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

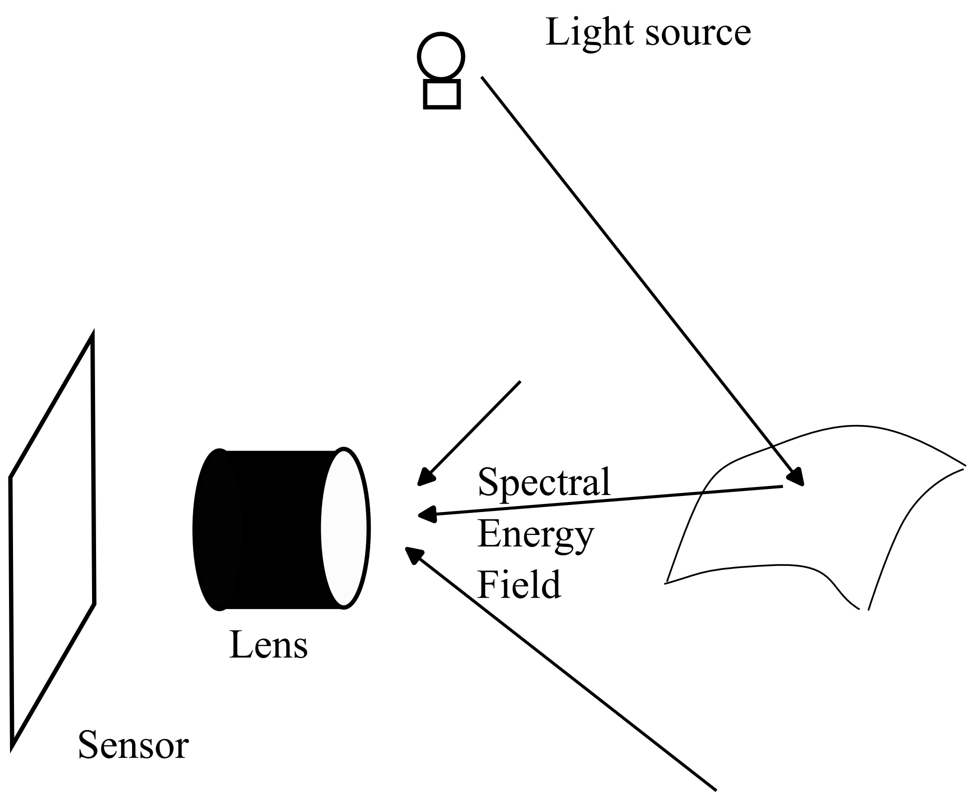

FIGURE 2.1: *A high-level model of imaging. Light leaves light sources and reflects from surfaces. Eventually, some light arrives at a camera and enters a lens system. Some of that light arrives at a photosensor inside the camera.*

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 –

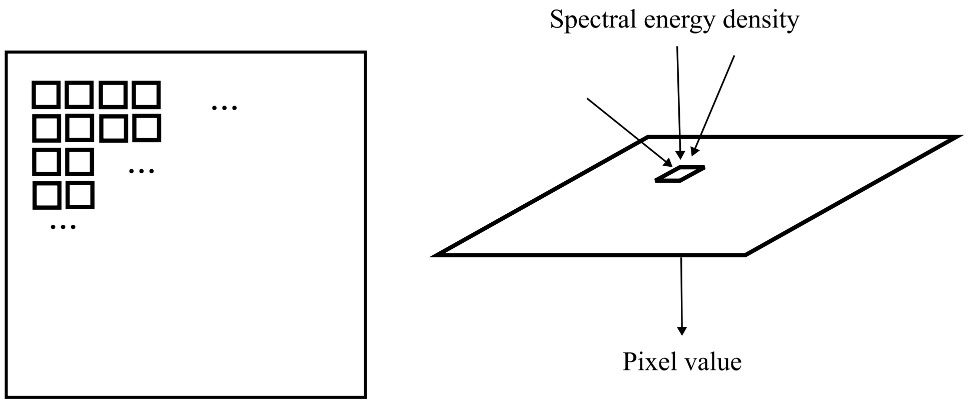

FIGURE 2.2: *The photosensor is divided into a grid of pixels, which are small sensitive locations. Each pixel receives an incoming spectral energy field, and turns it into a number. This number is typically a weighted average over a position in the sensor, a very small range of incoming directions and 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.*

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.* 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 interpolation*, 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).

