was $32 \times 32 \times 32$. You should check you agree with these figures, and you can tell by how much the image needed to be padded to achieve this (a drawing might help). A

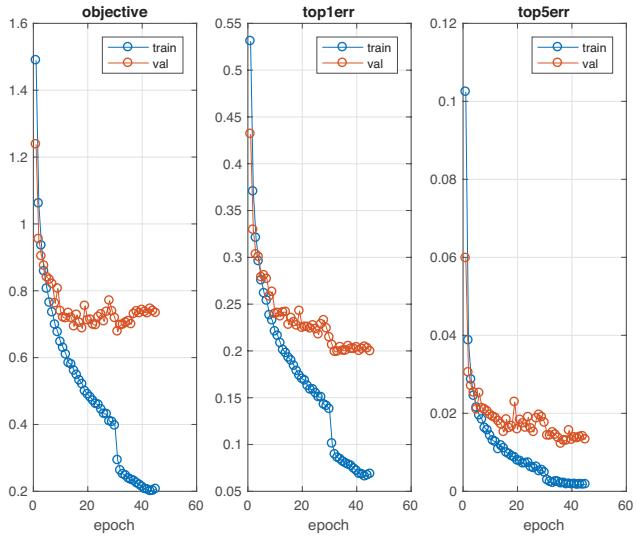


FIGURE 12.11: *This figure shows the results of training the network of Figure 12.10 on the CIFAR-10 training set. Loss, top-1 error and top-5 error for training and validation sets, plotted as a function of epoch for the network of the text. The loss (recorded here as "objective") is the log-loss. Note: the low validation error; the gap between train and validation error; and the very low top-5 error. The validation error is actually quite high for this dataset — you can find a league table at* `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`.

value in this data block is computed from a 5×5 window of pixels, so the receptive field is 5×5. Again, by convention, every convolutional layer has a bias term, so the total number of parameters in the first layer is $(5 \times 5 \times 3) \times 32 + 32$. The next layer is a 3×3 max pooling layer. The notation $3s2$ means that the pooling blocks have a stride of 2, so they overlap. The block is padded for this pooling layer, by attaching a single column at the right and a single row at the bottom to get a $33 \times 33 \times 32$ block. With this padding and stride, the pooling takes $33 \times 33 \times 32$ block and produces a $16 \times 16 \times 32$ block (you should check this with a pencil and paper drawing; it's right). The receptive field for values in this block is 7×7 (you should check this with a pencil and paper drawing; it's right, too).

The layer labelled "Apool 3s2" is an average pooling layer which computes an average in a 3×3 window, again with a stride of 2. The block is padded before this layer in the same way the block before the max pooling layer was padded. Eventually, we wind up with a 64 dimensional feature vector describing the image, and the convolutional layer and softmax that follow are logistic regression applied to that feature vector.

Just like MNIST, much of the information in a CIFAR-10 image is redundant. It's now somewhat harder to see the redundancies, but Figure 12.6 should make

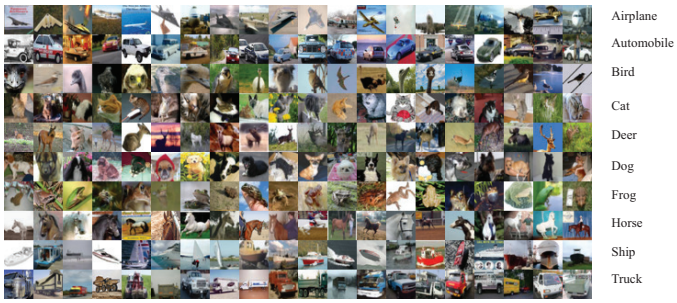


FIGURE 12.12: *Some of the approximately 2000 test examples misclassified by the network trained in the text. Each row corresponds to a category. The images in that row belong to that category, but are classified as belonging to some other category. At least some of these images look like "uncommon" views of the object or "strange" instances – it's plausible that the network misclassifies images when the view is uncommon or the object is a strange instance of the category.*

you suspect that some classes have different backgrounds than others. Figure 12.6 shows the class mean for each class. There are a variety of options for normalizing these images (more below). For this example, I whitened pixel values for each pixel in the image grid independently (procedure 12.1, which is widely used). Whitened images tend to be very hard for humans to interpret. However, the normalization involved deals with changes in overall image brightness and moderate shifts in color rather well, and can significantly improve classification.

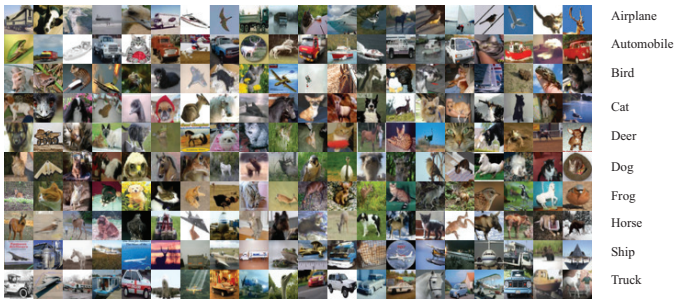


FIGURE 12.13: *Some of the approximately 2000 test examples misclassified by the network trained in the text. Each row corresponds to a category. The images in that row are classified as belonging to that category, but actually belong to another. At least some of these images look like "confusing" views — for example, you can find birds that do look like aircraft, and aircraft that do look like birds.*

Procedure: 12.1 *Simple image whitening*

At training time: Start with N training images $\mathcal{I}^{(i)}$. We assume these are 3D blocks of data. Write $I^{(i)}_{uvw}$ for the u, v, w'th location in the i'th image. Compute $\mathcal{M}$ and $\mathcal{S}$, where the u, v, w'th location in each is given by

$$\begin{aligned} M_{uvw} &= \frac{\sum_i I^{(i)}_{uvw}}{N} \\ S_{uvw} &= \sqrt{\frac{\sum_i (I^{(i)}_{uvw} - M^{(i)}_{uvw})^2}{N}} \end{aligned}$$

Choose some small number ϵ to avoid dividing by zero. Now the i'th whitened image, $\mathcal{W}^{(i)}$, has for its u, v, w'th location

$$W^{(i)}_{uvw} = (I^{(i)}_{uvw} - M_{uvw})/(S_{uvw} + \epsilon)$$

Use these whitened images to train.
At test time: For a test image $\mathcal{T}$, compute $\mathcal{W}$ which has for its u, v, w'th location

$$W_{uvw} = (T_{uvw} - M_{uvw})/(S_{uvw} + \epsilon)$$

and classify that.

I trained this network for 20 epochs using tutorial code circulated with Matconvnet. Mini-batches are pre-selected so that each training data item is touched once per epoch, so an epoch represents a single pass through the data. It is common in image classification to report loss, top-1 error and top-5 error. Top-1 error is

FIGURE 12.14: *Visualizing the patterns that the final stage ReLU's respond to for the simple CIFAR example. Each block of images shows the images that get the largest output for each of 10 ReLU's (the ReLU's were chosen at random from the 64 available in the top ReLU layer). Notice that these ReLU outputs don't correspond to class – these outputs go through a fully connected layer before classification – but each ReLU are clearly responds to a pattern, and different ReLU's respond more strongly to different patterns.*

the frequency that the correct class has the highest posterior. Top-5 error is the frequency that the correct class appears in the five classes with largest posterior. This can be useful when the top-1 error is large, because you may observe improvements in top-5 error even when the top-1 error doesn't change. Figure 12.11 shows the loss, top-1 error and top-5 error for training and validation sets plotted as a function of epoch. This classifier misclassifies about 2000 of the test examples, so it is hard to show all errors. Figure 12.12 shows examples from each class that are misclassified as belonging to some other class. Figure 12.13 shows examples that are that are misclassified into each class.

The phenomenon that ReLU's are pattern detectors is quite reliable. Figure 12.14 shows the 20 images that give the strongest responses for each of 10 ReLU's in the final ReLU layer. These ReLU's clearly have a quite strong theory of a pattern, and different ReLU's respond most strongly to quite different patterns. More sophisticated visualizations search for images that get the strongest response from units at various stages of complex networks; it's quite reliable that these images show a form of order or structure.

12.5 TRICKS AND QUIRKS

TODO: Other kinds of normalization

12.5.1 Quirks: Adversarial Examples

Adversarial examples are a curious experimental property of neural network image classifiers. Here is what happens. Assume you have an image $\mathbf{x}$ that is correctly

classified with label l. The network will produce a probability distribution over labels $P(L|\mathbf{x})$. Choose some label k that is not correct. It is possible to use modern optimization methods to search for a modification to the image $\delta\mathbf{x}$ such that

$$\begin{aligned} \delta\mathbf{x} \quad & \text{is small} \\ & \textit{and} \\ P(k|\mathbf{x}+\delta\mathbf{x}) \quad & \text{is large.} \end{aligned}$$

You might expect that $\delta\mathbf{x}$ is "large"; what is surprising is that mostly it is so tiny as to be imperceptible to a human observer. The property of being an adversarial example seems to be robust to image smoothing, simple image processing, and printing and photographing. The existence of adversarial examples raises the following, rather alarming, prospect: You could make a template that you could hold over a stop sign, and with one pass of a spraypaint can, turn that sign into something that is interpreted as a minimum speed limit sign by current computer vision systems. I haven't seen this demonstration done yet, but it appears to be entirely within the reach of modern technology, and it and activities like it offer significant prospects for mayhem.

What is startling about this behavior is that it is exhibited by networks that are very good at image classification, *assuming* that no-one has been fiddling with the images. So modern networks are very accurate on untampered pictures, but may behave very strangely in the presence of tampering. One can (rather vaguely) identify the source of the problem, which is that neural network image classifiers have far more degrees of freedom than can be pinned down by images. This observation doesn't really help, though, because it doesn't explain why they (mostly) work rather well, and it doesn't tell us what to do about adversarial examples. There have been a variety of efforts to produce networks that are robust to adversarial examples, but evidence right now is based only on experiment (some networks behave better than others) and we are missing clear theoretical guidance.

CHAPTER 13

Image Classification

CHAPTER 14

Object Detection

TODO: transformer based detection

CHAPTER 15

Semantic Segmentation

CHAPTER 16

Generating Images and Video from Random Numbers

PART FIVE

HOW IMAGES ARE MADE

CHAPTER 17

Cameras, Light and Shading

17.1 CAMERAS

17.1.1 The Pinhole Camera

A *pinhole camera* is a light-tight box with a very small hole in the front (Figure 17.1). Think about a point on the back of the box. The only light that arrives at that point must come through the hole, because the box is light-tight. If the hole is very small, then the light that arrives at the point comes from only one direction. This means that an inverted image of a scene appears at the back of the box (Figure 17.1). An appropriate sensor (CMOS sensor; CCD sensor; light sensitive film) at the back of the box will capture this image.

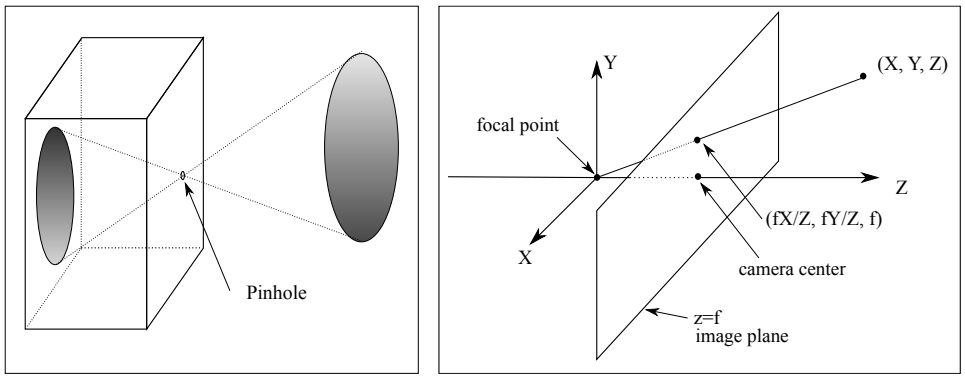


FIGURE 17.1: *The pinhole imaging model. On the* **left**, *a light-tight box with a pinhole in it views an object. The only light that a point on the back of the box sees comes through the very small pinhole, so that an inverted image is formed on the back face of the box. On the* **right**, *the usual geometric abstraction. The box doesn't affect the geometry, and is omitted. The pinhole has been moved to the back of the box, so that the image is no longer inverted. The image is formed on the plane $z = f$, by convention. Notice the coordinate system is* left-handed, *because the camera looks down the z-axis. This is because most people's intuition is that z* increases *as one moves into the image. The text provides some more detail on this point.*

Pinhole camera models produce an upside-down image. This is easily dealt with in practice (turn the image the right way up). An easy way to account for this is to assume the sensor is *in front* of the hole, so that the image is not upside-down. One could not build a camera like this (the sensor blocks light from the hole) but it is a convenient abstraction. There is a standard model of this camera, in a standard coordinate system. The coordinate system is left-handed even though coordinate

systems in 3D are usually right-handed coordinate systems. This is because most people's intuition is that z *increases* as one moves into the image. The pinhole – usually called the *focal point* – is at the origin, and the sensor is on the plane $z = f$. This plane is the *image plane*, and f is the *focal length*. We ignore any camera body and regard the image plane as infinite.

Under this highly abstracted camera model, almost any point in 3D will map to a point in the image plane. We *image* a point in 3D by constructing a ray through the 3D point and the focal point, and intersecting that ray with the image plane. The focal point has an important, distinctive, property: It cannot be imaged, and it is the only point that cannot be imaged.

Similar triangles yields that the point (X, Y, Z) in 3D is imaged to

$$(fX/Z, fY/Z, f)$$

on the sensor (Figure 17.1). Notice that the z-coordinate is the same for each point on the image plane, so it is quite usual to ignore it and use the model

$$(X, Y, Z) \to (fX/Z, fY/Z).$$

The focal length just scales the image. In standard camera models, other scaling effects occur as well, and we write projection as if $f = 1$, yielding

$$(X, Y, Z) \to (X/Z, Y/Z).$$

The projection process is known as *perspective projection*. The point where the z-axis intersects the image plane (equivalently, where the ray through the focal point perpendicular to the image plane intersects the image plane) is the *camera center*. Remarkably, in almost every publication in computer vision the camera is expressed in left-handed coordinates and everything else works in right-handed coordinates. The exercises demonstrate that there is no real difficulty here.

Remember this: *Most practical cameras can be modelled as a pinhole camera. The standard model of the pinhole camera maps*

$$(X, Y, Z) \to (X/Z, Y/Z).$$

Figure 17.1 shows important terminology (focal point; image plane; camera center).

17.1.2 Perspective Effects

Perspective projection has a number of important properties, summarized as:

- lines project to lines;
- more distant objects are smaller;

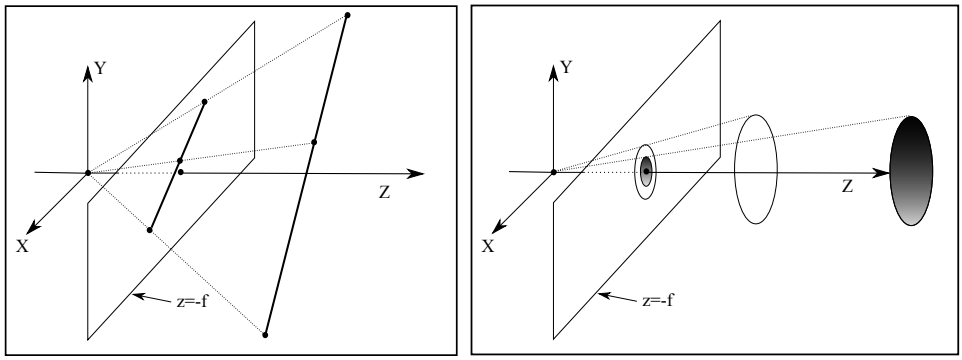


FIGURE 17.2: *Perspective projection maps almost any 3D line to a line in the image plane (***left***). Some rays from the focal point to points on the line are shown as dotted lines. The family of all such rays is a plane, and that plane must intersect the image plane in a line as long as the 3D line does not pass through the focal point. On the* **right***, two 3D objects viewed in perspective projection; the more distant object appears smaller in the image.*

- lines that are parallel in 3D meet in the image;
- planes have horizons;
- planes image as half-planes.

Lines project to lines: Almost every line in 3D maps to a line in the image. You can see this by noticing that the image of the 3D line is formed by intersecting rays from the focal point to each point on the 3D line with the image plane. But these rays form a plane, so we are intersecting a plane with the image plane, and so obtain a line (Figure 17.2). The exceptions are the 3D lines through the focal point – these project to points.

More distant objects are smaller: The further away an object is in 3D, the smaller the image of that object, because of the division by Z (Figure 17.2).

Lines that are parallel in 3D meet in the image: Now think about a set of infinitely long parallel railroad tracks. The sleepers supporting the tracks are all the same size. Distant sleepers are smaller than nearby sleepers, and arbitrarily distant sleepers are arbitrarily small. This means that parallel lines will meet in the image. The point at which the lines in a collection of parallel lines meet is known as the *vanishing point* for those lines (Figure 17.3). The vanishing point for a set of parallel lines can be obtained by intersecting the ray from the focal point and parallel to those lines with the image plane (Figure 17.3).

Planes have horizons: Now think about the image of a plane. As Figure 17.5 shows, the plane through the focal point and parallel to that plane produce a line in the image, known as the *horizon* of the plane.

Planes image as half-planes: For an abstract perspective camera, any point on the plane can be imaged to a point on the image plane. In practical cameras, we cannot image points that lie behind the camera in 3D. Now cast a ray through the focal point and some point $\mathbf{x}$ in the image plane. If $\mathbf{x}$ is on one side of

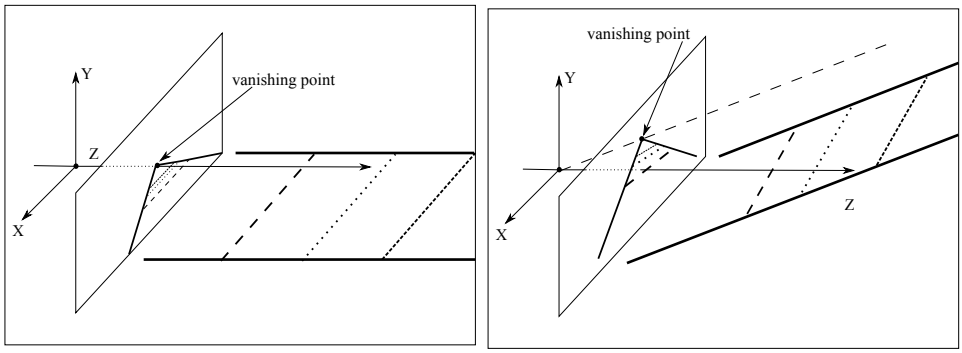


FIGURE 17.3: *Perspective projection maps a set of parallel lines to a set of lines that meet in a point. On the* **left**, *a set of lines parallel to the z-axis, with "railway sleepers" shown. As these sleepers get further away, they get smaller in the image, meaning the projected lines must meet. The vanishing point (the point where they meet) is obtained by intersecting the ray parallel to the lines and through the focal point with the image plane. On the* **right**, *a different pair of parallel lines with a different vanishing point. The figure establishes that, if there are more than two lines in the set of parallel lines, all will meet at the vanishing point.*

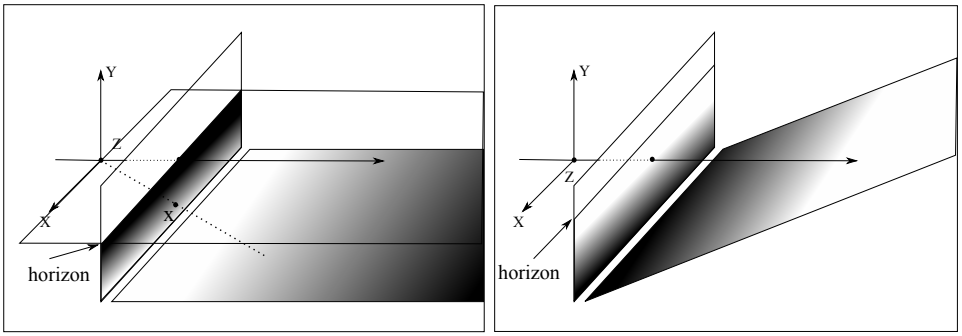


FIGURE 17.4: **Left** *shows a plane in 3D (in this case, $y = -1$). The intersection of the plane through the focal point parallel to the 3D plane (in this case, $y = 0$) and the image plane, forms an image line called the horizon. This line cuts the image plane into two parts. Construct the ray through the focal point and a point* $\mathbf{x}$ *in the image plane. For* $\mathbf{x}$ *on one side of the horizon, this ray will intersect the 3D plane in the half space $z > 0$ (and so in front of the camera, shown here). If* $\mathbf{x}$ *is on the other side of the horizon, the intersection will be in the half space $z < 0$ (and so behind the camera, where it cannot be seen).* **Right** *shows a different 3D plane with a different horizon. The gradients on the planes indicate roughly where points on the 3D plane appear in the image plane (light points map to light, dark to dark).*

the horizon, the ray will hit the plane in the $z > 0$ half space and so we can see the plane. If it is on the other side, it will hit the plane in the $z < 0$ half space, so we cannot see the plane.

Remember this: *Under perspective projection:*

- *points project to points;*
- *lines project to lines;*
- *more distant objects are smaller;*
- *lines that are parallel in 3D meet in the image;*
- *planes have horizons;*
- *planes image as half-planes.*

17.1.3 Scaled Orthographic Projection and Orthographic Projection

Under some circumstances, perspective projection can be simplified. Assume the camera views a set of points which are close to one another compared with the distance to the camera. Write $\mathbf{X}_i = (X_i, Y_i, Z_i)$ for the i'th point, and assume that $Z_i = Z(1+\epsilon_i)$, where ϵ_i is quite small. In this case, the distance to the set of points is much larger than the *relief* of the points, which is the distance from nearest to furthest point. The i'th point projects to $(fX_i/Z_i, fY_i/Z_i)$, which is approximately $(f(X_i/Z)(1-\epsilon_i), f(Y_i/Z)(1-\epsilon_i))$. Ignoring ϵ_i because it is small, we have the projection model

$$(X, Y, Z) \to (f/Z)(X, Y) = s(X, Y).$$

This model is usually known as *scaled orthograpic projection*. The model applies quite often. One important example is pictures of people. Very often, all body parts are roughly the same distance from the camera — think of a side view of a pedestrian seen from a motor car. Scaled orthographic projection applies in such cases. It is not always an appropriate model. For example, when a person is holding up a hand to block the camera's view, perspective effects can be significant (Figure **??**).

Occasionally, it is useful to rescale the camera (or assume that $f/Z = 1$), yielding $(X, Y, Z) \to (X, Y)$. This is known as *orthographic projection*.

Remember this: *Scaled orthographic projection maps*

$$(X, Y, Z) \to s(X, Y)$$

where s is some scale. The model applies when the distance to the points being viewed is much greater than their relief. Many views of people have this property.

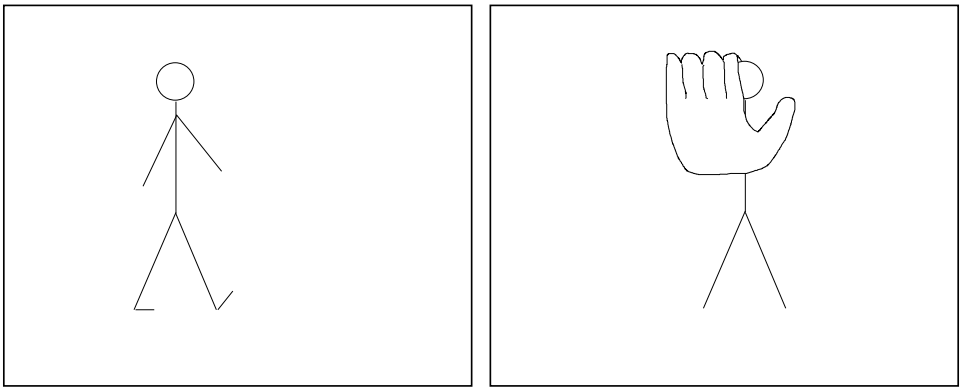


FIGURE 17.5: *The pedestrian on the* **left** *is viewed from some way away, so the distance to the pedestrian is much larger than the change in depth over the pedestrian. In this case, which is quite common for views of people, scaled orthography will apply. The celebrity on the* **right** *is holding a hand up to prevent the camera viewing their face; the hand is quite close to the camera, and the body is an armslength away. In this case, perspective effects are strong. The hand looks big because it is close, and the head looks small because it is far.*

17.1.4 Lenses

One practical version of a pinhole camera is a *camera obscura* – the box is built as a room, and you can stand in the room and see the view on the back wall (some examples are at `https://www.atlasobscura.com/lists/camera-obscura-places`; the internet yields amusing disputes about the correct plural form of the term). You can also build a simple pinhole camera with a matchbox, some tape, a pin, and some light sensitive film do the trick. Getting good images takes trouble, however.

A large hole in front of the camera will cause the image at the back to be brighter, but blurrier, because each point on the sensor will average light over all directions that happen to go through the hole. If the hole is smaller, the image will get sharper, but darker. In practical cameras, achieving an image that is both bright and focused is the job of the lens system. There may be one or several lenses that light passes through before reaching the sensor at the back of the camera. Each of these lenses is built from refracting materials. The shape and position of the lenses, together with the refractive index of the materials they are built of, determine the path that light follows through the lens system. Generally, the lens system is designed to collect as much light as possible at the input and produce a focused image on the image plane. Remarkably, the many or most lens systems result in an imaging geometry that can be modelled with a pinhole camera model, and lens system effects are ignored in all but quite specialized applications of computer vision.

Lens systems are designed and modelled using geometric optics, but lens designs always involve compromises. The result is that cameras with lenses differ from pinhole cameras in some ways that are worth knowing about, although they are not always important. First, in an abstract pinhole camera, all objects at what-

ever distance are in focus. Geometric optics means that a lens with this property admits very little light, so it is common to work with cameras that have a limited *depth of field* – the range of distances to the camera over which objects are in focus on the image plane. Second, manufacturing difficulties and cost considerations mean that lenses will have various *aberrations*. The net effect of most aberrations is a tendency to defocus some objects under some circumstances, but *chromatic aberrations* can cause colors to be less crisp and objects to have "halos" of color. Chromatic aberration occurs because light of different wavelengths takes slightly different paths through a refracting object. Various lens coatings can correct chromatic aberration, but the resulting lens system will be more expensive. Third, in most lens systems, the periphery of the image tends to be brighter than it would be in a pure pinhole camera. For more complex lens systems, an effect in the lens known as *vignetting* can darken the periphery somewhat. Finally, lenses may cause *geometric distortions* of the image. The most noticeable effect of these distortions is that straight lines in the world may project to curves in the image. Most common is *barrel distortion*, where a square is imaged as a bulging barrel; *pincushion distortion*, where the square bulges in rather than out, can occur (Figure **??**).

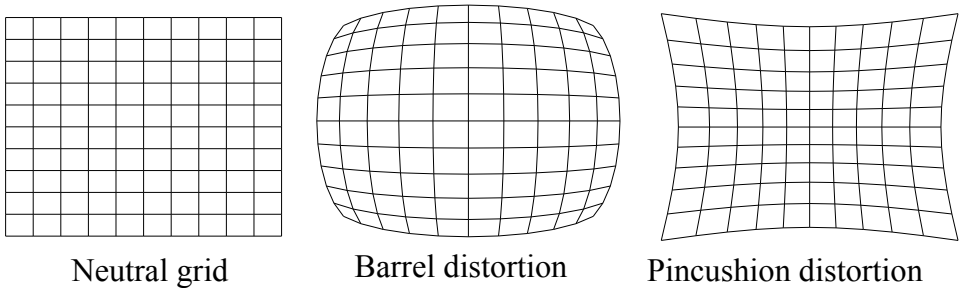


FIGURE 17.6: *On the* **left** *a neutral grid observed in a non-distorting lens (and viewed frontally to prevent any perspective distortion).* **Center** *shows the same grid, viewed in a lens that produces barrel distortion.* **Right***, the same grid, now viewed in a lens that produces pincushion distortion.*

17.2 LIGHT AND SURFACES

Three major phenomena determine the brightness of a pixel: the response of the camera to light, the fraction of light reflected from the surface to the camera, and the amount of light falling on the surface. Each can be dealt with quite straightforwardly.

Camera response: Modern camera sensors respond linearly to light. This linear response is adjusted in software, because humans find linear images confusing (such images tend to be too dark in most places, and too light in others). The *camera response function* or *CRF* determines what value is reported at each location. Typical CRF's are close to linear in mid-ranges, but have pronounced nonlinearities for darker and brighter illumination. This allows the camera to reproduce the very wide dynamic range of natural light without saturating.

Write $\mathbf{X}$ for a point in space that projects to $\mathbf{x}$ in the image, $I_{patch}(\mathbf{X})$ for

the intensity of the surface patch at $\mathbf{X}$, $C(\cdot)$ for the camera response function, and $I_{camera}(\mathbf{x})$ for the camera response at $\mathbf{x}$. Then our model is:

$$I_{camera}(\mathbf{x}) = C(I_{patch}(\mathbf{x})).$$

It is quite usual to assume that the camera response is linearly related to the intensity of the surface patch. In this case, $C(I_{patch}(\mathbf{x})) = kI_{patch}(\mathbf{x})$, and it is common to assume that k is known if needed. A CRF can be recovered from enough image data, if required (Section 19.1.1).

Surface reflection: Different points on a surface may reflect more or less of the light that is arriving. Darker surfaces reflect less light, and lighter surfaces reflect more. There is a rich set of possible physical effects, but most can be ignored. Section 17.2.1 describes the relatively simple model that is sufficient for almost all purposes in computer vision.

Illumination: The amount of light a patch receives depends on the overall intensity of the light, and on the geometry. The overall intensity could change because some *luminaires* (the formal term for light sources) might be shadowed, or might have strong directional components. Geometry affects the amount of light arriving at a patch because surface patches facing the light collect more radiation and so are brighter than surface patches tilted away from the light, an effect known as *shading*. Section 17.2.2 describes the most important model used in computer vision; Section 17.2.4 describes a much more complex model that is necessary to explain some important practical difficulties in shading inference.

17.2.1 Reflection at Surfaces

Most surfaces reflect light by a process of *diffuse reflection*. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. Examples are easy to identify with this test: most cloth has this property, as do most paints, rough wooden surfaces, most vegetation, and rough stone or concrete. The only parameter required to describe a surface of this type is its *albedo*, the fraction of the light arriving at the surface that is reflected. This does not depend on the direction in which the light arrives or the direction in which the light leaves. Surfaces with very high or very low albedo are difficult to make. For practical surfaces, albedo lies in the range 0.05 – 0.90 (see **?**, who argue the dynamic range is closer to 10 than the 18 implied by these numbers). Mirrors are not diffuse, because what you see depends on the direction in which you look at the mirror. The behavior of a perfect mirror is known as *specular reflection*. For an ideal mirror, light arriving along a particular direction can leave only along the *specular direction*, obtained by reflecting the direction of incoming radiation about the surface normal (Figure 17.7). Usually some fraction of incoming radiation is absorbed; on an ideal specular surface, this fraction does not depend on the incident direction.

If a surface behaves like an ideal specular reflector, you could use it as a mirror, and based on this test, relatively few surfaces actually behave like ideal specular reflectors. Imagine a near perfect mirror made of polished metal; if this surface suffers slight damage at a small scale, then around each point there will be a set of small facets, pointing in a range of directions. In turn, this means that

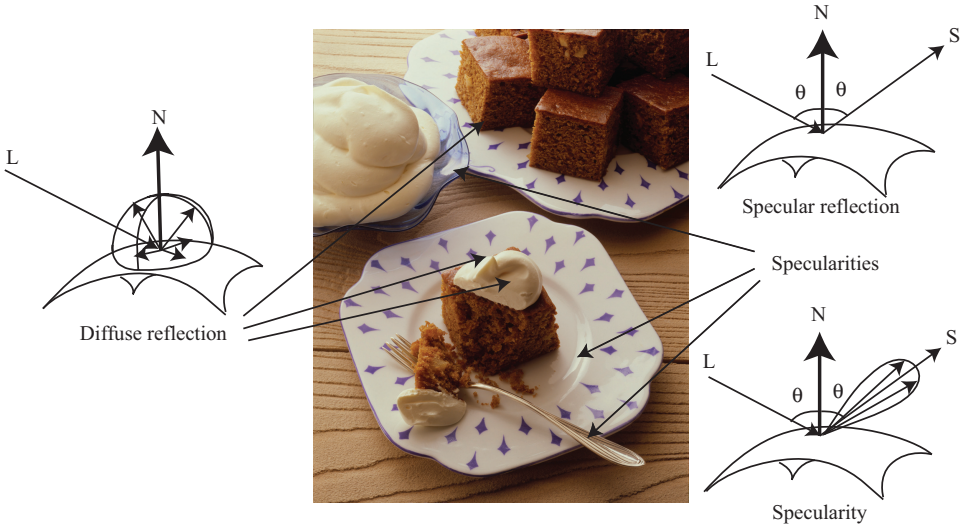


FIGURE 17.7: *The two most important reflection modes for computer vision are diffuse reflection (***left***), where incident light is spread evenly over the whole hemisphere of outgoing directions, and specular reflection (***right***), where reflected light is concentrated in a single direction. The specular direction* **S** *is coplanar with the normal and the source direction (***L***), and has the same angle to the normal that the source direction does. Most surfaces display both diffuse and specular reflection components. In most cases, the specular component is not precisely mirror like, but is concentrated around a range of directions close to the specular direction (***lower right***). This causes specularities, where one sees a mirror like reflection of the light source. Specularities, when they occur, tend to be small and bright. In the photograph, they appear on the metal spoon and on the plate. Large specularities can appear on flat metal surfaces (arrows). Most curved surfaces (such as the plate) show smaller specularities. Most of the reflection here is diffuse; some cases are indicated by arrows.* Martin Brigdale © Dorling Kindersley, used with permission.

light arriving in one direction will leave in several different directions because it strikes several facets, and so the specular reflections will be blurred. As the surface becomes less flat, these distortions will become more pronounced; eventually, the only specular reflection that is bright enough to see will come from the light source. This mechanism means that, in most shiny paint, plastic, wet, or brushed metal surfaces, one sees a bright blob—often called a *specularity*—along the specular direction from light sources, but few other specular effects. Specularities are easy to identify, because they are small and very bright (Figure 17.7; **?**). Most surfaces reflect only some of the incoming light in a specular component, and we can represent the percentage of light that is specularly reflected with a *specular albedo*. Although the diffuse albedo is an important material property that we will try to estimate from images, the specular albedo is largely seen as a nuisance and usually is not estimated.

For almost all purposes, it is enough to model all surfaces as being diffuse with specularities. This is the *lambertian+specular model*. Specularities are relatively seldom used in inference, and so there is no need for a formal model of their structure. Because specularities are small and bright, they are relatively easy to identify and remove with straightforward methods (find small bright spots, and replace them by smoothing the local pixel values). More sophisticated specularity finders use color information []. Thus, to apply the lambertian+specular model, we find and remove specularities, and then use Lambert's law (Section 17.2.2) to model image intensity.

17.2.2 Sources and Their Effects

The main source of illumination outdoors is the sun, whose rays all travel parallel to one another in a known direction because it is so far away. We model this behavior with a *distant point light source*. This is the most important model of lighting (because it is like the sun and because it is easy to use), and can be quite effective for indoor scenes as well as outdoor scenes. Because the rays are parallel to one another, a surface that faces the source cuts more rays (and so collects more light) than one oriented along the direction in which the rays travel. The amount of light collected by a surface patch in this model is proportional to the cosine of the angle θ between the illumination direction and the normal (Figure 17.8). The figure yields *Lambert's cosine law*, which states the brightness of a diffuse patch illuminated by a distant point light source is given by

$$I = \rho I_0 \cos\theta,$$

where I_0 is the intensity of the light source, θ is the angle between the light source direction and the surface normal, and ρ is the diffuse albedo. This law predicts that bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. We explore this cue in Section **??**.

If the surface cannot see the source, then it is in *shadow*. Since we assume that light arrives at our patch only from the distant point light source, our model suggests that shadows are deep black; in practice, they very seldom are, because the shadowed surface usually receives light from other sources. Outdoors, the most important such source is the sky, which is quite bright. Indoors, light reflected from other surfaces illuminates shadowed patches. This means that, for example, we tend to see few shadows in rooms with white walls, because any shadowed patch receives a lot of light from the walls. These effects are sometimes modelled by adding a constant *ambient illumination* term to the predicted intensity. The ambient term ensures that shadows are not too dark, but this is not a particularly good model of the spatial properties of interreflections. We have sketched the effects to be aware of in Section 17.2.4.

17.2.3 The Local Shading Model for Distant Luminaires

Surfaces reflect light onto one another (*interreflections*), meaning that the light arriving at a surface could have come directly from a luminaire, but it could also have been reflected from some other surface. Really accurate physical models of how

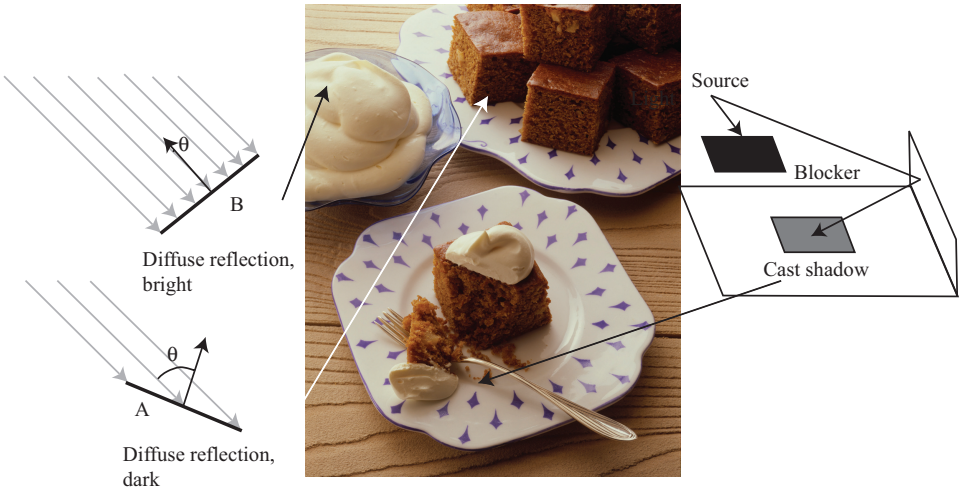


FIGURE 17.8: *The orientation of a surface patch with respect to the light affects how much light the patch gathers. We model surface patches as illuminated by a distant point source, whose rays are shown as light arrowheads. Patch A is tilted away from the source (θ is close to 90^0) and collects less energy, because it cuts fewer light rays per unit surface area. Patch B, facing the source (θ is close to 0^0), collects more energy, and so is brighter. Shadows occur when a patch cannot see a source. The shadows are not dead black, because the surface can see inter-reflected light from other surfaces. These effects are shown in the photograph. The darker surfaces are turned away from the illumination direction.* Martin Brigdale © Dorling Kindersley, used with permission.

light is distributed on scenes are now very well known [**?**] and are extremely useful in computer graphics. These models are very hard to use for inference, because every variable affects every other variable. For example, changes in the orientation of one surface element affect how much light it reflects onto every other surface element.

This means we must simplify the model, and so we must be using a model that isn't exact, meaning we need to keep track of what that model will do well and what it will do badly. The usual simplification is a *local shading model*, where we assume that shading is caused only by light that comes from the luminaire (i.e., that there are no interreflections).

Now assume that the luminaire is an infinitely distant source. For this case, write $\mathbf{N}(x)$ for the unit surface normal at $\mathbf{x}$, $\mathbf{S}$ for a vector pointing from $\mathbf{x}$ toward the source with length I_o (the source intensity), $\rho(\mathbf{x})$ for the albedo at $\mathbf{x}$, and $Vis(\mathbf{S}, \mathbf{x})$ for a function that is 1 when $\mathbf{x}$ can see the source and zero otherwise.

Then, the intensity at $\mathbf{x}$ is

$$I(\mathbf{x}) = \rho(\mathbf{x})\,(\mathbf{N}\cdot\mathbf{S})\,\mathrm{Vis}(\mathbf{S},\,\mathbf{x}) + \rho(\mathbf{x})A + M$$

Image intensity	=	Diffuse term	+	Ambient term	+	Specular (mirror-like) term

17.2.4 Shading Effects from Area Sources

The local shading model is a good rough and ready model, but it isn't right. It predicts dark shadows with sharp boundaries. These are quite common outdoors where the sun is the most important light source, but are uncommon indoors. To understand why, we must look at area sources.

An *area source* is an area that radiates light. Area sources occur quite commonly in natural scenes—an overcast sky is a good example—and in synthetic environments—for example, the fluorescent light boxes found in many industrial ceilings. Area sources are common in illumination engineering, because they tend not to cast strong shadows and because the illumination due to the source does not fall off significantly as a function of the distance to the source. Detailed models of area sources are complex, but a simple model is useful to understand shadows. Shadows from area sources are very different from shadows cast by point sources. One seldom sees dark shadows with crisp boundaries indoors. Instead, one could see no visible shadows, or shadows that are rather fuzzy diffuse blobs, or sometimes fuzzy blobs with a dark core (Figure 17.9). These effects occur indoors because rooms tend to have light walls and diffuse ceiling fixtures, which act as area sources. As a result, the shadows one sees are area source shadows.

To compute the intensity at a surface patch illuminated by an area source, we can break the source up into infinitesimal source elements, then sum effects from each element. If there is an occluder, then some surface patches may see none of the source elements. Such patches will be dark, and lie in the *umbra* (a Latin word meaning "shadow"). Other surface patches may see some, but not all, of the source elements. Such patches may be quite bright (if they see most of the elements), or relatively dark (if they see few elements), and lie in the *penumbra* (a compound of Latin words meaning "almost shadow"). One way to build intuition is to think of a tiny observer looking up from the surface patch. At umbral points, this observer will not see the area source at all whereas at penumbral points, the observer will see some, but not all, of the area source. An observer moving from outside the shadow, through the penumbra and into the umbra will see something that looks like an eclipse of the moon (Figure 17.9). The penumbra can be large, and can change quite slowly from light to dark. There might even be no umbral points at all, and, if the occluder is sufficiently far away from the surface, the penumbra could be very large and almost indistinguishable in brightness from the unshadowed patches. This is why many objects in rooms appear to cast no shadow at all (Figure 17.10).

17.3 DEPTH MEASUREMENT

The cameras of chapter 33.2 project points in 3D to points on an image plane. Building such cameras is now very well understood (and they are extremely cheap).

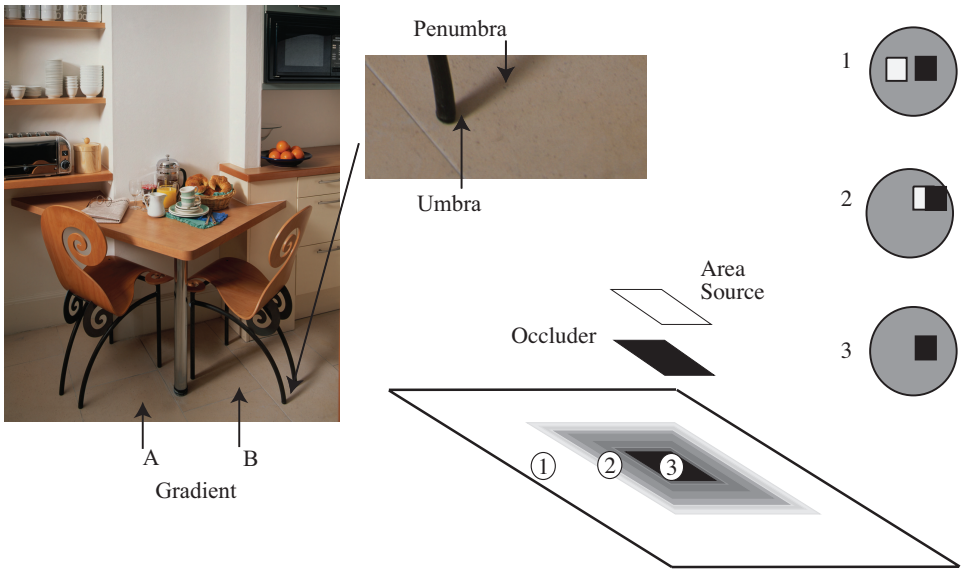


FIGURE 17.9: *Area sources generate complex shadows with smooth boundaries, because from the point of view of a surface patch, the source disappears slowly behind the occluder.* **Left:** *a photograph, showing characteristic area source shadow effects. Notice that A is much darker than B; there must be some shadowing effect here, but there is no clear shadow boundary. Instead, there is a fairly smooth gradient. The chair leg casts a complex shadow, with two distinct regions. There is a core of darkness (the* umbra*—where the source cannot be seen at all) surrounded by a partial shadow (*penumbra*— where the source can be seen partially). A good model of the geometry, illustrated* **right***, is to imagine lying with your back to the surface looking at the world above. At point 1, you can see all of the source; at point 2, you can see some of it; and at point 3, you can see none of it.* Peter Anderson © Dorling Kindersley, used with permission.

A lot is known about how to recover the points in 3D from the projected versions under various circumstances (some of this appears in chapters 33.2), but doing so can be inconvenient. It is often very useful to measure the 3D location of points directly.

17.3.1 Stereoscopic Depth Measurement

Stereo uses two cameras somewhat offset from one another. Figure 33.2 sketches this idea. The key is that if you know where the cameras are with respect to one another, and where a 3D point projects to in each of two perspective images, simple trigonometry will reveal where it is in 3D. Calibrating the relative geometry of the cameras is now well understood (Chapter 33.2), as is determining which (if any) point in the first image corresponds to which in the second (Chapter 33.2), and recovering a good depth model from this information (Chapter 33.2). Stereo rigs can be very cheap and accurate, and they have the great advantage that measurement

FIGURE 17.10: *The photograph on the* **left** *shows a room interior. Notice the lighting has some directional component (the vertical face indicated by the arrow is dark, because it does not face the main direction of lighting), but there are few visible shadows (for example, the chairs do not cast a shadow on the floor). On the* **right**, *a drawing to show why; here there is a small occluder and a large area source. The occluder is some way away from the shaded surface. Generally, at points on the shaded surface the incoming hemisphere looks like that at point 1. The occluder blocks out some small percentage of the area source, but the amount of light lost is too small to notice (compare figure 17.9).* Jake Fitzjones © Dorling Kindersley, used with permission.

is passive – one does not have to send signals into the environment.

But there are limits to stereopsis. Measuring large depths with two cameras that are close together requires highly accurate estimates of point positions in images. Figure 17.11 shows a simple geometry that illustrates the problem. The point $\mathbf{P}$ projects to $\mathbf{x}_1$ in camera 1, and to $\mathbf{x}_2$ in camera 2. Notice because of the carefully chosen camera geometry, the y-coordinates of $\mathbf{x}_1$ and $\mathbf{x}_2$ are the same; only the x-coordinates differ. Write x_1 for the x-coordinate of $\mathbf{x}_1$; X for the x-coordinate of P, and so on. From the triangles in that figure, we have

$$d = x_2 - x_1 = f\frac{(X - B) - X}{Z} = -f\frac{B}{Z}$$

meaning that as $\mathbf{P}$ gets further away, the *disparity* (difference between projected positions in left and right cameras) gets smaller, and so gets harder to measure. Resolving small differences in large depths is going to be hard. This means that either the *baseline* (distance between camera focal points, B in Figure 33.2) is large (and so the equipment is bulky) or one can't reliably measure large depths.

A second important limit is that some points will appear in one camera, but not in the other (an effect known as *Da Vinci stereopsis*, illustrated in Figure 17.13), and so their depth cannot be measured by stereo. The result is quite characteristic "holes" in depth maps obtained from stereo cameras .

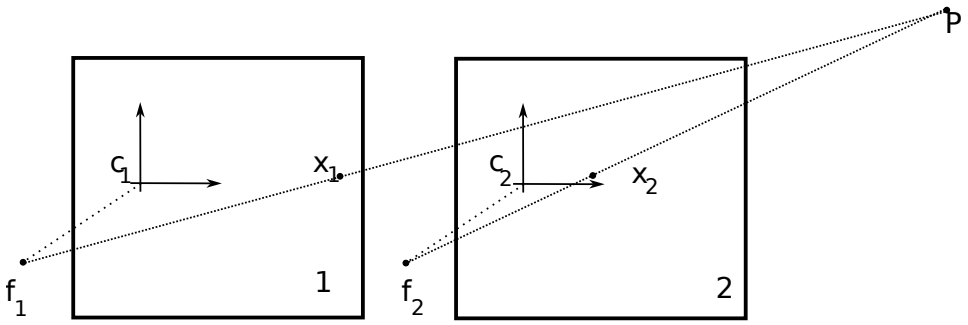


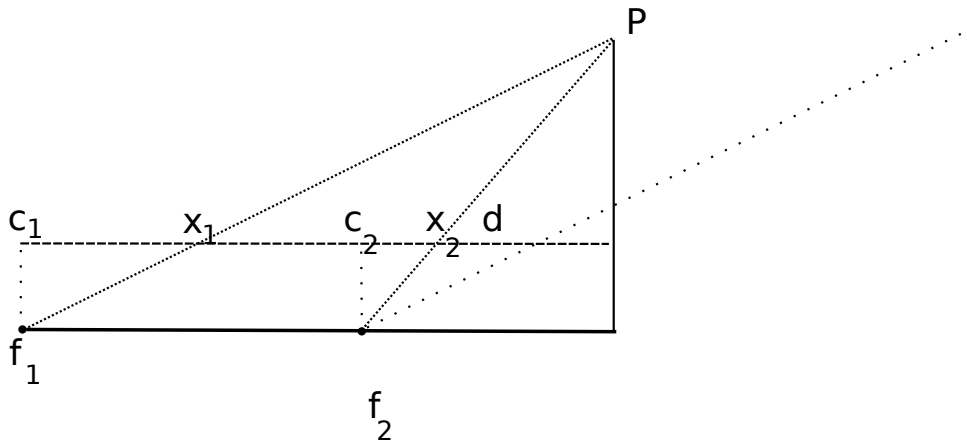
FIGURE 17.11: *When two pinhole cameras view a point, the 3D coordinates of the point can be reconstructed from the two images of that point. This applies for almost every configurations of the cameras. It is an elementary exercise in trigonometry (exercises) to determine* $\mathbf{P}$ *from the positions of the two focal points, the locations of the point in the two images, and the distance between the focal points. Considerable work can be required to find appropriate matching points, but the procedures required are now extremely well understood (Chapters 33.2). One can now buy camera systems that use this approach to report 3D point locations (often known as RGBD cameras). Here we show a specialized camera geometry, chosen to simplify notation. The second camera is translated with respect to the first, along a direction parallel to the image plane. The second camera is a copy of the first camera, so the image planes are parallel. In this geometry, the point being viewed shifts somewhat to the left in the right camera.*

17.3.2 Camera-Projector Stereo

The key difficulty in stereo is establishing which point in the left image corresponds to which in the right. This can be tricky even now for some kinds of object. One could use one camera and one projector. This projector is constructed to have geometry like that of a camera. Light leaves an analog of the focal point, and travels along rays through pixel locations. Modulation tricks mean the light through each different pixel location is uniquely identifiable. The geometry of Figure 17.11 still applies, but now the ray from $\mathbf{f}_1$ to $\mathbf{P}$ is a ray of emitted light.

A natural modulation trick is for the projector to display a sequence of (say) 8 patterns. Each pixel in each pattern is either dark or light. If the patterns are properly chosen, and if the camera observes all of them, you can think of each ray through the projector focal point as being tagged with eight bits. These eight bits identify the ray. Many rays will have the same bit pattern. If depth limits are known for the scene, and if the patterns are appropriately chosen, this ambiguity is not important.

For any baseline, there will be some practical limit to the largest and smallest depths that can be measured. This has an interesting consequence. In the geometry of Figure 17.11, imagine we fire a ray of modulated light from $\mathbf{f}_1$ through $\mathbf{x}_1$. If it is observed in camera 2 (it might not be, because the geometry of Figure 17.13 also still applies), we have a very good idea *where* it will be observed. The y-coordinate will not have changed and the disparity is limited by the depth range. This means



ht

FIGURE 17.12: **Right**: shows two important triangles in the plane spanned by the two focal points ($\mathbf{f}_1$, $\mathbf{f}_2$) and the point being viewed ($\mathbf{P}$). The extent of the shift leftwards (the *disparity*, d in the figure) reveals the depth to the point. Comparing triangle $\mathbf{f}_1$, $\mathbf{p}$, $\mathbf{R}$ with triangle $\mathbf{f}_2$, $\mathbf{p}$, $\mathbf{R}$ yields the relationship between depth, disparity and *baseline* (the distance between the two focal points.

we can use the same code for rays through two different points in camera 1 as long as they are sufficiently far apart.

Camera projector stereo uses the same geometry as two camera stereo, so that large depths are hard to measure without large baselines, and there will still be holes in depth maps.

17.3.3 Structured light

Structured light uses

17.3.4 Time of flight sensors and Lidar

could fire light out from a location, then wait till it returns. The length of the wait and the speed of light reveal the depth to the point (Figure 33.2).

17.4 CAMERA RESPONSE FUNCTIONS AND HDR IMAGES

PROBLEMS

17.1. Use Figure 33.2, and write B for the distance between $\mathbf{f}_L$ and $\mathbf{f}_R$ and $\mathbf{v}_L$ for the unit vector between $\mathbf{f}_L$ and $\mathbf{X}_L$.
(a) Show that the point

$$\mathbf{P} = \mathbf{f}_L + \mathbf{v}_L \left[\frac{B}{\cos\theta_L + \cos\theta_R \left(\frac{\sin\theta_L}{\sin\theta_R} \right)} \right]$$

(b) Show that the point

$$\mathbf{P} = \mathbf{f}_R + \mathbf{v}_R \left[\frac{B}{\cos\theta_R + \cos\theta_L \left(\frac{\sin\theta_R}{\sin\theta_L} \right)} \right]$$

(c) Under what circumstances could these two expressions produce different results? (hint: $\mathbf{f}_L$, $\mathbf{f}_R$, $\mathbf{X}_L$, $\mathbf{X}_R$ and $\mathbf{P}$ are coplanar, but what happens if $\mathbf{X}_L$ and $\mathbf{X}_R$ are measured with small errors?)

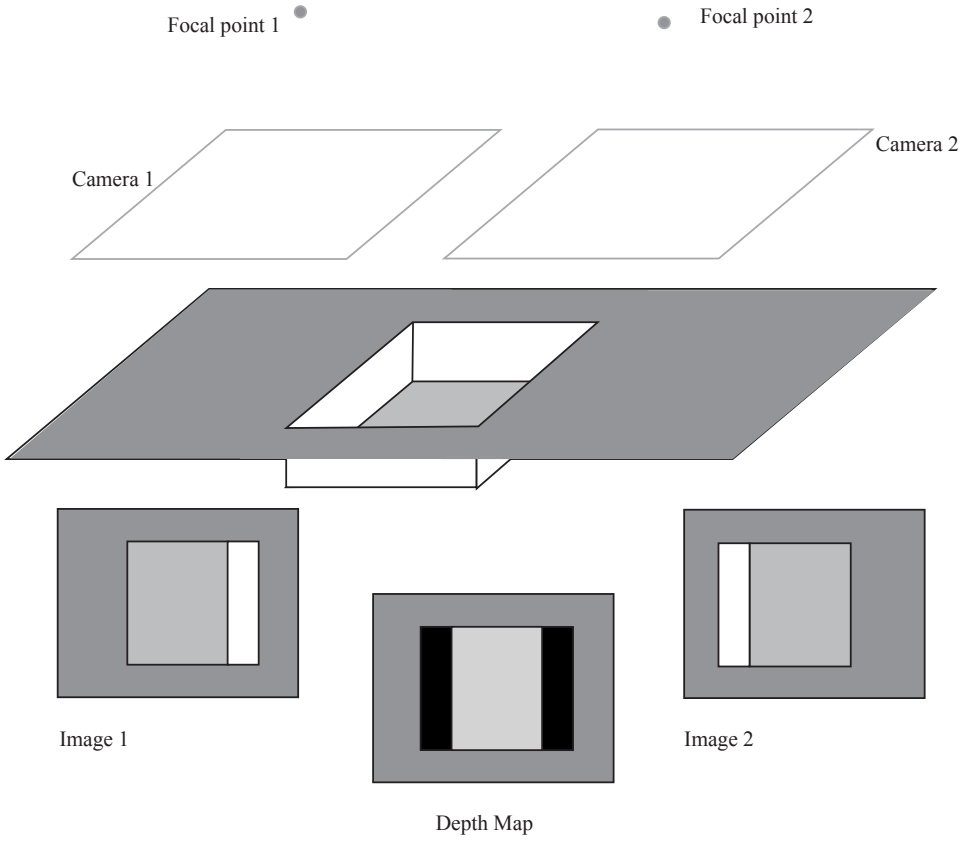


FIGURE 17.13: **Top** *shows two pinhole cameras viewing a rectangular depression in a flat surface. As the images show, camera on the left can see the right wall, and that on the right can see the left wall. This means that these walls cannot be reconstructed directly using trigonometry, and so the depth map will have holes in it. The depth map here is shown with a fairly common convention, where nearer surfaces are lighter, farther surfaces are darker, and holes are "infinitely far away".*

CHAPTER 18

Color Phenomena

The light receptors in cameras and in the eye respond more or less strongly to different wavelengths of light. Most cameras and most eyes have several different types of receptor, whose sensitivity to different wavelengths varies. Comparing the response of several types of sensor yields information about the distribution of energy with wavelength for the incoming light; this is color information. Color information can be used to remove shadows. The color of an object seen in an image depends on how the object was lit, but there are algorithms that can correct for this effect.

18.1 HUMAN COLOR PERCEPTION

The light coming out of sources or reflected from surfaces has more or less energy at different wavelengths, depending on the processes that produced the light. This distribution of energy with wavelength is sometimes called a *spectral energy density*; Figure 18.1 shows spectral energy densities for sunlight measured under a variety of different conditions. The visual system responds to light in a range of wavelengths from approximately 400nm to approximately 700nm. Light containing energy at just one wavelength looks deeply colored (these colors are known as *spectral colors*). The colors seen at different wavelengths have a set of conventional names, which originate with Isaac Newton (the sequence from 700nm to 400nm goes Red Orange Yellow Green Blue Indigo Violet, or **R**ichard **o**f **Y**ork **g**ot **b**listers **i**n **V**enice, although indigo is now frowned upon as a name because people typically cannot distinguish indigo from blue or violet). If the intensity is relatively uniform across the wavelengths, the light will look white.

Different kinds of color receptor in the human eye respond more or less strongly to light at different wavelengths, producing a signal that is interpreted as color by the human vision system. The precise interpretation of a particular light is a complex function of context; illumination, memory, object identity, and emotion can all play a part. The simplest question is to understand which spectral energy densities produce the same response from people under simple viewing conditions (Section 18.1.1). This yields a simple, linear theory of color matching that is accurate and extremely useful for describing colors. We sketch the mechanisms underlying the transduction of color in Section 18.1.2.

18.1.1 Color Matching

The simplest case of color perception is obtained when only two colors are in view on a black background. In a typical experiment, a subject sees a colored light—the *test light*—in one half of a split field (Figure 18.2). The subject can then adjust a mixture of lights in the other half to get it to match. The adjustments involve changing the intensity of some fixed number of *primaries* in the mixture.

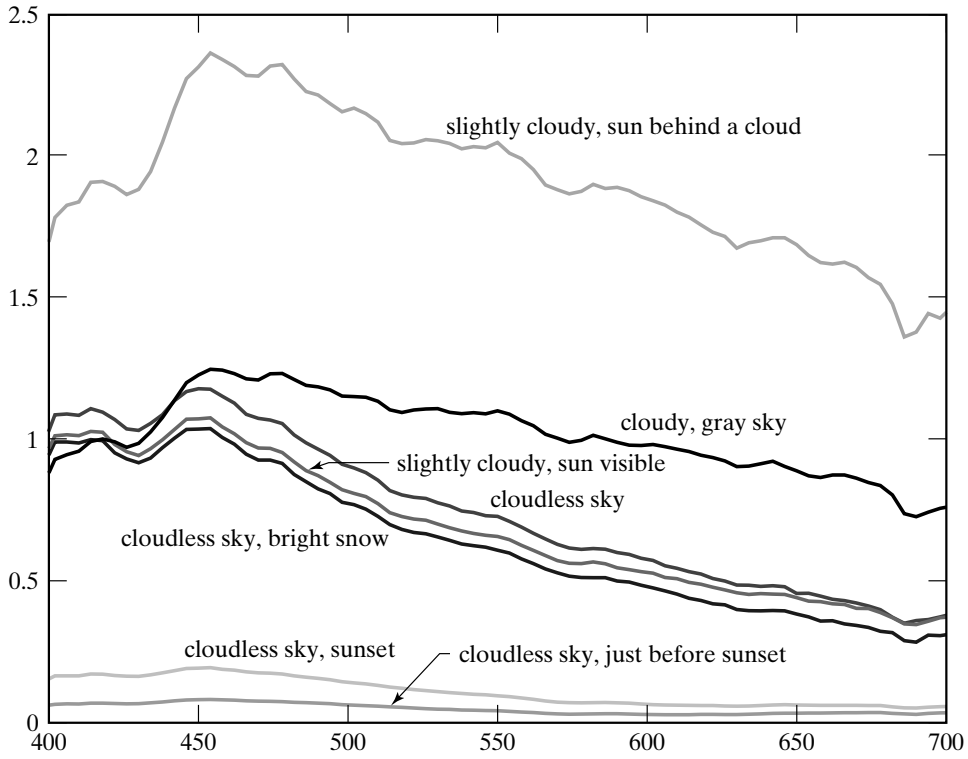


FIGURE 18.1: *Daylight has different amounts of power at different wavelengths. These plots show the spectral energy density of daylight measured at different times of day and under different conditions. The figure plots relative power against wavelength for wavelengths from 400 nm to 700 nm for a series of seven different daylight measurements, made by Jussi Parkkinen and Pertti Silfsten, of daylight illuminating a sample of barium sulphate (which gives a high reflectance white surface). At the foot of the plot, we show the names used for spectral colors of the relevant wavelengths. Plot from data obtainable at* `http://www.it.lut.fi/ip/research/color/database/database.html`.

Write T for the test light, an equals sign for a match, the weights—which are non-negative—as w_i, and the primaries P_i. A match can then be written in an algebraic form as

$$T = w_1 P_1 + w_2 P_2 + \ldots,$$

meaning that test light T matches the particular mixture of primaries given by $(w_1, w_2, \ldots)$. The situation is simplified if *subtractive matching* is allowed. In subtractive matching, the viewer can add some amount of some primaries to the *test light* instead of to the match. This can be written in algebraic form by allowing the weights in the expression above to be negative.

Under these conditions, most observers require only three primaries to match a test light. This phenomenon is known as the principle of *trichromacy*. However,

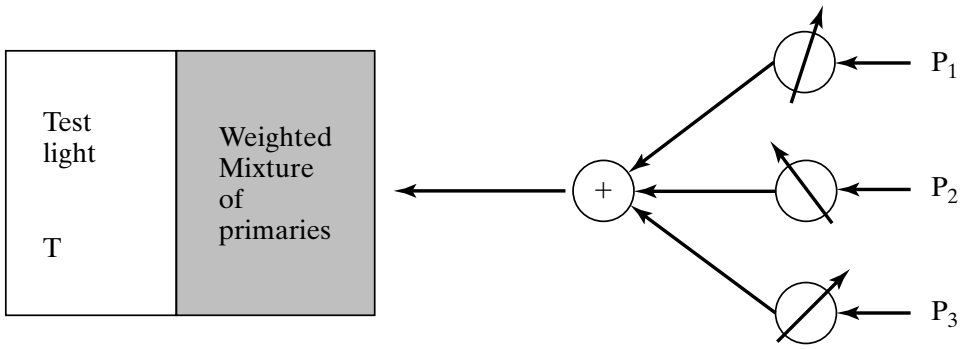


FIGURE 18.2: *Human perception of color can be studied by asking observers to mix colored lights to match a test light shown in a split field. The drawing shows the outline of such an experiment. The observer sees a test light T and can adjust the amount of each of three primaries in a mixture displayed next to the test light. The observer is asked to adjust the amounts so that the mixture looks the same as the test light. The mixture of primaries can be written as $w_1P_1 + w_2P_2 + w_3P_3$; if the mixture matches the test light, then we write $T = w_1P_1 + w_2P_2 + w_3P_3$. It is a remarkable fact that for most people three primaries are sufficient to achieve a match for many colors, and three primaries are sufficient for all colors if we allow subtractive matching (i.e., some amount of some of the primaries is mixed with the test light to achieve a match). Some people require fewer primaries. Furthermore, most people choose the same mixture weights to match a given test light.*

there are some caveats. First, subtractive matching must be allowed; second, the primaries must be independent, meaning that no mixture of two of the primaries may match a third. There is now clear evidence that trichromacy occurs because there are three distinct types of color transducer in the eye [**?**, **?**]. Given the same primaries and test light, most observers select the *same* mixture of primaries to match that test light, because most people have the same types of color receptor.

Matching is (to an accurate approximation) linear. This yields *Grassman's laws*. First, if we mix two test lights, then mixing the matches will match the result. Second, if two test lights can be matched with the same set of weights, then they will match each other. Finally, matching is linear: a test light with doubled intensity is matched by doubling the weights.

Given the same test light and set of primaries, most people use the same set of weights to match the test light. This, trichromacy, and Grassman's laws are about as true as any law covering biological systems can be. The exceptions include the following:

- people with too few kinds of color receptor as a result of genetic ill fortune (who may be able to match everything with fewer primaries);
- people with neural problems (who may display all sorts of effects, including a complete absence of the sensation of color);
- some elderly people (whose choice of weights differ from the norm because of

the development of macular pigment in the eye);

- very bright lights (whose hue and saturation look different from less bright versions of the same light);

- and very dark conditions (where the mechanism of color transduction is somewhat different than in brighter conditions).

18.1.2 Color Receptors

Human retinas contain two types of cell that are sensitive to light, differentiated by their shape. The light-sensitive region of a *cone* has a roughly conical shape, whereas that in a *rod* is roughly cylindrical. Cones largely dominate color vision. Cones are somewhat less sensitive to light than rods are, meaning that in low light, color vision is poor.

Trichromacy occurs because there are (usually!) three distinct types of cone in the eye that mediate color perception. Each of these types turns incident light into neural signals. The *principle of univariance* states that the activity of these cones is of one kind (i.e., they respond strongly or weakly, but do not signal the wavelength of the light falling on them). Univariance is a powerful idea because it gives us a good and simple model of human reaction to colored light: two lights will match if they produce the same receptor responses, *whatever their spectral energy densities*.

Write p_k for the response of the kth type of receptor, $\sigma_k(\lambda)$ for its sensitivity, $E(\lambda)$ for the light arriving at the receptor, and Λ for the range of visible wavelengths. We can obtain the overall response of a receptor by adding up the response to each separate wavelength in the incoming spectrum so that

$$p_k = \int_{\Lambda} \sigma_k(\lambda) E(\lambda) d\lambda.$$

Comparing color matching data for normal observers and those lacking one cone type yields the sensitivities of the three different kinds of cone to different wavelengths (Figure 18.3). The three types of cone are properly called *S cones*, *M cones*, and *L cones* (for their peak sensitivity being to short-, medium-, and long-wavelength light, respectively).

18.2 THE PHYSICS OF COLOR

Light sources can produce different amounts of light at different wavelengths, so incandescent lights look orange or yellow, and fluorescent lights look bluish. For most diffuse surfaces, albedo depends on wavelength, so that some wavelengths may be largely absorbed and others largely reflected. This means that most surfaces will look colored when lit by a white light. The light reflected from a colored surface is affected by both the color of the light falling on the surface, and by the surface. For example, a white surface lit by red light will reflect red light, and a red surface lit by white light will also reflect red light.

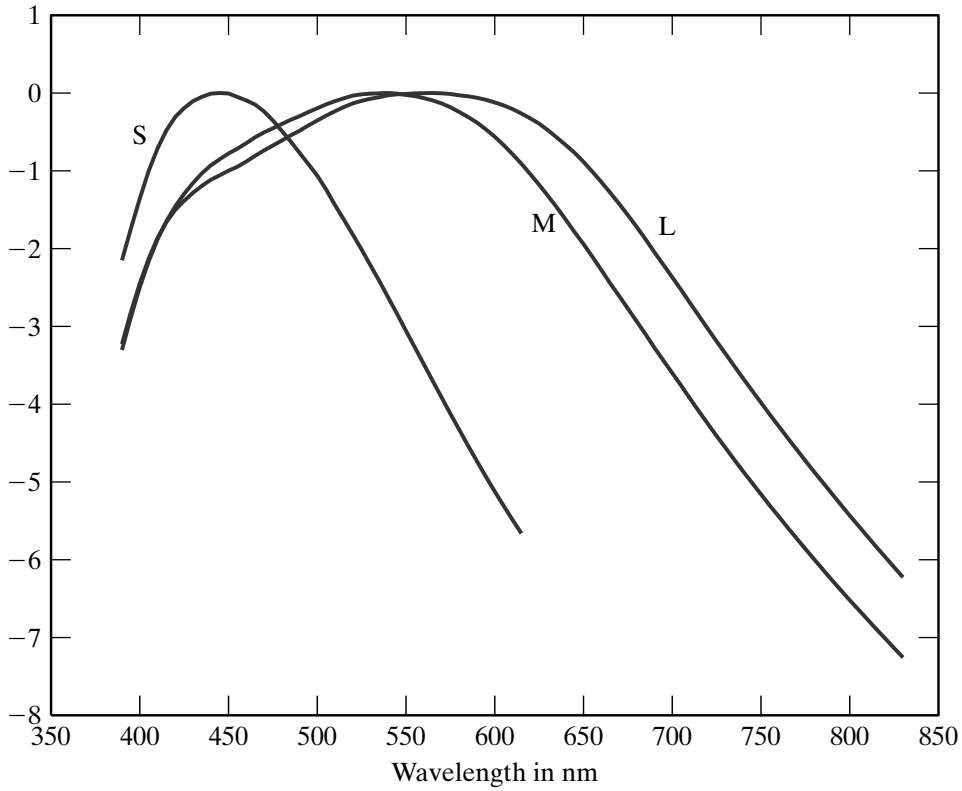


FIGURE 18.3: *There are three types of color receptor in the human eye, usually called* cones. *These receptors respond to all photons in the same way, but in different amounts. The figure shows the log of the relative spectral sensitivities of the three kinds of color receptor in the human eye, plotted against wavelength. On the wavelength axis, we have shown the color name usually associated with lights which contain energy only at that wavelength. The first two receptors—properly named the* long- *and* medium-wavelength *receptors—have peak sensitivities at quite similar wavelengths. The third receptor (*short-wavelength *receptor) has a different peak sensitivity. The response of a receptor to incoming light can be obtained by summing the product of the sensitivity and the spectral energy density of the light over all wavelengths. Notice that each receptor responds to quite a broad range of wavelengths. This means that human observers must perceive color by comparing the response of the receptors to one another, and that there must be many spectral energy densities that cannot be distinguished by humans.* Figures plotted from data disseminated by the Color and Vision Research Laboratories database, compiled by Andrew Stockman and Lindsey Sharpe, and available at `http://www.cvrl.org/`.

18.2.1 The Color of Light Sources

A patch of surface outdoors during the day is illuminated both by light that comes directly from the sun—usually called *daylight*—and by light from the sun that has

been scattered by the air (sometimes called *skylight* or *airlight*; the presence of clouds or snow can add other, important, phenomena). The color of daylight varies with time of day (Figure 18.1) and time of year.

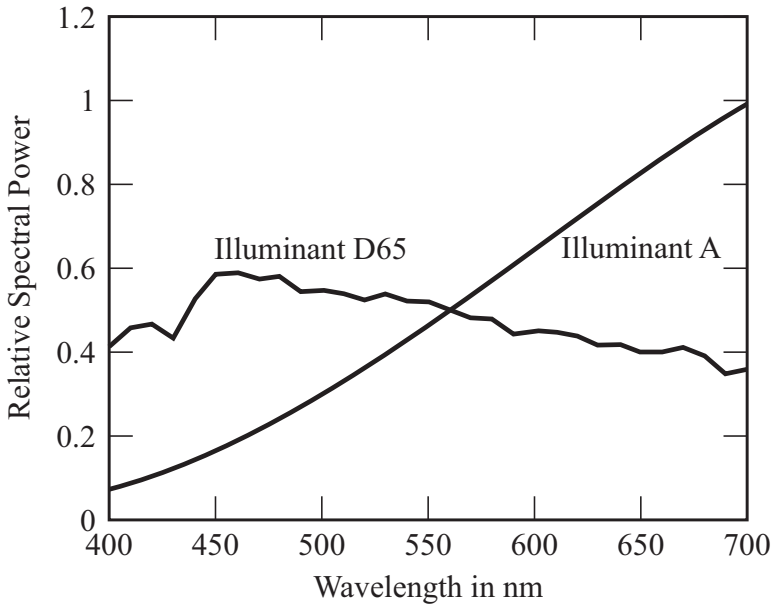


FIGURE 18.4: *There is a variety of illuminant models; the graph shows the relative spectral power distribution of two standard CIE models, illuminant A—which models the light from a 100W Tungsten filament light bulb, with color temperature 2800K—and illuminant D-65—which models daylight.* Figure plotted from data available at `http://www.cvrl.org/`.

For clear air, the intensity of radiation scattered by a unit volume depends on the fourth power of the frequency; this means that light of a long wavelength can travel much farther before being scattered than light of a short wavelength (this is known as *Rayleigh scattering*). This means that, when the sun is high in the sky, blue light is scattered out of the ray from the sun to the earth—meaning that the sun looks yellow—and can scatter from the sky into the eye—meaning that the sky looks blue. There are standard models of the spectral energy density of the sky at different times of day and latitude. Surprising effects occur when there are fine particles of dust in the sky (the larger particles cause much more complex scattering effects, usually modeled rather roughly by the *Mie scattering* model, described in **?** or in **?**).

Artificial Illumination

Typical artificial light sources are commonly of a small number of types:

- An *incandescent light* contains a metal filament that is heated to a high temperature. The spectrum roughly follows the black-body law (Section 18.2.1),

but the melting temperature of the element limits the color temperature of the light source, so the light has a reddish tinge.

- A *fluorescent light* works by generating high-speed electrons that strike gas within the bulb. The gas releases ultraviolet radiation, which causes phosphors coating the inside of the bulb to fluoresce. Typically the coating consists of three or four phosphors, which fluoresce in quite narrow ranges of wavelengths. Most fluorescent bulbs generate light with a bluish tinge, but some bulbs mimic natural daylight (Figure 18.5).

- In some bulbs, an arc is struck in an atmosphere consisting of gaseous metals and inert gases. Light is produced by electrons in metal atoms dropping from an excited state to a lower energy state. Typical of such lamps is strong radiation at a small number of wavelengths, which correspond to particular state transitions. The most common cases are *sodium arc lamps* and *mercury arc lamps*. Sodium arc lamps produce a yellow-orange light extremely efficiently and are quite commonly used for freeway lighting. Mercury arc lamps produce a blue-white light and are often used for security lighting.

- **TODO:** LED lights

Figure 18.5 shows a sample of spectra from different light bulbs.

Black Body Radiators

One useful abstraction is the *black body*, a body that reflects no light. A heated black body emits electromagnetic radiation. The spectral power distribution of this radiation depends only on the temperature of the body. If we write T for the temperature of the body in Kelvins, h for Planck's constant, k for Boltzmann's constant, c for the speed of light, and λ for the wavelength, we have

$$E(\lambda) \propto \frac{1}{\lambda^5} \frac{1}{(\exp(hc/k\lambda T) - 1)}.$$

This means that there is one parameter family of light colors corresponding to black body radiators—the parameter being the temperature—and so we can talk about the *color temperature* of a light source. This is the temperature of the black body that looks most similar. At relatively low temperatures, black bodies are red, passing through orange to a pale yellow-white to white as the temperature increases (Figure 18.10 shows this locus). When $hc \gg k\lambda T$, we have $1/(\exp(hc/k\lambda T) - 1) \approx \exp(-hc/k\lambda T)$, so

$$E(\lambda; T) = C\frac{\exp(-hc/k\lambda T)}{\lambda^5}$$

where C is the constant of proportionality; this model is somewhat easier to use than the exact model (Section 19.3.1).

18.2.2 The Color of Surfaces

The color of surfaces is a result of a large variety of mechanisms, including differential absorbtion at different wavelengths, refraction, diffraction, and bulk scattering

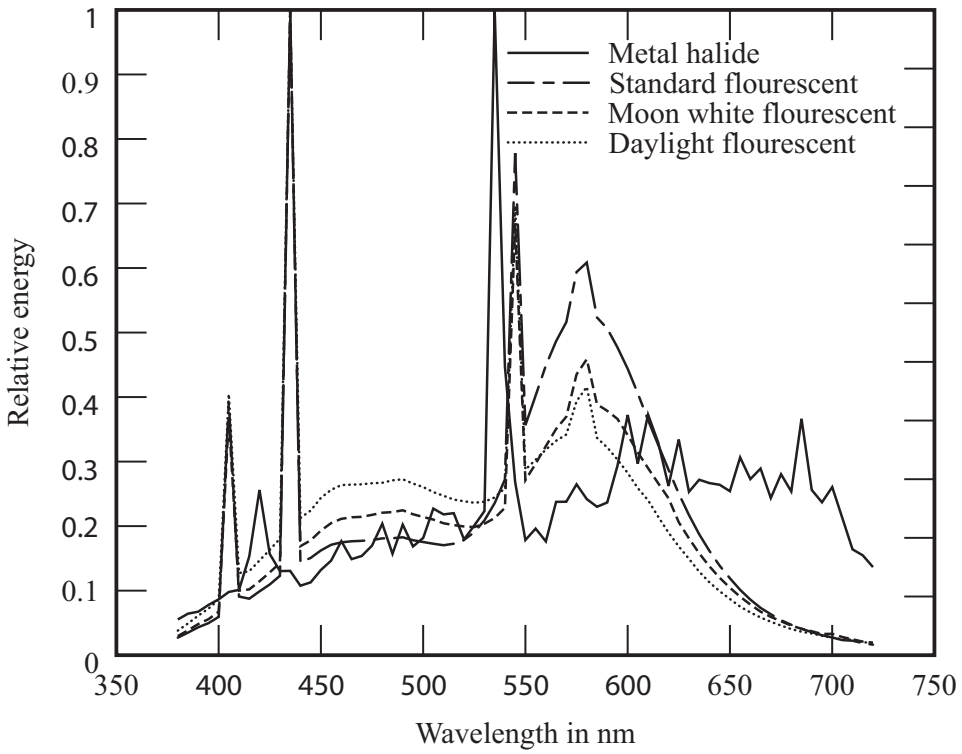


FIGURE 18.5: *The relative spectral power distribution of four different lamps from the Mitsubishi Electric Corporation. Note the bright, narrow bands that come from the flourescing phosphors in the fluorescent lamp.* The figure was plotted from data made available by the Coloring Info Pages at `http://www.colorpro.com/info/data/lamps.html`; the data was measured by Hiroaki Sugiura.

(for more details, see, for example **?**, **?**, **?**, or **?**). We can model surfaces as having a diffuse and a specular component, each of which has a wavelength-dependent albedo. The wavelength-dependent diffuse albedo is sometimes referred to as the *spectral reflectance* (sometimes abbreviated to *reflectance* or, less commonly, *spectral albedo*). Figures 18.6 and 18.7 show examples of spectral reflectances for a number of different natural objects.

There are two color regimes for specular reflection. If the surface is dielectric (i.e., does not conduct electricity), specularly reflected light tends to take the color of the light source. If the surface is a conductor, the specular albedo may depend quite strongly on wavelength, so that white light may result in colored specularities.

18.3 REPRESENTING COLOR

Describing colors accurately is a matter of great commercial importance. Many products are closely associated with specific colors—for example, the golden arches,

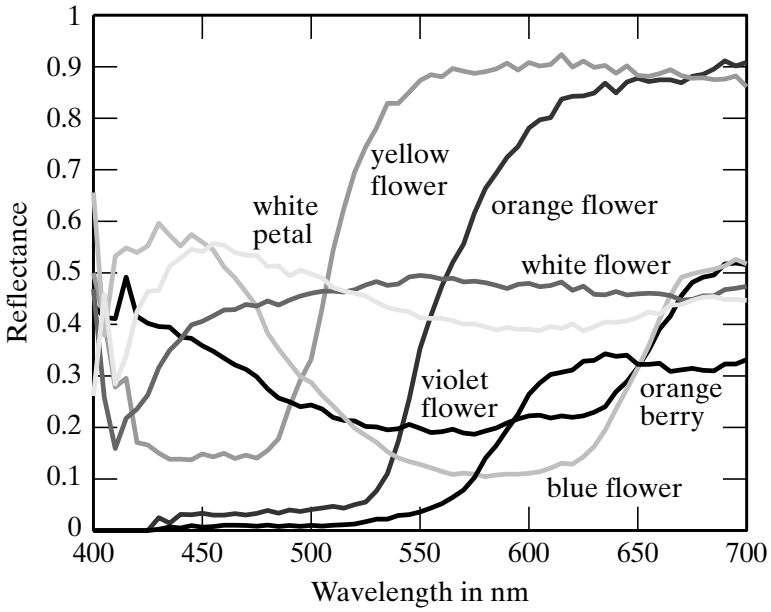


FIGURE 18.6: *Spectral albedoes for a variety of natural surfaces measured by Esa Koivisto, Department of Physics, University of Kuopio, Finland, plotted against wavelength in nanometers.* These figures were plotted from data available at `http://www.it.lut.fi/ip/research/color/database/database.html`.

the color of various popular computers, and the color of photographic film boxes—and manufacturers are willing to go to a great deal of trouble to ensure that different batches have the same color. This requires a standard system for talking about color. Simple color names are insufficient because relatively few people know many color names, and most people are willing to associate a large variety of colors with a given name. There are many linear and non-linear color spaces (**?** is a good reference). Generally, the choice of color space is driven by application. One important consideration is that, some color representations are more redundant than others. For example, the R, G and B layers in an RGB image are typically very similar, but a linear transformation can decorrelate these layers quite well (exercises). Redundancy is obviously a nuisance if one wishes to compress images. It is also a nuisance if one wishes to synthesize images, because the synthesis process must produce layers that are very, but not exactly, like each other. Another important consideration is consistency with perception. In some color spaces a small change in coordinates can result in a large change in perceived color. This is a problem if one wishes to control errors in color, for example, when mapping or synthesizing colors.

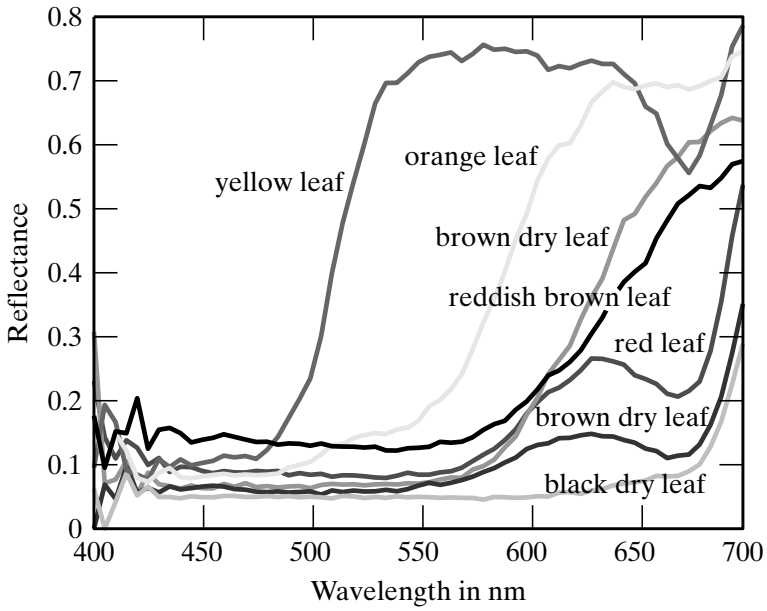


FIGURE 18.7: *Spectral albedoes for a variety of natural surfaces measured by Esa Koivisto, Department of Physics, University of Kuopio, Finland, plotted against wavelength in nanometers.* These figures were plotted from data available at `http://www.it.lut.fi/ip/research/color/database/database.html`.