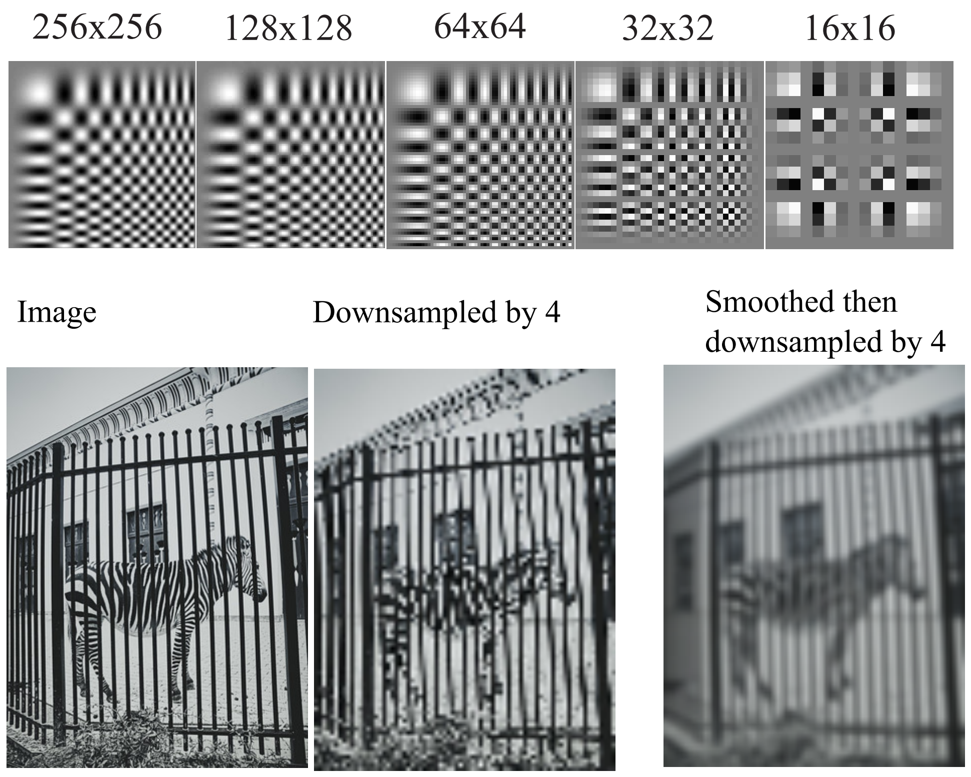

FIGURE 2.3: *Because each pixel in the sensor averages over a small range of directions and positions, the process mapping the input spectral energy distribution to pixel values can be thought of as sampling. On the* **left***, is a representation of the energy distribution as a continuous function of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

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 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)

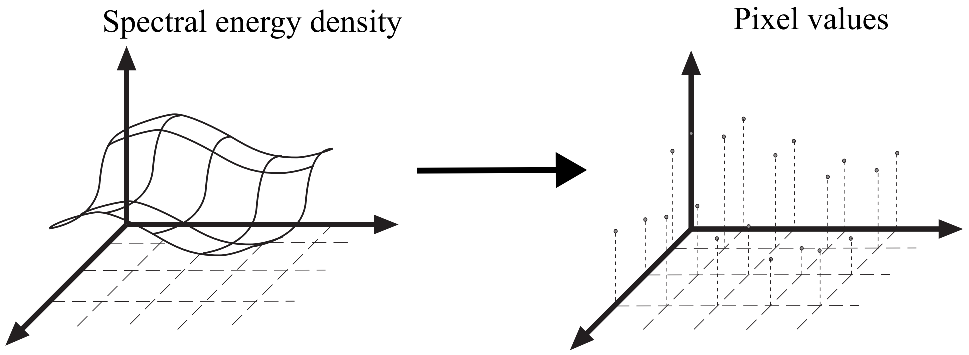

FIGURE 2.4: **Top row** *shows an image on the* **left***, and then a set of versions resampled to different grids then expanded to the original size. Notice how detail is lost in the resampling process. Some small boxes disappear, and others turn into large boxes.* **Bottom left** *shows a picture of 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.*

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 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