18.3.1 Additive Linear Color Spaces

There is a natural mechanism for representing color: agree on a standard set of primaries, and then describe any colored light by the three values of weights that people would use to match the light using those primaries. This approach extends to give a representation for surface colors as well if we use a standard light for illuminating the surface (and if the surfaces are equally clean, etc.). Performing a matching experiment each time we wish to describe a color can be practical (paint stores will mix paint to match a flake, for example), but a simpler procedure is available.

A *linear color space* is defined by a choice of primaries P_1, P_2, and P_3. These may not be physically realizable. One then obtains a set of *color matching functions* from the primaries by experiment. The color matching functions $f_1(\lambda)$, $f_2(\lambda)$, and $f_3(\lambda)$ have the property that, if a source $S(\lambda)$ is matched by $w_1P_1 + w_2P_2 + w_3P_3$, then

$$w_i = \int f_i(\lambda)S(\lambda)d\lambda.$$

There is a form of duality between primaries and color matching functions, so one can obtain a linear color space by constructing the color matching functions and then looking for primaries that produce these color matching functions. A variety of different systems have been standardized by the CIE (the *commission international d'éclairage*, which exists to create standards for such things).

The *CIE XYZ color space* is one quite popular standard. The color matching functions were chosen to be everywhere positive, so that the coordinates of any real light are always positive. It is not possible to obtain CIE X, Y, or Z primaries because for some wavelengths the value of their pectral energy density is negative. However, given color matching functions alone, one can specify the XYZ coordinates of a color and hence describe it.

Linear color spaces allow a number of useful graphical constructions that are more difficult to draw in three dimensions than in two, so it is common to intersect the XYZ space with the plane $X + Y + Z = 1$ (as shown in Figure 18.8) and draw the resulting figure using coordinates

$$(x, y) = \left(\frac{X}{X + Y + Z}, \frac{Y}{X + Y + Z} \right).$$

This space, which is often referred to as the *CIE xy color space* is shown in Figure 18.10. CIE xy is widely used in vision and graphics textbooks and in some applications, but is usually regarded by professional colorimetrists as out of date.

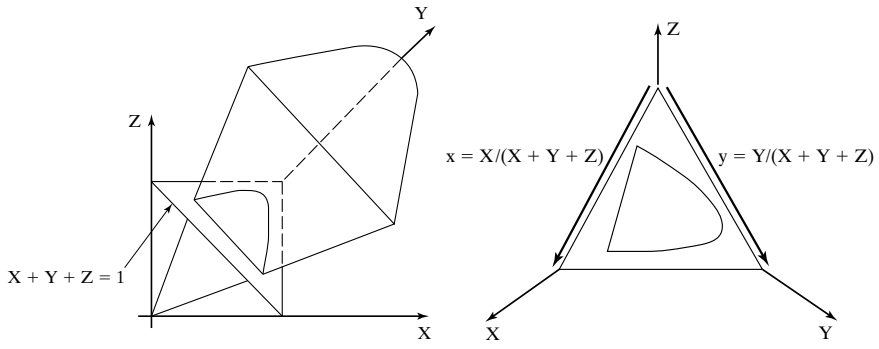


FIGURE 18.8: *The volume of all visible colors in the CIE XYZ coordinate space is a cone whose vertex is at the origin. Usually it is easier to suppress the brightness of a color, which we can do because, to a good approximation, perception of color is linear, and we do this by intersecting the cone with the plane* $X + Y + Z = 1$ *to get the CIE xy space shown in Figure 18.10.*

The *RGB color space* is a linear color space that formally uses single wavelength primaries (645.16 nm for R, 526.32 nm for G, and 444.44 nm for B; see Figure **??**). Informally, RGB uses whatever phosphors a monitor has as primaries. Available colors are usually represented as a unit cube—usually called the *RGB cube*—whose edges represent the R, G, and B weights. The cube is drawn in Figure 18.11.

The *opponent color space* is a linear color space derived from RGB. There is evidence that there are three kinds of color system in primates (e.g., see **?**; **?**). The oldest responds to intensity (i.e., light-dark comparisons). A more recent, but still old, color system compares blue with yellow. The most recent color system

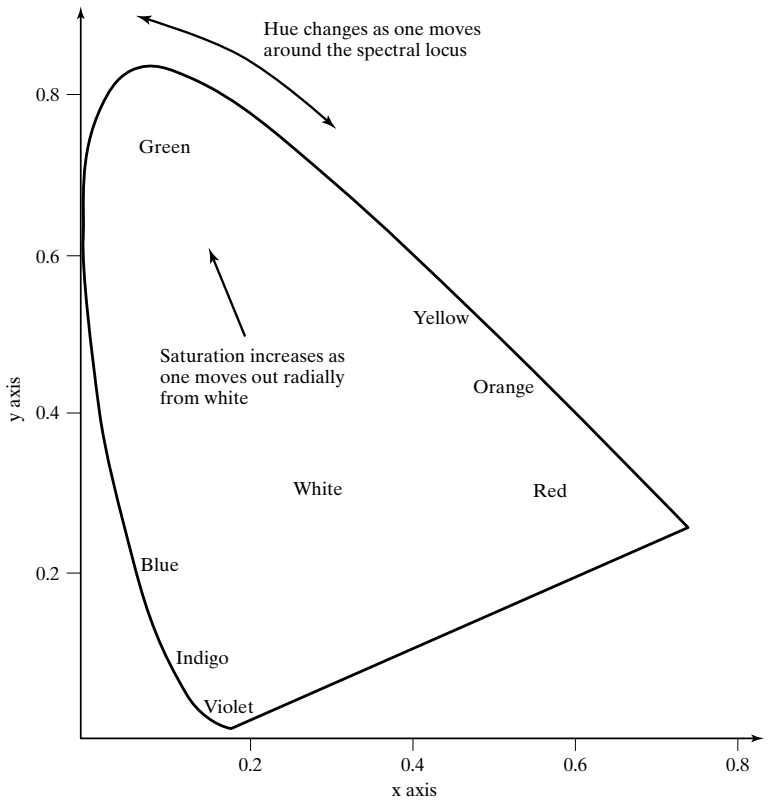


FIGURE 18.9: *The figure shows a constant brightness section of the standard 1931 standard CIE xy color space, with color names marked on the diagram. Generally, colors that lie farther away from the neutral point are more saturated—the difference between deep red and pale pink—and hue—the difference between green and red—as one moves around the neutral point.*

compares red with green. In some applications, it is useful to use a comparable representation. This can be obtained from RGB coordinates using $I = (R+G+B)/3$ for intensity, $(B - (R + G)/2)/I$ for the blue-yellow comparison (sometimes called B-Y), and $(R - G)/I$ for the red-green comparison (sometimes called R-G). Notice that B-Y (resp. R-G) is positive for strongly blue (resp. red) colors and negative for strongly yellow (resp. green) colors, and is intensity independent.

There are two useful constructions that work in linear color spaces, but are most commonly applied in CIE xy. First, because the color spaces are linear, and color matching is linear, all colors that can be obtained by mixing two primaries A and B lie on the line segment joining them plotted on the color space. Second, all colors that can be obtained by mixing three primaries A, B, and C lie in the triangle formed by the three primaries plotted on the color space. Typically, we use this construction to determine the set of colors (or *gamut*) that a set of monitor phosphors can display.

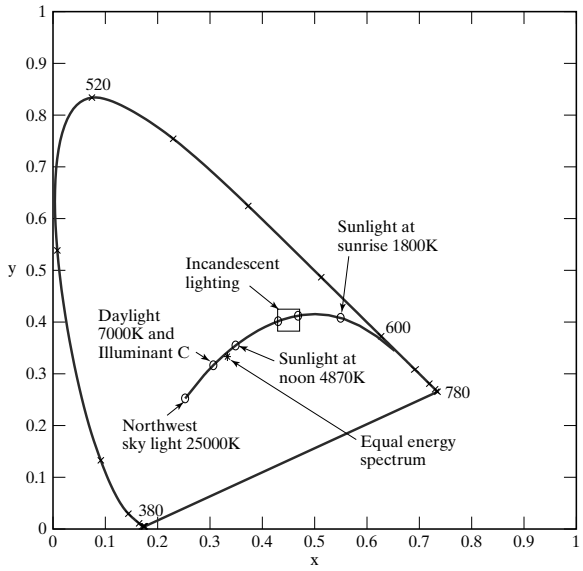


FIGURE 18.10: *The figure shows a constant brightness section of the standard 1931 standard CIE xy color space. This space has two coordinate axes. The curved boundary of the figure is often known as the* spectral locus*; it represents the colors experienced when lights of a single wavelength are viewed. The figure shows a locus of colors due to black-body radiators at different temperatures and a locus of different sky colors. Near the center of the diagram is the neutral point, the color whose weights are equal for all three primaries. CIE selected the primaries so that this light appears achromatic. Generally, colors that lie farther away from the neutral point are more saturated—the difference between deep red and pale pink—and hue—the difference between green and red—as one moves around the neutral point.*

18.3.2 Subtractive Mixing and Inks

Intuition from one's finger-painting days suggests that the primary colors should be red, yellow, and blue, and that yellow and blue mix to make green. The reason this intuition doesn't apply to monitors is that paints involve pigments—which mix subtractively—rather than lights. Pigments can behave in quite complex ways, but the simplest model is that pigments remove color from incident light, which is reflected from paper. Thus, red ink is really a dye that absorbs green and blue light—incident red light passes through this dye and is reflected from the paper. This is *subtractive color mixing*.

Color spaces for this kind of mixing can be quite complicated. In the simplest case, mixing is linear (or reasonably close to linear), and the *CMY space* applies. In this space, there are three primaries: *cyan* (a blue-green color), *magenta* (a purplish color), and *yellow*. These primaries should be thought of as subtracting a light primary from white light; cyan is $W - R$ (white − red); magenta is $W - G$ (white − green), and yellow is $W - B$ (white − blue). Now the appearance of mixtures can be evaluated by reference to the RGB color space. For example, cyan

and magenta mixed give

$$(W - R) + (W - G) = R + G + B - R - G = B,$$

that is, blue. Notice that $W + W = W$ because we assume that ink cannot cause paper to reflect more light than it does when uninked. Practical printing devices use at least four inks (cyan, magenta, yellow, and black) because mixing color inks leads to a poor black, it is difficult to ensure good enough registration between the three color inks to avoid colored haloes around text, and color inks tend to be more expensive than black inks. One reason that fingerpainting is hard is that the color resulting from mixing paints can be quite hard to predict. This is because the outcome depends very strongly on details such as the specific pigment in the paint, the size of pigment particles, the medium in which the pigment is suspended, the care put into stirring the mixture, and similar parameters; usually, we do not have enough detailed information to use a full physical model of these effects. A useful study of this difficult topic is [**?**].

18.3.3 Non-linear Color Spaces

The coordinates of a color in a linear space may not necessarily encode properties that are common in language or are important in applications. Useful color terms include: *hue*, the property of a color that varies in passing from red to green; *saturation*, the property of a color that varies in passing from red to pink; and *brightness* (sometimes called *lightness* or *value*, the property that varies in passing from black to white. Another difficulty with linear color spaces is that the individual coordinates do not capture human intuitions about the topology of colors; it is a common intuition that hues form a circle, in the sense that hue changes from red through orange to yellow, and then green, and from there to cyan, blue, purple, and then red again. Another way to think of this is to picture local hue relations: red is next to purple and orange; orange is next to red and yellow; yellow is next to orange and green; green is next to yellow and cyan; cyan is next to green and blue; blue is next to cyan and purple; and purple is next to blue and red. Each of these local relations works, and globally they can be modeled by laying hues out in a circle. This means that no individual coordinate of a linear color space can model hue, because that coordinate has a maximum value that is far away from the minimum value.

Applying a non-linear transformation to the RGB space can produce a color space that respects these relations. The *HSV space* (for hue, saturation, and value), is obtained by looking down the center axis of the RGB cube. Because RGB is a linear space, brightness—called *value* in HSV—varies with scale out from the origin. We can flatten the RGB cube to get a 2D space of constant value and for neatness deform it to be a hexagon. This gets the structure shown in Figure 18.11, where hue is given by an angle that changes as one goes round the neutral point and saturation changes as one moves away from the neutral point.

In some applications, it is important to know whether a color difference would be noticeable to a human viewer. One can determine *just noticeable differences* by modifying a color shown to observers until they can only just tell it has changed in a comparison with the original color. With an appropriate choice of non-linear

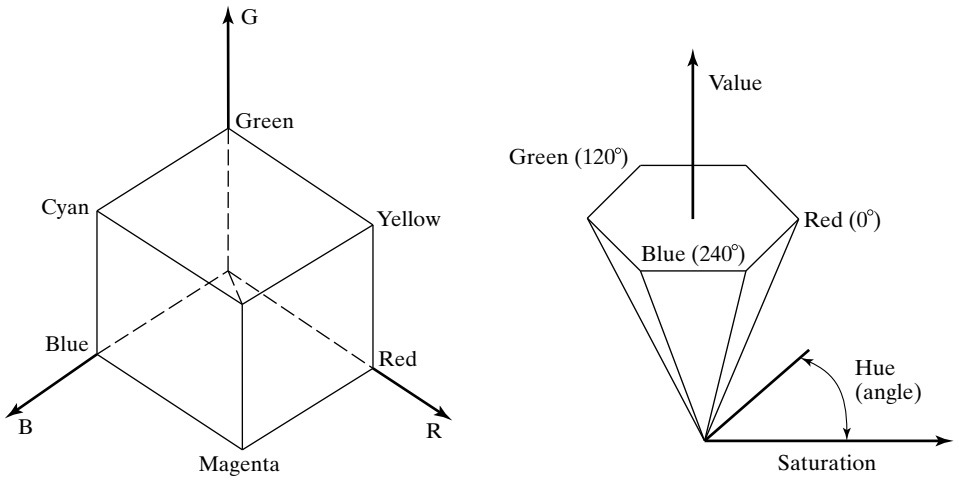


FIGURE 18.11: *On the* **left**, *we see the RGB cube; this is the space of all colors that can be obtained by combining three primaries (R, G, and B—usually defined by the color response of a monitor) with weights between zero and one. It is common to view this cube along its neutral axis—the axis from the origin to the point (1, 1, 1)—to see a hexagon. This hexagon codes hue (the property that changes as a color is changed from green to red) as an angle, which is intuitively satisfying. On the* **right**, *we see a cone obtained from this cross-section, where the distance along a generator of the cone gives the value (or brightness) of the color, the angle around the cone gives the hue, and the distance out gives the saturation of the color.*

transformation applied to linear color coordinates, one can find a *uniform color space*. In such a space, if the distance in coordinate space is below some threshold, a human observer would not be able to tell the colors apart.

A uniform space can be obtained from CIE XYZ using a projective transformation to obtain the *CIE u′v′ space CIE u'v' space*. The coordinates are:

$$(u', v') = \left(\frac{4X}{X + 15Y + 3Z}, \frac{9Y}{X + 15Y + 3Z} \right).$$

Generally, the distance between coordinates in $u\prime$, $v\prime$ space is a fair indicator of the significance of the difference between two colors. Of course, this omits differences in brightness.

CIE LAB is now almost universally the most popular uniform color space. Coordinates of a color in LAB are obtained as a non-linear mapping of the XYZ

coordinates:

$$L^* = 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16$$
$$a^* = 500\left[\left(\frac{X}{X_n}\right)^{\frac{1}{3}} - \left(\frac{Y}{Y_n}\right)^{\frac{1}{3}}\right]$$
$$b^* = 200\left[\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - \left(\frac{Z}{Z_n}\right)^{\frac{1}{3}}\right]$$

Here X_n, Y_n, and Z_n are the X, Y, and Z coordinates of a reference white patch. The reason to care about the LAB space is that it is substantially uniform. In some problems, it is important to understand how different two colors will look *to a human observer*, and differences in LAB coordinates give a good guide.

CHAPTER 19

Using Color Models

19.1 SIMPLE INFERENCE FROM SHADING

19.1.1 Radiometric Calibration and High Dynamic Range Images

The intensity of light travelling through a point in space in some direction is represented with a unit known as *radiance*. The intensity of light arriving at a point on a surface averaged over some range of directions is known as *irradiance*. Sensors average the irradiance over the area of a pixel to obtain incoming power E. This power is summed for some time period Δt to obtain the amount of energy the pixel receives. In turn, the energy determines the pixel intensity value reported by the imaging system. A property called *reciprocity* means that the response is a function of $E\Delta t$ alone. In particular, we will get the same outcome if we image one patch of intensity E for time Δt and another patch of intensity E/k for time $k\Delta t$. The actual response that the sensor produces is a function of $E\Delta t$. Determining this function from data is known as *radiometric calibration*.

Radiometric calibration has a number of applications. For example, we might want to compare renderings of a scene with pictures of the scene, and to do that we need to work in real radiometric units and so must calibrate the camera radiometrically. We might want to use pictures of a scene to estimate the lighting in that scene so we can postrender new objects into the scene, which would need to be lit correctly. Again, we would need to use radiometric units and so need to calibrate the camera.

Likely the most important application is *high dynamic range imaging* or *HDR imaging*. Many scenes have bright spots that are very much brighter than the dark spots. The dynamic range is the ratio of brightest to darkest spot. The camera response function (CRF) is typically somewhat linear over some range, and sharply non-linear near the top and bottom of this range, so that the camera can capture very dark and very light patches without saturation. However, it is quite easy to find scenes where the dynamic range is so big that images in a reasonable camera loses information. Either the brightest points are saturated or the darkest points are very close to zero. In either case, color and relative intensity information is lost. However, if we have multiple images of the scene, obtained with different values of Δt, then we can recover information that would otherwise be lost. Using a small Δt will allow very bright locations to be measured accurately (though mid range locations will be dark, and dark locations will be lost). Using a large Δt will allow very dark locations to be measured accurately (though mid range locations will be bright, and bright locations will be lost). If the CRF is known, then for each location at each Δt_i we can compute the value of $E\Delta t_i$ and so recover E for each location exactly.

Now assume we have multiple registered images, each obtained using a dif-

ferent exposure time. At the i, j'th pixel, we know the image intensity value $I_{ij}^{(k)}$ for the k'th exposure time, we know the value of the k'th exposure time Δt_k, and we know that the intensity of the corresponding surface patch E_{ij} is the same for each exposure, but we do not know the value of E_{ij}. Write the camera response function f, so that

$$I_{ij}^{(k)} = f(E_{ij}\Delta t_k).$$

There are now several possible approaches to solve for f. We could assume a parametric form—say, polynomial—then solve using least squares. Notice that we must solve not only for the parameters of f, but also for E_{ij}. For a color camera, we solve for calibration of each channel separately. **?** have studied the polynomial case in detail. Though the solution is not unique, ambiguous solutions are strongly different from one another, and most cases are easily ruled out. Furthermore, one does not need to know exposure times with exact accuracy to estimate a solution, as long as there are sufficient pixel values; instead, one estimates f from a fixed set of exposure times, then estimates the exposure times from f, and then re-estimates. This procedure is stable.

Alternatively, because the camera response is monotonic, we can work with its inverse $g = f^{-1}$, take logs, and write

$$\log g(I_{ij}^{(k)}) = \log E_{ij} + \log \Delta t_k.$$

We can now estimate the values that g takes at each point and the E_{ij} by placing a smoothness penalty on g. In particular, we minimize

$$\sum_{i,j,k} (\log g(I_{ij}^{(k)}) - (\log E_{ij} + \log \Delta t_k))^2 + \text{smoothness penalty on } g$$

by choice of g. **?** penalize the second derivative of g. Once we have a radiometrically calibrated camera, estimating a high dynamic range image is relatively straightforward. We have a set of registered images, and at each pixel location, we seek the estimate of radiance that predicts the registered image values best. In particular, we assume we know f. We seek an E_{ij} such that

$$\sum_k w(I_{ij})(I_{ij}^{(k)} - f(E_{ij}\Delta t_k))^2$$

is minimized. Notice the weights because our estimate of f is more reliable when I_{ij} is in the middle of the available range of values than when it is at larger or smaller values.

19.1.2 Inferring Lightness and Illumination

If we could estimate the albedo of a surface from an image, then we would know a property of the surface itself, rather than a property of a picture of the surface. Such properties are often called *intrinsic representations*. They are worth estimating, because they do not change when the imaging circumstances change. It might seem that albedo is difficult to estimate, because there is an ambiguity linking albedo and illumination; for example, a high albedo viewed under middling illumination

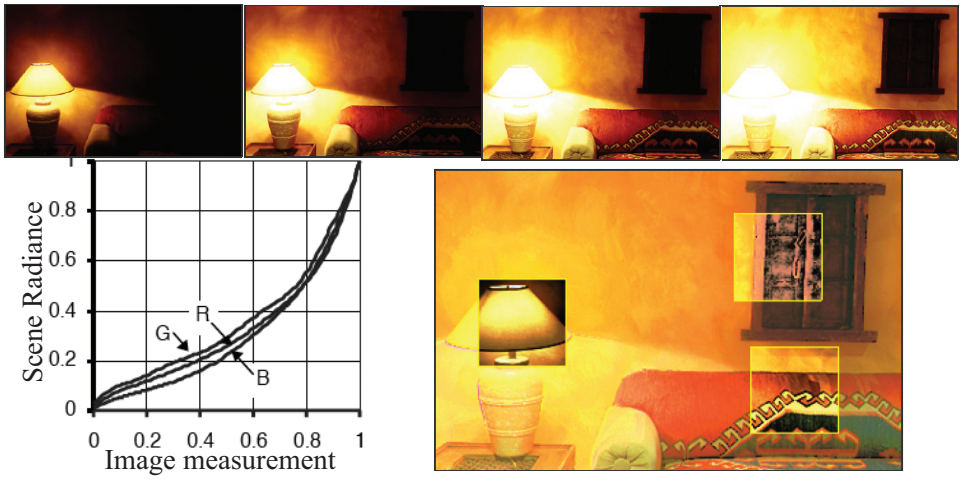


FIGURE 19.1: *It is possible to calibrate the radiometric response of a camera from multiple images obtained at different exposures. The* **top** *row shows four different exposures of the same scene, ranging from darker (shorter shutter time) to lighter (longer shutter time). Note how, in the dark frames, the lighter part of the image shows detail, and in the light frames, the darker part of the image shows detail; this is the result of non-linearities in the camera response. On the* **bottom left**, *we show the inferred calibration curves for each of the R, G, and B camera channels. On the* **bottom right**, *a composite image illustrates the results. The dynamic range of this image is far too large to print; instead, the main image is normalized to the print range. Overlaid on this image are boxes where the radiances in the box have also been normalized to the print range; these show how much information is packed into the high dynamic range image.*

will give the same brightness as a low albedo viewed under bright light. However, humans can report whether a surface is white, gray, or black (the *lightness* of the surface), despite changes in the intensity of illumination (the *brightness*). This skill is known as *lightness constancy*. There is a lot of evidence that human lightness constancy involves two processes: one process compares the brightness of various image patches and uses this comparison to determine which patches are lighter and which darker; the second establishes some form of absolute standard to which these comparisons can be referred (e.g. **?**).

It is worth reviewing early algorithms for estimating lightness briefly, because the underlying principles remain useful. These algorithms were developed in the context of simple scenes. In particular, we assume that the scene is flat and frontal; that surfaces are diffuse, or that specularities have been removed; and that the camera responds linearly. In this case, the camera response C at a point $\mathbf{x}$ is the product of an illumination term, an albedo term, and a constant that comes from the camera gain:

$$C(\mathbf{x}) = k_c I(\mathbf{x})\rho(\mathbf{x}).$$

If we take logarithms, we get

$$\log C(\mathbf{x}) = \log k_c + \log I(\mathbf{x}) + \log \rho(\mathbf{x}).$$

We now make a second set of assumptions:

- First, we assume that albedoes change only quickly over space. This means that a typical set of albedoes will look like a collage of papers of different grays. This assumption is quite easily justified: There are relatively few continuous changes of albedo in the world (the best example occurs in ripening fruit), and changes of albedo often occur when one object occludes another (so we would expect the change to be fast). This means that spatial derivatives of the term $\log \rho(\mathbf{x})$ are either zero (where the albedo is constant) or large (at a change of albedo).
- Second, illumination changes only slowly over space. This assumption is somewhat realistic. For example, the illumination due to a point source will change relatively slowly unless the source is very close, so the sun is a particularly good source for this method, as long as there are no shadows. As another example, illumination inside rooms tends to change very slowly because the white walls of the room act as area sources. This assumption fails dramatically at shadow boundaries, however. We have to see these as a special case and assume that either there are no shadow boundaries or that we know where they are.

These assumptions are sometimes called *Mondrian world* assumptions.

The earliest algorithm is the Retinex algorithm of **?**; this took several forms, most of which have fallen into disuse. The key insight of Retinex is that small gradients are changes in illumination, and large gradients are changes in lightness. We can use this by differentiating the log transform, throwing away small gradients, and integrating the results [**?**]. Doing this, or something like it, is widely known as Retinex. There is a constant of integration missing, so lightness ratios are available, but absolute lightness measurements are not. Figure **??** illustrates the process for a one-dimensional example, where differentiation and integration are easy.

This approach can be extended to two dimensions as well. Differentiating and thresholding is easy: at each point, we estimate the magnitude of the gradient; if the magnitude is less than some threshold, we set the gradient vector to zero; otherwise, we leave it alone. The difficulty is in integrating these gradients to get the log albedo map. The thresholded gradients may not be the gradients of an image because the mixed second partials may not be equal (integrability again; compare with Section 19.1.3).

The problem can be rephrased as a minimization problem: choose the log albedo map whose gradient is most like the thresholded gradient. This is a relatively simple problem because computing the gradient of an image is a linear operation. The x-component of the thresholded gradient is scanned into a vector $\mathbf{p}$, and the y-component is scanned into a vector $\mathbf{q}$. We write the vector representing log-albedo as $\mathbf{l}$. Now the process of forming the x derivative is linear, and so there is some matrix $\mathcal{M}_x$, such that $\mathcal{M}_x\mathbf{l}$ is the x derivative; for the y derivative, we write the corresponding matrix $\mathcal{M}_y$.

Form the gradient of the log of the image
At each pixel, if the gradient magnitude is below
  a threshold, replace that gradient with zero
Reconstruct the log-albedo by solving the minimization
  problem described in the text
Obtain a constant of integration
Add the constant to the log-albedo, and exponentiate

Algorithm 19.1: *Determining the Lightness of Image Patches.*

The problem becomes finding the vector $\mathbf{l}$ that minimizes

$$| \mathcal{M}_x\mathbf{l} - \mathbf{p} |^2 + | \mathcal{M}_y\mathbf{l} - \mathbf{q} |^2 .$$

This is a quadratic minimization problem, and the answer can be found by a linear process. Some special tricks are required because adding a constant vector to $\mathbf{l}$ cannot change the derivatives, so the problem does not have a unique solution. We explore the minimization problem in the exercises.

The constant of integration needs to be obtained from some other assumption. There are two obvious possibilities:

- we can assume that the *brightest patch is white*;
- we can assume that the *average lightness is constant*.

We explore the consequences of these models in the exercises.

More sophisticated algorithms are now available, but there were no quantitative studies of performance until recently. Grosse *et al.* built a dataset for evaluating lightness algorithms, and show that a version of the procedure we describe performs extremely well compared to more sophisticated algorithms [**?**]. The major difficulty with all these approaches is caused by shadow boundaries, which we discuss in Section 19.3.1.

19.1.3 Photometric Stereo: Shape from Multiple Shaded Images

It is possible to reconstruct a patch of surface from a series of pictures of that surface taken under different illuminants. First, we need a camera model. For simplicity, we choose an orthographic camera situated so that the point (x, y, z) in space is imaged to the point (x, y) in the camera (the method can be extended to the other camera models described in Chapter **??**).

In this case, to measure the shape of the surface, we need to obtain the depth to the surface. This suggests representing the surface as $(x, y, f(x, y))$—a representation known as a *Monge patch* after the French military engineer who first used it (Figure 19.3). This representation is attractive because we can determine a unique point on the surface by giving the image coordinates. Notice that to obtain a measurement of a solid object, we would need to reconstruct more than one patch because we need to observe the back of the object.

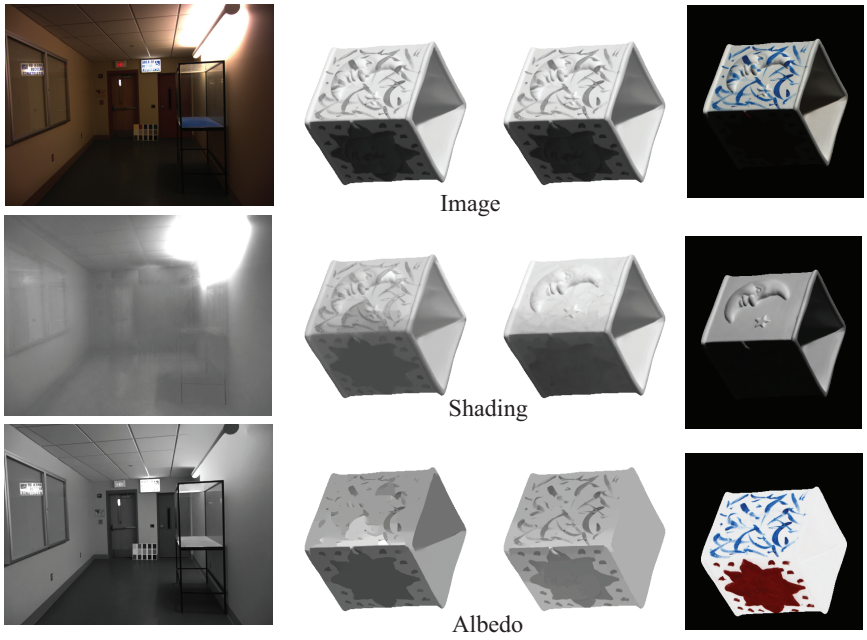


FIGURE 19.2: *Retinex remains a strong algorithm for recovering albedo from images. Here we show results from the version of Retinex described in the text applied to an image of a room (***left***) and an image from a collection of test images due to ?. The* **center-left** *column shows results from Retinex for this image, and the* **center-right** *column shows results from a variant of the algorithm that uses color reasoning to improve the classification of edges into albedo versus shading. Finally, the* **right** *column shows the correct answer, known by clever experimental methods used when taking the pictures. This problem is very hard; you can see that the albedo images still contain some illumination signal.* Part of this figure courtesy Kevin Karsch, U. Illinois.

Photometric stereo is a method for recovering a representation of the Monge patch from image data. The method involves reasoning about the image intensity values for several different images of a surface in a fixed view illuminated by different sources. This method recovers the height of the surface at points corresponding to each pixel; in computer vision circles, the resulting representation is often known as a *height map*, *depth map*, or *dense depth map*.

Fix the camera and the surface in position, and illuminate the surface using a point source that is far away compared with the size of the surface. We adopt a local shading model and assume that there is no ambient illumination (more about this later) so that the brightness at a point $\mathbf{x}$ on the surface is

$$B(\mathbf{x}) = \rho(\mathbf{x})\mathbf{N}(\mathbf{x}) \cdot \mathbf{S}_1,$$

where $\mathbf{N}$ is the unit surface normal and $\mathbf{S}_1$ is the source vector. We can write $B(x, y)$ for the radiosity of a point on the surface because there is only one point on

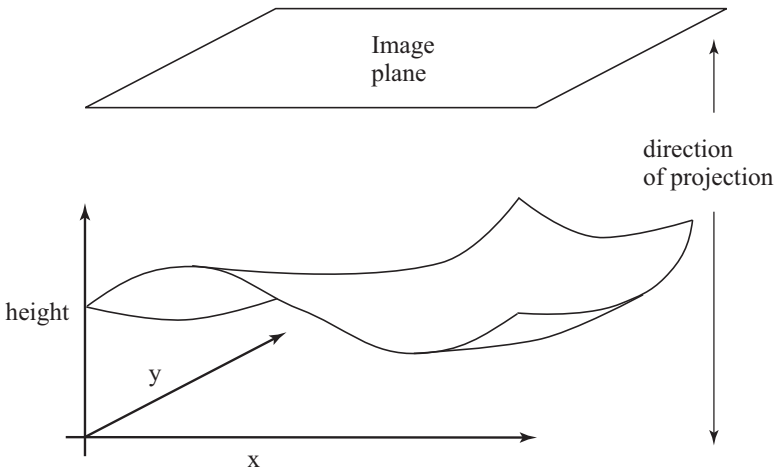


FIGURE 19.3: *A Monge patch is a representation of a piece of surface as a height function. For the photometric stereo example, we assume that an orthographic camera—one that maps (x, y, z) in space to (x, y) in the camera—is viewing a Monge patch. This means that the shape of the surface can be represented as a function of position in the image.*

the surface corresponding to the point (x, y) in the camera. Now we assume that the response of the camera is linear in the surface radiosity, and so have that the value of a pixel at (x, y) is

$$\begin{aligned} I(x,y) &= kB(\mathbf{x}) \\ &= kB(x,y) \\ &= k\rho(x,y)\mathbf{N}(x,y) \cdot \mathbf{S}_1 \\ &= \mathbf{g}(x,y) \cdot \mathbf{V}_1, \end{aligned}$$

where $\mathbf{g}(x, y) = \rho(x, y)\mathbf{N}(x, y)$ and $\mathbf{V}_1 = k\mathbf{S}_1$, where k is the constant connecting the camera response to the input radiance.

In these equations, $\mathbf{g}(x, y)$ describes the surface, and $\mathbf{V}_1$ is a property of the illumination and of the camera. We have a dot product between a vector field $\mathbf{g}(x, y)$ and a vector $\mathbf{V}_1$, which could be measured; with enough of these dot products, we could reconstruct $\mathbf{g}$ and so the surface.

Now if we have n sources, for each of which $\mathbf{V}_i$ is known, we stack each of these $\mathbf{V}_i$ into a known matrix $\mathcal{V}$, where

$$\mathcal{V} = \begin{pmatrix} \mathbf{V}_1^T \\ \mathbf{V}_2^T \\ \cdots \\ \mathbf{V}_n^T \end{pmatrix}.$$

For each image point, we stack the measurements into a vector

$$\mathbf{i}(x, y) = \{I_1(x, y), I_2(x, y), \ldots, I_n(x, y)\}^T.$$

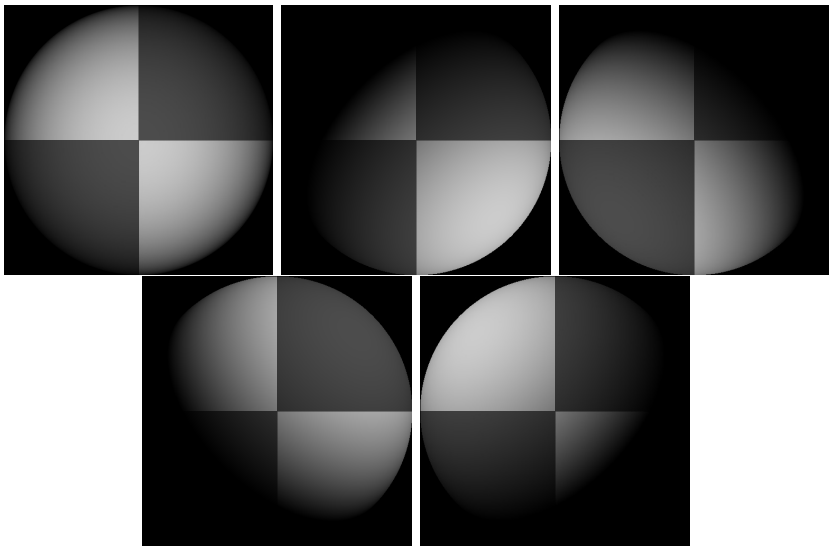


FIGURE 19.4: *Five synthetic images of a sphere, all obtained in an orthographic view from the same viewing position. These images are shaded using a local shading model and a distant point source. This is a convex object, so the only view where there is no visible shadow occurs when the source direction is parallel to the viewing direction. The variations in brightness occuring under different sources code the shape of the surface.*

Notice that we have one vector per image point; each vector contains all the image brightnesses observed at that point for different sources. Now we have

$$\mathbf{i}(x, y) = \mathcal{V}\mathbf{g}(x, y),$$

and $\mathbf{g}$ is obtained by solving this linear system—or rather, one linear system per point in the image. Typically, $n > 3$, so that a least-squares solution is appropriate. This has the advantage that the residual error in the solution provides a check on our measurements.

Substantial regions of the surface might be in shadow for one or the other light (see Figure 19.4). We assume that all shadowed regions are known, and deal only with points that are not in shadow for any illuminant. More sophisticated strategies can infer shadowing because shadowed points are darker than the local geometry predicts.

We can extract the albedo from a measurement of $\mathbf{g}$ because $\mathbf{N}$ is the unit normal. This means that $|\mathbf{g}(x, y)| = \rho(x, y)$. This provides a check on our measurements as well. Because the albedo is in the range zero to one, any pixels where $|\mathbf{g}|$ is greater than one are suspect—either the pixel is not working or $\mathcal{V}$ is incorrect. Figure 19.5 shows albedo recovered using this method for the images shown in Figure 19.4.

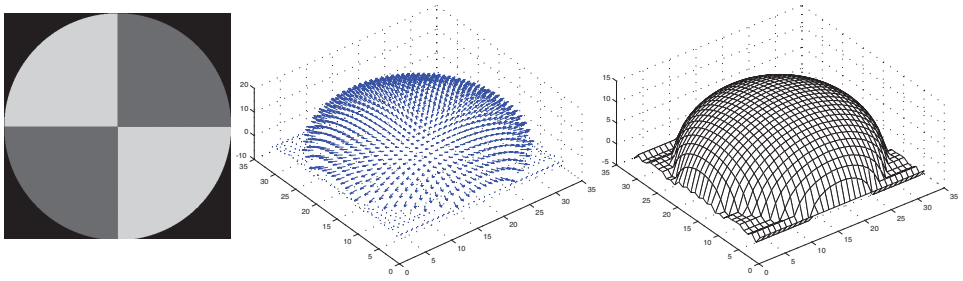


FIGURE 19.5: *The image on the* **left** *shows the magnitude of the vector field* $\mathbf{g}(x, y)$ *recovered from the input data of Figure 19.4 represented as an image—this is the reflectance of the surface. The* **center** *figure shows the normal field, and the* **right** *figure shows the height field.*

We can extract the surface normal from $\mathbf{g}$ because the normal is a unit vector

$$\mathbf{N}(x, y) = \frac{\mathbf{g}(x, y)}{|\mathbf{g}(x, y)|}.$$

Figure 19.5 shows normal values recovered for the images of Figure 19.4.

The surface is $(x, y, f(x, y))$, so the normal as a function of (x, y) is

$$\mathbf{N}(x, y) = \frac{1}{\sqrt{1 + \frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2}} \left\{ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, 1 \right\}^T.$$

To recover the depth map, we need to determine $f(x, y)$ from measured values of the unit normal.

Assume that the measured value of the unit normal at some point (x, y) is $(a(x, y), b(x, y), c(x, y))$. Then

$$\frac{\partial f}{\partial x} = \frac{a(x, y)}{c(x, y)} \text{ and } \frac{\partial f}{\partial y} = \frac{b(x, y)}{c(x, y)}.$$

We have another check on our data set, because

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x},$$

so we expect that

$$\frac{\partial \left(\frac{a(x,y)}{c(x,y)} \right)}{\partial y} - \frac{\partial \left(\frac{b(x,y)}{c(x,y)} \right)}{\partial x}$$

should be small at each point. In principle it should be zero, but we would have to estimate these partial derivatives numerically and so should be willing to accept small values. This test is known as a test of *integrability*, which in vision applications always boils down to checking that mixed second partials are equal.

Obtain many images in a fixed view under different illuminants
Determine the matrix $\mathcal{V}$ from source and camera information

Inferring albedo and normal:
For each point in the image array that is not shadowed
  Stack image values into a vector $\mathbf{i}$
  Solve $\mathcal{V}\mathbf{g} = \mathbf{i}$ to obtain $\mathbf{g}$ for this point
  Albedo at this point is $|\mathbf{g}|$
  Normal at this point is $\frac{\mathbf{g}}{|\mathbf{g}|}$
  p at this point is $\frac{N_1}{N_3}$
  q at this point is $\frac{N_2}{N_3}$
end
Check: is $(\frac{\partial p}{\partial y} - \frac{\partial q}{\partial x})^2$ small everywhere?

Integration:
Top left corner of height map is zero
For each pixel in the left column of height map
  height value = previous height value + corresponding q value
end
For each row
  For each element of the row except for leftmost
    height value = previous height value + corresponding p value
  end
end

Algorithm 19.2: *Photometric Stereo.*

Assuming that the partial derivatives pass this sanity test, we can reconstruct the surface up to some constant depth error. The partial derivative gives the change in surface height with a small step in either the x or the y direction. This means we can get the surface by summing these changes in height along some path. In particular, we have

$$f(x,y) = \oint_C \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) \cdot d\mathbf{l} + c,$$

where C is a curve starting at some fixed point and ending at (x, y), and c is a constant of integration, which represents the (unknown) height of the surface at the start point. The recovered surface does not depend on the choice of curve (exercises). Another approach to recovering shape is to choose the function $f(x, y)$ whose partial derivatives most look like the measured partial derivatives. Figure 19.5 shows the reconstruction obtained for the data shown in Figure 19.4.

Current reconstruction work tends to emphasize geometric methods that reconstruct from multiple views. These methods are very important, but often require feature matching, as we shall see in Chapters **??** and **??**. This tends to mean that it is hard to get very high spatial resolution, because some pixels are consumed

FIGURE 19.6: *Photometric stereo could become the method of choice to capture complex deformable surfaces. On the* **top**, *three images of a garment, lit from different directions, which produce the reconstruction shown on the* **top right**. *A natural way to obtain three different images at the same time is to use a color camera; if one has a red light, a green light, and a blue light, then a single color image frame can be treated as three images under three separate lights. On the* **bottom**, *an image of the garment captured in this way, which results in the photometric stereo reconstruction on the* **bottom right**.

in resolving features. Recall that resolution (which corresponds roughly to the spatial frequencies that can be reconstructed accurately) is not the same as accuracy (which involves a method providing the right answers for the properties it estimates). Feature-based methods are capable of spectacularly accurate reconstructions. Because photometric cues have such spatial high resolution, they are a topic of considerable current interest. One way to use photometric cues is to try and match pixels with the same brightness across different cameras; this is difficult, but produces impressive reconstructions. Another is to use photometric stereo ideas. For some applications, photometric stereo is particularly atractive because one can get reconstructions from a single view direction—this is important, because we cannot always set up multiple cameras. In fact, with a trick, it is possible to get reconstructions from a single frame. A natural way to obtain three different images at the same time is to use a color camera; if one has a red light, a green light and a blue light, then a single color image frame can be treated as three images under three separate lights, and photometric stereo methods apply. In turn, this means that photometric stereo methods could be used to recover high-resolution reconstructions of deforming surfaces in a relatively straightforward way. This is particularly useful when it is difficult to get many cameras to view the object. Figure 19.6 shows one application to reconstructing cloth in video (from **?**), where multiple view reconstruction is complicated by the need to synchronize frames (al-

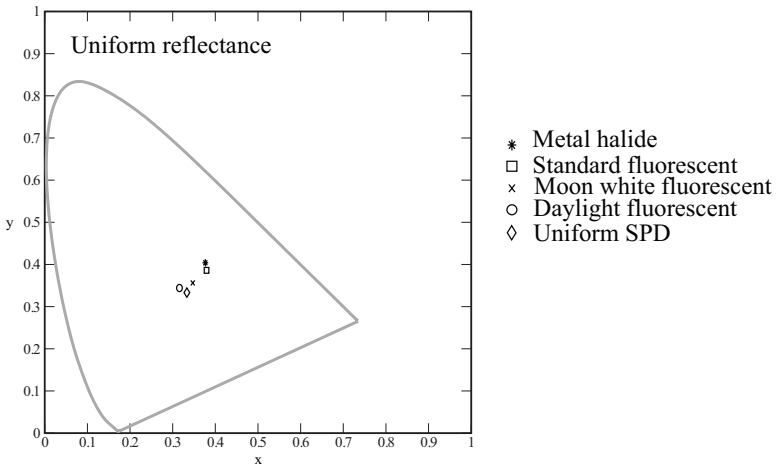


FIGURE 19.7: *Light sources can have quite widely varying colors. This figure shows the color of the four light sources of Figure 18.5, compared with the color of a uniform spectral power distribution, plotted in CIE x, y coordinates.*

ternatives are explored in, for example, **?** or **?**).

19.2 A MODEL OF IMAGE COLOR

Assume that an image pixel is the image of some surface patch. Many phenomena affect the color of this pixel. The main effects are: the camera response to illumination; the choice of camera receptors; the amount of light that arrives at the surface; the color of light arriving at the surface; the dependence of the diffuse albedo on wavelength; and specular components. We have already dealt with the camera response (Section 19.1.1) and we will assume that the camera is linear, or has been radiometrically calibrated. A quite simple model can be used to separate the other effects.

Assume that the surfaces that we are dealing with can be described by the diffuse+specular model. Write $\mathbf{x}$ for a point, λ for wavelength, $E(\mathbf{x}, \lambda)$ for the spectral energy density of the light leaving a surface, $\rho(\mathbf{x}, \lambda)$ for the albedo of a surface as a function of wavelength and position, $S_d(\mathbf{x}, \lambda)$ for the spectral energy density of the light source (which may vary with position; for example, the intensity might change), and $S_i(\mathbf{x}, \lambda)$ for the spectral energy density of interreflected light. Then we have that:

$$\begin{aligned} E(\mathbf{x}, \lambda) &= [\text{diffuse term}] + (\text{specular term}) \\ &= [(\text{direct term}) + (\text{interreflected term})] + (\text{specular term}) \\ &= (\rho(\mathbf{x}, \lambda)(\text{geometric term}))[(S_d(\mathbf{x}, \lambda) + S_i(\mathbf{x}, \lambda))] + (\text{specular term}). \end{aligned}$$

The geometric terms represent how intensity is affected by surface normal. Notice that the diffuse term is affected both by the color of the surface and by the color of the light (examples in Figures 19.7 and 19.8).

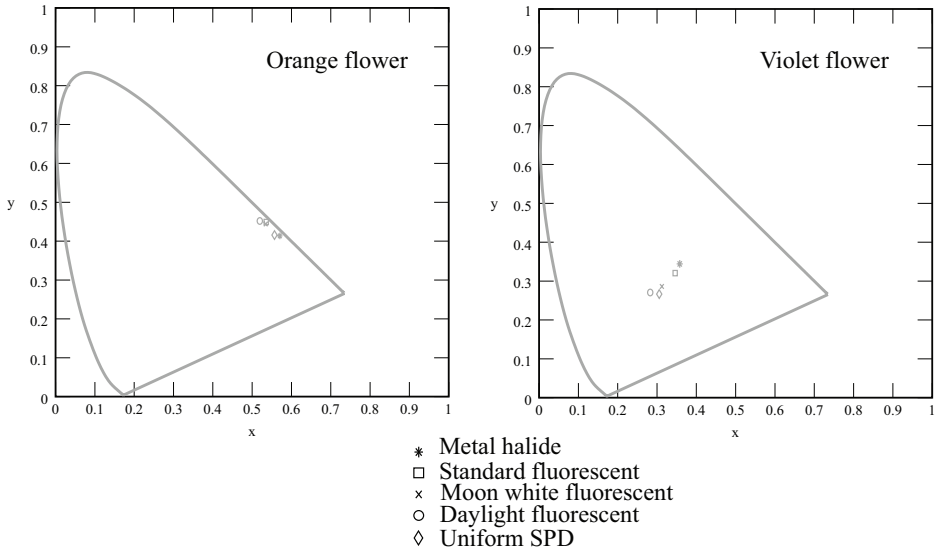


FIGURE 19.8: *The color of a light source affects the color of surfaces lit by the source. The different colors obtained by lighting the violet flower of Figure 18.6 (***left***) and the orange flower of Figure 18.6 (***right***) with the four light sources of Figure 18.5.*

Because the camera is linear, the pixel value at $\mathbf{x}$ is a sum of terms corresponding to each of the terms in $E(\vec{x}, \lambda)$. Write $\mathbf{d}(\mathbf{x})$ for the color taken by a flat patch facing the light source at $\mathbf{x}$ with the same albedo as the actual patch there, $g(\mathbf{x})$ for a geometric term (explained below), $\mathbf{i}(\mathbf{x})$ for the contribution of the interreflected term, $\mathbf{s}(\mathbf{x})$ for the unit intensity color of the specular term, and $g_s(\mathbf{x})$ for a geometric term (explained below). Then we have:

$$\begin{aligned}\mathbf{C}(\mathbf{x}) &= [(\text{direct term}) + (\text{interreflected term})] + (\text{specular term}) \\ &= g_d(\mathbf{x})\mathbf{d}(\mathbf{x}) + \mathbf{i}(\mathbf{x}) + g_s(\mathbf{x})\mathbf{s}(\mathbf{x}).\end{aligned}$$

Generally, to work with this model, we ignore $\mathbf{i}(\mathbf{x})$; we identify and remove specularities, using the methods of Section **??**, and so assume that $\mathbf{C}(\mathbf{x}) = g_d(\mathbf{x})\mathbf{d}(\mathbf{x})$.

19.2.1 The Diffuse Term

There are two diffuse components. One, $\mathbf{i}(\mathbf{x})$, is due to interreflections. Interreflections can be a significant source of colored light. If a large colored surface reflects light onto another surface, that surface's color can change quite substantially. This is an effect that people find hard to see, but which is usually fairly easy to spot in photographs. There are no successful models for removing these color shifts, most likely because they can be very hard to predict. This is because many different surface reflectances can have the same color, so that two surfaces with the same color (but different reflectances) can have quite differently colored interreflections.

The interreflection term is often small, and usually is simply ignored.

Ignoring the interreflected component, the diffuse term is

$$g_d(\mathbf{x})\mathbf{d}(\mathbf{x}).$$

Here $\mathbf{d}(\mathbf{x})$ is the *image* color of an equivalent *flat* surface facing the light source and viewed under the same light. The geometric term, $g_d(\mathbf{x})$, varies relatively slowly over space and accounts for the change in brightness due to the orientation of the surface.

We can model the dependence of $\mathbf{d}(\mathbf{x})$ on the light and on the surface by assuming we are viewing flat, diffuse surfaces, illuminated from infinitely far behind the camera. In this case, there will be no effects due to specularities or to surface orientation. The color of light arriving at the camera will be determined by two factors: first, the wavelength-dependent albedo of the surface that the light is leaving; and second, the wavelength-dependent intensity of the light falling on that surface. If a patch of perfectly diffuse surface with diffuse albedo $\rho(\lambda)$ is illuminated by a light whose spectrum is $S(\lambda)$, the spectrum of the reflected light is $\rho(\lambda)S(\lambda)$. Assume the camera has linear photoreceptors, and the k'th type of photoreceptor has sensitivity $\sigma_k(\lambda)$. If a linear photoreceptor of the kth type sees this surface patch, its response is:

$$p_k = \int_{\Lambda} \sigma_k(\lambda)\rho(\lambda)S(\lambda)d\lambda,$$

where Λ is the range of all relevant wavelengths.

The main engineering parameter here is the photoreceptor sensitivities $\sigma_k(\lambda)$. For some applications such as shadow removal (Section 19.3.1), it can be quite helpful to have photoreceptor sensitivities that are "narrow-band" (i.e., the photoreceptors respond to only one wavelength). Usually, the only practical methods to change the photoreceptor sensitivities are to either put colored filters in front of the camera or to use a different camera. Using a different camera doesn't work particularly well, because manufacturers try to have sensitivities that are reasonably compatible with human receptor sensitivities. They do this so that cameras give about the same responses to colored lights that people do; as a result, cameras tend to have quite similar receptor sensitivities. There are three ways to proceed: install narrow-band filters in front of the lens (difficult to do and seldom justified); apply a transformation to the receptor outputs that makes them behave more like narrow-band receptors (often helpful, if the necessary data are available, **?**;**?**); or assume that they are narrow-band receptors and tolerate any errors that result (generally quite successful).

19.2.2 The Specular Term

The specular component will have a characteristic color, and its intensity will change with position. We can model the specular component as

$$g_s(\mathbf{x})\mathbf{s}(\mathbf{x}),$$

where $\mathbf{s}(\mathbf{x})$ is the unit intensity *image* color of the specular reflection at that pixel, and $g_s(\mathbf{x})$ is a term that varies from pixel to pixel, and models the amount of energy

specularly reflected. We expect $g_s(\mathbf{x})$ to be zero at most points, and large at some points.

The color $\mathbf{s}(\mathbf{x})$ of the specular component depends on the material. Generally, metal surfaces have a specular component that is wavelength dependent and so takes on a characteristic color that depends on the metal (gold is yellow, copper is orange, platinum is white, and osmium is blue or purple). Surfaces that do not conduct—*dielectric surfaces*— have a specular component that is independent of wavelength (e.g., the specularities on a shiny plastic object are the color of the light). Section **??** describes how these properties can be used to find specularities, and to find image regions corresponding to metal or plastic objects.

19.3 INFERENCE FROM COLOR

Our color model supports a variety of inferences. Here we show methods to remove shadows (Section 19.3.1) and to infer surface color (Section 19.3.2).

19.3.1 Shadow Removal Using Color

Lightness methods make the assumption that "fast" edges in images are due to changes in albedo (Section 19.1.2). This assumption is usable, but fails badly at shadows, particularly shadows in sunlight outdoors (Figure 19.10), where there can be a large and fast change of image brightness. People usually are not fooled into believing that a shadow is a patch of dark surface, so must have some method to identify shadow edges. Home users often like editing and improving photographs, and programs that could remove shadows from images would be valuable. A shadow removal program would work something like a lightness method: find all edges, identify the shadow edges, remove those, and then integrate to get the picture back.

There are some cues for finding shadow edges that seem natural, but don't work well. One might assume that shadow edges have very large dynamic range (which albedo edges can't have; see Section 17.2.1), but this is not always the case. One might assume that, at a shadow edge, there was a change in brightness but not in color. It turns out that this is not the case for outdoor shadows, because the lit region is illuminated by yellowish sunlight, and the shadowed region is illuminated by bluish light from the sky, or sometimes by interreflected light from buildings, and so on. However, a really useful cue can be obtained by modelling the different light sources.

We assume that light sources are black bodies, so that their spectral energy density is a function of temperature. We assume that surfaces are diffuse. We use the simplified black-body model of Section 18.2.1, where, writing T for the temperature of the body in Kelvins, h for Planck's constant, k for Boltzmann's constant, c for the speed of light, and λ for the wavelength, we have

$$E(\lambda; T) = C\frac{\exp(-hc/k\lambda T)}{\lambda^5}$$

(C is some constant of proportionality). Now assume that the color receptors each respond only at one wavelength, which we write λ_k for the k'th receptor, so that $\sigma_k(\lambda) = \delta(\lambda - \lambda_k)$. If we view a surface with spectral albedo $\rho(\lambda)$ illuminated by

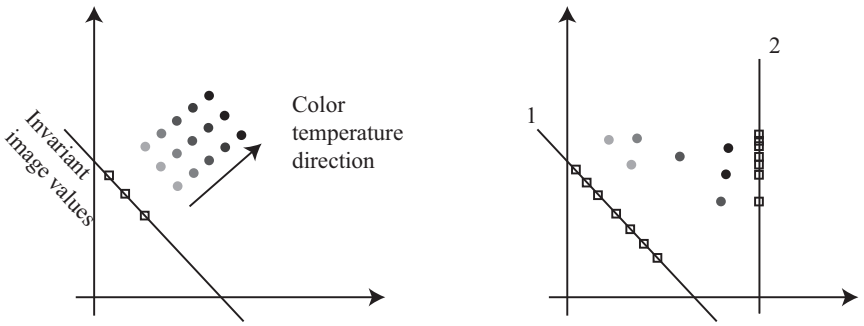


FIGURE 19.9: *Changing the color temperature of the light under which a surface is viewed moves the (c_1, c_2) coordinates of that surface along the color temperature direction (***left***; the different gray patches represent the same surface under different lights). If we now project the coordinates along the (c_1, c_2) direction onto some line, we obtain a value that doesn't change when the illuminant color temperature changes. This is the invariant value for that pixel. Generally, we do not know enough about the imaging system to estimate the color temperature direction. However, we expect to see many different surfaces in each scene; this suggests that the right choice of color temperature direction on the* **right** *is 1 (where there are many different types of surface) rather than 2 (where the range of invariant values is small).*

one of these sources at temperature T, the response of the j'th receptor will be

$$r_j = \int \sigma_j(\lambda)\rho(\lambda)K\frac{\exp(-hc/k\lambda T)}{\lambda^5}d\lambda = K\rho(\lambda_j)\frac{\exp(-hc/k\lambda_j T)}{\lambda_j^5}.$$

We can form a color space that is very well behaved by taking $c_1 = \log(r_1/r_3)$, $c_2 = \log(r_2/r_3)$, because

$$\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} + \frac{1}{T}\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

where $a_1 = \log\rho(\lambda_1) - \log\rho(\lambda_3) + 5\log\lambda_3 - 5\log\lambda_1$ and $b_1 = (hc/k)(1/\lambda_3 - 1/\lambda_1)$ (and a_2, b_2 follow). Notice that, when one changes the color temperature of the source, the (c_1, c_2) coordinates move along a straight line. The direction of the line depends on the sensor, *but not on the surface.* Call this direction the **color temperature direction**. The intercept of the line depends on the surface.

Now consider a world of colored surfaces, and map the image colors to this space. There is a family of parallel lines in this space, whose direction is the color temperature direction. Different surfaces may map to different lines. If we change the color temperature of the illuminant, then each color in this space will move along the color temperature direction, but colors will not move from line to line. We now represent a surface color by its line. For example, we could construct a line through the origin that is perpendicular to color temperature direction, then represent a surface color by distance along this line (Figure 19.9). We can represent each pixel

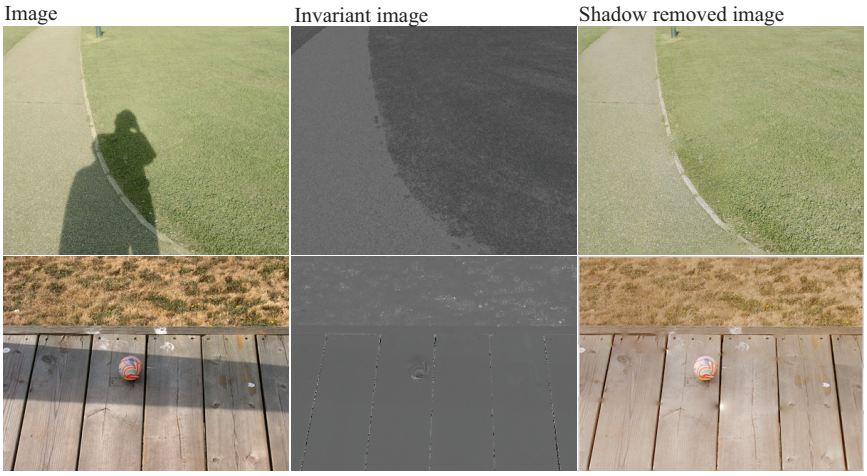


FIGURE 19.10: *The invariant of the text and of Figure 19.9 does not change value when a surface is shadowed. Finlayson* et al. *use this to build a shadow removal system that works by (a) taking image edges; (b) forming an invariant image; then (c) using that invariant image to identify shadow edges; and finally (d) integrating only non-shadow edges to form the result. The results are quite convincing.*

in the image in this space, and in this representation the color image becomes a gray-level image, *where the gray level does not change inside shadows* (because a shadow region just has a different color temperature to the non-shadowed region). **?** calls this the *invariant image*. Any edge that appears in the image but not in the invariant image is a shadow edge, so we can now apply our original formula: find all edges, identify the shadow edges, remove those, and then integrate to get the picture back.

Of course, under practical circumstances, usually we do not know enough about the sensors to evaluate the as and bs that define this family of lines, so we cannot get the invariant image directly. However, we can infer a direction in (c_1, c_2) space that is a good estimate by a form of entropy reasoning. We must choose a color temperature direction. Assume the world is rich in differently colored surfaces. Now consider two surfaces S_1 and S_2. If $\mathbf{c}_1$ (the (c_1, c_2) values for S_1) and $\mathbf{c}_2$ are such that $\mathbf{c}_1 - \mathbf{c}_2$ is parallel to the color temperature direction, we can choose T_1 and T_2 so that S_1 viewed under light with color temperature T_1 will look the same as S_2 viewed under light with color temperature T_2. We expect this to be uncommon, because surfaces tend not to mimic one another in this way. This means we expect that colors will tend to spread out when we project along a good estimate of the color temperature direction. A reasonable measure of this spreading out is the *entropy* of the histogram of projected colors. We can now estimate the invariant image, without knowing anything about the sensor. We search directions in (c_1, c_2) space, projecting all the image colors along that direction; our estimate of the color temperature direction is the one where this projection yields the largest entropy. From this we can compute the invariant image, and so apply our shadow removal

strategy above. In practice, the method works well, though great care is required with the integration procedure to get the best results (Figure 19.10).

19.3.2 Color Constancy: Surface Color from Image Color

In our model, the image color depends on both light color and on surface color. If we light a green surface with white light, we get a green image; if we light a white surface with a green light, we also get a green image. This makes it difficult to name surface colors from pictures. We would like to have an algorithm that can take an image, discount the effect of the light, and report the actual color of the surface being viewed.

This process is called *color constancy*. Humans have some form of color constancy algorithm. People are often unaware of this, and inexperienced photographers are sometimes surprised that a scene photographed indoors under fluorescent lights has a blue cast, whereas the same scene photographed outdoors may have a warm orange cast. The simple linear models of Section 18.3 can predict the color an observer will perceive when shown an isolated spot of light of a given power spectral distribution. But if this spot is part of a larger, more complex scene, these models can give wildly inaccurate predictions. This is because the human color constancy algorithm uses various forms of scene information to decide what color to report. Demonstrations by **?**, which are illustrated in Figure 19.11, give convincing examples of this effect. It is surprisingly difficult to predict what colors a human will see in a complex scene (**?**; **?**; [**?**]; [**?**]; [**?**]). This is one of the many difficulties that make it hard to produce really good color reproduction systems.

Human color constancy is not perfectly accurate, and people can choose to disregard information from their color constancy system. As a result, people can often report:

- the color a surface would have in white light (often called *surface color*);
- the color of the light arriving at the eye (a useful skill that allows artists to paint surfaces illuminated by colored lighting); and
- the color of the light falling on the surface.

The model of image color in Section 19.2 is

$$\mathbf{C}(\mathbf{x}) = g_d(\mathbf{x})\mathbf{d}(\mathbf{x}) + g_s(\mathbf{x})\mathbf{s}(\mathbf{x}) + \mathbf{i}(\mathbf{x}).$$

We decided to ignore the interreflection term $\mathbf{i}(\mathbf{x})$. In principle, we could use the methods of Section **??** to generate new images without specularities. This brings us to the term $g_d(\mathbf{x})\mathbf{d}(\mathbf{x})$. Assume that $g_d(\mathbf{x})$ is a constant, so we are viewing a flat, frontal surface. The resulting term, $\mathbf{d}(\mathbf{x})$, models the world as a collage of flat, frontal, diffuse colored surfaces. Such worlds are sometimes called *Mondrian worlds*, after the painter. Notice that, under our assumptions, $\mathbf{d}(\mathbf{x})$ consists of a set of patches of fixed color. We assume that there is a single illuminant that has a constant color over the whole image. This term is a conglomeration of illuminant, receptor, and reflectance information. It is impossible to disentangle these completely in a realistic world. However, current algorithms can make quite

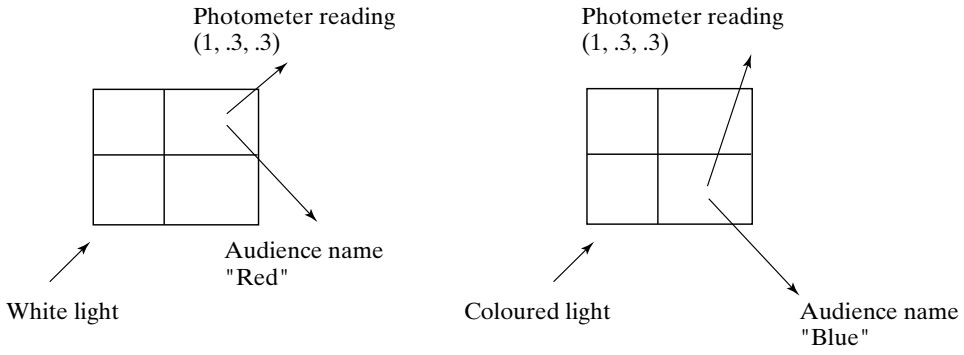


FIGURE 19.11: *Land showed an audience a quilt of rectangles of flat colored papers—since known as a Mondrian for a purported resemblance to the work of that artist—illuminated using three slide projectors, casting red, green and blue light respectively. He used a photometer to measure the energy leaving a particular spot in three different channels, corresponding to the three classes of receptor in the eye. He recorded the measurement, and asked the audience to name the patch. Assume the answer was "red" (on the* **left***). Land then adjusted the slide projectors so that some other patch reflected light that gave the same photometer measurements, and asked the audience to name that patch. The reply would describe the patch's color in white light—if the patch looked blue in white light, the answer would be "blue" (on the* **right***). In later versions of this demonstration, Land put wedge-shaped neutral density filters into the slide projectors so that the color of the light illuminating the quilt of papers would vary slowly across the quilt. Again, although the photometer readings vary significantly from one end of a patch to another, the audience sees the patch as having a constant color.*

usable estimates of surface color from image colors given a well-populated world of colored surfaces and a reasonable illuminant.

Recall from Section 19.2 that if a patch of perfectly diffuse surface with diffuse spectral reflectance $\rho(\lambda)$ is illuminated by a light whose spectrum is $E(\lambda)$, the spectrum of the reflected light is $\rho(\lambda)E(\lambda)$ (multiplied by some constant to do with surface orientation, which we have already decided to ignore). If a linear photoreceptor of the kth type sees this surface patch, its response is:

$$p_k = \int_\Lambda \sigma_k(\lambda)\rho(\lambda)E(\lambda)d\lambda,$$

where Λ is the range of all relevant wavelengths, and $\sigma_k(\lambda)$ is the sensitivity of the kth photoreceptor.

Finite-Dimensional Linear Models

This response is linear in the surface reflectance and linear in the illumination, which suggests using linear models for the families of possible surface reflectances and illuminants. A *finite-dimensional linear model* models surface spectral albedoes and illuminant spectral energy density as a weighted sum of a finite number of basis

functions. We need not use the same bases for reflectances and for illuminants.

If a finite-dimensional linear model of surface reflectance is a reasonable description of the world, any surface reflectance can be written as

$$\rho(\lambda) = \sum_{j=1}^{n} r_j \phi_j(\lambda),$$

where the $\phi_j(\lambda)$ are the basis functions for the model of reflectance, and the r_j vary from surface to surface. Similarly, if a finite-dimensional linear model of the illuminant is a reasonable model, any illuminant can be written as

$$E(\lambda) = \sum_{i=1}^{m} e_i \psi_i(\lambda),$$

where the $\psi_i(\lambda)$ are the basis functions for the model of illumination.

When both models apply, the response of a receptor of the kth type is

$$\begin{aligned} p_k &= \int \sigma_k(\lambda) \left(\sum_{j=1}^{n} r_j \phi_j(\lambda) \right) \left(\sum_{i=1}^{m} e_i \psi_i(\lambda) \right) d\lambda \\ &= \sum_{i=1,j=1}^{m,n} e_i r_j \left(\int \sigma_k(\lambda) \phi_j(\lambda) \psi_i(\lambda) \right) d\lambda \\ &= \sum_{i=1,j=1}^{m,n} e_i r_j g_{ijk}, \end{aligned}$$

where we expect that the

$$g_{ijk} = \int \sigma_k(\lambda) \phi_j(\lambda) \psi_i(\lambda) d\lambda$$

are known, as they are components of the world model (they can be learned from observations; see the exercises).

Inferring Surface Color

The finite-dimensional linear model describes the interaction between illumination color, surface color, and image color. To infer surface color from image color, we need some sort of assumption. There are several plausible cues that can be used.

Specular reflections at dielectric surfaces have uniform specular albedo. We could find the specularities with the methods of that section, then recover surface color using this information. At a specularity, we have

$$p_k = \int \sigma_k(\lambda) \sum_{i=1}^{m} e_i \psi_i(\lambda) d\lambda,$$

and so if we knew the spectral sensitivities of the sensor and the basis functions ψ_i, we could solve for e_i by solving a linear system. Now we know all e_i, and all p_k for

each pixel. We can solve the linear system

$$p_k = \sum_{i=1,j=1}^{m,n} e_i r_j g_{ijk}$$

in the unknown r_j to recover reflectance coefficients.

Known average reflectance is another plausible cue. In this case, we assume that the spatial average of reflectance in all scenes is constant and known (e.g., we might assume that all scenes have a spatial average of reflectance that is dull gray). In the finite-dimensional basis for reflectance, we can write this average as

$$\sum_{j=1}^{n} \overline{r_j} \phi_j(\lambda).$$

Now if the average reflectance is constant, the average of the receptor responses must be constant too (if the imaging process is linear; see the discussion), and the average of the response of the kth receptor can be written as:

$$\overline{p_k} = \sum_{i=1,j=1}^{m,n} e_i g_{ijk} \overline{r_j}.$$

We know $\overline{p_k}$ and $\overline{r_j}$, and so have a linear system in the unknown light coefficients e_i. We solve this, and then recover reflectance coefficients at each pixel, as for the case of specularities. For reasonable choices of reflectors and dimension of light and surface basis, this linear system will have full rank.

The **gamut** of a color image is revealing. The gamut is the set of different colors that appears in the image. Generally, it is difficult to obtain strongly colored pixels under white light with current imaging systems. Furthermore, if the picture is taken under strongly colored light, that will tend to bias the gamut. One doesn't see bright green pixels in images taken under deep red light, for example. As a result, the image gamut is a source of information about the illumination. If an image gamut contains two pixel values—call them $\mathbf{p}_1$ and $\mathbf{p}_2$—then it must be possible to take an image *under the same illuminant* that contains the value $t\mathbf{p}_1 + (1-t)\mathbf{p}_2$ for $0 \leq t \leq 1$ (because we could mix the colorants on the surfaces). This means that the illuminant information depends on the convex hull of the image gamut. There are now various methods to exploit these observations. There is usually more than one illuminant consistent with a given image gamut, and geometric methods can be used to identify the consistent illuminants. This set can be narrowed down using probabilistic methods (for example, images contain lots of different colors [**?**]) or physical methods (for example, the main sources of illumination are the sun and the sky, well modelled as black bodies [**?**]).

19.4 NOTES

There are a number of important general resources on the use of color. We recommend **?**, **?**, **?**, **?**, **?**, **?**. **?** contains an enormous amount of helpful information. Recent textbooks with an emphasis on color include **?**, **?**, **?**, **?** and **?**.

Trichromacy and Color Spaces

Until quite recently, there was no conclusive explanation of why trichromacy applied, although it was generally believed to be due to the presence of three different types of color receptor in the eye. Work on the genetics of photoreceptors can be interpreted as confirming this hunch (see **?** and **?**), although a full explanation is still far from clear because this work can also be interpreted as suggesting many individuals have more than three types of photoreceptor [**?**].

There is an astonishing number of color spaces and color appearance models available. The important issue is not in what coordinate system one measures color, but how one counts the difference, so color metrics may still bear some thought.

Color metrics are an old topic; usually, one fits a metric tensor to MacAdam ellipses. The difficulty with this approach is that a metric tensor carries the strong implication that you can measure differences over large ranges by integration, whereas it is very hard to see large-range color comparisons as meaningful. Another concern is that the weight observers place on a difference in a Maxwellian view and the semantic significance of a difference in image colors are two very different things.

Specularity Finding

The specularity finding method we describe is due to **?**, with improvements due to **?**, [**?**], and to **?**. Specularities can also be detected because they are small and bright [**?**], because they differ in color and motion from the background [**?**, **?**, **?**], or because they distort patterns [**?**]. Specularities are a prodigious nuisance in reconstruction, because specularities cause matching points in different images to have different colors; various motion-based strategies have been developed to remove them in these applications [**?**, **?**, **?**].

Color Constancy

Land reported a variety of color vision experiments (**?**, [**?**], [**?**], [**?**]). Finite-dimensional linear models for spectral reflectances can be supported by an appeal to surface physics as spectral absorption lines are thickened by solid state effects. The main experimental justifications for finite-dimensional linear models of surface reflectance are measurements, by **?**, of the surface reflectance of a selection of standard reference surfaces known as *Munsell chips*, and measurements of a selection of natural objects by **?**. **?** performed a principal axis decomposition of his data to obtain a set of basis functions, and **?** fitted weighted sums of these functions to Krinov's date to get good fits with patterned deviations. The first three principal axes explained in each case a high percentage of the sample variance (near 99 %), and hence a linear combination of these functions fitted all the sampled functions rather well. More recently, **?** fitted Cohen's [**?**] basis vectors to a large set of data, including Krinov's [**?**] data, and further data on the surface reflectances of Munsell chips, and concluded that the dimension of an accurate model of surface reflectance was on the order of five or six.

Finite-dimensional linear models are an important tool in color constancy. There is a large collection of algorithms that follow rather naturally from the approach. Some algorithms exploit the properties of the linear spaces involved (**?**;

?; **?**). Illumination can be inferred from: reference objects [**?**]; specular reflections (Judd [**?**] writing in 1960 about early German work in surface color perception refers to this as "a more usual view"; recent work includes [**?**, **?**, **?**, **?**]); the average color [**?**, **?**, **?**]; and the gamut (**?**, **?**, **?**, [**?**]).

The structure of the family of maps associated with a change in illumination has been studied quite extensively. The first work is due to Von Kries (who didn't think about it quite the way we do). He assumed that color constancy was, in essence, the result of independent lightness calculations in each channel, meaning that one can rectify an image by scaling each channel independently. This practice is known as Von Kries' law. The law boils down to assuming that the family of maps consists of diagonal matrices. Von Kries' law has proved to be a remarkably good law [**?**]. Current best practice involves applying a linear transformation to the channels and then scaling the result using diagonal maps (**?**, [**?**]).

Reference datasets are available for testing methods [**?**]. Color constancy methods seem to work quite well in practice [**?**, **?**]; whether this is good enough is debated [**?**, **?**]. Probabilistic methods can be applied to color constancy [**?**]. Prior models on illumination are a significant cue [**?**].

There is surprisingly little work on color constancy that unifies a study of the spatial variation in illumination with solutions for surface color, which is why we were reduced to ignoring a number of terms in our color model. Ideally, one would work in shadows and surface orientation, too. Again, the whole thing looks like an inference problem to us, but a subtle one. The main papers on this extremely important topic are **?**, **?**. There is substantial room for research here, too.

Interreflections between colored surfaces lead to a phenomenon called *color bleeding*, where each surface reflects colored light onto the other. The phenomenon can be surprisingly large in practice. People seem to be quite good at ignoring it entirely, to the extent that most people don't realize that the phenomenon occurs at all. Discounting color bleeding probably uses spatial cues. Some skill is required to spot really compelling examples. The best known to the authors is occasionally seen in southern California, where there are many large hedges of white oleander by the roadside. White oleander has dark leaves and white flowers. Occasionally, in bright sunlight, one sees a hedge with yellow oleander flowers; a moment's thought attributes the color to the yellow service truck parked by the road reflecting yellow light onto the white flowers. One's ability to discount color bleeding effects seems to have been disrupted by the dark leaves of the plant breaking up the spatial pattern. Color bleeding contains cues to surface color that are quite difficult to disentangle (see **?**, **?**, and **?** for studies).

It is possible to formulate and attack color constancy as an inference problem [**?**, **?**]. The advantage of this approach is that, for given data, the algorithm could report a range of possible surface colors, with posterior weights.

CHAPTER 20

Camera Matrices

20.1 SIMPLE PROJECTIVE GEOMETRY

Draw a pattern on a plane, then view that pattern with a perspective camera. The distortions you observe are more interesting than are predicted by simple rotation, translation and scaling. For example, if you drew parallel lines, you might see lines that intersect at a vanishing point – this doesn't happen under rotation, translation and scaling. *Projective geometry* can be used to describe the set of transformations produced by a perspective camera.

20.1.1 Homogeneous Coordinates and Projective Spaces

The coordinates that every reader will be most familiar with are known as *affine coordinates*. In affine coordinates, a point on the plane is represented by 2 numbers, a point in 3D is represented with 3 numbers, and a point in k dimensions is represented with k numbers. Now adopt the convention that a point in k dimensions is represented by $k+1$ numbers *not all of which are zero*. Two representations $\mathbf{X}_1$ and $\mathbf{X}_2$ represent the same point (write $\mathbf{X}_1 \equiv \mathbf{X}_2$) if there is some $\lambda \neq 0$ so that

$$\mathbf{X}_1 = \lambda \mathbf{X}_2.$$

These coordinates are known as *homogeneous coordinates*, and will offer a particularly convenient representation of perspective projection.

> **Remember this:** *In homogeneous coordinates, a point in a k dimensional space is represented by $k+1$ coordinates $(X_1, \ldots, X_{k+1})$, together with the convention that*
>
> $$(X_1, \ldots, X_{k+1}) \equiv \lambda(X_1, \ldots, X_{k+1}) \text{ for } \lambda \neq 0.$$

The space represented by $k+1$ homogeneous coordinates is different from the space represented by k affine coordinates in important but subtle ways. We start with a 1D space. In homogenous coordinates, we represent a point on a 1D space with two coordinates, so (X_1, X_2) (by convention, homogeneous coordinates are written with capital letters). Two sets of homogeneous coordinates (U_1, U_2) and (V_1, V_2) represent different points if there is no $\lambda \neq 0$ such that $\lambda(U_1, U_2) = (V_1, V_2)$. Now consider the set of all the distinct points, which is known as the *projective line*. Any point on an ordinary line (the *affine line*) has a corresponding point on the projective line. In affine coordinates, a point on the affine line is given by a single coordinate x. This point can be identified with the point on the projective line

given by $(X_1, X_2) = \lambda(x, 1)$ (for $\lambda \neq 0$) in homogeneous coordinates. Notice that the projective line has an "extra point" – $(X_1, 0)$ are the homogeneous coordinates of a single point (check this), but this point would be "at infinity" on the affine line.

Example: 20.1 *Seeing the point at infinity*

You can actually see the point at infinity. Recall that lines that are parallel in the world can intersect in the image at a vanishing point. This vanishing point turns out to be the image of the point "at infinity" on the parallel lines. For example, on the plane $y = -1$ in the camera coordinate system, draw two lines $(1, -1, t)$ and $(-1, -1, t)$ (these lines are in Figure 33.2). Now these lines project to $(f1/t, f(-1/t), f)$ and $(f(-1/t), f(-1/t), f)$ on the image plane, and their vanishing point is $(0, 0, f)$. This vanishing point occurs when the parameter t reaches infinity. The exercises work this example in homogeneous coordinates.

There isn't anything special about the point on the projective line given by $(X_1, 0)$. You can see this by identifying x on the affine line with $(X_1, X_2) = \lambda(1, x)$ (for $\lambda \neq 0$). Now $(X_1, 0)$ is a point like any other, and $(0, X_2)$ is "at infinity". A little work establishes that there is a 1-1 mapping between the projective line and a circle (exercises).

Higher dimensional spaces follow the same pattern. In affine coordinates, a point in a k dimensional affine space (eg an *affine plane*; *affine 3D space*; etc) is given by k coordinates $(x_1, x_2, \ldots, x_k)$. The space described by $k + 1$ homogeneous coordinates is a *projective space* (a *projective plane*; *projective 3D space*; etc). A point $(x_1, x_2, \ldots, x_k)$ in a k dimensional affine space can be identified with $(X_1, X_2, \ldots, X_{k+1}) = \lambda(x_1, x_2, \ldots, x_k, 1)$ (for $\lambda \neq 0$) in the k dimensional projective space. The points in the projective space given by $(X_1, X_2, \ldots, 0)$ have no corresponding points in the affine space. Notice that this set of points is a $k - 1$ dimensional space in homogeneous coordinates. When $k = 2$, this set is a projective line, and is referred to as the *line at infinity*, and the whole space is known as the *projective plane*. As the exercises show, you can see the line at infinity: the horizon of a plane in the image is actually the image of the line at infinity in that plane.

When $k = 3$, this set is itself a projective plane, and is known as the *plane at infinity*; the whole space is sometimes known as *projective 3-space*. Notice this means that 3D projective space is obtained by "sewing" a projective plane to the 3D affine space we are accustomed to. The piece of the projective space "at infinity" isn't special, using the same argument as above. The particular line (resp. plane) that is "at infinity" is chosen by the homogeneous coordinate you divide by. There is an established convention in computer vision of dividing by the last homogeneous coordinate and talking about the line at infinity and the plane at infinity.

Remember this: *The k dimensional space represented by $k+1$ homogeneous coordinates is a projective space. You can represent a point $(x_1, \ldots, x_k$ in affine k space in this projective space as $(x_1, \ldots, x_k, 1)$. Not every point in the projective space can be obtained like this – the points $(X_1, \ldots, X_k, 0)$ are "extra". These points form a projective $k-1$ space which is thought of as being "at infinity". Important cases are $k = 1$ (the projective line with a point at infinity); $k = 2$ (the projective plane with a line at infinity).*

20.1.2 Lines and Planes in Projective Space

Lines on the affine plane form one important example of homogeneous coordinates. A line is the set of points (x, y) where $ax + by + c = 0$. We can use the coordinates (a, b, c) to represent a line. If $(d, e, f) = \lambda(a, b, c)$ for $\lambda \neq 0$ (which is the same as $(d, e, f) \equiv (a, b, c)$), then (d, e, f) and (a, b, c) represent the same line. This means the coordinates we are using for lines are homogeneous coordinates, and the family of lines in the affine plane is a projective plane. Notice that encoding lines using affine coordinates must leave out some lines. For example, if we insist on using $(u, v, 1) = (a/c, b/c, 1)$ to represent lines, the corresponding equation of the line would be $ux + vy + 1 = 0$. But no such line can pass through the origin – our representation has left out every line through the origin.

Lines on the projective plane work rather like lines on the affine plane. Write the points on our line using homogeneous coordinates to get

$$(x, y, 1) = (X_1/X_3, X_2/X_3, 1)$$

or equivalently (X_1, X_2, X_3) where $X_1 = xX_3$, $X_2 = yX_3$. Substitute to find the equation of the corresponding line on the projective plane, $aX_1 + bX_2 + cX_3 = 0$, or $\mathbf{a}^T\mathbf{X} = 0$. There is an interesting point here. A set of three homogenous coordinates can be used to describe either a point on the projective plane or a line on the projective plane.

Remember this: *A line on the projective plane is the set of points $\mathbf{X}$ such that*

$$\mathbf{a}^T\mathbf{X} = 0.$$

Here $\mathbf{a}$ is a vector of homogeneous coordinates specifying the particular line.

Remember this: *Write* $\mathbf{P}_1$ *and* $\mathbf{P}_2$ *for two points on the projective plane that are represented in homogeneous coordinates and are different. From the exercises, the line through these two points is given by*

$$\mathbf{a} = \mathbf{P}_1 \times \mathbf{P}_2.$$

From the exercises, a parametrization of this line is given by

$$U\mathbf{P}_1 + V\mathbf{P}_2.$$

Planes in projective 3-space work rather like lines on the projective plane. The locus of points (x, y, z) where $ax + by + cz + d = 0$ is a plane in affine 3-space. Because (a, b, c, d) and $\lambda(a, b, c, d)$ give the same plane, we have that (a, b, c, d) are homogeneous coordinates for a plane in 3D. We can write the points on the plane using homogeneous coordinates to get

$$(x, y, z, 1) = (X_1/X_4, X_2/X_4, X_3/X_4, 1)$$

or equivalently

$$(X_1, X_2, X_3, X_4) \text{ where } X_1 = xX_4, X_2 = yX_4, X_3 = zX_4.$$

Substitute to find the equation of the corresponding plane in projective 3-space $aX_1 + bX_2 + cX_3 + dX_4 = 0$ or $\mathbf{a}^T\mathbf{X} = 0$. A set of four homogenous coordinates can be used to describe either a point in projective 3-space or a plane in projective 3-space.