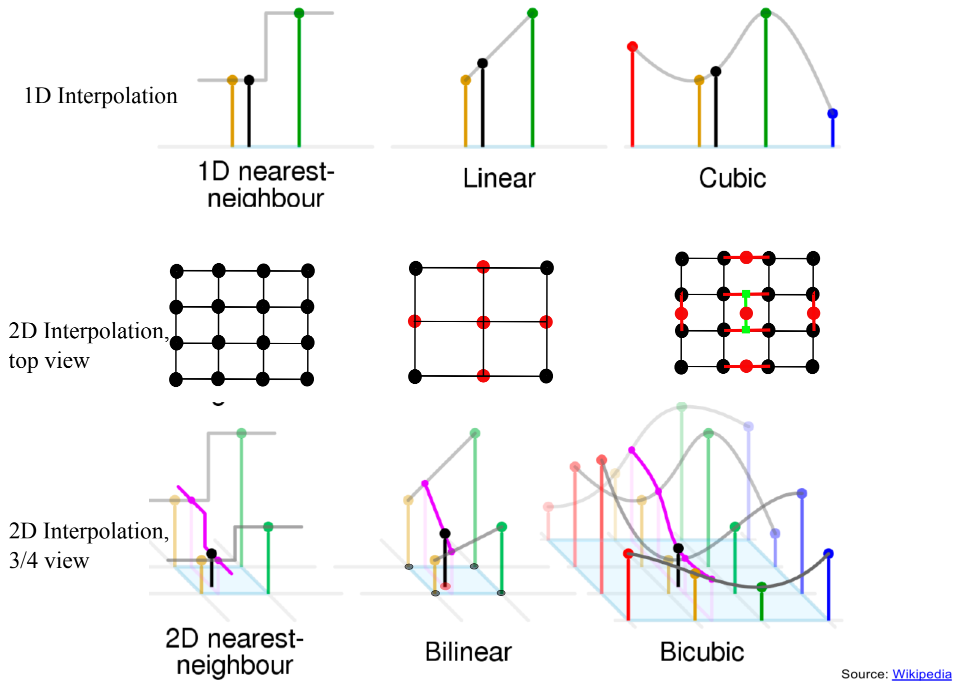

FIGURE 2.5: **Top row** *shows interpolation in 1D. To obtain a value for a location between samples, we could: choose the value of the closest sample (nearest neighbors,* **left***; connect the two closest samples with a line segment, then sample that (linear interpolation,* **center***); or construct 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.*

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.*

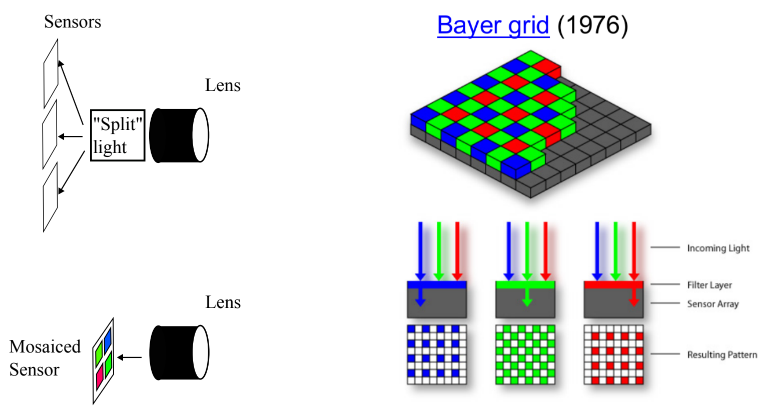

FIGURE 2.6: *There are two main ways to obtain color images. One can (as in* **top left***) build a multiccd camera with three imaging sensors. Each has a different response to wavelengths. The cheaper and lighter alternative is to use one imaging sensor (***bottom left***) but have a mosaic of pixels 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.*

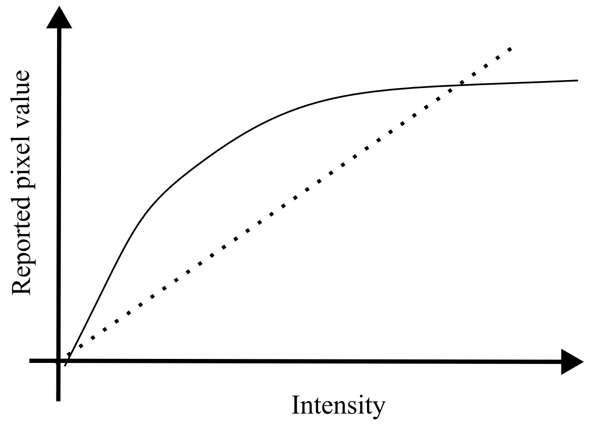

FIGURE 2.7: *A typical camera response function, mapping the response a linear sensor would compute to the output recorded by the camera. Notice that locations that would be quite dark for a linear sensor will be lighter; but as the linear sensor gets very bright, the output recorded by the camera grows 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).*

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

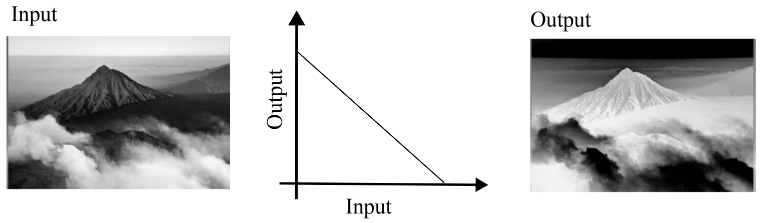

FIGURE 2.8: *Mapping individual pixel values using the mapping in the* **center** *will transform the image on the* **left** *to that on the* **right***. This function maps light pixel values to dark ones, and vice versa, and is often called a negative.*

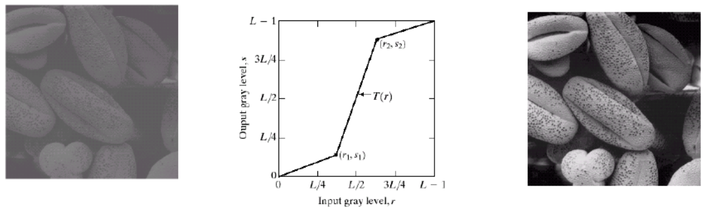

FIGURE 2.9: *Mapping individual pixel values using the mapping in the* **center** *will transform the image on the* **left** *to that on the* **right***. Notice how the mapping compresses the range of dark and light pixels, and expands that of mid-range pixels, so adjusting the contrast of the image.*

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:

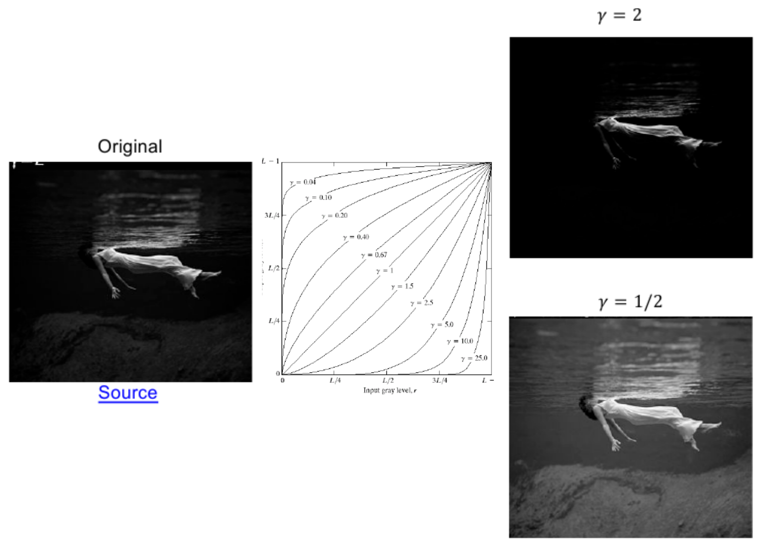

FIGURE 2.10: *Many imaging and rendering devices have a response that is a power of the input, so that output* $= C input^{\gamma}$*, where* γ *is a parameter of the device. One can simulate this effect by applying a transform like those shown in the* **center** *(curves for several values of* γ*). Note 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.

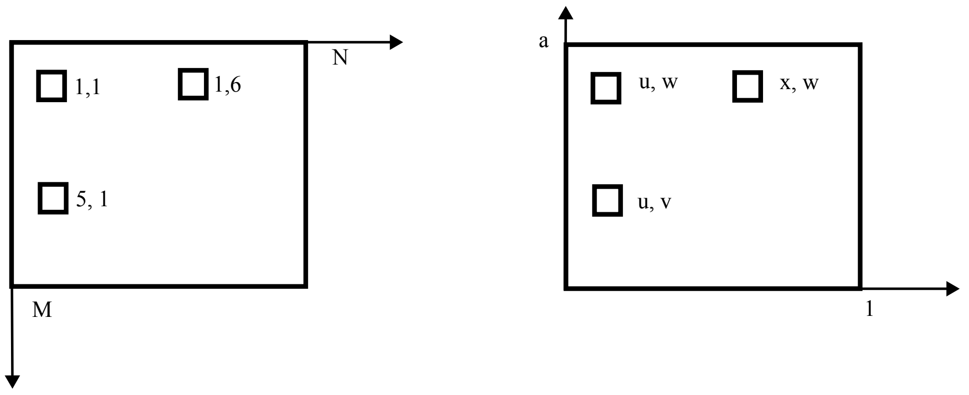

FIGURE 2.11: *Two common coordinate systems for images. On the* **left***, the origin is at the top left corner, and we count in pixels. This is an $M \times N$ image. I will use the convention $\mathcal{I}_{ij}$ for points in this coordinate system, so the top right pixel is $\mathcal{I}_{16}$. On the 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.