Remember this: *A plane in projective 3D is the set of points* $\mathbf{X}$ *such that*

$$\mathbf{a}^T\mathbf{X} = 0.$$

Here $\mathbf{a}$ *is a vector of homogeneous coordinates specifying the particular plane.*

Remember this: *Write $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ for three points in projective 3D that are represented in homogeneous coordinates, are different points, and are not collinear. From the exercises, the plane through these points is given by*

$$\mathbf{a} = \mathit{NullSpace}\left(\begin{bmatrix} \mathbf{P}_1^T \\ \mathbf{P}_2^T \\ \mathbf{P}_3^T \end{bmatrix}\right).$$

From the exercises, a parametrization of this plane is given by

$$U\mathbf{P}_1 + V\mathbf{P}_2 + W\mathbf{P}_3.$$

20.1.3 Homographies

Write $\mathbf{X} = (X_1, X_2, X_3)$ for the coordinates of a point on the projective plane. Now consider $\mathbf{V} = \mathcal{M}\mathbf{X}$, where $\mathcal{M}$ is a 3×3 matrix with non-zero determinant. We can interpret $\mathbf{V}$ as a point on the projective plane, and in fact $\mathcal{M}$ is a mapping from the projective plane to itself. There is something to check here. Write $\mathcal{M}(\mathbf{X})$ for the point that $\mathbf{X}$ maps to, etc. Because $\mathbf{X} \equiv \lambda\mathbf{X}$ (for $\lambda \neq 0$), we must have that $\mathcal{M}(\mathbf{X}) \equiv \mathcal{M}(\lambda\mathbf{X})$ otherwise one point would map to several points. But

$$\mathcal{M}(\mathbf{X}) = \mathcal{M}\mathbf{X} \equiv \lambda\mathcal{M}\mathbf{X} = \mathcal{M}(\lambda\mathbf{X})$$

so $\mathcal{M}$ is a mapping. Such mappings are known as *homographies*. You should check that $\mathcal{M}^{(-1)}$ is the inverse of $\mathcal{M}$, and is a homography. You should check that $\mathcal{M}$ and $\lambda\mathcal{M}$ represent the same homography. Homographies are interesting to us because any view of a plane by a perspective (or orthographic) camera is a homography, and a variety of useful tricks rest on understanding homographies.

Any homography will map every line to a line. Write $\mathbf{a}$ for the line in the projective plane whose points satisfy $\mathbf{a}^T\mathbf{X} = 0$. Now apply the homography $\mathcal{M}$ to those points to get $\mathbf{V} = \mathcal{M}\mathbf{X}$. Notice that

$$\mathbf{a}^T\mathcal{M}^{(-1)}\mathbf{V} = \mathbf{a}^T\mathbf{X} = 0,$$

so that the line $\mathbf{a}$ transforms to the line $\mathcal{M}^{(-T)}\mathbf{a}$. Homographies are easily inverted.

Remember this: *A homography is a mapping from the projective plane to the projective plane. Assume $\mathcal{M}$ is a 3×3 matrix with non-zero determinant; then the homography represented by $\mathcal{M}$ maps the point with homogeneous coordinates $\mathbf{X}$ to the point with homogeneous coordinates $\mathcal{M}\mathbf{X}$. The two matrices $\mathcal{M}$ and $\lambda\mathcal{M}$ represent the same homography, and the inverse of this homography is represented by $\mathcal{M}^{-1}$. The homography represented by $\mathcal{M}$ will map the line represented by $\mathbf{a}$ to the line represented by $\mathcal{M}^{-T}\mathbf{a}$.*

20.2 CAMERA MATRICES AND TRANSFORMATIONS

20.2.1 Perspective and Orthographic Camera Matrices

In affine coordinates we wrote perspective projection as $(X, Y, Z) \rightarrow (X/Z, Y/Z)$ (remember, we will account for f later). Now write the 3D point in homogeneous coordinates, so

$$\mathbf{X} = (X_1, X_2, X_3, X_4) \text{ where } X_1 = XX_4, \text{ etc.}$$

Write the point in the image plane in homogeneous coordinates as well, to obtain

$$\mathbf{I} = (I_1, I_2, I_3) \text{ where } I_1 = (X/Z)I_3 \text{ and } I_2 = (Y/Z)I_3.$$

So we could use

$$\mathbf{I} = (X, Y, Z) \equiv (X/Z, Y/Z, 1) \equiv (X_1/X_4, X_2/X_4, X_3/X_4) \equiv (X_1, X_2, X_3).$$

Notice that (X, Y, Z) is a natural choice of homogeneous coordinates for the point in the image plane. This means that, in homogeneous coordinates, we can represent perspective projection as

$$(X_1, X_2, X_3, X_4) \rightarrow (X_1, X_2, X_3) \equiv (X_1, X_2, X_3).$$

or

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}$$

where the matrix is known as the *perspective camera matrix* (write $\mathcal{C}_p$). Notice that this representation preserves the property that the focal point of the camera cannot be imaged, and is the only such point. The focal point can be represented in homogeneous coordinates by $(0, 0, 0, T)$, for $T \neq 0$. This maps to $(0, 0, 0)$, which is meaningless in homogeneous coordinates. You should check no other point maps to $(0, 0, 0)$.

Remember this: *The perspective camera matrix is*

$$\mathcal{C}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

In affine coordinates, in the right coordinate system and assuming that the scale is chosen to be one, scaled orthographic perspective can be written as $(X, Y, Z) \rightarrow (X, Y)$. Following the argument above, we obtain in homogeneous coordinates

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}$$

where the matrix is known as the *orthographic camera matrix* (write $\mathcal{C}_o$).

Remember this: *The orthographic camera matrix is*

$$\mathcal{C}_o = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

20.2.2 Cameras in World Coordinates

The camera matrix describes a perspective (resp. orthographic) projection for a camera in a specific coordinate system – the focal point is at the origin, the camera is looking down the z-axis, and so on. In the more general case, the camera is placed somewhere in world coordinates looking in some direction, and we need to account for this. Furthermore, the camera matrix assumes that points in the camera are reported in a specific coordinate system. The pixel locations reported by a practical camera might not be in that coordinate system. For example, many cameras place the origin at the top left hand corner. We need to account for this effect, too.

A general perspective camera transformation can be written as:

$$\begin{aligned} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} &= \begin{bmatrix} \text{Transformation} \\ \text{mapping image} \\ \text{plane coords to} \\ \text{pixel coords} \end{bmatrix} \mathcal{C}_p \begin{bmatrix} \text{Transformation} \\ \text{mapping world} \\ \text{coords to camera} \\ \text{coords} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} \\ &= \mathcal{T}_i \mathcal{C}_p \mathcal{T}_e \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} \end{aligned}$$

The parameters of $\mathcal{T}_i$ are known as *camera intrinsic parameters* or *camera intrinsics*, because they are part of the camera, and typically cannot be changed. The parameters of $\mathcal{T}_e$ are known as *camera extrinsic parameters* or *camera extrinsics*, because they can be changed.

20.2.3 Camera Extrinsic Parameters

The transformation $\mathcal{T}_e$ represents a rigid motion (equivalently, a *Euclidean transformation*, which consists of a 3D rotation and a 3D translation). In affine coordinates, any Euclidean transformation maps the vector $\mathbf{x}$ to

$$\mathcal{R}\mathbf{x} + \mathbf{t}$$

where $\mathcal{R}$ is an appropriately chosen 3D rotation matrix (check the endnotes if you can't recall) and $\mathbf{t}$ is the translation. Any map of this form is a Euclidean

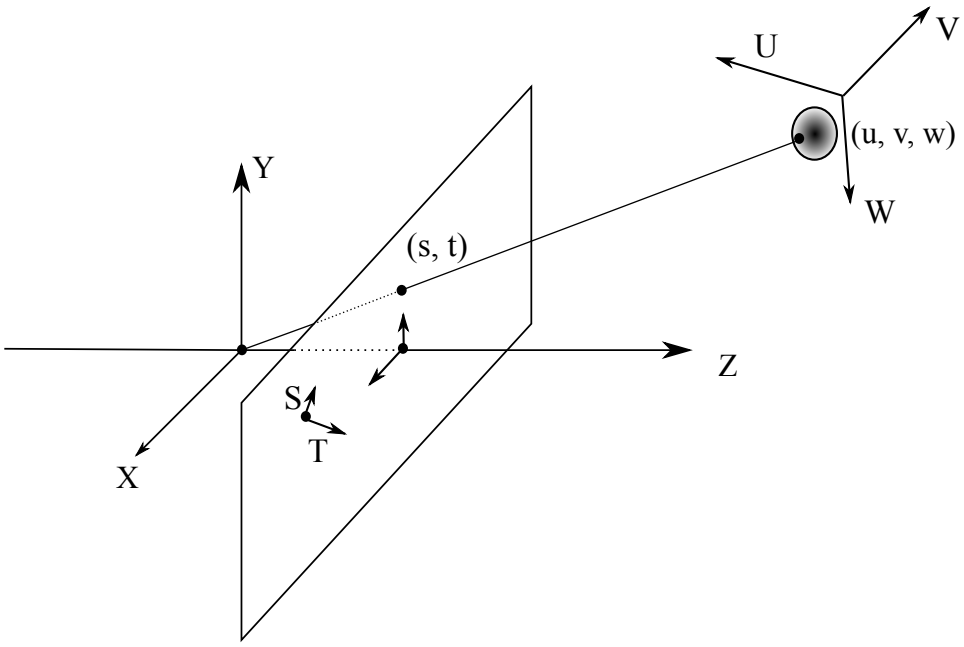


FIGURE 20.1: *A perspective camera (in its own coordinate system, given by X, Y and Z axes) views a point in world coordinates (given by (u, v, w) in the UVW coordinate system) and reports the position of points in ST coordinates. We must model the mapping from (u, v, w) to (s, t), which consists of a transformation from the UVW coordinate system to the XYZ coordinate system followed by a perspective projection followed by a transformation to the ST coordinate system.*

transformation. You should confirm the transformation that maps the vector $\mathbf{X}$ representing a point in 3D in homogeneous coordinates to

$$\lambda \begin{bmatrix} \mathcal{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \mathbf{X}$$

represents a Euclidean transformation, but in homogeneous coordinates. It follows that any map of this form is a Euclidean transformation. Because $\mathcal{T}_e$ represents a Euclidean transformation, it must have this form. The exercises explore some properties of $\mathcal{T}_e$.

20.2.4 Camera Intrinsic Parameters

Camera intrinsic parameters must model a possible coordinate transformation in the image plane from projected world coordinates (write (x, y)) to pixel coordinates (write (u, v)), together with a possible change of focal length. This change is caused by the image plane being further away from, or closer to, the focal point. The coordinate transformation is not arbitrary (Figure 20.2). Typically, the origin of the pixel coordinates is usually not at the camera center. Write Δx for the step in the image plane from pixel (i, j) to $(i+1, j)$ and Δy for the step to $(i, j+1)$. These

are vectors parallel to the camera coordinate axes. The vector Δx may not be perpendicular to the vector Δy, causing *skew*. For many cameras, $\|\Delta x\|$ is different from $\|\Delta y\|$ – such cameras have *non-square pixels*, and $\|\Delta x\|/\|\Delta y\|$ is known as the *aspect ratio* of the pixel. Furthermore, $\|\Delta x\|$ is not usually one unit in world coordinates.

There is one tricky point here. Rotating the world about the Z axis has an effect equivalent to rotating the camera coordinate system (Figure **??**). This means we cannot tell whether this rotation is the result of a change in the extrinsics (the world rotated) or the intrinsics (the camera coordinate system rotated). By convention, there is no rotation in the intrinsics, so a pure rotation of the image is always the result of the world rotating.

There are two possible parametrizations of camera intrinsics. Recall f is the focal length of the camera. Write (c'_x, c'_y) for the location of the camera center in pixel coordinates; a for the aspect ratio of the pixels ; and k' for the skew. Then $\mathcal{T}_i$ is parametrized as

$$\begin{bmatrix} \|\Delta x\| & k' & c'_x \\ 0 & \|\Delta y\| & c'_y \\ 0 & 0 & 1/f \end{bmatrix} \equiv \begin{bmatrix} af\|\Delta y\| & fk' & fc'_x \\ 0 & f\|\Delta y\| & fc'_y \\ 0 & 0 & 1 \end{bmatrix}$$

Notice in this case we are distinguishing between scaling resulting from $\|\Delta y\|$ and scaling resulting from the focal length. This is unusual, but can occur. More usual is to conflate these effects and parametrize the intrinsics as

$$\begin{bmatrix} as & k & c_x \\ 0 & s & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where $s = f\|\Delta y\|$, $a = \|\Delta x\|\Delta y$, $k = fk'$, $c_x = fc'_x$, $c_y = fc'_y$.

Remember this: *A general perspective camera can be written in homogeneous coordinates as:*

$$\begin{aligned} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} &= \mathcal{T}_i \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathcal{T}_e \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} \\ &= \begin{bmatrix} as & k & c_x \\ 0 & s & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \mathcal{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} \end{aligned}$$

where $\mathcal{R}$ is a rotation matrix.

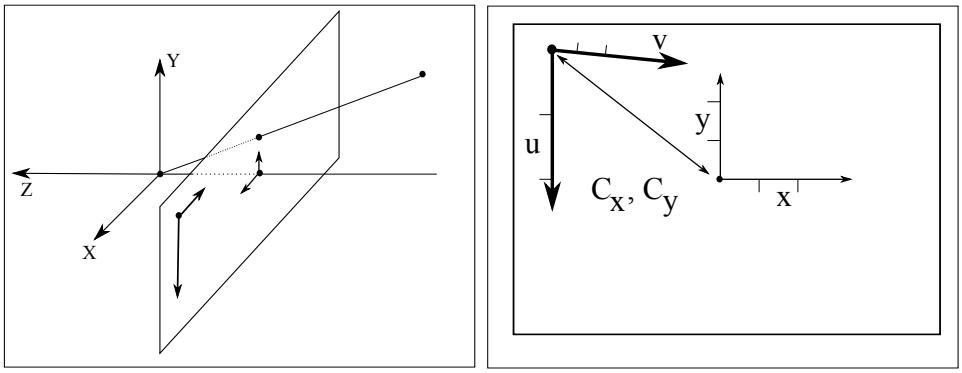


FIGURE 20.2: *The camera reports pixel values in pixel coordinates, which are not the same as world coordinates. The camera intrinsics represent the transformation between world coordinates and pixel coordinates. On the* **left**, *a camera (as in Figure 17.1), with the camera coordinate system shown in heavy lines. On the* **right**, *a more detailed view of the image plane. The camera coordinate axes are marked* (u, v) *and the image coordinate axes* (x, y)*. It is hard to determine* f *from the figure, and we will conflate scaling due to* f *with scaling resulting from the change to camera coordinates. The camera coordinate system's origin is not at the camera center, so* (c_x, c_y) *are not zero. I have marked unit steps in the coordinate system with ticks. Notice that the* v*-axis is not perpendicular to the* u*-axis (so* k *-* the skew *- is not zero). Ticks in the* u, v *axes are not the same distance apart as ticks on the* x, y *axes, meaning that* s *is not one. Furthermore,* u *ticks are further apart than* v *ticks, so that* a *is not one.*

By the arguments above, a general orthographic camera transformation can be written as:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \mathcal{T}_i \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathcal{T}_e \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}$$

PROBLEMS

20.1. We construct the vanishing point of a pair of parallel lines in homogeneous coordinates.

- **(a)** Show that the set of points in homogeneous coordinates in 3D given by $(s, -s, t, s)$ (for s, t parameters) form a line in 3D.
- **(b)** Now image the line $(s, -s, t, s)$ in 3D in a standard perspective camera with focal length 1. Show the result is the line $(s, -s, t)$ in the image plane.
- **(c)** Now image the line $(-s, -s, t, s)$ in 3D in a standard perspective camera with focal length 1. Show the result is the line $(-s, -s, t)$ in the image plane.
- **(d)** Show that the lines $(s, -s, t)$ and $(-s, -s, t)$ intersect in the point $(0, 0, t)$.

20.2. We construct the horizon of a plane for a standard perspective camera with

focal length 1. Write $\mathbf{a} = [a_1, a_2, a_3, a_4]^T$ for the coefficients of the plane, so that for every point $\mathbf{X}$ on the plane we have $\mathbf{a}^T\mathbf{X} = 0$.

(a) Show that the plane given by $\mathbf{u} = [a_1, a_2, a_3, 0]$ is parallel to the plane given by $\mathbf{a}$, and passes through $(0, 0, 0, 1)$.

(b) Write the points on the image plane $(u, v, 1) \equiv (U, V, W)$ in homogeneous coordinates. Show that the horizon of the plane is the set of points $\mathbf{u}$ in the image plane given by $\mathbf{l}^T\mathbf{u} = 0$, where $\mathbf{l} = [a_1, a_2, a_3]^T$.

20.3. A pinhole camera with focal point at the origin and image plane at $z = f$ views two parallel lines $\mathbf{u} + t\mathbf{w}$ and $\mathbf{v} + t\mathbf{w}$. Write $\mathbf{w} = [w_1, w_2, w_3]^T$, etc.

(a) Show that the vanishing point of these lines, on the image plane, is given by $(f\frac{w_1}{w_3}, f\frac{w_2}{w_3})$.

(b) Now we model a family of pairs of parallel lines, by writing $\mathbf{w}(r, s) = r\mathbf{a} + s\mathbf{b}$, for any (r, s). In this model, $\mathbf{u} + t\mathbf{w}(r, s)$ and $\mathbf{v} + t\mathbf{w}(r, s)$ are the pair of lines, and (r, s) chooses the direction. First, show that this family of vectors lies in a plane. Now show that the vanishing point for the (r, s)'th pair is $(f\frac{ra_1+sb_1}{ra_3+sb_3}, f\frac{ra_2+sb_2}{ra_3+sb_3})$.

(c) Show that the family of vanishing points $(f\frac{ra_1+sb_1}{ra_3+sb_3}, f\frac{ra_2+sb_2}{ra_3+sb_3})$ lies on a straight line in the image. Do this by constructing $\mathbf{c}$ such that $\mathbf{c}^T\mathbf{a} = \mathbf{c}^T\mathbf{b} = 0$. Now write $(x(r, s), y(r, s)) = (-f\frac{ra_1+sb_1}{ra_3+sb_3}, -f\frac{ra_2+sb_2}{ra_3+sb_3})$ and show that $c_1x(r, s) + c_2y(r, s) + c_3 = 0$.

20.4. All points on the projective plane with homogeneous coordinates $(U, V, 0)$ lie "at infinity" (divide by zero). As we have seen, these points form a projective line.

(a) Show this line is represented by the vector of coefficients $(0, 0, C)$.

(b) A homography $\mathcal{M} = \left[\mathbf{m}_1^T; \mathbf{m}_2^T; \mathbf{m}_3^T\right]$ is applied to the projective plane. Show that the line whose coefficients are $\mathbf{v}_3$ maps to the line at infinity.

(c) Now write the homography as $\mathcal{M} = \left[\mathbf{m}'_1, \mathbf{m}'_2, \mathbf{m}'_3\right]$ (so $\mathbf{m}'$ are columns). Show that the homography maps the points at infinity to a line given in parametric form as $s\mathbf{m}'_1 + t\mathbf{m}'_2$.

(d) Now write $\mathbf{n}$ for a non-zero vector such that $\mathbf{n}^T\mathbf{m}'_1 = \mathbf{n}^T\mathbf{m}'_2 = 0$. Show that, for any point $\mathbf{x}$ on the line given in parametric form as $s\mathbf{m}'_1 + t\mathbf{m}'_2$, we have $\mathbf{n}^T\mathbf{x} = 0$. Is $\mathbf{n}$ unique?

(e) Use the results of the previous subexercises to show that for any given line, there are some homographies that map that line to the line at infinity.

(f) Use the results of the previous subexercises to show that for any given line, there are some homographies that map the line at infinity to that line.

20.5. We will show that there is no significant difference between choosing a right-handed camera coordinate system and a left-handed camera coordinate system. Notice that, in a right handed camera coordinate system (where the camera looks down the negative z-axis rather than the positive z-axis) the image plane is at $z = -f$.

(a) Show that, in a right-handed coordinate system, a pinhole camera maps

$$(X, Y, Z) \to (-fX/Z, -fY/Z).$$

(b) Show that the argument in the text yields a camera matrix of the form

$$\mathcal{C}'_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/f & 0 \end{bmatrix}.$$

(c) Show that, if one allows the scale in $\mathcal{T}_i$ to be negative, one could still use

$$\mathcal{C}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}$$

as a camera matrix.

CHAPTER 21

Using Camera Models

21.1 CAMERA CALIBRATION FROM A 3D REFERENCE

Camera calibration involves estimating the intrinsic parameters of the camera, and perhaps lens parameters if needed, from one or more images. There are numerous strategies, all using versions of the following recipe: build a *calibration object*, where the positions of some points (*calibration points*) are known; view that object from one or more viewpoints; obtain the image locations of the calibration points; and solve an optimization problem to recover camera intrinsics and perhaps lens parameters. As one would expect, much depends on the choice of calibration object. If all the calibration points sit on an object, the extrinsics will yield the *pose* (for position and orientation) of the object with respect to the camera. We use a two step procedure: formulate the optimization problem, then find a good starting point.

21.1.1 Formulating the Optimization Problem

The optimization problem is relatively straightforward to formulate. Notation is the main issue. We have N reference points $\mathbf{s}_i = [s_{x,i}, s_{y,i}, s_{z,i}]$ with known position in some reference coordinate system in 3D. The measured location in the image for the i'th such point is $\hat{\mathbf{t}}_i = \left[\hat{t}_{x,i}, \hat{t}_{y,i}\right]$. There may be measurement errors, so the $\hat{\mathbf{t}}_i = \mathbf{t}_i + \xi_i$, where ξ_i is an error vector and $\mathbf{t}_i$ is the unknown true position of the image point. We will assume the magnitude of error does not depend on direction in the image plane (it is *isotropic*), so it is natural to minimize the squared magnitude of the error

$$\sum_i \xi_i^T \xi_i. \tag{21.1}$$

The main issue here is writing out expressions for ξ_i in the appropriate coordinates. Write $\mathcal{T}_i$ for the intrinsic matrix whose u, v'th component will be i_{uv}; $\mathcal{T}_e$ for the extrinsic transformation, whose u, v'th component will be e_{uv}. Recalling that $\mathcal{T}_i$ is upper triangular, and engaging in some manipulation, we obtain

$$\sum_i \xi_i^T \xi_i = \sum_i (t_{x,i} - p_{x,i})^2 + (t_{y,i} - p_{y,i})^2 \tag{21.2}$$

where

$$\begin{aligned} p_{x,i} &= \frac{i_{11} g_{x,i} + i_{12} g_{y,i} + i_{13} g_{z,i}}{g_{z,3}} \\ p_{y,i} &= \frac{i_{22} g_{x,i} + i_{23} g_{z,i}}{g_{z,i}} \end{aligned}$$

and

$$\begin{aligned} g_{x,i} &= e_{11}s_{x,i} + e_{12}s_{y,i} + e_{13}s_{z,i} + e_{14} \\ g_{y,i} &= e_{21}s_{x,i} + e_{22}s_{y,i} + e_{23}s_{z,i} + e_{24} \\ g_{z,i} &= e_{31}s_{x,i} + e_{32}s_{y,i} + e_{33}s_{z,i} + e_{34} \end{aligned}$$

(which you should check as an exercise). This is a constrained optimization problem, because $\mathcal{T}_e$ is a Euclidean transformation. The constraints here are

$$\begin{aligned} 1 - \sum_v e_{1v}^2 = 0 \text{ and } 1 - \sum_v e_{2v}^2 = 0 \text{ and } 1 - \sum_v e_{3v}^2 = 0 \\ \sum_v e_{1v}e_{2v} = 0 \text{ and } \sum_v e_{1v}e_{3v} = 0 \text{ and } 1 - \sum_v e_{2v}e_{3v} = 0 \end{aligned} .$$

We might just throw this into a constrained optimizer (review Section 33.2), but good behavior requires a good start point. This can be obtained by a little manipulation, which I work through in the next section. Some readers may prefer to skip this at first (or even higher) reading because it's somewhat specialized, but it shows how the practical application of some tricks worth knowing.

21.1.2 Setting up a Start Point

Write $\mathbf{C}_j^T$ for the j'th row of the camera matrix, and $\mathbf{S}_i = [s_{x,i}, s_{y,i}, s_{z,i}, 1]^T$ for homogeneous coordinates representing the i'th point in 3D. Then, assuming no errors in measurement, we have

$$\hat{t}_{x,i} = \frac{\mathbf{C}_1^T\mathbf{S}_i}{\mathbf{C}_3^T\mathbf{S}_i} \text{ and } \hat{t}_{y,i} = \frac{\mathbf{C}_2^T\mathbf{S}_i}{\mathbf{C}_3^T\mathbf{S}_i}, \tag{21.3}$$

which we can rewrite as

$$\mathbf{C}_3^T\mathbf{S}_i\hat{t}_{x,i} - \mathbf{C}_1^T\mathbf{S}_i = 0 \text{ and } \mathbf{C}_3^T\mathbf{S}_i\hat{t}_{y,i} - \mathbf{C}_2^T\mathbf{S}_i = 0. \tag{21.4}$$

We now have two homogenous linear equations in the camera matrix elements for each pair (3D point, image point). There are a total of 12 degrees of freedom in the camera matrix, meaning we can recover a least squares solution from six point pairs. The solution will have the form $\lambda\mathcal{P}$ where λ is an unknown scale and $\mathcal{P}$ is a known matrix. This is a natural consequence of working with homogeneous equations, but also a natural consequence of working with homogeneous coordinates. You should check that if $\mathcal{P}$ is a projection from projective 3D to the projective plane, $\lambda\mathcal{P}$ will yield the same projection as long as $\lambda \neq 0$.

This is enough information to recover the focal point of the camera. Recall that the focal point is the single point that images to $[0, 0, 0]^T$. This means that if we are presented with a 3×4 matrix claiming to be a camera matrix, we can determine what the focal point of that camera is without fuss – just find the null space of the matrix. Notice that we do not need to know λ to estimate the null space.

Remember this: *Given a 3×4 camera matrix $\mathcal{P}$, the homogeneous coordinates of the focal point of that camera are given by $\mathbf{X}$, where $\mathcal{P}\mathbf{X} = [0, 0, 0]^T$*

There is an important relationship between the focal point of the camera and the extrinsics. Assume that, in the world coordinate system, the focal point can be represented by $\left[\mathbf{f}^T, 1\right]^T$. This point must be mapped to $[0, 0, 0, 1]^T$ by $\mathcal{T}_e$. Because we can recover $\mathbf{f}$ from $\mathcal{P}$ easily, we have an important constraint on $\mathcal{T}_e$, given in the box.

Remember this: *Assume camera matrix $\mathcal{P}$ has null space $\lambda\mathbf{u} = \lambda\left[\mathbf{f}^T, 1\right]^T$. Then we must have $\mathcal{T}_e\mathbf{u} = [0, 0, 0, 1]^T$, so we must have*

$$\mathcal{T}_e = \begin{bmatrix} \mathcal{R} & -\mathcal{R}\mathbf{f} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{21.5}$$

This means that, if we know $\mathcal{R}$, we can recover the translation from the focal point. We must now recover the intrinsic transformation and $\mathcal{R}$ from what we know.

$$\lambda\mathcal{P} = \mathcal{T}_i \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathcal{R} & -\mathcal{R}\mathbf{f} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathcal{T}_i\mathcal{R} & -\mathcal{T}_i\mathcal{R}\mathbf{f} \end{bmatrix} \tag{21.6}$$

We do not know λ, but we do know $\mathcal{P}$. Now write $\mathcal{P}_l$ for the left 3×3 block of $\mathcal{P}$, and recall that $\mathcal{T}_i$ is upper triangular and $\mathcal{R}$ orthonormal. The first question is the sign of λ. We expect $\mathsf{Det}(\mathcal{R}) = 1$, and $\mathsf{Det}(\mathcal{T}_i) > 0$, so $\mathsf{Det}(\mathcal{P}_l)$ should be positive. This yields the sign of λ – choose a sign $s \in \{-1, 1\}$ so that $\mathsf{Det}(s\mathcal{P}_l)$ is positive.

We can now factor $s\mathcal{P}_l$ into an upper triangular matrix $\mathcal{T}$ and an orthonormal matrix $\mathcal{Q}$. This is an RQ factorization (Section 33.2). Recall we could not distinguish between scaling caused by the focal length and scaling caused by pixel scale, so that

$$\mathcal{T}_i = \begin{bmatrix} as & k & c_x \\ 0 & s & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{21.7}$$

In turn, we have $\lambda = s(1/t_{33})$, $c_y = (t_{23}/t_{33})$, $s = (t_{22}/t_{33})$, $c_x = (t_{13}/t_{33})$, $k = (t_{12}/t_{33})$, and $a = (t_{11}/t_{22})$.

Procedure: 21.1 *Calibrating a Camera using 3D Reference Points*

For N reference points $\mathbf{s}_i = [s_{x,i}, s_{y,i}, s_{z,i}]$ with known position in some reference coordinate system in 3D, write the measured location in the image for the i'th such point $\hat{\mathbf{t}}_i = \left[\hat{t}_{x,i}, \hat{t}_{y,i}\right]$. Now minimize

$$\sum_i \xi_i^T \xi_i = \sum_i (\hat{t}_{x,i} - p_{x,i})^2 + (\hat{t}_{y,i} - p_{y,i})^2 \tag{21.8}$$

where

$$\begin{aligned} p_{x,i} &= \frac{i_{11} g_{x,i} + i_{12} g_{y,i} + i_{13} g_{i,3}}{g_{i,3}} \\ p_{y,i} &= \frac{i_{22} g_{x,i} + i_{23} g_{i,3}}{g_{i,3}} \end{aligned}$$

and

$$\begin{aligned} g_{x,i} &= e_{11} s_{x,i} + e_{12} s_{y,i} + e_{13} s_{z,i} + e_{14} \\ g_{y,i} &= e_{21} s_{x,i} + e_{22} s_{y,i} + e_{23} s_{z,i} + e_{24} \\ g_{z,i} &= e_{31} s_{x,i} + e_{32} s_{y,i} + e_{33} s_{z,i} + e_{34} \end{aligned}$$

subject to:

$$1 - \sum_v e_{j,1v}^2 = 0 \text{ and } 1 - \sum_v e_{j,2v}^2 = 0 \text{ and } 1 - \sum_v e_{j,3v}^2 = 0$$

$$\sum_v e_{j,1v} e_{j,2v} = 0 \text{ and } \sum_v e_{j,1v} e_{j,3v} = 0 \text{ and } 1 - \sum_v e_{j,2v} e_{j,3v} = 0 \quad .$$

Use the start point of procedure 21.2

Procedure: 21.2 *Calibrating a Camera using 3D Reference Points: Start Point*

Estimate the rows of the camera matrix $\mathbf{C}_i$ using at least six points and

$$\mathbf{C}_3^T \mathbf{S}_i \hat{t}_{x,i} - \mathbf{C}_1^T \mathbf{S}_i = 0 \text{ and } \mathbf{C}_3^T \mathbf{S}_i \hat{t}_{y,i} - \mathbf{C}_2^T \mathbf{S}_i = 0. \tag{21.9}$$

Write $\lambda\mathcal{P}$ for the 1D family of solutions to this set of homogeneous linear equations, organized into 3×4 matrix form. Compute the vector $\mathbf{n} = \left[\mathbf{f}^T, 1\right]$ such that $\mathcal{P}\mathbf{n}$. Write $\mathcal{P}_l$ for the left 3×3 block of $\mathcal{P}$. Choose $s \in \{-1, 1\}$ such that $\mathsf{Det}\,(s\mathcal{P}_l) > 0$. Use RQ factorization to obtain $\mathcal{T}$ and $\mathcal{Q}$ such that $s\mathcal{P}_l = \mathcal{T}\mathcal{Q}$. Then the start point for the intrinsic parameters is:

$$\begin{bmatrix} a \\ s \\ k \\ c_x \\ c_y \end{bmatrix} = \begin{bmatrix} (t_{11}/t_{22}) \\ (t_{22}/t_{33}) \\ (t_{12}/t_{33}) \\ (t_{13}/t_{33}) \\ (t_{23}/t_{33}) \end{bmatrix} \tag{21.10}$$

and for $\mathcal{T}_e$ is:

$$\begin{bmatrix} \mathcal{Q} & -\mathcal{Q}\mathbf{f} \\ \mathbf{0} & 1 \end{bmatrix}. \tag{21.11}$$

21.2 CALIBRATING THE EFFECTS OF LENS DISTORTION

Now assume the lens applies some form of geometric distortion, as in Section 33.2. There are now strong standard models of the major lens distortions (Section 33.2). We will now estimate lens parameters, camera intrinsics and camera extrinsics from a view of a calibration object (as in Section 33.2; note the methods of Section 33.2 apply to this problem too). As in those sections, we use a two step procedure: formulate the optimization problem (Section 33.2), then find a good starting point (Section 33.2).

21.2.1 Modelling Geometric Lens Distortion

Geometric distortions caused by lenses are relatively easily modelled by assuming the lens causes (x, y) in the image plane to map to $(x+\delta x, y+\delta y)$ in the image plane. We seek a model for $\delta x, \delta y$ that has few parameters and that captures the main effects. A natural model of barrel distortion is that points are "pulled" toward the camera center, with points that are further from the center being "pulled" more. Similarly, pincushion distortion results from points being "pushed" away from the camera center, with distant points being pushed further (Figure **??**).

Set up a polar coordinate system (r, θ) in the image plane using the image center as the origin. The figure and description suggest that barrel and pincushion distortion can be described by a map $(r, \theta) \rightarrow (r + \delta r, \theta)$. We model δr as a polynomial in r. Brown and Conrady [] established the model $\delta r = k_1 r^3 + k_2 r^5$ as sufficient for a wide range of distortions, and we use $(r, \theta) \rightarrow (r + k_1 r^3 + k_2 r^5, \theta)$

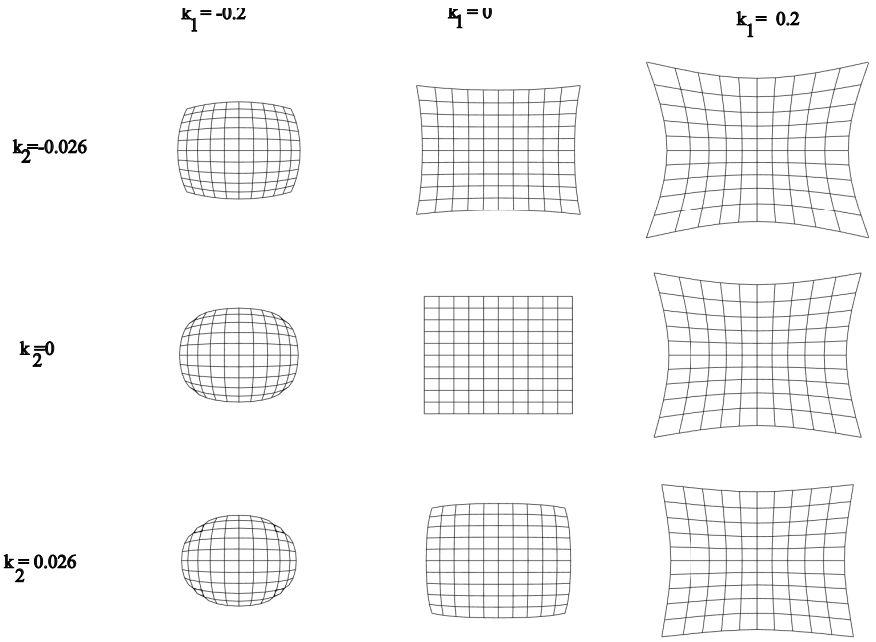


FIGURE 21.1: *The effects of k_1 and k_2 on a neutral grid (***center***), showing how the parameters implement various barrel or pincushion distortions. Notice how k_2 slightly changes the shape of the curves that k_1 produces from straight lines in the grid.*

for unknown k_1, k_2. We must map this model to image coordinates to obtain a map $(x, y) \rightarrow (x + \delta x, y + \delta y)$. Since $\cos\theta = x/r$, $\sin\theta = y/r$, we have $(x, y) \rightarrow (x + x(k_1(x^2 + y^2) + k_2(x^2 + y^2)^2), y + y(k_1(x^2 + y^2) + k_2(x^2 + y^2)^2))$. Figure 21.1 shows distortions resulting from different choices of k_1 and k_2. This model is known as a *radial distortion model*.

More sophisticated lens distortion models account for the lens being off-center. This causes *tangential distortion* (Figure 21.2). The most commonly used model of tangential distortion is a map $(x, y) \rightarrow (x + p_1(x^2 + y^2 + 2x^2) + 2p_2xy, y + p_2(x^2 + y^2 + 2y^2) + 2p_1xy)$ (derived from []; more detail in, for example []).

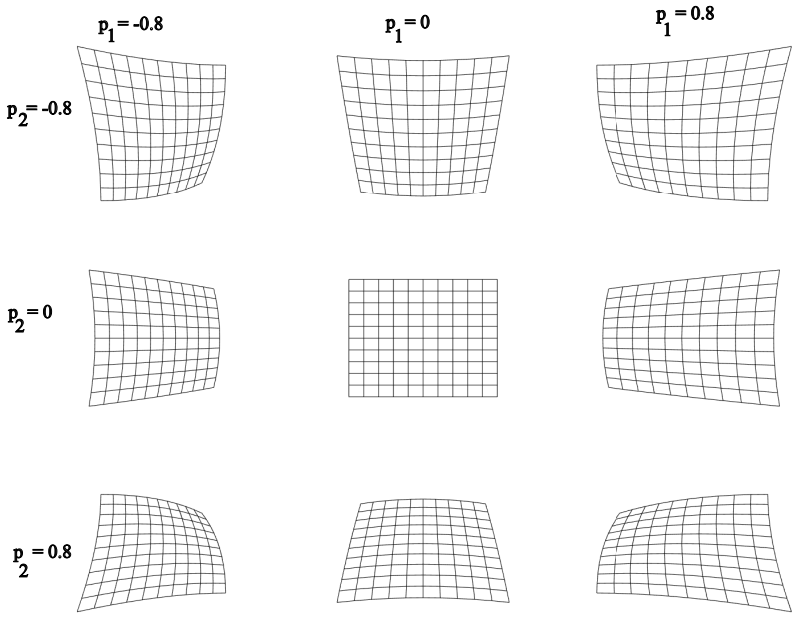


FIGURE 21.2: *The effects of p_1 and p_2 on a neutral grid (***center***), showing how the parameters implement various distortions. These parameters model effects that occur because the lens is off-center; note the grid "turning away" from the lens.*

Remember this: *A full lens distortion model is*

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x + x(k_1(x^2+y^2) + k_2(x^2+y^2)^2) + p_1(x^2+y^2+2x^2) + 2p_2xy \\ y + y(k_1(x^2+y^2) + k_2(x^2+y^2)^2) + p_2(x^2+y^2+2y^2) + 2p_1xy \end{pmatrix}. \tag{21.12}$$

for k_1, k_2, p_1, p_2 parameters. It is common to ignore tangential distortion and focus on radial distortion by setting $p_1 = p_2 = 0$.

21.2.2 Lens Calibration: Formulating the Optimization Problem

Again, the optimization problem is relatively straightforward to formulate. Write $\hat{\mathbf{t}}_i = [t_{x,i}, t_{y,i}]$ for the measured x, y position in the image plane of the i'th reference point. We have that $\hat{\mathbf{t}}_i = \mathbf{t}_i + \xi_i$, where ξ_i is an error vector and $\mathbf{t}_i$ is the true (unknown) position of the i'th point. Again, assume the error is isotropic, so it is

natural to minimize

$$\sum_i \xi_i^T \xi_i. \tag{21.13}$$

We obtain expressions for $\xi_{i,j}$ in the appropriate coordinates as in Section 33.2, and using the notation of that section, but now accounting for the effects of the lens. We have

$$\sum_i \xi_i^T \xi_i = \sum_i (t_{x,i} - l_{x,i})^2 + (t_{y,i} - l_{y,i})^2 \tag{21.14}$$

where

$$\begin{aligned} l_{x,i} &= p_{x,i} + p_{x,i}(k_1(p_{x,i}^2 + p_{y,i}^2) + k_2(p_{x,i}^2 + p_{y,i}^2)^2) + p_1(p_{x,i}^2 + p_{y,i}^2 + 2p_{x,i}^2) + 2p_2 p_{x,i} p_{y,i} \\ l_{y,i} &= p_{y,i} + p_{y,i}(k_1(p_{x,i}^2 + p_{y,i}^2) + k_2(p_{x,i}^2 + p_{y,i}^2)^2) + p_2(p_{x,i}^2 + p_{y,i}^2 + 2p_{y,i}^2) + 2p_1 p_{x,i} p_{y,i} \end{aligned}$$

(which models the effect of the lens on the imaged points). The imaged points are

$$\begin{aligned} p_{x,i} &= \frac{i_{11} g_{x,i} + i_{12} g_{y,i} + i_{13} g_{z,i}}{g_{z,i}} \\ p_{y,i} &= \frac{i_{22} g_{x,i} + i_{23} g_{z,i}}{g_{z,i}} \end{aligned}$$

and, as before, we have

$$\begin{aligned} g_{x,i} &= e_{11} s_{x,i} + e_{12} s_{y,i} + e_{13} s_{z,i} + e_{14} \\ g_{y,i} &= e_{21} s_{x,i} + e_{22} s_{y,i} + e_{23} s_{z,i} + e_{24} \\ g_{z,i} &= e_{31} s_{x,i} + e_{32} s_{y,i} + e_{33} s_{z,i} + e_{34}. \end{aligned}$$

(which you should check as an exercise). As before, this is a constrained optimization problem, because $\mathcal{T}_e$ is a Euclidean transformation. The constraints here are

$$\begin{aligned} 1 - \sum_v e_{j,1v}^2 = 0 \text{ and } 1 - \sum_v e_{j,2v}^2 = 0 \text{ and } 1 - \sum_v e_{j,3v}^2 = 0 \\ \sum_v e_{j,1v} e_{j,2v} = 0 \text{ and } \sum_v e_{j,1v} e_{j,3v} = 0 \text{ and } 1 - \sum_v e_{j,2v} e_{j,3v} = 0 \end{aligned} .$$

As in Section 33.2, simply dropping this problem into a constrained optimizer is not a particularly good approach. If we assume the lens distortion is minor, we can obtain a start point for the intrinsics and the extrinsics using Section 33.2. We then use those parameters, together with $k_1 = 0$, $k_2 = 0$, $p_1 = 0$ and $p_2 = 0$, as a start point.