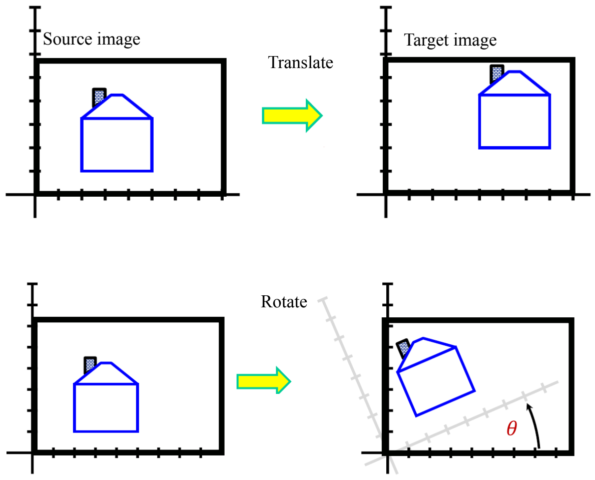

FIGURE 2.12: *Write $\mathcal{S}$ for a source image and $\mathcal{T}$ for a target image.* **Top row** *shows an image translation which maps the source pixel at (x, y) to the location $x + t_x$, $y + t_y$ in the target (so $\mathcal{T}(x + t_x, y + t_y) = \mathcal{S}(x, y)$). You should confirm that 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

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

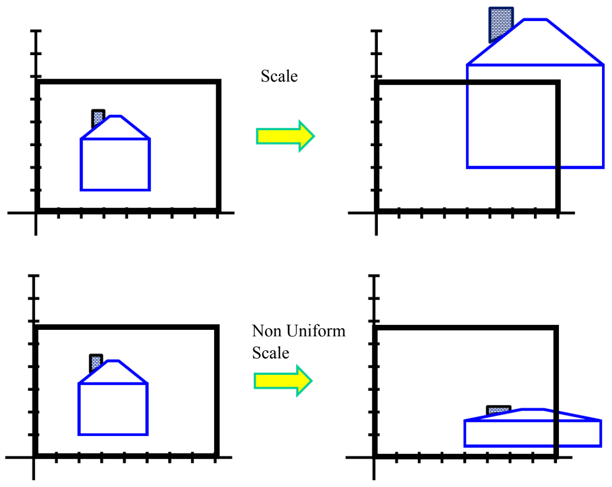

FIGURE 2.13: **Top row** *shows a uniform image scale which maps the source pixel at* (x, y) *to the location* sx, sy *in the target (so* $\mathcal{T}(sx, sy) = \mathcal{S}(x, y)$*). You should confirm that for this figure,* $s > 1$*.* **Bottom row** *shows a non-uniform image scale which maps the source 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

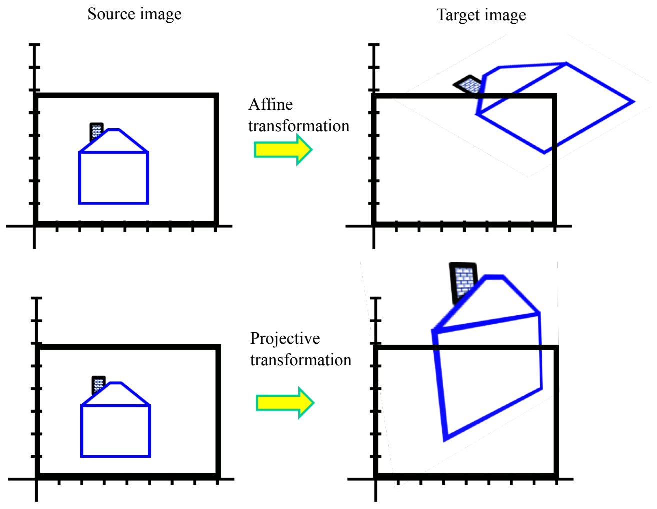

FIGURE 2.14: **Top row** *shows an affine transformation of the image which maps the source pixel at* (x, y) *to the location* $ax + by + c$, $dx + ey + f$ *in the target (so* $\mathcal{T}(ax + by + c, dx + ey + f) = \mathcal{S}(x, y)$*). You should confirm that, for this figure,* $(ae - bd) > 0$*.* **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