CHAPTER 22

A Camera Above a Ground Plane

Imagine a camera is moving above a ground plane. Using registration tools together with camera matrices makes means we can calibrate the camera's intrinsics, reason about the position and orientation of the camera, and reconstruct the pattern on the ground plane. In turn, this reconstruction can yield estimates of what objects are moving and whether there are objects that have relief ("stick out" from the ground).

22.1 PIPH: PERPENDICULAR IMAGE PLANE AND FIXED HEIGHT

Assume the camera moves at fixed height above a ground plane, and the ground plane is at right angles to the image plane (call this configuration *PIPH* for short). This is a good model for a camera on (say) an autonomous car or a taxiing aircraft. Figure 22.1 shows the notation, etc. Here the focal length is f, the ground plane is the plane $y = -h$ in the camera coordinate system (remember, z is depth into the scene). Remarkably, we can calibrate the camera with elementary geometric reasoning in a configuration like this, at least for simple cameras.

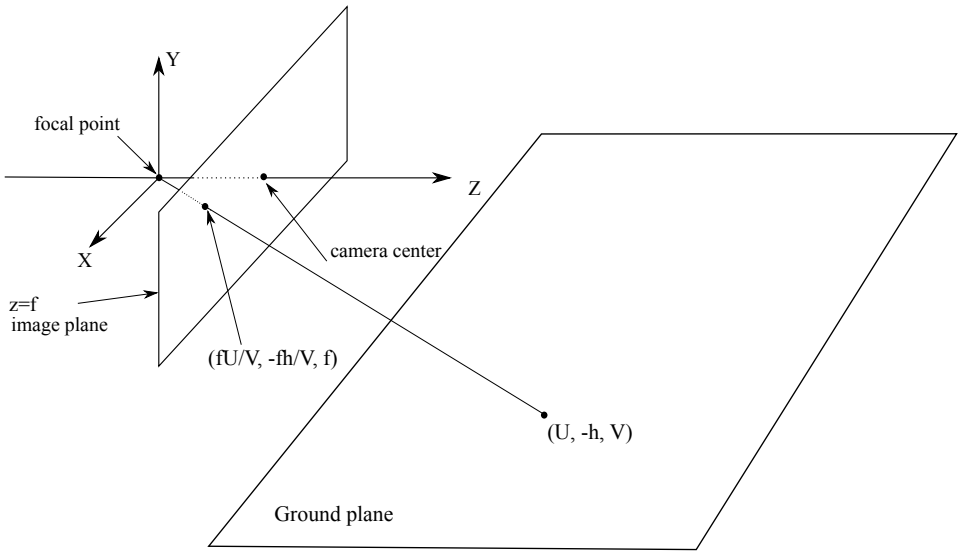


FIGURE 22.1: *A perspective camera with its image plane at right angles to a ground plane ($y = -h$), imaging a point on the ground plane.*

22.1.1 PIPH Geometry

From Figure 22.1, in the camera coordinate system, the point $(u, -h, v)$ on the ground plane in intersects the image plane at $(fu/v, -fh/v, f)$. These are affine coordinates for a point in 3D. Homogeneous coordinates for the point on the image plane are $(u/v, -h/v, 1)$ or equivalently $(u, -h, v)$. Similarly, homogeneous coordinates for the point on the ground plane are $(u, v, 1)$.

To get the transformation from the ground plane in world coordinates to the image in image coordinates, we must account for extrinsics and intrinsics. The homography from the ground plane to the image will be

$$\mathcal{T}_{\mathrm{g}\to\mathrm{i}} = \mathcal{T}_{\mathrm{int}} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -h \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}} = \begin{bmatrix} as & k & c_x \\ 0 & s & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -h \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}}.$$

Here $\mathcal{T}_{\mathrm{int}}$ comes from the camera intrinsics and $\mathcal{T}_{\mathrm{ext}}$ (which represents extrinsics) is a rotation and translation *in the ground plane*. This transformation is present because the coordinate system on the ground plane may not be directly below the focal point and aligned with the camera.

You should check that, in PIPH geometry, the horizon of the ground plane is horizontal in the image and passes through c_y (Figure 22.1 should help), so we can determine c_y from an image. Write (i_x, i_y) for the affine coordinates of a point in the image. If we ensure that the horizon is the line $i_y = 0$ (which we can do by a simple subtraction), then $c_y = 0$. In these coordinates, we have

$$\mathcal{T}_{\mathrm{g}\to\mathrm{i}} = \begin{bmatrix} as & k & c_x \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -h \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}} = \begin{bmatrix} as & c_x & -hk \\ 0 & 0 & -sh \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}}.$$

This is *not* an affine transformation. However

$$\begin{aligned} \mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{g}\to\mathrm{i}} \\ &= \begin{bmatrix} as & c_x & -hk \\ 0 & 1 & 0 \\ 0 & 0 & sh \end{bmatrix} \mathcal{T}_{\mathrm{ext}} \\ &\equiv \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & -\frac{k}{s} \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathcal{T}_{\mathrm{ext}} \end{aligned}$$

(recall $\equiv$ means that they are the same homography; one is a scaling of the other, which doesn't matter in homogeneous coordinates). This means $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}}$ is an affine transformation. This is a powerful fact. If we know some points on the ground plane and corresponding points in the image, we can recover $\mathcal{T}_{\mathrm{g}\to\mathrm{i}}$, premultiply by $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}$ (which we know), then read off some camera parameters. Remarkably, we can also estimate camera ground plane motion and the pattern on the ground plane *without* calibrating the camera. These estimates are up to scale – we cannot

get the magnitude of the translation or the size of objects on the ground plane without other information.

22.1.2 PIPH Calibration

Now assume we have a set of points $\mathbf{p}_i$ on the ground plane, and we can find the corresponding points $\mathbf{q}_i$ on the image plane. Fit $\mathcal{T}_{\mathrm{g}\rightarrow\mathrm{i}}$ to this information using procedure 11.2, to obtain $\mathcal{F}$. Now

$$\begin{aligned}\mathcal{C}_{\mathrm{g}\rightarrow\mathrm{i}}\mathcal{F} &\equiv \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & -\frac{k}{s} \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathcal{T}_{\text{ext}} \\ &= \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & -\frac{k}{s} \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathcal{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathcal{M} & \mathbf{u} \\ \mathbf{0}^T & 1 \end{bmatrix}\end{aligned}$$

where $\mathcal{T}_{\text{ext}}$ is a Euclidean transformation on the plane. We cannot recover $\frac{k}{s}$ from $\mathbf{u}$ because we don't know $\mathbf{t}$, the translation from the ground plane coordinate system to the camera coordinate system. However, for many cameras $k = 0$, and we assume this is the case for our camera. We have

$$\mathcal{M} = \begin{bmatrix} \frac{a}{h} & -\frac{c_x}{sh} \\ 0 & \frac{1}{sh} \end{bmatrix} \mathcal{R}$$

and we can factor $\mathcal{M}$ using an RQ factorization (see procedure 32.1). Doing so yields $\frac{h}{a}$, c_x and sh.

Notice there is a weakness in this procedure. The homography $\mathcal{T}_{\mathrm{g}\rightarrow\mathrm{i}}$ has a known, special parametric form and we did not impose this form when we estimated the homography. The correct way to resolve this is to minimize the error between $\mathcal{T}_{\mathrm{g}\rightarrow\mathrm{i}}(\mathbf{p}_i)$ and $\mathbf{q}_i$ for a homography of the correct form, using our estimates to provide a start point. This is an example of general recipe for calibration that we shall see again – first, make an estimate of parameters to provide a start point, then polish that estimate using an optimization problem

Procedure: 22.1 *PIPH Calibration: Overview*

Given a set of points $\mathbf{p}_i$ on the ground plane, corresponding points $\mathbf{q}_i$ on the image plane, and a camera known to be in PIPH geometry, estimate camera intrinsics and extrinsics by:

- Assuming $k = 0$;
- Obtaining a start point for $\frac{h}{a}$, c_x, sh and extrinsic parameters as below;
- Polishing the start point by optimization.

Procedure: 22.2 *PIPH Calibration: Initialization*

Initial: Fit $\mathcal{T}_{\mathrm{g}\to\mathrm{i}}$ to the points using procedure 11.2, to obtain $\mathcal{F}$. Now compute

$$\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{F} = \begin{bmatrix} \mathcal{M} & \mathbf{u} \\ \mathbf{0} & 1 \end{bmatrix}.$$

Intrinsics start point: Use an RQ factorization on $\mathcal{M}$ to obtain $\mathcal{M} = \mathcal{R}\mathcal{Q}$; $\mathcal{R}$ is upper triangular, and yields $\frac{h}{a}$, c_x, sh. **Extrinsics start point:** We have

$$\mathcal{T}_{\mathrm{ext}} = \begin{bmatrix} \mathcal{Q} & \mathbf{u} \\ \mathbf{0} & 1 \end{bmatrix}$$

Procedure: 22.3 *PIPH Calibration: Optimization*

Solve the optimization problem

$$\sum_i (q_{i,x} - w_{i,x})^2 + (q_{i,y} - w_{i,y})^2$$

where

$$\begin{aligned} w_{i,x} &= \frac{h_{11}p_{i,x} + h_{12}p_{i,y} + h_{13}}{h_{31}p_{i,x} + h_{32}p_{i,y} + h_{33}} \\ w_{i,y} &= \frac{h_{21}p_{i,x} + h_{22}p_{i,y} + h_{23}}{h_{31}p_{i,x} + h_{32}p_{i,y} + h_{33}} \\ \mathcal{H} &= \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & 0 \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathcal{Q} & \mathbf{u} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ \mathcal{Q} &= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \end{aligned}$$

and the parameters are $\frac{a}{h}$, c_x, sh, θ and $\mathbf{u}$ using the start point of procedure 22.2.

22.1.3 Using PIPH to Estimate Motion

Imagine the camera captures an image $\mathcal{I}_n$ at frame n, moves rigidly, then captures $\mathcal{I}_{n+1}$. The camera image plane stays perpendicular to the ground plane, and the height of the focal point doesn't change. We can recover the camera motion and some camera parameters in this case. We know that $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}_n}$ and $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}_{n+1}}$ are both affine. Notice that

$$\mathcal{T}_{\mathrm{i}_n\to\mathrm{i}_{n+1}} = \mathcal{T}_{\mathrm{g}\to\mathrm{i}_{n+1}}\mathcal{T}^{-1}_{\mathrm{g}\to\mathrm{i}_n}.$$

We can *measure* $\mathcal{T}_{\mathrm{i}_n \to \mathrm{i}_{n+1}}$ by finding interest points in the two images, then using Procedure 11.2. Write $\mathcal{F}$ for the measured transformation. We must have that

$$\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{F}\mathcal{C}^{-1}_{\mathrm{g}\to\mathrm{i}} = \mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}_{n+1}}\mathcal{T}^{-1}_{\mathrm{g}\to\mathrm{i}_n}\mathcal{C}^{-1}_{\mathrm{g}\to\mathrm{i}} = \left[\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}_{n+1}}\right]\left[\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}_n}\right]^{-1}$$

and so $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{F}\mathcal{C}^{-1}_{\mathrm{g}\to\mathrm{i}}$ is affine. In fact, we know the form of this matrix, which is

$$\begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & 0 \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathcal{T}_{\mathrm{ext}_{n+1}}\mathcal{T}^{-1}_{\mathrm{ext}_n} \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & 0 \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1}.$$

Now $\mathcal{E}_{n\to n+1} = \mathcal{T}_{\mathrm{ext}_{n+1}}\mathcal{T}^{-1}_{\mathrm{ext}_n}$ is the camera motion on the ground plane. Notice that

$$\begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & 0 \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \frac{c_x}{sa} & 0 \\ 0 & \frac{1}{sa} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{a}{h} & 0 & 0 \\ 0 & \frac{a}{h} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and recall that isotropic scaling commutes with rotation (Section 33.2), to find that

$$\mathcal{F} = \begin{bmatrix} 1 & \frac{c_x}{sa} & 0 \\ 0 & \frac{1}{sa} & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathcal{E} \begin{bmatrix} 1 & \frac{c_x}{sa} & 0 \\ 0 & \frac{1}{sa} & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1}.$$

Now write $\mathcal{M} = \mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{F}\mathcal{C}^{-1}_{\mathrm{g}\to\mathrm{i}}$. The upper 2×2 block of $\mathcal{M}$ is

$$\begin{bmatrix} \cos\theta + \frac{c_x}{sa}\sin\theta & -(sa + \frac{c_x^2}{sa})\sin\theta \\ \frac{1}{sa}\sin\theta & \cos\theta - \frac{c_x}{sa}\sin\theta \end{bmatrix}$$

so we can recover the rotation from

$$\cos\theta = m_{11} + m_{22},$$

and some calibration parameters from

$$\begin{aligned} s^2a^2 &= \frac{1-\cos^2\theta}{m_{12}^2} \\ \left(\frac{c_x}{as}\right)^2 &= \frac{(m_{22}-\cos\theta)^2}{(1-\cos^2\theta)}. \end{aligned}$$

This means we can recover as and c_x if we can determine the signs of the square roots. But a and s are necessarily positive and we can obtain the sign of c_x by elementary reasoning about the camera, so we can recover the signs. Now if the camera translates by $[t_x, t_y]$, then the translation component of $\mathcal{M}$ is

$$\begin{bmatrix} m_{13} \\ m_{23} \end{bmatrix} = \begin{bmatrix} -\frac{h}{a}t_x + \frac{hc_x}{a}t_y \\ -sht_y \end{bmatrix}$$

so that

$$\frac{-(as)m_{13} + \frac{c_x}{as}m_{23}}{-m_{23}} = \frac{t_x}{t_y}.$$

Now we *observe* u and v, and can recover as and c_x, so we know the *direction* but not magnitude of the translation. Equivalently, we can recover the movement of the camera up to scale.

Procedure: 22.4 *PIPH Motion Estimation*

Given an image $\mathcal{I}_n$ at frame n and $\mathcal{I}_{n+1}$ at frame $n+1$, where an uncalibrated camera with $k = 0$ moves rigidly, with image plane perpendicular to the ground plane, and the height of the focal point fixed but unknown, obtain an estimate of $\mathcal{T}_{\mathcal{I}_n \to \mathcal{I}_{n+1}}$; write $\mathcal{F}$ for this estimate. Find this by identifying interest points in $\mathcal{I}_n$ and $\mathcal{I}_{n+1}$, and fitting a homography to these interest points (Procedure 11.2). Then $\mathcal{M} = \mathcal{C}_{\mathrm{g}\to\mathrm{i}} \mathcal{F} \mathcal{C}_{\mathrm{g}\to\mathrm{i}}^{-1}$ is affine. We have

$$\begin{aligned} \cos\theta &= m_{11} + m_{22} \\ s^2a^2 &= \frac{1-\cos^2\theta}{m_{12}^2} \\ \left(\frac{c_x}{as}\right)^2 &= \frac{(m_{22}-\cos\theta)^2}{(1-\cos^2\theta)} \end{aligned}$$

yielding rotation and some camera parameters. The translation is recovered from

$$\frac{-(as)m_{13} + \frac{c_x}{as}m_{23}}{-m_{23}} = \frac{t_x}{t_y}.$$

22.1.4 The Pattern on the Ground Plane

We can recover the pattern on the ground plane up to scale as well from two images. Write the true pattern $\mathcal{P}$. Recall that $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}\mathcal{T}_{\mathrm{g}\to\mathrm{i}}$ is affine, which means that $\mathcal{T}_{\mathrm{i}\to\mathrm{g}}\mathcal{C}_{\mathrm{g}\to\mathrm{i}}^{-1}$ is affine as well (Section 33.2). This means that if we apply the homography $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}$ to the image, we will obtain a pattern that is within an affine transformation of the ground plane, and we can determine the form of the affine transformation. This is easy to do, because $\mathcal{C}_{\mathrm{g}\to\mathrm{i}}$ is known.

We have

$$\mathcal{T}_{\mathrm{g}\to\mathrm{i}} = \begin{bmatrix} as & c_x & -hk \\ 0 & 0 & -sh \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}}.$$

so that

$$\mathcal{T}_{\mathrm{i}\to\mathrm{g}}\mathcal{C}_{\mathrm{g}\to\mathrm{i}}^{-1} = \mathcal{T}_{\mathrm{ext}}^{-1} \begin{bmatrix} as & c_x & -hk \\ 0 & 0 & -sh \\ 0 & 1 & 0 \end{bmatrix}^{-1} \mathcal{C}_{\mathrm{g}\to\mathrm{i}}^{-1}.$$

Now as in Section 22.1.2, we assume $k = 0$. We have

$$\mathcal{C}_{\mathrm{g}\to\mathrm{i}} \begin{bmatrix} as & c_x & 0 \\ 0 & 0 & -sh \\ 0 & 1 & 0 \end{bmatrix} \equiv \begin{bmatrix} \frac{a}{h} & \frac{c_x}{sh} & 0 \\ 0 & \frac{1}{sh} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

so that

$$\mathcal{T}_{\mathrm{i}\to\mathrm{g}}\mathcal{C}^{-1}_{\mathrm{g}\to\mathrm{i}} = \mathcal{T}^{-1}_{\mathrm{ext}} \begin{bmatrix} \frac{h}{a} & 0 & 0 \\ 0 & \frac{h}{a} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -c_x & 0 \\ 0 & as & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

In turn, if we know c_x and as, we can recover the image plane pattern up to scale. As Section 22.1.3 shows, these parameters can be estimated from two distinct views of the ground plane.

Procedure: 22.5 *PIPH Pattern Estimation*

Given an image $\mathcal{I}_n$ at frame n and $\mathcal{I}_{n+1}$ at frame $n+1$, where an uncalibrated camera with $k = 0$ moves rigidly, with image plane perpendicular to the ground plane, and the height of the focal point fixed but unknown, obtain camera parameters as and c_x from procedure 22.4. Write

$$\mathcal{T}_{\mathrm{partial}} = \begin{bmatrix} 1 & -c_x & 0 \\ 0 & as & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then the ground plane pattern is within a scale of

$$\mathcal{T}^{-1}_{\mathrm{partial}}\mathcal{C}_{\mathrm{g}\to\mathrm{i}}(\mathcal{I}_n)$$

22.1.5 Off Perpendicular Image Planes

All the procedures above can be extended to deal with an off-perpendicular image plane if one is allowed a single calibration step. This step essentially estimates the angle between the image plane and the ground plane. In particular, notice that the methods of Section 22.1.3 and 22.5 depend on the fact that a *known* homography applied to the image yields something that is within an affine transformation of the ground plane.

When the image plane is not perpendicular to the ground plane, the homography from ground plane to image can be derived from Figure 33.2 (assuming $k = 0$) as

$$\mathcal{P}_{\mathrm{g}\to\mathrm{i}} = \begin{bmatrix} as & 0 & c_x \\ 0 & s & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \gamma & -h \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}} = \begin{bmatrix} as & c_x & -hk \\ 0 & s\gamma + c_y & -hs \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_{\mathrm{ext}}.$$

You should check that

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & \frac{1}{s\gamma + c_y} & -1 \end{bmatrix} \mathcal{P}_{\mathrm{g}\to\mathrm{i}} \equiv \begin{bmatrix} \frac{a}{h} & \frac{c_x}{hs} & 0 \\ 0 & \frac{1}{hs} & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathcal{T}_{\mathrm{ext}}.$$

This means that, if we can estimate $s\gamma + c_y$, we can apply the strategies of the previous section. A simple strategy for doing so is to image a set of reference points on the ground plane, compute the homography from ground plane to image $\mathcal{P}_{\mathrm{g}\to\mathrm{i}}$, then obtain

$$\mathcal{D}_w = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & w & -1 \end{bmatrix}$$

such that $\mathcal{D}_w\mathcal{P}_{\mathrm{g}\to\mathrm{i}}$ is affine.

22.1.6 PIPH Mosaics

22.2 CAMERA CALIBRATION FROM PLANE REFERENCES

It is possible to calibrate a camera fully with a plane calibration object, but you need to have several views. Plane patterns are easy to make and easy to disseminate. Obtain a plane pattern with a set of easily localized points (a checkerboard is good) where the locations of those points on the plane are known in world units. So if one is using a checkerboard, one might know that the checks are square and 10cm on edge, for example. Lay this down flat, and take a set of images of it from different views. In each view the calibration points should be visible.

For each view, we will compute the homography from the calibration object's plane to the image plane using point correspondences (Section 33.2). It turns out that these homographies yield constraints on the camera matrix (Section 33.2) and these constraints yield a camera estimate (Section 33.2). This estimate is a start point for an optimization problem (Section 33.2, very much on the lines of Section 33.2).

22.2.1 Constraining Intrinsics with Homographies

Each map from the pattern to an image is a homography. Choose the world coordinate system so that the pattern lies on the plane $z = 0$. Doing so just changes the camera *extrinsics*, so no generality has been lost, but it allows us to write the homography in a useful form. Recall the camera is

$$\mathcal{T}_i\mathcal{C}_p\mathcal{T}_{e,j}$$

where $\mathcal{T}_{e,j}$ is the euclidean transformation giving the extrinsics for the j'th view. This is applied to a set of points $(s_{x,i}, s_{y,i}, 0, 1)$. In turn, the homography for the j'th view must have the form

$$\lambda_j\mathcal{M}_j = \mathcal{T}_i\left[\mathbf{r}_{1,j}, \mathbf{r}_{2,j}, \mathbf{t}_j\right]$$

(where $\mathbf{r}_{1,j}$, $\mathbf{r}_{2,j}$ are the first two *columns* of the rotation matrix in $\mathcal{T}_{e,j}$ and $\mathbf{t}_j$ is the translation). We do not know λ_j (which is non-zero) because scaling the homography matrix yields the same homography. Now write $\mathcal{N}_j = \mathcal{T}_i^{(-1)}\mathcal{M}_j = [\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3]$. We must have that

$$\mathbf{n}_1^T\mathbf{n}_1 - \mathbf{n}_2^T\mathbf{n}_2 = 0 \text{ and } \mathbf{n}_1^T\mathbf{n}_2 = 0.$$

These equations constrain the unknown values of $\mathcal{T}_i$, and we get two for each homography. In turn, with sufficient views (and so homographies), we can estimate $\mathcal{T}_i$.

22.2.2 Estimating Intrinsics from Homographies

In the j'th view of the plane calibration object, we recover a homography $\mathcal{M}_j$. Now write $\mathcal{N}_j = \mathcal{T}_i^{(-1)}\mathcal{M}_j = [\mathbf{n}_{j,1}, \mathbf{n}_{j,2}, \mathbf{n}_{j,3}]$. We know from Section 33.2 that

$$\mathbf{n}_{j,1}^T\mathbf{n}_{j,1} - \mathbf{n}_{j,2}^T\mathbf{n}_{j,2} = 0 \text{ and } \mathbf{n}_{j,1}^T\mathbf{n}_{j,2} = 0.$$

Now write $\mathcal{A} = (\mathcal{T}_i^{(-T)}\mathcal{T}_i^{(-1)})$ (which is unknown). These two constraints are linear homogenous equations in the entries of $\mathcal{A}$, which is 3×3 but symmetric, and so has 6 unknown parameters. If we have 3 homographies, we will have 6 constraints, and can use least squares to recover a 1D family of solutions $\lambda\mathcal{B}$, where $\mathcal{B}$ is *known* and λ is a scale. We now need to find $\mathcal{T}_i$ and λ so that $\lambda\mathcal{B}$ is close to $(\mathcal{T}_i^{(-T)}\mathcal{T}_i^{(-1)}))$.

There are constraints here. Write $\mathcal{U} = \mathcal{T}_i^{(-1)}$. Recall $\mathcal{T}_i$ is upper triangular, and $i_{33} = 1$. This means that $\mathcal{U}$ is upper triangular, and $u_{33} = 1$. We will find $\mathcal{U}$ and λ by finding $\mathcal{V}$ such that $\mathcal{V}^T\mathcal{V}$ is closest to $\mathcal{B}$, then computing $\mathcal{U} = (1/v_{33})\mathcal{V}$.

Finding $\mathcal{V}$ is straightforward. We obtain the closest symmetric matrix to $\mathcal{B}$, then apply a Cholesky factorization (Section 33.2). The factorization could be modified if a very small number appears on the diagonal, but this event is most unlikely. We now invert $\mathcal{U}$ to obtain an estimate $\mathcal{E}$ of $\mathcal{T}_i$. Recall this has the form

$$\begin{bmatrix} as' & k' & c_x' \\ 0 & s' & c_y' \\ 0 & 0 & 1 \end{bmatrix}.$$

so we have $c_x' = e_{13}$, $c_y' = e_{23}$, $s' = e_{22}$, $a = e_{11}/e_{22}$ and $k' = e_{12}$. This is usually an acceptable start point for optimization.

22.2.3 Estimating Extrinsics from Homographies

We have an estimate of the camera intrinsics, and now need an estimate of the extrinsics for each view. Recall from Section **??** that

$$\lambda_j\mathcal{M}_j = \mathcal{T}_i\left[\mathbf{r}_{1,j}, \mathbf{r}_{2,j}, \mathbf{t}_j\right]$$

(where $\mathbf{r}_{1,j}$, $\mathbf{r}_{2,j}$ are the first two *columns* of the rotation matrix in $\mathcal{T}_{e,j}$ and $\mathbf{t}_j$ is the translation). We have estimates of $\mathcal{M}_j$ and of $\mathcal{T}_i$, but we do not know λ_j. We can solve for λ_j by noticing that the first two columns of

$$\lambda_j\mathcal{T}_i^{-1}\mathcal{M}_j = \lambda_j\mathcal{Q}_j = \lambda_j\left[\mathbf{q}_{1,j}, \mathbf{q}_{2,j}, \mathbf{q}_{3,j}\right]$$

are unit vectors, and are normal to one another. For example, we might estimate

$$\lambda_j = \sqrt{\frac{2}{\mathbf{q}_{1,j}^T\mathbf{q}_{1,j} + \mathbf{q}_{2,j}^T\mathbf{q}_{2,j}}}$$

and from this follows the estimate

$$\mathcal{T}_{e,j} = \begin{bmatrix} \lambda_j\mathbf{q}_{1,j} & \lambda_j\mathbf{q}_{2,j} & \lambda_j^2\mathbf{q}_{1,j} \times \mathbf{q}_{2,j} & \lambda_j\mathbf{q}_{3,j} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

22.2.4 Formulating the Optimization Problem

We will calibrate the camera by solving an optimization problem. All calibration points will lie on the plane $z = 0$ in world coordinates, and we will have more than one view of that plane. Write $\mathbf{t}_{ij} = [t_{x,ij}, t_{y,ij}]$ for the measured x, y position in the image plane of the i'th reference point in the j'th view. We have that $\mathbf{t}_{ij} = \hat{\mathbf{t}}_{ij} + \xi_{ij}$, where ξ_{ij} is an error vector and $\hat{\mathbf{t}}_{ij}$ is the true (unknown) position. Again, assume the error is isotropic, so it is natural to minimize

$$\sum_{ij} \xi_{ij}^T \xi_{ij}.$$

The main issue here is writing out expressions for ξ_{ij} in the appropriate coordinates. Write $\mathcal{T}_i$ for the intrinsic matrix whose u, v'th component will be i_{uv}; $\mathcal{T}_{e,j}$ for the j'th extrinsic transformation, whose u, v'th component will be e_{uv}; and $\mathbf{s}_i = [s_{x,i}, s_{y,i}, 0]$ for the known coordinates of the i'th reference point in the coordinate frame of the reference points. Recalling that $\mathcal{T}_i$ is lower triangular, and engaging in some manipulation, we obtain

$$\sum_{ij} \xi_{ij}^T \xi_{ij} = \sum_{i} (t_{x,ij} - p_{x,ij})^2 + (t_{y,ij} - p_{y,ij})^2$$

where

$$\begin{aligned} p_{x,ij} &= \frac{i_{11} g_{x,ij} + i_{12} g_{y,ij} + i_{13} g_{z,ij}}{g_{z,ij}} \\ p_{y,ij} &= \frac{i_{22} g_{x,ij} + i_{23} g_{z,ij}}{g_{z,ij}} \end{aligned}$$

and

$$\begin{aligned} g_{x,ij} &= e_{11,j} s_{x,i} + e_{12,j} s_{y,i} + e_{14,j} \\ g_{y,ij} &= e_{21,j} s_{x,i} + e_{22,j} s_{y,i} + e_{24,j} \\ g_{z,ij} &= e_{31,j} s_{x,i} + e_{32,j} s_{y,i} + e_{34,j} \end{aligned}$$

(which you should check as an exercise – notice the missing $s_{z,i}$ terms!). This is a constrained optimization problem, because $\mathcal{T}_e$ is a Euclidean transformation. The constraints here are

$$1 - \sum_v e_{j,1v}^2 = 0 \text{ and } 1 - \sum_v e_{j,2v}^2 = 0 \text{ and } 1 - \sum_v e_{j,3v}^2 = 0$$
$$\sum_v e_{j,1v} e_{j,2v} = 0 \text{ and } \sum_v e_{j,1v} e_{j,3v} = 0 \text{ and } 1 - \sum_v e_{j,2v} e_{j,3v} = 0 \quad .$$

As in Section 33.2, we could just throw this into a constrained optimizer (review Section 33.2), but good behavior requires a good start point.

Procedure: 22.6 *Calibrating a Camera from Multiple Homographies*

Procedure: 22.7 *Calibrating a Camera from Multiple Homographies: Start Point*

PART SIX

WORKING WITH TWO IMAGES

CHAPTER 23

Pairs of Cameras

23.1 GEOMETRY

Two perspective cameras view a point $\mathbf{X}$ in 3D; we see $\mathbf{x}_1$ in the first camera and $\mathbf{x}_2$ in the second. As section 33.2 sketched, knowing something about the relative geometry of the cameras and where the point appears in each camera will reveal the 3D coordinates of the point. We will use some form of search to link points in the first and second cameras. But not any point in camera 2 could correspond to $\mathbf{x}_1$, and understanding this constraint reveals a great deal of information about the relative configuration of the cameras.

All of this geometry can be done without using coordinates. Figure 33.2 shows two general perspective cameras viewing a point. Notice the line joining the two focal points of the cameras. This line intersects each image plane in an important point, known as the *epipole* for that image plane.

23.1.1 The Fundamental and Essential Matrices

23.2 INFERENCE

23.2.1 Recovering Fundamental Matrices from Correspondences

23.2.2 RANSAC: Searching for Good Points

An alternative to modifying the cost function is to search the collection of data points for good points. This is quite easily done by an iterative process: First, we choose a small subset of points and fit to that subset, then we see how many other points fit to the resulting object. We continue this process until we have a high probability of finding the structure we are looking for.

For example, assume that we are fitting a line to a data set that consists of about 50% outliers. We can fit a line to only two points. If we draw pairs of points uniformly and at random, then about a quarter of these pairs will consist entirely of good data points. We can identify these good pairs by noticing that a large collection of other points lie close to the line fitted to such a pair. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

? formalized this approach into an algorithm — search for a random sample that leads to a fit on which many of the data points agree. The algorithm is usually called `RANSAC`, for RANdom SAmple Consensus, and is displayed in Algorithm 23.1. To make this algorithm practical, we need to choose three parameters.

The Number of Samples Required Our samples consist of sets of points drawn uniformly and at random from the data set. Each sample contains the minimum number of points required to fit the abstraction we wish to fit. For

Determine:
    n — the smallest number of points required (eg., for lines, $n = 2$,
      for circles, $n = 3$)
    k — the number of iterations required
    t — the threshold used to identify a point that fits well
    d — the number of nearby points required
      to assert a model fits well
Until k iterations have occurred
    Draw a sample of n points from the data
      uniformly and at random
    Fit to that set of n points
    For each data point outside the sample
      Test the distance from the point to the structure
        against t; if the distance from the point to the structure
        is less than t, the point is close
    end
    If there are d or more points close to the structure
      then there is a good fit. Refit the structure using all
      these points. Add the result to a collection of good fits.
end
Use the best fit from this collection, using the
  fitting error as a criterion

Algorithm 23.1: *RANSAC: fitting structures using random sample consensus*

example, if we wish to fit lines, we draw pairs of points; if we wish to fit circles, we draw triples of points, and so on. We assume that we need to draw n data points, and that w is the fraction of these points that are good (we need only a reasonable estimate of this number). Now the expected value of the number of draws k required to get one point is given by

$$\begin{aligned}\mathrm{E}[k] &= 1P(\text{one good sample in one draw}) + 2P(\text{one good sample in two draws}) + \ldots \\ &= w^n + 2(1-w^n)w^n + 3(1-w^n)^2 w^n + \ldots \\ &= w^{-n}\end{aligned}$$

(where the last step takes a little manipulation of algebraic series). We would like to be fairly confident that we have seen a good sample, so we wish to draw more than w^{-n} samples; a natural thing to do is to add a few standard deviations to this number. The standard deviation of k can be obtained as

$$SD(k) = \frac{\sqrt{1-w^n}}{w^n}.$$

An alternative approach to this problem is to look at a number of samples that guarantees a low probability z of seeing only bad samples. In this case, we have

$$(1-w^n)^k = z,$$

which means that

$$k = \frac{\log(z)}{\log(1 - w^n)}.$$

It is common to have to deal with data where w is unknown. However, each fitting attempt contains information about w. In particular, if n data points are required, then we can assume that the probability of a successful fit is w^n. If we observe a long sequence of fitting attempts, we can estimate w from this sequence. This suggests that we start with a relatively low estimate of w, generate a sequence of attempted fits, and then improve our estimate of w. If we have more fitting attempts than the new estimate of w predicts, then the process can stop. The problem of updating the estimate of w reduces to estimating the probability that a coin comes up heads or tails given a sequence of fits.

Telling Whether a Point Is Close We need to determine whether a point lies close to a line fitted to a sample. We do this by determining the distance between the point and the fitted line, and testing that distance against a threshold d; if the distance is below the threshold, the point lies close. In general, specifying this parameter is part of the modeling process. In general, obtaining a value for this parameter is relatively simple. We generally need only an order of magnitude estimate, and the same value applies to many different experiments. The parameter is often determined by trying a few values and seeing what happens; another approach is to look at a few characteristic data sets, fitting a line by eye, and estimating the average size of the deviations.

The Number of Points That Must Agree Assume that we have fitted a line to some random sample of two data points. We need to know whether that line is good. We do this by counting the number of points that lie within some distance of the line (the distance was determined in the previous section). In particular, assume that we know the probability that an outlier lies in this collection of points; write this probability as y. We should like to choose some number of points t such that the probability that all points near the line are outliers, y^t, is small (say less than 0.05). Notice that $y \leq (1 - w)$ (because some outliers should be far from the line) so we could choose t such that $(1 - w)^t$ is small.

23.2.3 Visual Odometry

CHAPTER 24

Stereopsis

CHAPTER 25

Optic Flow

PART SEVEN

WORKING WITH IMAGE SEQUENCES

CHAPTER 26

Structure from Motion

CHAPTER 27

Filtering

CHAPTER 28

Tracking

CHAPTER 29

SLAM

29.1 EKF-SLAM

CHAPTER 30

3D Models from Images

TOOLS

CHAPTER 31

Classification and Basic Neural Networks

31.1 LOGISTIC REGRESSION

31.1.1 Classifier Basics

A *classifier* is a predictor that accepts some description – say, an image or features describing that image – and predicts a class. Typically, classifiers are learned from *labelled data*, a set of N examples $\mathbf{x}_i$ each with a class label y_i, where the class label is taken from a total of C classes. Learning takes these examples and produces a predictor that can predict the class of future examples. Generally, this predictor is chosen to predict training samples well. Much of what follows will deal with *how* this is done for various different kinds of predictors, but here we deal with generalities that apply to all cases.

Typically, we apply a classifier to data that hasn't been seen in training and whose labels aren't (and may never be) known. For example, we might need to label transactions as "sound" or "fraudulent", but may never investigate whether the label was right. A good classifier is one that causes its user the least loss (or the most profit) when used. In the example, it might be profitable to tolerate some level of fraud (perhaps because doing so means that customers are not put off). Measuring how good a classifier is in these terms requires knowing the expected cost of errors, which is often difficult. It is quite usual to evaluate *accuracy* (the fraction of classification attempts that get the right answer) or *error rate* (the fraction of classification attempts that get the wrong answer) instead.

What is important is accuracy on *future data* that hasn't been seen in training. For this to occur, training data must be "similar" to future data in some way. Assume future data are samples from the joint distribution $P(\mathbf{x}, y)$, though we don't see y. It is possible to prove a variety of bounds on error if training data are independent identically distributed (IID) samples from that joint distribution, but little is known about other cases as of writing. An important mystery in computer vision is that future data is quite often somewhat different from training data, without any major problems occurring.

Notation: 31.1 $\mathbb{I}_{[f(\mathbf{x})=y]}(\mathbf{x}, y)$*: Indicator function*

$\mathbb{I}_{[f(\mathbf{x})=y]}(\mathbf{x}, y)$ is the *indicator function* that takes the value 1 when its condition (here $f(\mathbf{x}) = y$) is true, otherwise 0.

Notation: 31.2 $\mathbb{E}_p[f]$*: Expectation*

$\mathbb{E}_p[f]$ is the expectation of f under the probability distribution p.

Assume both training and future data are IID samples from some joint distribution $P(\mathbf{x}, y)$. The accuracy of a classifier $f(\mathbf{x})$ is $\mathbb{E}_{P(\mathbf{x},y)}\left[\mathbb{I}_{[f(\mathbf{x})=y]}(\mathbf{x}, y)\right]$. Because we don't know $P(\mathbf{x}, y)$ we cannot compute this directly, but if we had a set of IID samples – a *test set* $\mathcal{T}$ – from this distribution we could estimate the accuracy as

$$\mathbb{E}_{P(\mathbf{x},y)}\left[\mathbb{I}_{[f(\mathbf{x})=y]}(\mathbf{x}, y)\right] \approx \frac{1}{N_t}\sum_{u\in\mathcal{T}} \mathbb{I}_{[f(\mathbf{x}_u)=y_u)]}(\mathbf{x}_u, y_u)$$

(which is the fraction of the samples in $\mathcal{T}$ that are correctly classified by f). There is one very important caveat. The accuracy of f on training samples must be an optimistic estimate of accuracy, because f was chosen to be accurate on those training samples. This means that $\mathcal{T}$ must consist of examples that were not used in training. So we take the labelled data, split it into two components (train and test), and then use one to train f and the other to evaluate f. One can improve the estimate of accuracy by *cross-validation* – repeating this process using different random splits, then averaging the resulting estimates of accuracy.

Remember this: *A classifier predicts a label from a representation. Classifiers are evaluated by accuracy or error rate, estimated on data* not used in training*. The standard recipe splits training data into two components (train and test), uses one to train the classifier and the other to evaluate it. Never evaluate on data that was used in training, because your estimate will be wrong.*

31.1.2 Logistic Regression

To classify, we will construct a model of $P(c = v|\mathbf{x})$ (the *posterior distribution* of the class) and then choose the class with the highest value of the posterior. The model

$$\log P(c = v|\mathbf{x}) = \mathbf{w}_v^T\mathbf{x} + b_v + K$$

results in a classification procedure known as *logistic regression*. The K – which is the same for each class – ensures that the probabilities sum to one. A simple calculation yields that

$$P(c = v|\mathbf{x}) = \frac{e^{\mathbf{w}_v^T\mathbf{x}+b_v}}{\sum_{j=1}^{C} e^{\mathbf{w}_v^T\mathbf{x}+b_v}}.$$

Notation: 31.3 $\mathbf{s}(\mathbf{u})$: *Softmax*

The *softmax* function of a d dimensional vector $\mathbf{u}$ is given by

$$\mathbf{s}(\mathbf{u}) = [e^{u_1}, \ldots, e^{u_d}] \left(\frac{1}{\sum_{j=1}^{d} e^{u_d}} \right).$$

Now write $\mathcal{W} = \left[{\mathbf{w_1}}^T; \ldots; \mathbf{w}_C^T \right]$ and $\mathbf{b} = [b_1; \ldots; b_C]$; then the vector of posterior probabilities (one per class) can be written as

$$\begin{aligned} \mathbf{s}(\mathcal{W}\mathbf{x} + \mathbf{b}) &= \mathbf{s}(\mathbf{u}) \\ &\text{where} \\ \mathbf{u} &= \mathcal{W}\mathbf{x} + \mathbf{b} \end{aligned}$$

where we adjust $\theta = \{\mathcal{W}, \mathbf{b}\}$ to obtain the best classification results.

31.1.3 The Cross-entropy Loss and Regularization

The essential difficulty here is to choose the $\theta = \{\mathcal{W}, \mathbf{b}\}$ that results in the best behavior. We do so by writing a cost function that estimates the error rate of the classification, then searching for θ that makes that function small. A natural cost function looks at the log likelihood of the data under the probability model produced from the outputs of the units. If the i'th example is from class j, we would like

$$-\log p(\text{class} = j | \mathbf{x}_i, \theta)$$

to be small (notice the sign here; it's usual to minimize negative log likelihood). We can compress notation by encoding the class of an example using a *one hot* vector $\mathbf{y}_i$, which is C dimensional. If the i'th example is from class j, then the j'th component of $\mathbf{y}_i$ is 1, and all other components in the vector are 0. I will write y_{ij} for the j'th component of $\mathbf{y}_i$. The components of $\mathbf{y}_i$ can be used as switches to obtain a loss function

$$\begin{aligned} \frac{1}{N} \sum_i L_{\log}(y_i, \mathbf{x}_i, \theta) &= \frac{1}{N} \sum_i \left[-\log p(\text{class of example } i | \mathbf{x}_i, \theta) \right] \\ &= \frac{1}{N} \sum_{i \in \text{data}} \left[\left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathcal{M}\mathbf{x}_i + \mathbf{b}) \right\} \right] \end{aligned}$$

(recall the j'th component of $\log \mathbf{s}$ is $\log s_j$). This loss is variously known as *log-loss* or *cross-entropy loss*.

Remember this: *Write θ for the parameters of a classifier. The log-loss or cross-entropy loss is given by $\frac{1}{N} \sum_i [-\log p(\text{class of example } i | \mathbf{x}_i, \theta)]$*

The goal of training is to find a classifier that performs well on held out data. Such a classifier should have small loss on the training data, but this is not sufficient. Imagine $\mathbf{x}_i$ is a training example, and $\mathbf{x}$ is near this training example, so that $\|\mathbf{x} - \mathbf{x}_i\|_2$ is small. It is likely that $\mathbf{x}$ will have the same label as $\mathbf{x}_i$ if the two are close enough, but if $\|\mathcal{W}(\mathbf{x} - \mathbf{x}_i)\|_2$ is big, then the classifier might predict a different label. In turn, ensuring that $\|\mathcal{W}(\mathbf{x} - \mathbf{x}_i)\|_2$ is not much larger than $\|\mathbf{x} - \mathbf{x}_i\|_2$ is likely to improve performance on held out data.

The *Frobenius norm* of a matrix $\mathcal{M}$ is $\|\mathcal{M}\|_F = \sum_{ij} m_{ij}^2$. It has the property that $\|\mathcal{M}\mathbf{x} - \mathcal{M}\mathbf{x}_i\|_2 \leq \|\mathcal{M}\|_F \|\mathbf{x} - \mathbf{x}_i\|_2$. In turn, we would like to achieve a small loss with a $\mathcal{M}$ that has small Frobenius norm. For some (currently unknown) λ, we will minimize

$$\frac{1}{N} \sum_i L_{\log}(y_i, \mathbf{x}_i, \mathcal{M}) + \lambda \|\mathcal{M}\|_F$$

(sometimes known as the *regularized training loss*; the term $\|\mathcal{M}\|_F$ is one example of a *regularizer*).

Remember this: *Generally, one regularizes a training loss to control error on future examples.*

We will choose the value of λ by search. Experience shows that quite large changes in λ have relatively little effect on the classifier, so we search a discrete set of values (typically, in decades, so $1e-3, 1e-2, 1e-1$ and so on). For each value of λ, we find and evaluate a classifier. We do this by splitting the training set into a training portion and validation portion; training on the training portion with the chosen value of λ; then evaluating on the validation portion. Because the classifier was not trained on the validation portion, the estimate of accuracy is unbiased and we can use it to choose λ. However, we do not have the best possible classifier for that value of λ because we did not use the validation portion in training. We retrain using the selected value of λ and all the training data. Finally, we evaluate this classifier using the test dataset.

Procedure: 31.1 *Evaluating a classifier for unknown regularization constant λ*

- If this hasn't been done, split the labelled data into training and test sets.
- Choose a discrete set of likely values, typically in decades, so $1e-3, 1e-2, 1e-1$ and so on.
- For each value of λ, find and evaluate a classifier, by:
  - splitting the training set into a training portion and validation portion;
  - training on the training portion with the chosen value of λ;
  - evaluating on the validation portion.
- Use the estimates to choose λ.
- Train with this λ on the entire training set.
- Evaluate on the test dataset

31.1.4 Training with Stochastic Gradient Descent

We have a function $g(\theta)$, and we wish to obtain a value of θ that achieves the minimum for that function. Solving in closed form doesn't work (try it!). Typical numerical methods take a point $\theta^{(n)}$, update it to $\theta^{(n+1)}$, then check to see whether the result is a minimum. This process is started from a start point. The choice of start point may or may not matter for general problems, but for our problem a random start point is fine. The update is usually obtained by computing a direction $\mathbf{p}^{(n)}$ such that for small values of η, $g(\theta^{(n)} + \eta\mathbf{p}^{(n)})$ is smaller than $g(\theta^{(n)})$. Such a direction is known as a *descent direction*. We must then determine how far to go along the descent direction, a process known as *line search*.

Obtaining a descent direction: One method to choose a descent direction is *gradient descent*, which uses the negative gradient of the function. We can write a Taylor series expansion for the function $g(\theta^{(n)} + \eta\mathbf{p}^{(n)})$. We have that

$$g(\theta^{(n)} + \eta\mathbf{p}^{(n)}) = g(\theta^{(n)}) + \eta\left[(\nabla g)^T\mathbf{p}^{(n)}\right] + O(\eta^2)$$

This means that we can expect that if

$$\mathbf{p}^{(n)} = -\nabla g(\theta^{(n)}),$$

for small values of η, $g(\mathbf{u}^{(n)} + \eta\mathbf{p}^{(n)})$ will be less than $g(\mathbf{u}^{(n)})$. This works (as long as g is differentiable, and quite often when it isn't) because g must go down for at least small steps in this direction.

But recall that our cost function is a sum of one error cost per example, together with the regularizer. This means the cost function looks like

$$g(\theta) = \left[(1/N)\sum_{i=1}^{N} g_i(\theta)\right] + g_0(\theta),$$

as a function of θ. Gradient descent would require us to form

$$-\nabla g(\theta) = -\left(\left[(1/N)\sum_{i=1}^{N} \nabla g_i(\theta)\right] + \nabla g_0(\theta)\right)$$

and then take a small step in this direction. But if N is large, this is unattractive, as we might have to sum a lot of terms. This happens a lot in building classifiers, where you might quite reasonably expect to deal with millions (billions; perhaps trillions) of examples. Touching each example at each step really is impractical.

Stochastic gradient descent is an algorithm that replaces the exact gradient with an approximation that has a random error, but is simple and quick to compute. The term

$$(\frac{1}{N})\sum_{i=1}^{N} \nabla g_i(\theta).$$

is a population mean, and we know (or should know!) how to deal with those. We can estimate this term by drawing a random sample (a *batch*) of N_b (the *batch size*) examples, with replacement, from the population of N examples, then computing the mean for that sample. We approximate the population mean by

$$(\frac{1}{N_b})\sum_{j \in \text{batch}} \nabla g_j(\theta).$$

The batch size is usually determined using considerations of computer architecture (how many examples fit neatly into cache?) or of database design (how many examples are recovered in one disk cycle?). One common choice is $N_b = 1$, which is the same as choosing one example uniformly and at random. We form

$$\mathbf{p}_{N_b}^{(n)} = -\left(\left[(1/N_b)\sum_{j \in \text{batch}} \nabla g_i(\theta^{(n)})\right] + \nabla g_0(\theta^{(n)})\right)$$

and then take a small step along $\mathbf{p}_{N_b}^{(n)}$. Our new point becomes

$$\theta^{(n+1)} = \theta^{(n)} + \eta \mathbf{p}_{N_b}^{(n)},$$

where η is called the *steplength* (or sometimes *step size* or *learning rate*, even though it isn't the size or the length of the step we take, or a rate!).

Because the expected value of the sample mean is the population mean, if we take many small steps along $\mathbf{p}_{N_b}$, they should average out to a step backwards along the gradient. This approach is known as stochastic gradient descent because

we're not going along the gradient, but along a random vector which is the gradient only in expectation. It isn't obvious that stochastic gradient descent is a good idea. Although each step is easy to take, we may need to take more steps. The question is then whether we gain in the increased speed of the step what we lose by having to take more steps. Current theory suggests we do gain, and there are other benefits. For example, stochastic gradient descent appears to regularize very large models in quite unexpected ways. In practice, the approach is hugely successful for training classifiers.

Choosing a steplength: Because evaluating g is hard, we can't search for the η that gives the best value of g. Instead, we use an η that is large at the start — so that the method can explore large changes in the values of the classifier parameters — and small steps later — so that it settles down. The choice of how η gets smaller is often known as a *steplength schedule* or *learning rate schedule*. Often, you can tell how many steps are required to have seen the whole dataset; this is called an *epoch*. A common steplength schedule sets the steplength in the e'th epoch to be

$$\eta^{(e)} = \frac{m}{e+n},$$

where m and n are constants chosen by experiment with small subsets of the dataset.

There is no good test for whether stochastic gradient descent has converged to the right answer, because natural tests involve evaluating the gradient and the function, and doing so is expensive. More usual is to plot the error as a function of iteration on the validation set, and interrupt or stop training when the error has reached an acceptable level. The error (resp. accuracy) should vary randomly (because the steps are taken in directions that only approximate the gradient) but should decrease (resp. increase) overall as training proceeds (because the steps do approximate the gradient). sometimes known as *learning curves*.

> **Remember this:** *Stochastic gradient descent is the dominant training paradigm for classifiers. Stochastic gradient descent uses a sample of the training data to estimate the gradient. Doing so results in a fast but noisy gradient estimate. This is particularly valuable when training sets are very large (as they should be). Steplengths are chosen according to a steplength schedule, and there is no test for convergence other than evaluating classifier behavior on validation data.*

31.1.5 Example: Classifying MNIST with Logistic Regression

I will use the MNIST dataset of handwritten digits as a source of examples in this and future chapters. This dataset is very widely used to check simple methods. It was originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. You can find this dataset in several places. The original dataset is at `http://yann.lecun.com/exdb/mnist/`. Each data item is a 28×28 grey level image of a handwritten digit. Each comes with a label from 0 to 9. The images

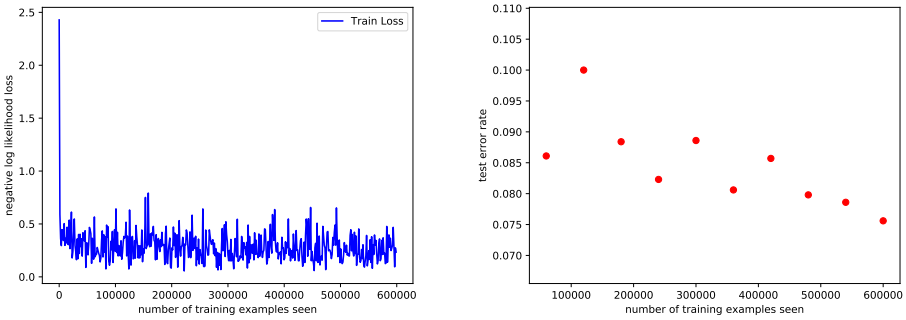


FIGURE 31.1: *On the* **left**, *the learning curve for a logistic regression classifier trained on MNIST data. Note the loss falls off quickly, then declines very slowly. The loss plotted here is the loss for a particular batch after a step has been taken using the gradient on that batch. Although the step follows the gradient, it may cause the loss to rise because it goes too far along the gradient direction — there is no search for a step length that guarantees descent and there are no second order terms here. Nonetheless, because the steps are small and approximately in the right direction, the loss declines. On the* **right**, *the error rate for the test set plotted at the end of each epoch. Notice how this declines, but not monotonically.*

were originally binary (ink and no-ink pixels) but are usually now seen in grey-level form, but pixels are either very dark or very light. For most images, the digit is pretty clear to a human observer, but some images contain quite mysterious digits.

MNIST can be classified with logistic regression. I straightened each image into a feature vector, then proceeded. I used a simple PyTorch program (Section 33.2 for PyTorch), with stochastic gradient descent as the optimizer. I used a learning rate scheduler that multiplied the learning rate by 0.9 after each epoch. I used the standard test-train split (60, 000 test and 10 000 train), and computed the test error rate each epoch. Figure 33.2 shows the *learning curve* (loss plotted against number of training images) and the test error rate. The error rate may seem small, but good MNIST classifiers obtain very much smaller error rates.

31.2 SIMPLE NEURAL NETWORKS

The one problem with logistic regression is that its performance depends on the features (the $\mathbf{x}$). The key trick in neural networks is to learn a transformation of these features to produce new, hopefully better, features. We do this by building simple feature transformation layers, then stacking them.

31.2.1 Layers and Units

We will compose *layers* – functions that accept vector inputs (and often parameters) and produce vector outputs – to form a classifier. Write the r'th function $\mathbf{o}^{(r)}$. functions don't). The r'th layer receives parameters $\theta^{(r)}$ (which will be empty if the layer doesn't need parameters). In this notation, the output of a network

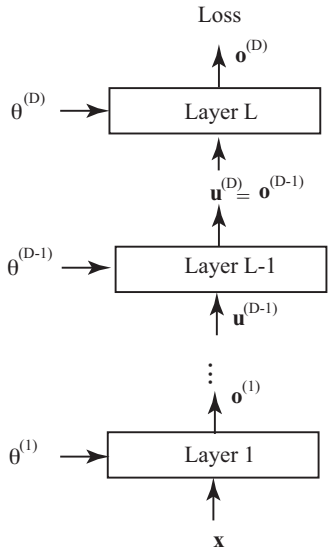


FIGURE 31.2: *Notation for layers, inputs, and parameters, for reference.*

applied to $\mathbf{x}$ could be written as

$$\mathbf{o}^{(D)}(\mathbf{o}^{(D-1)}(\ldots(\mathbf{o}^{1}(\mathbf{x},\theta^{(1)}),\theta^{(2)}),\ldots),\theta^{(D)})$$

which is messy. More clean is to write

$$\begin{aligned}
&\mathbf{o}^{(D)} \\
&\text{where} \\
\mathbf{o}^{(D)} &= \mathbf{o}^{(D)}(\mathbf{u}^{(D)},\theta^{(D)}) \\
\mathbf{u}^{(D)} &= \mathbf{o}^{(D-1)}(\mathbf{u}^{(D-1)},\theta^{(D-1)}) \\
\ldots &= \ldots \\
\mathbf{u}^{(2)} &= \mathbf{o}^{(1)}(\mathbf{u}^{(1)},\theta^{1}) \\
\mathbf{u}^{(1)} &= \mathbf{x}.
\end{aligned}$$

These equations really are a map for a computation. You feed in $\mathbf{x}$; this gives $\mathbf{u}^{(1)}$; which gives $\mathbf{u}^{(2)}$; and so on, up to $\mathbf{o}^{(D)}$. This is important, because it allows us to write an expression for the gradient fairly cleanly. (Figure 31.2 captures some of this).

There are a number of important standard layers.

- A *softmax layer* forms $\mathbf{o} = \mathbf{s}(\mathbf{u})$ (and has no parameters); we have already seen this layer.
- A *fully connected layer* or *fc layer* forms $\mathbf{o} = \mathcal{M}\mathbf{u} + \mathbf{b}$ (and its parameters are $\mathcal{M}$, $\mathbf{b}$). The logistic regression of Section 33.2 is a fully connected layer followed by a softmax layer.

- A *ReLU layer* applies $F(u) = \mathsf{max}(0, u)$ (a *ReLU*) to each element of the input vector.
- A *tanh layer* applies $F(u) = \tanh(u)$ to each element of the input vector.
- A *sigmoid layer* applies $F(u) = 1/(1 + e^{-u})$ to each element of the input vector.

We will see important variants of fully connected layers obtained by constraining the form of $\mathcal{M}$. Composing layers does not always yield anything interesting. For example, having two fully connected layers rather than one would still yield a logistic regression, because composing two affine functions yields another affine function. This new logistic regression classifier would have a weird parametrization, but it would still be a logistic regression. But consider a fc layer, followed by a ReLU layer, followed by another fc layer, followed by a softmax. We have something genuinely new. The fc-softmax pair at the output is a familiar logistic regression. But the fc-ReLU pair at the input maps the original feature representation into a new feature representation. If we can train this set of layers, we expect better accuracy than the original logistic regression, because we are adjusting the original features to obtain a new set that works better with a logistic regression. And we could stack more layers between the input and the logistic regression, in the hope of getting even better features.

An fc-layer followed by a non-linearity can be seen as a layer of *unit*s (another term is *perceptron*s). A unit takes a vector $\mathbf{x}$ of inputs and uses a vector $\mathbf{w}$ of parameters (known as the *weights*), a scalar b (known as the *bias*), and a nonlinear function F to form its output, which is

$$F(\mathbf{w}^T\mathbf{x} + b).$$

Units are sometimes referred to as *neurons*, and there is a large and rather misty body of vague speculative analogy linking devices built out of units to neuroscience. There is no reason to engage with this analogy here (or perhaps anywhere).

31.2.2 Training a Multi-layer Classifier

We now form a classifier out of a set of layers. There will be multiple layers, followed by an fc layer and then a softmax. The fc layer needs to produce a C dimensional vector, because there are C classes, but we'll be vague about the other layers for the moment. Our loss will have two terms: the cross-entropy term, and a regularization term. We will train this classifier with stochastic gradient descent on the loss. The key question is determining the gradient with respect to parameters.

The cross-entropy term is an average of per-item losses, and so takes the form

$$\frac{1}{N}\sum_i L(\mathbf{y}_i, \mathbf{o}^{(D)}(\mathbf{x}_i, \theta)).$$

The regularization term (which we won't always use) is a term that depends on the parameters. The gradient of the regularization term is easy.

Now imagine we have chosen a minibatch of M examples. We must compute the gradient of the cost function. The penalty term is easily dealt with, but the

per-item loss term is something of an exercise in the chain rule. We drop the index for the example, and so have to handle the gradient of

$$\begin{aligned} & L(\mathbf{y}, \mathbf{o}^{(D)}) \\ & \quad \text{where} \\ \mathbf{o}^{(D)} &= \mathbf{o}^{(D)}(\mathbf{u}^{(D)}, \theta^{(D)}) \\ \mathbf{u}^{(D)} &= \mathbf{o}^{(D-1)}(\mathbf{u}^{(D-1)}, \theta^{(L-1)}) \\ \ldots &= \ldots \\ \mathbf{u}^{(2)} &= \mathbf{o}^{(1)}(\mathbf{u}^{(1)}, \theta^{1}) \\ \mathbf{u}^{(1)} &= \mathbf{x}. \end{aligned}$$

Again, think of these equations as a map for a computation.

You should check (using whatever form of the chain rule you recall) that

$$\nabla_{\theta^{(2)}} L = (\nabla_{\mathbf{s}} L) \times J_{\mathbf{s};\mathbf{o}} \times J_{\mathbf{o};\theta^{(2)}}.$$

Notation: 31.4 $\mathcal{J}_{\mathbf{f};\mathbf{x}}$*: Jacobian*

The *Jacobian* $\mathcal{J}_{\mathbf{f};\mathbf{x}}$ is a matrix of partial derivatives

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \ldots & \frac{\partial f_1}{\partial x_{\#(\mathbf{x})}} \\ \ldots & \ldots & \ldots \\ \frac{\partial f_{\#(\mathbf{f})}}{\partial x_1} & \ldots & \frac{\partial f_{\#(\mathbf{s})}}{\partial f_{\#(\mathbf{o})}} \end{pmatrix}$$

(in some circles, this is called the derivative of $\mathbf{f}$, but this convention can become confusing). The Jacobian simplifies writing out the chain rule. Notice that the subscript marks which derivatives are being computed.

Now consider $\nabla_{\theta} L$ and use the chain rule. We have

$$\nabla_{\theta^{(D)}} L(\mathbf{y}, \mathbf{o}^{(D)}) = (\nabla_{\mathbf{o}^{(D)}} L) \times J_{\mathbf{o}^{(D)};\theta^{(D)}}$$

Now think about $\nabla_{\theta^{(D-1)}} L$. The loss depends on $\theta^{(D-1)}$ in a somewhat roundabout way; layer $D-1$ uses $\theta^{(D-1)}$ to produce its outputs, and these are fed into layer D *as that layer's inputs.* So we must have

$$\nabla_{\theta^{(D-1)}} L(\mathbf{y}_i, \mathbf{o}^{(D)}(\mathbf{x}_i, \theta)) = (\nabla_{\mathbf{o}^{(D)}} L) \times J_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \times J_{\mathbf{o}^{(D-1)};\theta^{(D-1)}}$$

(look carefully at the subscripts on the Jacobians, and remember that $\mathbf{u}^{(D)} = \mathbf{o}^{(D-1)}$). And $\mathbf{o}^{(D)}$ depends on $\theta^{(D-2)}$ through $\mathbf{u}^{(D)}$ which is a function of $\mathbf{u}^{(D-1)}$ which is a function of $\theta^{(D-2)}$, so that

$$\nabla_{\theta^{(D-2)}} L(\mathbf{y}_i, \mathbf{o}^{(D)}(\mathbf{x}_i, \theta)) = (\nabla_{\mathbf{o}^{(D)}} L) \times J_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \times J_{\mathbf{o}^{(D-1)};\mathbf{u}^{(D-1)}} \times J_{\mathbf{o}^{(D-2)};\theta^{(D-2)}}$$

(again, look carefully at the subscripts on each of the Jacobians, and remember that $\mathbf{u}^{(D)} = \mathbf{o}^{(D-1)}$ *and* $\mathbf{u}^{(D-1)} = \mathbf{o}^{(D-2)}$).

We can now get to the point. We have a recursion:

$$\begin{aligned}
\mathbf{v}^{(D)} &= (\nabla_{\mathbf{o}^{(D)}} L) \\
\nabla_{\theta^{(D)}} L &= \mathbf{v}^{(D)} \mathcal{J}_{\mathbf{o}^{(D)};\theta^{(D)}} \\
\nabla_{\theta^{(D-1)}} L &= \mathbf{v}^{(D)} \mathcal{J}_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \mathcal{J}_{\mathbf{o}^{(D-1)};\theta^{(D-1)}} \\
&\dots \\
\nabla_{\theta^{(i-1)}} L &= \mathbf{v}^{(D)} \mathcal{J}_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \dots \mathcal{J}_{\mathbf{o}^{(i)};\mathbf{u}^{(i)}} \mathcal{J}_{\mathbf{o}^{(i-1)};\theta^{(i-1)}} \\
&\dots
\end{aligned}$$

But look at the form of the products of the matrices. We don't need to remultiply all those matrices; instead, we are attaching a new term to a product we've already computed. All this is more cleanly written as:

$$\begin{aligned}
\mathbf{v}^{(D)} &= \left(\nabla^{(D)}_{\mathbf{o}} L\right) \\
\nabla_{\theta^{(D)}} L &= \mathbf{v}^{(D)} \mathcal{J}_{\mathbf{o}^{(D)};\theta^{(D)}} \\
\mathbf{v}^{(D-1)} &= \mathbf{v}^{(D)} \mathcal{J}_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \\
\nabla_{\theta^{(D-1)}} L &= \mathbf{v}^{(D-1)} \mathcal{J}_{\mathbf{o}^{(D-1)};\theta^{(D-1)}} \\
&\dots \\
\mathbf{v}^{(i-1)} &= \mathbf{v}^{(i)} \mathcal{J}_{\mathbf{o}^{(i)};\mathbf{u}^{(i)}} \\
\nabla_{\theta^{(i-1)}} L &= \mathbf{v}^{(i-1)} \mathcal{J}_{\mathbf{o}^{(i-1)};\theta^{(i-1)}} \\
&\dots
\end{aligned}$$

I have not added notation to keep track of the point at which the partial derivative is evaluated (it should be obvious, and we have quite enough notation already). When you look at this recursion, you should see that, to evaluate $\mathbf{v}^{(i-1)}$, you will need to know $\mathbf{u}^{(k)}$ for $k \geq i-1$. This suggests the following strategy. We compute the $\mathbf{u}$'s (and, equivalently, $\mathbf{o}$'s) with a "forward pass", moving from the input layer to the output layer. Then, in a "backward pass" from the output to the input, we compute the gradient. Doing this is often referred to as *backpropagation*.

Remember this: *The gradient of a multilayer network follows from the chain rule. A straightforward recursion known as backpropagation yields an efficient algorithm for evaluating the gradient. Information flows up the network to compute outputs, then back down to get gradients.*

31.2.3 Dropout and Redundant Units

A very useful regularization strategy is to try and ensure that no unit relies too much on the output of any other unit. One can do this as follows. At each training step, randomly select some units, set their outputs to zero (and reweight the inputs

of the units receiving input from them), and then take the step. Now units are trained to produce reasonable outputs even if some of their inputs are randomly set to zero — units can't rely too much on one input, because it might be turned off. Notice that this sounds sensible, but it isn't quite a proof that the approach is sound; that comes from experiment. The approach is known as *dropout*.

At test time, there is no dropout. Every unit computes its usual output in the usual way. This creates an important training issue. Write p for the probability that a unit is dropped out, which will be the same for all units subject to dropout. You should think of the expected output of the i'th unit at *training* time as $(1-p)o_i$ (because with probability p, it is zero). But at test time, the next unit will see o_i; so at training time, you should reweight the inputs by $1/(1-p)$. In exercises, we will use packages that arrange all the details for us.

Remember this: *Dropout can force units to look at inputs from all of a set of redundant units, and so regularize a network.*

31.2.4 Housekeeping

There are now several software environments that can accept a description of a network as a map of a computation, like the one above, and automatically construct a code that implements that network. In essence, the user writes a map, provides inputs, and decides what to do with gradients to get descent. These environments support the necessary housekeeping to map a network onto a GPU, evaluate the network and its gradients on the GPU, train the network by updating parameters, and so on. The easy availability of these environments has been an important factor in the widespread adoption of neural networks.

At time of writing, the main environments available are

- **Darknet:** This is an open source environment developed by Joe Redmon. You can find it at `https://pjreddie.com/darknet/`. There is some tutorial material there.
- **Matconvnet:** This is an environment for MATLAB users, originally written by Andrea Vedaldi and supported by a community of developers. You can find it at `http://www.vlfeat.org/matconvnet`. There is a tutorial at that URL.
- **MXNet:** This is a software framework from Apache that is supported on a number of public cloud providers, including Amazon Web Services and Microsoft Azure. It can be invoked from a number of environments, including R and MATLAB). You can find it at `https://mxnet.apache.org`.
- **PaddlePaddle:** This is an environment developed at Baidu research. You can find it at `http://www.paddlepaddle.org`. There is tutorial material on that page; I understand there is a lot of tutorial material in Chinese, but I

can't read Chinese and so can't find it or offer URL's. You should search the web for more details.

- **PyTorch:** This is an environment developed at Facebook's AI research. You can find it at `https://pytorch.org`. There video tutorials at `https://pytorch.org/tutorials/`.

- **Tensorflow:** This is an environment developed at Google. You can find it at `https://www.tensorflow.org`. There is extensive tutorial material at `https://www.tensorflow.org/tutorials/`.

- **Keras:** This is an environment developed by François Chollet, intended to offer high-level abstractions independent of what underlying computational framework is used. It is supported by the TensorFlow core library. You can find it at `https://keras.io`. There is tutorial material at that URL.

Each of these environments has their own community of developers. It is now common in the research community to publish code, networks and datasets openly. This means that, for much cutting edge research, you can easily find a code base that implements a network; and all the parameter values that the developers used to train a network; and a trained version of the network; and the dataset they used for training and evaluation. But these aren't the only environments. You can find a useful comparison at `https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software` that describes many other environments.

Remember this: *Training even a simple network involves a fair amount of housekeeping code. There are a number of software environments that simplify setting up and training complicated neural networks.*

31.2.5 Example: Multi-layer MNIST

Using multiple layers significantly improves MNIST classification accuracy. I straightened each image into a feature vector, then proceeded. The network I used is shown in Figure **??**. I used a simple PyTorch program (Section 33.2 for PyTorch), with stochastic gradient descent as the optimizer. I used a learning rate scheduler that multiplied the learning rate by 0.9 after each epoch. I used the standard test-train split (60, 000 test and 10 000 train), and computed the test error rate each epoch. Figure 33.2 shows the learning curve and the test error rate. The error rate may seem small, but good MNIST classifiers obtain very much smaller error rates.

31.2.6 Augmentation and Ensembles

Three important practical issues that need to be addressed to build very strong image classifiers.

- **Data sparsity:** Datasets of images are never big enough to show all effects accurately. This is because an image of a horse is still an image of a horse even if it has been through a small rotation, or has been resized to be a bit bigger or smaller, or has been cropped differently, and so on. There is no way to take account of these effects in the architecture of the network.
- **Data compliance:** We want each image fed into the network to be the same size.
- **Network variance:** The network we have is never the best network; training started at a random set of parameters, and has a strong component of randomness in it. For example, most minibatch selection algorithms select random minibatches. Training the same architecture on the same dataset twice will not yield the same network.

All three can be addressed by some care with training and test data.

Generally, the way to address data sparsity is *data augmentation*, by expanding the training dataset to include different rotations, scalings, and crops of images. Doing so is relatively straightforward. You take each training image, and generate a collection of extra training images from it. You can obtain this collection by: resizing and then cropping the training image; using different crops of the same training image (assuming that training images are a little bigger than the size of image you will work with); rotating the training image by a small amount, resizing and cropping; and so on.

There are some cautions. When you rotate then crop, you need to be sure that no "unknown" pixels find their way into the final crop. You can't crop too much, because you need to ensure that the modified images are still of the relevant class, and too aggressive a crop might cut out the horse (or whatever) entirely. This somewhat depends on the dataset. If each image consists of a tiny object on a large background, and the objects are widely scattered, crops need to be cautious; but if the object covers a large fraction of the image, the cropping can be quite aggressive.

Cropping is usually the right way to ensure that each image has the same size. Resizing images might cause some to stretch or squash, if they have the wrong aspect ratio. This likely isn't a great idea, because it will cause objects to stretch or squash, making them harder to recognize. It is usual to resize images to a convenient size without changing the aspect ratio, then crop to a fixed size.

There are two ways to think about network variance (at least!). If the network you train isn't the best network (because it can't be), then it's very likely that training multiple networks and combining the results in some way will improve classification. You could combine results by, for example, voting. Small improvements can be obtained reliably like this, but the strategy is often deprecated because it isn't particularly elegant or efficient. A more usual approach is to realize that the network might very well handle one crop of a test image rather better than others (because it isn't the best network, etc.). Small improvements in performance can be obtained very reliably by presenting multiple crops of a test image to a given network, and combining the results for those crops.

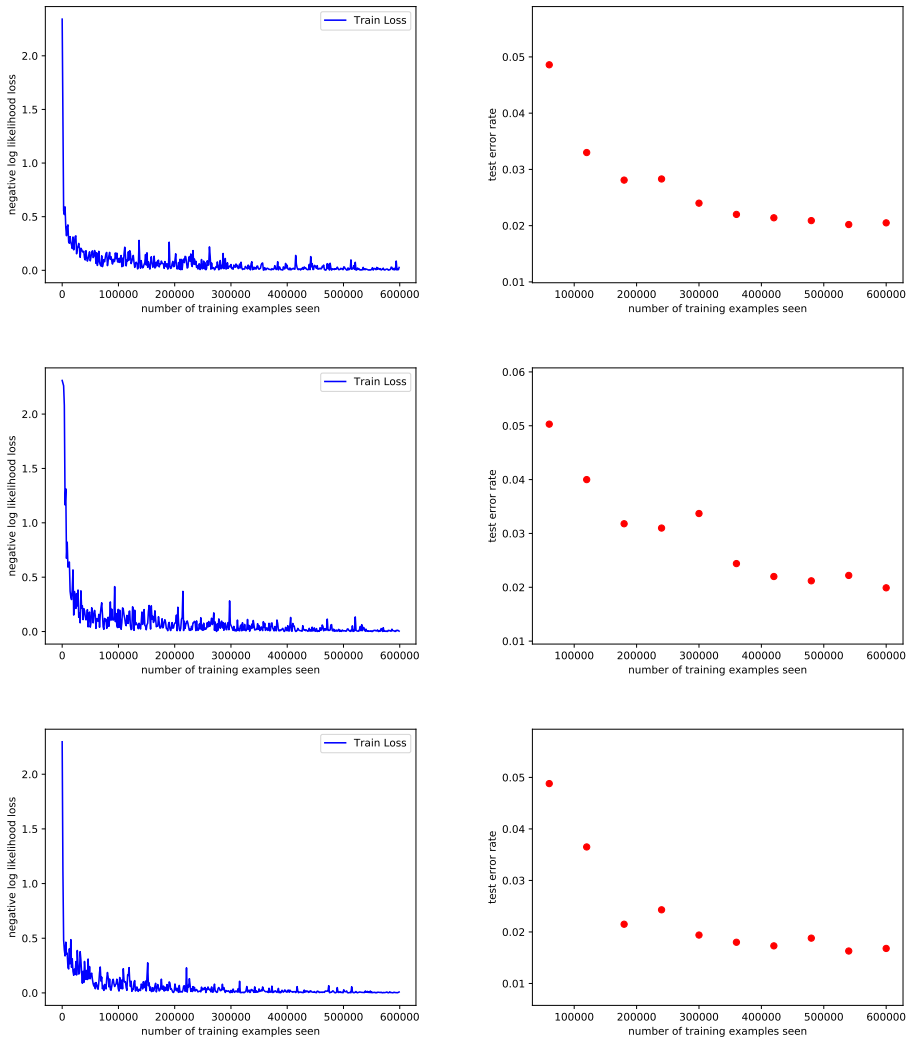


FIGURE 31.3: *On the* **left**, *the learning curve for a logistic regression classifier trained on MNIST data. Note the loss falls off quickly, then declines very slowly. The loss plotted here is the loss for a particular batch after a step has been taken using the gradient on that batch. Although the step follows the gradient, it may cause the loss to rise because it goes too far along the gradient direction — there is no search for a step length that guarantees descent and there are no second order terms here. Nonetheless, because the steps are small and approximately in the right direction, the loss declines. On the* **right**, *the error rate for the test set plotted at the end of each epoch. Notice how this declines, but not monotonically.*

CHAPTER 32

Some useful Matrix and Transformation Facts

32.1 NOMENCLATURE

32.1.1 Types of Matrix

32.1.2 Types of Transformation

TODO: rotations commute with scales

32.2 LEAST SQUARES

32.2.1 General Linear Systems

32.2.2 Homogeneous Equations

32.3 TRICKS

The *trace* of matrix $\mathcal{M}$ is the sum of values along the diagonal. It is applicable only to square matrices. In the exercises, you will show that the trace is linear, that

$$\mathsf{Tr}\left(\mathcal{AB}\right) = \mathsf{Tr}\left(\mathcal{BA}\right)$$

and that

$$\mathsf{Tr}\left(\mathcal{ABC}\right) = \mathsf{Tr}\left(\mathcal{CAB}\right) = \mathsf{Tr}\left(\mathcal{BCA}\right),$$

all facts sufficiently well worth remembering to deserve being put in a box.

Remember this: *The trace of $\mathcal{M}$ is $\sum_i m_{ii}$. We have that the trace is linear, that*

$$\mathsf{Tr}\left(\mathcal{AB}\right) = \mathsf{Tr}\left(\mathcal{BA}\right)$$

and that

$$\mathsf{Tr}\left(\mathcal{ABC}\right) = \mathsf{Tr}\left(\mathcal{CAB}\right) = \mathsf{Tr}\left(\mathcal{BCA}\right).$$

The *Frobenius norm* is the matrix norm obtained by summing squared entries of the matrix. We write

$$\|\mathcal{A}\|_F = \sum_{i,j} a_{ij}^2.$$

In the exercises, you will show that

$$\|\mathcal{A}\|_F = \mathsf{Tr}\left(\mathcal{A}^T\mathcal{A}\right)$$

Remember this: *The Frobenius norm of $\mathcal{M}$ is $\sum_{ij} m_{ij}^2$. It can be computed as $\mathsf{Tr}\left(\mathcal{M}^T\mathcal{M}\right)$.*

32.3.1 RQ Factorization

Any real square matrix $\mathcal{M}$ can be factored into $\mathcal{R}\mathcal{Q}$, where $\mathcal{R}$ is upper triangular and $\mathcal{Q}$ is orthonormal. We do the 2×2 case, then write out in general form. Write $\mathcal{M} = \left[\mathbf{m}_1^T; \mathbf{m}_2^T\right]$ (etc). Then

$$\mathcal{M} = \left[m_{11}\mathbf{q}_1^T + m_{12}\mathbf{q}_2^T; m_{22}\mathbf{q}_2^T\right].$$

The upper triangular form of $\mathcal{R}$ means we can solve for terms in a convenient order, so

- $m_{22} = \sqrt{\mathbf{m}_2^T\mathbf{m}_2}$ and $\mathbf{q}_2 = (1/m_{22})\mathbf{m}_2$ (because the second row of $\mathcal{Q}$ must be a unit vector);
- and $m_{12} = \mathbf{m}_1^T\mathbf{q}_2$ (because $\mathbf{q}_1$ and $\mathbf{q}_2$ are orthonormal);
- and so $m_{11} = \sqrt{\mathbf{l}_1^T\mathbf{l}_1}$ and $\mathbf{q}_1 = (1/m_{11})\mathbf{l}_1$ where $\mathbf{l}_1 = \mathbf{m}_1 - m_{12}\mathbf{q}_2$

This logic works for larger matrices, where the main nuisance is notation.

Procedure: 32.1 *RQ Factorization*

Now assume $\mathcal{M}$ is $d \times d$. Then

$$\mathcal{M} = \left[\sum_{i=1}^{d} m_{1i}\mathbf{q}_i^T; \sum_{i=2}^{d} m_{2i}\mathbf{q}_i^T; \ldots; m_{dd}\mathbf{q}_d^T\right]$$

We have that

- $m_{dd} = \sqrt{\mathbf{m}_d^T\mathbf{m}_d}$ and $\mathbf{q}_d = (1/m_{dd})\mathbf{m}_d$;
- and for u ranging from $d-1$ to 1:
  - $m_{ud} = \mathbf{m}_u^T\mathbf{q}_d$;
  - $m_{uu} = \sqrt{\mathbf{l}_u^T\mathbf{l}_u}$;
  - and $\mathbf{q}_u = (1/m_{uu})\mathbf{l}_u$;
  - where $\mathbf{l}_u = \mathbf{m}_u - \sum_{v=u+1} v = dm_{uv}\mathbf{q}_v$.

RQ factorization is a variant of the more commonly used QR factorization (orthonormal times upper triangular) which is useful for camera calibration.

32.3.2 Cholesky Factorization

A symmetric, positive definite matrix $\mathcal{M}$ can be factored as $\mathcal{U}^T\mathcal{U}$, where $\mathcal{U}$ is upper triangular. An example (below) will show why this works, but it is most unwise to write your own Cholesky factorization because the best behavior comes from pivoting, etc. to preserve numerical precision. Notice that if you want to factor the matrix into lower triangular factors (so $\mathcal{M} = \mathcal{L}^T\mathcal{L}$), a version of the same procedure will work.

I will plod through Cholesky factorization of a 3×3 matrix to illustrate the principle. Assume the factorization exists. Then we have

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{12} & m_{22} & m_{23} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & 0 \\ u_{12} & u_{22} & 0 \\ u_{13} & u_{23} & u_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

If you multiply out, you will notice $u_{11} = \sqrt{m_{11}}$. For concreteness, choose the positive square root (this is usual – there are actually 8 factorizations of this matrix, but they differ only by choice of sign of various square roots). Now $u_{12} = m_{11}/u_{11}$ and $u_{13} = m_{13}/u_{11}$. Similarly, $u_{22} = \sqrt{m_{22} - u_{12}^2}$ (and again, choose the positive square root) and $u_{23} = (m_{23} - u_{13})/u_{22}$. Finally, $u_{33} = \sqrt{m_{33} - u_{13}^2 - u_{23}^2}$. Notice the trick – if we compute the elements of $\mathcal{U}$ in the right order, there is only ever one unknown. This procedure is clearly going to fail if we attempt to take a square root of a negative number. A little manipulation will establish that this occurs only if $\mathcal{M}$ is not positive definite.

CHAPTER 33

Tools for High Dimensional Data

TODO: Curse of dimension **TODO:** PCA **TODO:** Simple multidimensional scaling **TODO:** TSNE

33.1 PRINCIPAL COMPONENTS ANALYSIS

33.1.1 Mean and Covariance

For one-dimensional data, we wrote

$$\mathsf{mean}\left(\{x\}\right) = \frac{\sum_i x_i}{N}.$$

This expression is meaningful for vectors, too, because we can add vectors and divide by scalars. We write

$$\mathsf{mean}\left(\{\mathbf{x}\}\right) = \frac{\sum_i \mathbf{x}_i}{N}$$

and call this the mean of the data. Notice that each component of $\mathsf{mean}\left(\{\mathbf{x}\}\right)$ is the mean of that component of the data. There is not an easy analogue of the median, however (how do you order high dimensional data?) and this is a nuisance. Notice that, just as for the one-dimensional mean, we have

$$\mathsf{mean}\left(\{\mathbf{x} - \mathsf{mean}\left(\{\mathbf{x}\}\right)\}\right) = 0$$

(i.e. if you subtract the mean from a data set, the resulting data set has zero mean).

Variance, standard deviation and correlation can each be seen as an instance of a more general operation on data. Extract two components from each vector of a dataset of vectors, yielding two 1D datasets of N items; write $\{x\}$ for one and $\{y\}$ for the other. The i'th element of $\{x\}$ corresponds to the i'th element of $\{y\}$ (the i'th element of $\{x\}$ is one component of some bigger vector $\mathbf{x}_i$ and the i'th element of $\{y\}$ is another component of this vector). We can define the covariance of $\{x\}$ and $\{y\}$.

Definition: 33.1 *Covariance*

Assume we have two sets of N data items, $\{x\}$ and $\{y\}$. We compute the covariance by

$$\mathsf{cov}\left(\{x\}, \{y\}\right) = \frac{\sum_i (x_i - \mathsf{mean}\left(\{x\}\right))(y_i - \mathsf{mean}\left(\{y\}\right))}{N}$$

Covariance measures the tendency of corresponding elements of $\{x\}$ and of $\{y\}$ to be larger than (resp. smaller than) the mean. The correspondence is defined by the order of elements in the data set, so that x_1 corresponds to y_1, x_2 corresponds to y_2, and so on. If $\{x\}$ tends to be larger (resp. smaller) than its mean for data points where $\{y\}$ is also larger (resp. smaller) than its mean, then the covariance should be positive. If $\{x\}$ tends to be larger (resp. smaller) than its mean for data points where $\{y\}$ is smaller (resp. larger) than its mean, then the covariance should be negative.

Notice that

$$\mathsf{std}\left(x\right)^2 = \mathsf{var}\left(\{x\}\right) = \mathsf{cov}\left(\{x\}, \{x\}\right)$$

which you can prove by substituting the expressions. Recall that variance measures the tendency of a dataset to be different from the mean, so the covariance of a dataset with itself is a measure of its tendency not to be constant. More important is the relationship between covariance and correlation, in the box below.

Remember this:

$$\mathsf{corr}\left(\{(x, y)\}\right) = \frac{\mathsf{cov}\left(\{x\}, \{y\}\right)}{\sqrt{\mathsf{cov}\left(\{x\}, \{x\}\right)}\sqrt{\mathsf{cov}\left(\{y\}, \{y\}\right)}}.$$

This is occasionally a useful way to think about correlation. It says that the correlation measures the tendency of $\{x\}$ and $\{y\}$ to be larger (resp. smaller) than their means for the same data points, *compared to* how much they change on their own.

33.1.2 The Covariance Matrix

Working with covariance (rather than correlation) allows us to unify some ideas. In particular, for data items which are d dimensional vectors, it is straightforward to compute a single matrix that captures all covariances between all pairs of components — this is the covariance matrix.

Definition: 33.2 *Covariance Matrix*

The covariance matrix is:

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \frac{\sum_i (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))^T}{N}$$

Notice that it is quite usual to write a covariance matrix as Σ, and we will follow this convention.

Covariance matrices are often written as Σ, whatever the dataset (you get to figure out precisely which dataset is intended, from context). Generally, when we

want to refer to the j, k'th entry of a matrix $\mathcal{A}$, we will write $\mathcal{A}_{jk}$, so Σ_{jk} is the covariance between the j'th and k'th components of the data.

Definition: 33.3 *Properties of the covariance matrix*

- The j, k'th entry of the covariance matrix is the covariance of the j'th and the k'th components of $\mathbf{x}$, which we write $\mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$.
- The j, j'th entry of the covariance matrix is the variance of the j'th component of $\mathbf{x}$.
- The covariance matrix is symmetric.
- The covariance matrix is always positive semi-definite; it is positive definite, *unless* there is some vector $\mathbf{a}$ such that $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}_i\}\right) = 0$ for all i.

Proposition:

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right)_{jk} = \mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$$

Proof: Recall

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \frac{\sum_i (\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right))^T}{N}$$

and the j, k'th entry in this matrix will be

$$\frac{\sum_i (x_i^{(j)} - \mathsf{mean}\left(\left\{x^{(j)}\right\}\right))(x_i^{(k)} - \mathsf{mean}\left(\left\{x^{(k)}\right\}\right))^T}{N}$$

which is $\mathsf{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right)$.

Proposition:

$$\text{Covmat}\left(\{\mathbf{x}_i\}\right)_{jj} = \Sigma_{jj} = \text{var}\left(\left\{x^{(j)}\right\}\right)$$

Proof:

$$\begin{aligned}\text{Covmat}\left(\{\mathbf{x}\}\right)_{jj} &= \text{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(j)}\right\}\right) \\ &= \text{var}\left(\left\{x^{(j)}\right\}\right)\end{aligned}$$

Proposition:

$$\text{Covmat}\left(\{\mathbf{x}\}\right) = \text{Covmat}\left(\{\mathbf{x}\}\right)^T$$

Proof: We have

$$\begin{aligned}\text{Covmat}\left(\{\mathbf{x}\}\right)_{jk} &= \text{cov}\left(\left\{x^{(j)}\right\}, \left\{x^{(k)}\right\}\right) \\ &= \text{cov}\left(\left\{x^{(k)}\right\}, \left\{x^{(j)}\right\}\right) \\ &= \text{Covmat}\left(\{\mathbf{x}\}\right)_{kj}\end{aligned}$$

Proposition: *Write* $\Sigma = \mathsf{Covmat}(\{\mathbf{x}\})$. *If there is no vector* $\mathbf{a}$ *such that* $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})) = 0$ *for all* i, *then for any vector* $\mathbf{u}$, *such that* $\|\mathbf{u}\| > 0$,

$$\mathbf{u}^T\Sigma\mathbf{u} > 0.$$

If there is such a vector $\mathbf{a}$, *then*

$$\mathbf{u}^T\Sigma\mathbf{u} \geq 0.$$

Proof: We have

$$\begin{aligned}\mathbf{u}^T\Sigma\mathbf{u} &= \frac{1}{N}\sum_i \left[\mathbf{u}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))\right]\left[(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))^T\mathbf{u}\right] \\ &= \frac{1}{N}\sum_i \left[\mathbf{u}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))\right]^2.\end{aligned}$$

Now this is a sum of squares. If there is some $\mathbf{a}$ such that $\mathbf{a}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})) = 0$ for every i, then the covariance matrix must be positive semidefinite (because the sum of squares could be zero in this case). Otherwise, it is positive definite, because the sum of squares will always be positive.

33.2 REPRESENTING DATA ON PRINCIPAL COMPONENTS

We have seen that a blob of data can be translated so that it has zero mean, then rotated so the covariance matrix is diagonal. In this coordinate system, we can set some components to zero, and get a representation of the data that is still accurate. The rotation and translation can be undone, yielding a dataset that is in the same coordinates as the original, but lower dimensional. The new dataset is a good approximation to the old dataset. All this yields a really powerful idea: we can choose a small set of vectors, so that each item in the original dataset can be represented as the mean vector plus a weighted sum of this set. This representation means we can think of the dataset as lying on a low dimensional space inside the original space. It's an experimental fact that this model of a dataset is usually accurate for real high-dimensional data, and it is often an extremely convenient model. Furthermore, representing a dataset like this very often suppresses noise – if the original measurements in your vectors are noisy, the low dimensional representation may be closer to the true data than the measurements are.

We start with a dataset of N d-dimensional vectors $\{\mathbf{x}\}$. We translate this dataset to have zero mean, forming a new dataset $\{\mathbf{m}\}$ where $\mathbf{m}_i = \mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})$. We diagonalize $\mathsf{Covmat}(\{\mathbf{m}\}) = \mathsf{Covmat}(\{\mathbf{x}\})$ to get

$$\mathcal{U}^T\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \Lambda$$

and form the dataset $\{\mathbf{r}\}$, using the rule

$$\mathbf{r}_i = \mathcal{U}^T \mathbf{m}_i = \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}\,(\{\mathbf{x}\})).$$

We saw the mean of this dataset is zero, and the covariance is diagonal. Most high dimensional datasets display another important property: many, or most, of the diagonal entries of the covariance matrix are very small. This means we can build a low dimensional representation of the high dimensional dataset that is quite accurate.

33.2.1 Approximating Blobs

The covariance matrix of $\{\mathbf{r}\}$ is diagonal, and the values on the diagonal are interesting. It is quite usual for high dimensional datasets to have a small number of large values on the diagonal, and a lot of small values. This means that the blob of data is really a low dimensional blob in a high dimensional space. For example, think about a line segment (a 1D blob) in 3D. As another example, look at Figure **??**; the scatterplot matrix strongly suggests that the blob of data is flattened (eg look at the petal width vs petal length plot).

Now assume that $\mathsf{Covmat}\,(\{\mathbf{r}\})$ has many small and few large diagonal entries. In this case, the blob of data represented by $\{\mathbf{r}\}$ admits an accurate low dimensional representation. The data set $\{\mathbf{r}\}$ is d-dimensional. We will try to represent it with an s dimensional dataset, and see what error we incur. Choose some $s < d$. Now take each data point $\mathbf{r}_i$ and replace the last $d - s$ components with 0. Call the resulting data item $\mathbf{p}_i$. We should like to know the average error in representing $\mathbf{r}_i$ with $\mathbf{p}_i$.

This error is

$$\frac{1}{N}\sum_i \left[(\mathbf{r}_i - \mathbf{p}_i)^T (\mathbf{r}_i - \mathbf{p}_i)\right].$$

Write $r_i^{(j)}$ for the j' component of $\mathbf{r}_i$, and so on. Remember that $\mathbf{p}_i$ is zero in the last $d - s$ components. The mean error is then

$$\frac{1}{N}\sum_i \left[\sum_{j=s+1}^{j=d} \left(r_i^{(j)}\right)^2\right].$$

But we know this number, because we know that $\{\mathbf{r}\}$ has zero mean. The error is

$$\sum_{j=s+1}^{j=d} \left[\frac{1}{N}\sum_i \left(r_i^{(j)}\right)^2\right] = \sum_{j=s+1}^{j=d} \mathsf{var}\left(\left\{r^{(j)}\right\}\right)$$

which is the sum of the diagonal elements of the covariance matrix from r, r to d, d. Equivalently, writing λ_i for the i'th eigenvalue of $\mathsf{Covmat}\,(\{x\})$ and assuming the eigenvalues are sorted in descending order, the error is

$$\sum_{j=s+1}^{j=d} \lambda_j$$

FIGURE 33.1: *On the* **left**, *the translated and rotated blob of figure* **??**. *This blob is stretched — one direction has more variance than another. Setting the y coordinate to zero for each of these datapoints results in a representation that has relatively low error, because there isn't much variance in these values. This results in the blob on the* **right**. *The text shows how the error that results from this projection is computed.*

FIGURE 33.2: *A panel plot of the bodyfat dataset of figure* **??**, *now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Each scatterplot is on the same set of axes, so you can see that the dataset extends more in some directions than in others. You should notice that, in some directions, there is very little variance. This suggests that replacing the coefficient in those directions with zero (as in figure 33.1) should result in a representation of the data that has very little error.*

If this sum is small compared to the sum of the first s components, then dropping the last $d - s$ components results in a small error. In that case, we could think about the data as being s dimensional. Figure 33.1 shows the result of using this approach to represent the blob I've used as a running example as a 1D dataset.

This is an observation of great practical importance. As a matter of experimental fact, a great deal of high dimensional data produces relatively low dimensional blobs. We can identify the main directions of variation in these blobs, and use them to understand and to represent the dataset.

33.2.2 Example: Transforming the Height-Weight Blob

Translating a blob of data doesn't change the scatterplot matrix in any interesting way (the axes change, but the picture doesn't). Rotating a blob produces really interesting results, however. Figure 33.2 shows the dataset of Figure **??**, translated to the origin and rotated to diagonalize it. Now we do not have names for each component of the data (they're linear combinations of the original components), but each pair is now not correlated. This blob has some interesting shape features. Figure 33.2 shows the gross shape of the blob best. Each panel of this figure has the same scale in each direction. You can see the blob extends about 80 units in direction 1, but only about 15 units in direction 2, and much less in the other two directions. You should think of this blob as being rather cigar-shaped; it's long in one direction, but there isn't much in the others. The cigar metaphor isn't perfect (have you seen a four-dimensional cigar recently?), but it's helpful. You can think of each panel of this figure as showing views down each of the four axes of the cigar.

Now look at figure 33.3. This shows the same rotation of the same blob of data, but now the scales on the axis have changed to get the best look at the detailed shape of the blob. First, you can see that blob is a little curved (look at the projection onto direction 2 and direction 4). There might be some effect here worth studying. Second, you can see that some points seem to lie away from the

FIGURE 33.3: *A panel plot of the bodyfat dataset of figure* **??**, *now rotated so that the covariance between all pairs of distinct dimensions is zero. Now we do not know names for the directions — they're linear combinations of the original variables. Compare this figure with figure 33.3; in that figure, the axes were the same, but in this figure I have scaled the axes so you can see details. Notice that the blob is a little curved, and there are several data points that seem to lie some way away from the blob, which I have numbered.*

FIGURE 33.4: *The data of Figure* **??**, *represented by translating and rotating so that the covariance is diagonal, projecting off the two smallest directions, then undoing the rotation and translation. This blob of data is two dimensional (because we projected off two dimensions – figure 33.2 suggested this was safe), but is represented in a four dimensional space. You can think of it as a thin two dimensional pancake of data in the four dimensional space (you should compare to Figure* **??** *on page* **??***). It is a good representation of the original data. Notice that it looks slightly thickened on edge, because it isn't aligned with the coordinate system – think of a view of a flat plate at a slight slant.*

main blob. I have plotted each data point with a dot, and the interesting points with a number. These points are clearly special in some way.

The problem with these figures is that the axes are meaningless. The components are weighted combinations of components of the original data, so they don't have any units, etc. This is annoying, and often inconvenient. But I obtained Figure 33.2 by translating, rotating and projecting data. It's straightforward to undo the rotation and the translation – this takes the projected blob (which we know to be a good approximation of the rotated and translated blob) back to where the original blob was. Rotation and translation don't change distances, so the result is a good approximation of the original blob, but now in the original blob's coordinates. Figure 33.4 shows what happens to the data of Figure **??**. This is a two dimensional version of the original dataset, embedded like a thin pancake of data in a four dimensional space. Crucially, it represents the original dataset quite accurately.

33.2.3 Representing Data on Principal Components

Now consider undoing the rotation and translation for our projected dataset $\{\mathbf{p}\}$. We would form a new dataset $\{\hat{\mathbf{x}}\}$, with the i'th element given by

$$\hat{\mathbf{x}}_i = \mathcal{U}\mathbf{p}_i + \mathsf{mean}\left(\{\mathbf{x}\}\right)$$

(you should check this expression). But this expression says that $\hat{\mathbf{x}}_i$ is constructed by forming a weighted sum of the first s columns of $\mathcal{U}$ (because all the other components of $\mathbf{p}_i$ are zero), then adding $\mathsf{mean}\left(\{\mathbf{x}\}\right)$. If we write $\mathbf{u}_j$ for the j'th

column of $\mathcal{U}$ and w_{ij} for a weight value, we have

$$\hat{\mathbf{x}}_i = \sum_{j=1}^{s} w_{ij}\mathbf{u}_j + \text{mean}\left(\{\mathbf{x}\}\right).$$

What is important about this sum is that s is usually a lot less than d. In turn, this means that we are representing the dataset using a lower dimensional dataset. We choose an s dimensional flat subspace of d dimensional space, and represent each data item with a point that lies on in that subset. The $\mathbf{u}_j$ are known as *principal components* (sometimes *loadings*) of the dataset; the $r_i^{(j)}$ are sometimes known as *scores*, but are usually just called *coefficients*. Forming the representation is called *principal components analysis* or *PCA*. The weights w_{ij} are actually easy to evaluate. We have that

$$w_{ij} = r_i^{(j)} = (\mathbf{x}_i - \text{mean}\left(\{\mathbf{x}\}\right))^T \mathbf{u}_j.$$

Remember this: *Data items in a d dimensional data set can usually be represented with good accuracy as a weighted sum of a small number s of d dimensional vectors, together with the mean. This means that the dataset lies on an s-dimensional subspace of the d-dimensional space. The subspace is spanned by the principal components of the data.*

33.2.4 The Error in a Low Dimensional Representation

We can easily determine the error in approximating $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$. The error in representing $\{\mathbf{r}\}$ by $\{\mathbf{p}\}$ was easy to compute. We had

$$\frac{1}{N}\sum_i \left[(\mathbf{r}_i - \mathbf{p}_i)^T (\mathbf{r}_i - \mathbf{p}_i)\right] = \sum_{j=s+1}^{j=d} \text{var}\left(\left\{r^{(j)}\right\}\right) = \sum_{j=s+1}^{j=d} \lambda_j$$

If this sum is small compared to the sum of the first s components, then dropping the last $d - s$ components results in a small error.

The average error in representing $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$ is now easy to get. Rotations and translations do not change lengths. This means that

$$\frac{1}{N}\sum_i |\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 = \frac{1}{N}\sum_i |\mathbf{r}_i - \mathbf{p}_i\|^2 = \sum_{j=s+1}^{j=d} \lambda_j$$

which is easy to evaluate, because these are the values of the $d - s$ eigenvalues of $\text{Covmat}\left(\{\mathbf{x}\}\right)$ that we decided to ignore. Now we could choose s by identifying how much error we can tolerate. More usual is to plot the eigenvalues of the covariance matrix, and look for a "knee", like that in Figure **??**. You can see that the sum of remaining eigenvalues is small.

Procedure: 33.1 *Principal Components Analysis*

Assume we have a general data set $\mathbf{x}_i$, consisting of N d-dimensional vectors. Now write $\Sigma = \mathsf{Covmat}(\{\mathbf{x}\})$ for the covariance matrix.
Form $\mathcal{U}$, Λ, such that

$$\Sigma\mathcal{U} = \mathcal{U}\Lambda$$

(these are the eigenvectors and eigenvalues of Σ). Ensure that the entries of Λ are sorted in decreasing order. Choose r, the number of dimensions you wish to represent. Typically, we do this by plotting the eigenvalues and looking for a "knee" (Figure **??**). It is quite usual to do this by hand.
Constructing a low-dimensional representation: For $1 \leq j \leq s$, write $\mathbf{u}_i$ for the i'th column of $\mathcal{U}$. Represent the data point $\mathbf{x}_i$ as

$$\hat{\mathbf{x}}_i = \mathsf{mean}(\{\mathbf{x}\}) + \sum_{j=1}^{s} \left[\mathbf{u}_j^T (\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))\right] \mathbf{u}_j$$

The error in this representation is

$$\frac{1}{N}\sum_i \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 = \sum_{j=s+1}^{j=d} \lambda_j$$

33.2.5 Extracting a Few Principal Components with NIPALS

If you remember the curse of dimension, you should have noticed something of a problem in my account of PCA. When I described the curse, I said one consequence was that forming a covariance matrix for high dimensional data is hard or impossible. Then I described PCA as a method to understand the important dimensions in high dimensional datasets. But PCA appears to rely on covariance, so I should not be able to form the principal components in the first place. In fact, we can form principal components without computing a covariance matrix.

I will now assume the dataset has zero mean, to simplify notation. This is easily achieved. You subtract the mean from each data item at the start, and add the mean back once you've finished. As usual, we have N data items, each a d dimensional column vector. We will now arrange these into a matrix,

$$\mathcal{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \cdots \\ \mathbf{x}_N^T \end{pmatrix}$$

where each *row* of the matrix is a data vector. Now assume we wish to recover the first principal component. This means we are seeking a vector $\mathbf{u}$ and a set of N numbers w_i such that $w_i\mathbf{u}$ is a good approximation to $\mathbf{x}_i$. Now we can stack the w_i into a column vector $\mathbf{w}$. We are asking that the matrix $\mathbf{w}\mathbf{u}^T$ be a good

approximation to $\mathcal{X}$, in the sense that $\mathbf{w}\mathbf{u}^T$ encodes as much of the variance of $\mathcal{X}$ as possible.

The *Frobenius norm* is a term for the matrix norm obtained by summing squared entries of the matrix. We write

$$\|\mathcal{A}\|_F{}^2 = \sum_{i,j} a_{ij}^2.$$

In the exercises, you will show that the right choice of $\mathbf{w}$ and $\mathbf{u}$ minimizes the cost

$$\|\mathcal{X} - \mathbf{w}\mathbf{u}^T\|_F{}^2$$

which we can write as

$$C(\mathbf{w}, \mathbf{u}) = \sum_{ij} (x_{ij} - w_i u_j)^2 .$$

Now we need to *find* the relevant $\mathbf{w}$ and $\mathbf{u}$. Notice there is not a unique choice, because the pair $(s\mathbf{w}, (1/s)\mathbf{u})$ works as well as the pair $(\mathbf{w}, \mathbf{u})$. We will choose $\mathbf{u}$ such that $|\mathbf{u}\| = 1$. There is still not a unique choice, because you can flip the signs in $\mathbf{u}$ and $\mathbf{w}$, but this doesn't matter. At the right $\mathbf{w}$ and $\mathbf{u}$, the gradient of the cost function will be zero.

The gradient of the cost function is a set of partial derivatives with respect to components of $\mathbf{w}$ and $\mathbf{u}$. The partial with respect to w_k is

$$\frac{\partial C}{\partial w_k} = \sum_j (x_{kj} - w_k u_j) u_j$$

which can be written in matrix vector form as

$$\nabla_{\mathbf{w}} C = (\mathcal{X} - \mathbf{w}\mathbf{u}^T)\mathbf{u}.$$

Similarly, the partial with respect to u_l is

$$\frac{\partial C}{\partial u_l} = \sum_i (x_{il} - w_i u_l) w_i$$

which can be written in matrix vector form as

$$\nabla_{\mathbf{u}} C = (\mathcal{X}^T - \mathbf{u}\mathbf{w}^T)\mathbf{w}.$$

At the solution, these partial derivatives are zero. Notice that, if we know the right $\mathbf{u}$, then the equation $\nabla_{\mathbf{w}} C = 0$ is linear in $\mathbf{w}$. Similarly, if we know the right $\mathbf{w}$, then the equation $\nabla_{\mathbf{u}} C = 0$ is linear in $\mathbf{u}$. This suggests an algorithm. First, assume we have an estimate of $\mathbf{u}$, say $\mathbf{u}^{(n)}$. Then we could choose the $\mathbf{w}$ that makes the partial wrt $\mathbf{w}$ zero, so

$$\hat{\mathbf{w}} = \frac{\mathcal{X}\mathbf{u}^{(n)}}{(\mathbf{u}^{(n)})^T\mathbf{u}^{(n)}}.$$

Now we can update the estimate of $\mathbf{u}$ by choosing a value that makes the partial wrt $\mathbf{u}$ zero, using our estimate $\hat{\mathbf{w}}$, to get

$$\hat{\mathbf{u}} = \frac{\mathcal{X}^T \hat{\mathbf{w}}}{(\hat{\mathbf{w}})^T \hat{\mathbf{w}}}.$$

We need to rescale to ensure that our estimate of $\mathbf{u}$ has unit length. Write $s = \sqrt{(\hat{\mathbf{u}})^T \hat{\mathbf{u}}}$ We get

$$\mathbf{u}^{(n+1)} = \frac{\hat{\mathbf{u}}}{s}$$

and

$$\mathbf{w}^{(n+1)} = s\hat{\mathbf{w}}.$$

This iteration can be started by choosing some row of $\mathcal{X}$ as $\mathbf{u}^{(0)}$. You can test for convergence by checking $\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\|$. If this is small enough, then the algorithm has converged.

To obtain a second principal component, you form $\mathcal{X}^{(1)} = \mathcal{X} - \mathbf{w}\mathbf{u}^T$ and apply the algorithm to that. You can get many principal components like this, but it's not a good way to get all of them (eventually numerical issues mean the estimates are poor). The algorithm is widely known as NIPALS (for Non-linear Iterative Partial Least Squares).

33.2.6 Principal Components and Missing Values

Now imagine our dataset has missing values. We assume that the values are not missing in inconvenient patterns — if, for example, the k'th component was missing for every vector then we'd have to drop it — but don't go into what precise kind of pattern is a problem. Your intuition should suggest that we can estimate a few principal components of the dataset without particular problems. The argument is as follows. Each entry of a covariance matrix is a form of average; estimating averages in the presence of missing values is straightforward; and, when we estimate a few principal components, we are estimating far fewer numbers than when we are estimating a whole covariance matrix, so we should be able to make something work. This argument is sound, if vague.

The whole point of NIPALS is that, if you want a few principal components, you don't need to use a covariance matrix. This simplifies thinking about missing values. NIPALS is quite forgiving of missing values, though missing values make it hard to use matrix notation. Recall I wrote the cost function as $C(\mathbf{w}, \mathbf{u}) = \sum_{ij}(x_{ij} - w_i u_j)^2$. Notice that missing data occurs in $\mathcal{X}$ because there are x_{ij} whose values we don't know, but there is no missing data in $\mathbf{w}$ or $\mathbf{u}$ (we're estimating the values, and we always have *some* estimate). We change the sum so that it ranges over only the known values, to get

$$C(\mathbf{w}, \mathbf{u}) = \sum_{ij \in \text{known values}} \left(x_{ij} - w_i u_j\right)^2.$$

Now we need a shorthand to ensure that sums run over only known values. Write $\mathcal{V}(k)$ for the set of column (resp. row) indices of known values for a given row (resp.

column index) k. So $i \in \mathcal{V}(k)$ means all i such that x_{ik} is known *or* all i such that x_{ki} is known (the context will tell you which). We have

$$\frac{\partial C}{\partial w_k} = \sum_{j \in \mathcal{V}(k)} (x_{kj} - w_k u_j)\, u_j$$

and

$$\frac{\partial C}{\partial u_l} = \sum_{i \in \mathcal{V}(l)} (x_{il} - w_i u_l)\, w_i.$$

These partial derivatives must be zero at the solution. This means we can use $\mathbf{u}^{(n)}$, $\mathbf{w}^{(n)}$ to estimate

$$\hat{w}_k = \frac{\sum_{j \in \mathcal{V}(k)} x_{kj} u_j^{(n)}}{\sum_j u_j^{(n)} u_j^{(n)}}$$

and

$$\hat{u}_l = \frac{\sum_{i \in \mathcal{V}(l)} x_{il} \hat{w}_l}{\sum_i \hat{w}_i \hat{w}_i}$$

We then normalize as before to get $\mathbf{u}^{(n+1)}$, $\mathbf{w}^{(n+1)}$.

Procedure: 33.2 *Obtaining some principal components with NIPALS*

We assume that $\mathcal{X}$ has zero mean. Each row is a data item. Start with $\mathbf{u}^0$ as some row of $\mathcal{X}$. Write $\mathcal{V}(k)$ for the set of indices of known values for a given row or column index k. Now iterate

- compute
$$\hat{w}_k = \frac{\sum_{j \in \mathcal{V}(k)} x_{kj} u^{(n)_j}}{\sum_j u_j^{(n)} u_j^{(n)}}$$
and
$$\hat{u}_l = \frac{\sum_{i \in \mathcal{V}(l)} x_{il} \hat{w}_l}{\sum_i \hat{w}_i \hat{w}_i};$$
- compute $s = \sqrt{(\hat{\mathbf{u}})^T \hat{\mathbf{u}}}$, and
$$\mathbf{u}^{(n+1)} = \frac{\hat{\mathbf{u}}}{s}$$
and
$$\mathbf{w}^{(n+1)} = s\hat{\mathbf{w}};$$
- Check for convergence by checking that $\left\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\right\|$ is small.

This procedure yields a single principal component representing the highest variance in the dataset. To obtain the next principal component, replace $\mathcal{X}$ with $\mathcal{X} - \mathbf{w}\mathbf{u}^T$ and repeat the procedure. This process will yield good estimates of the first few principal components, but as you generate more principal components, numerical errors will become more significant.

33.2.7 PCA as Smoothing

Assume that each data item $\mathbf{x}_i$ is noisy. We use a simple noise model. Write $\tilde{\mathbf{x}}_i$ for the true underlying value of the data item, and ξ_i for the value of a normal random variable with zero mean and covariance $\sigma^2\mathcal{I}$. Then we use the model

$$\mathbf{x}_i = \tilde{\mathbf{x}}_i + \xi_i$$

(so the noise in each component is independent, has zero mean, and has variance σ^2; this is known as *additive, zero-mean, independent gaussian noise*). You should think of the measurement $\mathbf{x}_i$ as an estimate of $\tilde{\mathbf{x}}_i$. A principal component analysis of $\mathbf{x}_i$ can produce an estimate of $\tilde{\mathbf{x}}_i$ that is closer than the measurements are.

There is a subtlety here, because the noise is random, but we see the values of the noise. This means that $\mathsf{Covmat}(\{\xi\})$ (i.e. the covariance of the observed

numbers) is the value of a random variable (because the noise is random) whose mean is $\sigma^2\mathcal{I}$ (because that's the model). The subtlety is that $\mathsf{mean}\left(\{\xi\}\right)$ will not necessarily be exactly $\mathbf{0}$ and $\mathsf{Covmat}\left(\{\xi\}\right)$ will not necessarily be exactly $\sigma^2\mathcal{I}$. The weak law of large numbers tells us that $\mathsf{Covmat}\left(\{\xi\}\right)$ will be extremely close to its expected value (which is $\sigma^2\mathcal{I}$) for a large enough dataset. We will assume that $\mathsf{mean}\left(\{\xi\}\right) = \mathbf{0}$ and $\mathsf{Covmat}\left(\{\xi\}\right) = \sigma^2\mathcal{I}$.

The first step is to write $\tilde{\Sigma}$ for the covariance matrix of the true underlying values of the data, and $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$ for the covariance of the observed data. Then it is straightforward that

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right) = \tilde{\Sigma} + \sigma^2\mathcal{I}$$

because the noise is independent of the measurements. Notice that if $\mathcal{U}$ diagonalizes $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$, it will also diagonalize $\tilde{\Sigma}$. Write $\tilde{\Lambda} = \mathcal{U}^T\tilde{\Sigma}\mathcal{U}$. We have

$$\mathcal{U}^T\mathsf{Covmat}\left(\{\mathbf{x}\}\right)\mathcal{U} = \Lambda = \tilde{\Lambda} + \sigma^2\mathcal{I}.$$

Now think about the diagonal entries of Λ. If they are large, then they are quite close to the corresponding components of $\tilde{\Lambda}$, but if they are small, it is quite likely they are the result of noise. But these eigenvalues are tightly linked to error in a PCA representation.

In PCA (procedure 33.1), the d dimensional data point $\mathbf{x}_i$ is represented by

$$\hat{\mathbf{x}}_i = \mathsf{mean}\left(\{\mathbf{x}\}\right) + \sum_{j=1}^{s}\left[\mathbf{u}_j^T\left(\mathbf{x}_i - \mathsf{mean}\left(\{\mathbf{x}\}\right)\right)\right]\mathbf{u}_j$$

where $\mathbf{u}_j$ are the principal components. This representation is obtained by setting the coefficients of the $d - s$ principal components with small variance to zero. The error in representing $\{\mathbf{x}\}$ with $\{\hat{\mathbf{x}}\}$ follows from section 33.2.4 and is

$$\frac{1}{N}\sum_i \left\|\mathbf{x}_i - \hat{\mathbf{x}}_i\right\|^2 = \sum_{j=s+1}^{j=d}\lambda_j.$$

Now consider the error in representing $\tilde{\mathbf{x}}_i$ (which we don't know) by $\mathbf{x}_i$ (which we do). The average error over the whole dataset is

$$\frac{1}{N}\sum_i \left\|\mathbf{x}_i - \tilde{\mathbf{x}}_i\right\|^2.$$

Because the variance of the noise is $\sigma^2\mathcal{I}$, this error must be $d\sigma^2$. Alternatively, we could represent $\tilde{\mathbf{x}}_i$ by $\hat{\mathbf{x}}_i$. The average error of this representation over the whole dataset will be

$$\begin{aligned}\frac{1}{N}\sum_i \left\|\hat{\mathbf{x}}_i - \tilde{\mathbf{x}}_i\right\|^2 &= \text{Error in components that are preserved} + \\ &\quad \text{Error in components that are zeroed} \\ &= s\sigma^2 + \sum_{j=s+1}^{d}\tilde{\lambda}_u.\end{aligned}$$

Now if, for $j > s$, $\tilde{\lambda}_j < \sigma^2$, this error is smaller than $d\sigma^2$. We don't know which s guarantees this unless we know σ^2 and $\tilde{\lambda}_j$ which often doesn't happen. But it's usually possible to make a safe choice, and so *smooth* the data by reducing noise. This smoothing works because the components of the data are correlated. So the best estimate of each component of a high dimensional data item is likely not the measurement – it's a prediction obtained from all measurements. The projection onto principal components is such a prediction.

Remember this: *Given a d dimensional dataset where data items have had independent random noise added to them, representating each data item on $s < d$ principal components can result in a representation which is on average closer to the true underlying data than the original data items. The choice of s is application dependent.*

33.3 VISUALIZATION

33.3.1 Simple Visualization with Principal Coordinate Analysis

33.3.2 TSNE

33.3.3 the other mapper

CHAPTER 34

Clustering Methods

CHAPTER 35

SILS

35.1 SHIFT INVARIANT LINEAR SYSTEMS

Convolution represents the effect of a large class of system. In particular, most imaging systems have, to a good approximation, three significant properties:

- **Superposition:** We expect that

$$R(f + g) = R(f) + R(g);$$

  that is, the response to the sum of stimuli is the sum of the individual responses.

- **Scaling:** The response to a zero input is zero. Taken with superposition, we have that the response to a scaled stimulus is a scaled version of the response to the original stimulus; that is,

$$R(kf) = kR(f).$$

  A device that exihibits superposition and scaling is *linear*.

- **Shift invariance:** In a *shift invariant* system, the response to a translated stimulus is just a translation of the response to the stimulus. This means that, for example, if a view of a small light aimed at the center of the camera is a small, bright blob, then if the light is moved to the periphery, we should see the same small, bright blob, only translated.

A device that is linear and shift invariant is known as a *shift invariant linear system*, or often just as a *system*.

The response of a shift invariant linear system to a stimulus is obtained by convolution. We demonstrate this first for systems that take discrete inputs—say, vectors or arrays—and produce discrete outputs. We then use this to describe the behavior of systems that operate on continuous functions of the line or the plane, and from this analysis we obtain some useful facts about convolution.

35.1.1 Discrete Convolution

In the 1D case, we have a shift invariant linear system that takes a vector and responds with a vector. This case is the easiest to handle because there are fewer indices to look after. The 2D case—a system that takes an array and responds with an array—follows easily. In each case, we assume that the input and output are infinite dimensional. This allows us to ignore some minor issues that arise at the boundaries of the input. We deal with these in Section 35.1.3.

Discrete Convolution in One Dimension

We have an input vector $\mathbf{f}$. For convenience, we assume that the vector has infinite length and its elements are indexed by the integers (i.e., there is an element with index -1, say). The ith component of this vector is f_i. Now $\mathbf{f}$ is a weighted sum of basis elements. A convenient basis is a set of elements that have a one in a single component and zeros elsewhere. We write

$$\mathbf{e}_0 = \ldots 0, 0, 0, 1, 0, 0, 0, \ldots$$

This is a data vector that has a 1 in the zeroth place, and zeros elsewhere. Define a shift operation, which takes a vector to a shifted version of that vector. In particular, the vector $\texttt{Shift}(\mathbf{f}, i)$ has, as its jth component, the $j - i$th component of $\mathbf{f}$. For example, $\texttt{Shift}(\mathbf{e}_0, 1)$ has a zero in the first component. Now, we can write

$$\mathbf{f} = \sum_i f_i \texttt{Shift}(\mathbf{e}_0, i).$$

We write the response of our system to a vector $\mathbf{f}$ as

$$R(\mathbf{f}).$$

Now, because the system is shift invariant, we have

$$R(\texttt{Shift}(\mathbf{f}, k)) = \texttt{Shift}(R(\mathbf{f}), k).$$

Furthermore, because it is linear, we have

$$R(k\mathbf{f}) = kR(\mathbf{f}).$$

This means that

$$\begin{aligned} R(\mathbf{f}) &= R\left(\sum_i f_i \texttt{Shift}(\mathbf{e}_0, i)\right) \\ &= \sum_i R(f_i \texttt{Shift}(\mathbf{e}_0, i)) \\ &= \sum_i f_i R(\texttt{Shift}(\mathbf{e}_0, i)) \\ &= \sum_i f_i \texttt{Shift}(R(\mathbf{e}_0), i)). \end{aligned}$$

This means that to obtain the system's response to any data vector, we need to know only its response to $\mathbf{e}_0$. This is usually called the system's *impulse response*. Assume that the impulse response can be written as $\mathbf{g}$. We have

$$R(\mathbf{f}) = \sum_i f_i \texttt{Shift}(\mathbf{g}, i) = \mathbf{g} * \mathbf{f}.$$

This defines an operation—the 1D, discrete version of convolution—which we write with a $*$.

This is all very well, but it doesn't give us a particularly easy expression for the output. If we consider the jth element of $R(\mathbf{f})$, which we write as R_i, we must have

$$R_j = \sum_i g_{j-i} f_i,$$

which conforms to (and explains the origin of) the form used in Section **??**.

Discrete Convolution in Two Dimensions We now use an array of values and write the i, jth element of the array $\mathcal{D}$ as D_{ij}. The appropriate analogy to an impulse response is the response to a stimulus that looks like

$$\mathcal{E}_{00} = \begin{array}{ccccc} \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 1 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

If $\mathcal{G}$ is the response of the system to this stimulus, the same considerations as for 1D convolution yield a response to a stimulus $\mathcal{F}$, that is,

$$R_{ij} = \sum_{u,v} G_{i-u,j-v} F_{uv},$$

which we write as

$$\mathcal{R} = \mathcal{G} * *\mathcal{H}.$$

35.1.2 Continuous Convolution

There are shift invariant linear systems that produce a continuous response to a continuous input; for example, a camera lens takes a set of radiances and produces another set, and many lenses are approximately shift invariant. A brief study of these systems allows us to study the information lost by approximating a continuous function—the incoming radiance values across an image plane—by a discrete function—the value at each pixel.

The natural description is in terms of the system's response to a rather unnatural function, the δ-function, which is not a function in formal terms. We do the derivation first in one dimension to make the notation easier.

Convolution in One Dimension

We obtain an expression for the response of a continuous shift invariant linear system from our expression for a discrete system. We can take a discrete input and replace each value with a box straddling the value; this gives a continuous input function. We then make the boxes narrower and consider what happens in the limit.

Our system takes a function of one dimension and returns a function of one dimension. Again, we write the response of the system to some input $f(x)$ as $R(f)$; when we need to emphasize that f is a function, we write $R(f(x))$. The response

is also a *function*; occasionally, when we need to emphasize this fact, we write $R(f)(u)$. We can express the linearity property in this notation by writing

$$R(kf) = kR(f)$$

(for k some constant) and the shift invariance property by introducing a `Shift` operator, which takes functions to functions:

$$\texttt{Shift}(f, c) = f(u - c).$$

With this `Shift` operator, we can write the shift invariance property as

$$R(\texttt{Shift}(f, c)) = \texttt{Shift}(R(f), c).$$

We define the *box* function as:

$$box_\epsilon(x) = \begin{cases} 0 & abs(x) > \frac{\epsilon}{2} \\ 1 & abs(x) < \frac{\epsilon}{2} \end{cases} .$$

The value of $box_\epsilon(\epsilon/2)$ does not matter for our purposes. The input function is $f(x)$. We construct an even grid of points x_i, where $x_{i+1} - x_i = \epsilon$. We now construct a vector $\mathbf{f}$ whose ith component (written f_i) is $f(x_i)$. This vector can be used to represent the function.

We obtain an approximate representation of f by $\sum_i f_i \texttt{Shift}(box_\epsilon, x_i)$. We apply this input to a shift invariant linear system; the response is a weighted sum of shifted responses to box functions. This means that

$$\begin{aligned} R\left(\sum_i f_i \texttt{Shift}(box_\epsilon, x_i)\right) &= \sum_i R(f_i \texttt{Shift}(box_\epsilon, x_i)) \\ &= \sum_i f_i R(\texttt{Shift}(box_\epsilon, x_i)) \\ &= \sum_i f_i \texttt{Shift}(R(\frac{box_\epsilon}{\epsilon}\epsilon), x_i) \\ &= \sum_i f_i \texttt{Shift}(R(\frac{box_\epsilon}{\epsilon}), x_i)\epsilon. \end{aligned}$$

So far, everything has followed our derivation for discrete functions. We now have something that looks like an approximate integral if $\epsilon \to 0$.

We introduce a new device, called a δ-function, to deal with the term box_ϵ/ϵ. Define

$$d_\epsilon(x) = \frac{box_\epsilon(x)}{\epsilon}.$$

The δ-function is:

$$\delta(x) = \lim_{\epsilon \to 0} d_\epsilon(x).$$

We don't attempt to evaluate this limit, so we need not discuss the value of $\delta(0)$. One interesting feature of this function is that, for practical shift invariant

linear systems, the response of the system to a δ-function exists and has *compact support* (i.e., is zero except on a finite number of intervals of finite length). For example, a good model of a δ-function in 2D is an extremely small, extremely bright light. If we make the light smaller and brighter while ensuring the total energy is constant, we expect to see a small but finite spot due to the defocus of the lens. The δ-function is the natural analogue for $\mathbf{e}_0$ in the continuous case.

This means that the expression for the response of the system,

$$\sum_i f_i \texttt{Shift}(R(\frac{box_\epsilon}{\epsilon}), x_i)\epsilon,$$

turns into an integral as ϵ limits to zero. We obtain

$$\begin{aligned} R(f) &= \int \{R(\delta)(u-x')\}\, f(x')dx' \\ &= \int g(u-x')f(x')dx', \end{aligned}$$

where we have written $R(\delta)$—which is usually called the **impulse response** of the system—as g and have omitted the limits of the integral. These integrals could be from $-\infty$ to ∞, but more stringent limits could apply if g and h have compact support. This operation is called **convolution** (again), and we write the foregoing expression as

$$R(f) = (g * f).$$

Convolution is *commutative*, meaning

$$(g * h)(x) = (h * g)(x).$$

Convolution is *associative*, meaning that

$$(f * (g * h)) = ((f * g) * h).$$

This latter property means that we can find a single shift invariant linear system that behaves like the composition of two different systems. This will be useful when we discuss sampling.

Convolution in Two Dimensions

The derivation of convolution in two dimensions requires more notation. A box function is now given by $box_{\epsilon^2}(x,y) = box_\epsilon(x)box_\epsilon(y)$; we now have

$$d_\epsilon(x,y) = \frac{box_{\epsilon^2}(x,y)}{\epsilon^2}.$$

The δ-function is the limit of $d_\epsilon(x,y)$ function as $\epsilon \to 0$. Finally, there are more terms in the sum. All this activity results in the expression

$$\begin{aligned} R(h)(x,y) &= \int\int g(x-x', y-y')h(x',y')dxdy \\ &= (g **h)(x,y), \end{aligned}$$

where we have used two $*$s to indicate a two-dimensional convolution. Convolution in 2D is *commutative*, meaning that

$$(g * *h) = (h * *g),$$

and *associative*, meaning that

$$((f * *g) * *h) = (f * *(g * *h)).$$

A natural model for the impulse response of a two-dimensional system is to think of the pattern seen in a camera viewing a very small, distant light source (which subtends a very small viewing angle). In practical lenses, this view results in some form of fuzzy blob, justifying the name *point spread function*, which is often used for the impulse response of a 2D system. The point spread function of a linear system is often known as its *kernel*.

35.1.3 Edge Effects in Discrete Convolutions

In practical systems, we cannot have infinite arrays of data. This means that when we compute the convolution, we need to contend with the edges of the image; at the edges, there are pixel locations where computing the value of the convolved image requires image values that don't exist. There are a variety of strategies we can adopt:

- **Ignore these locations**, which means that we report only values for which every required image location exists. This has the advantage of probity, but the disadvantage that the output is smaller than the input. Repeated convolutions can cause the image to shrink quite drastically.
- **Pad the image with constant values**, which means that, as we look at output values closer to the edge of the image, the extent to which the output of the convolution depends on the image goes down. This is a convenient trick because we can ensure that the image doesn't shrink, but it has the disadvantage that it can create the appearance of substantial gradients near the boundary.
- **Pad the image in some other way**. For example, we might think of the image as a doubly periodic function so that if we have an $n \times m$ image, then column $m + 1$—required for the purposes of convolution—would be the same as column $m - 1$. This can create the appearance of substantial second derivative values near the boundary.

Index

Index: Procedures

Index: Remember This

Index: Notation

Index: Resources

Index: TODO