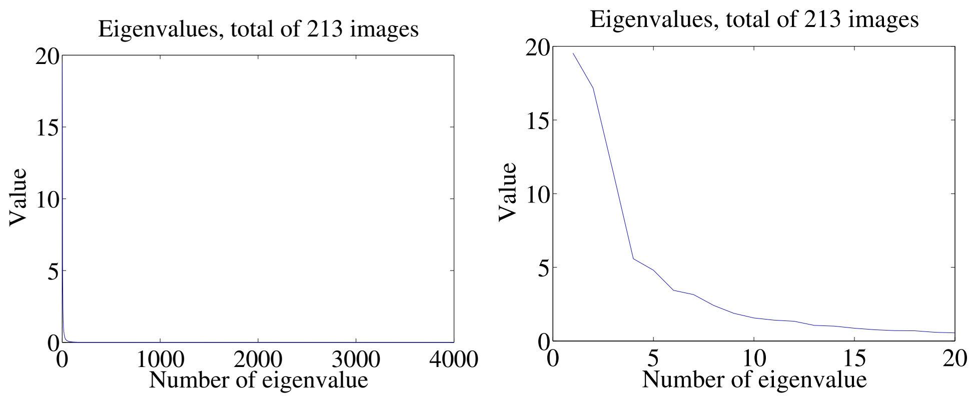

FIGURE 2.15: *On the* **left**,*the eigenvalues of the covariance of the Japanese facial expression dataset; there are 4096, so it's hard to see the curve (which is packed to the left). On the* **right**, *a zoomed version of the curve, showing how quickly the values of the eigenvalues get small.*

"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.

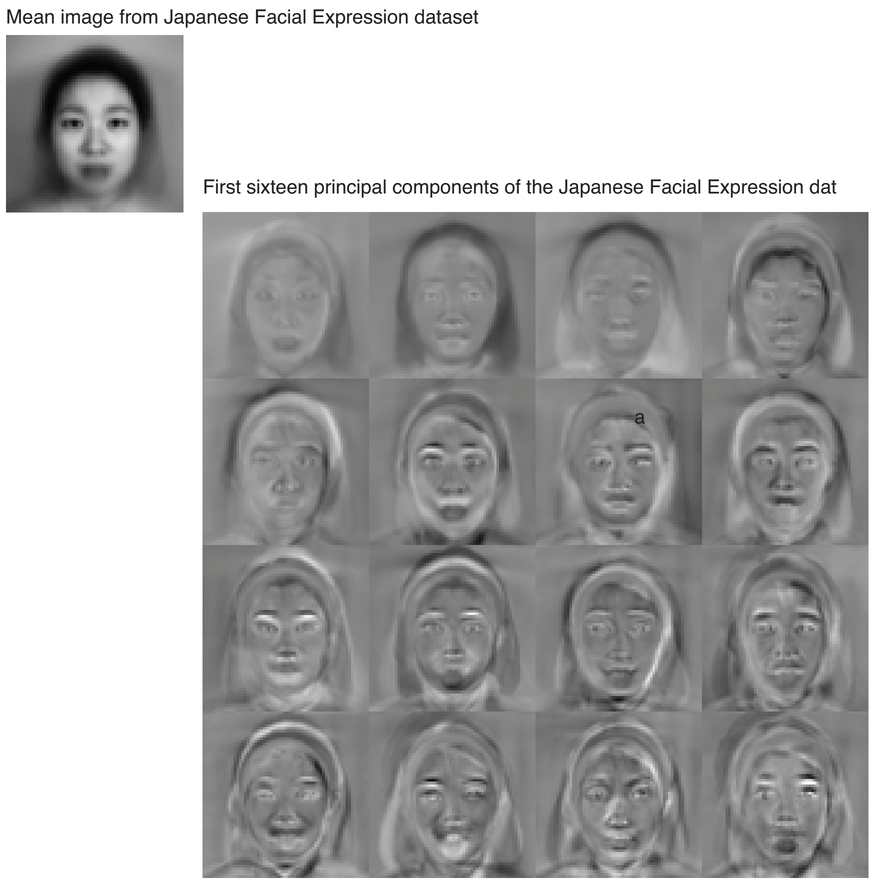
Mean image from Japanese Facial Expression dataset
First sixteen principal components of the Japanese Facial Expression dat
a

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

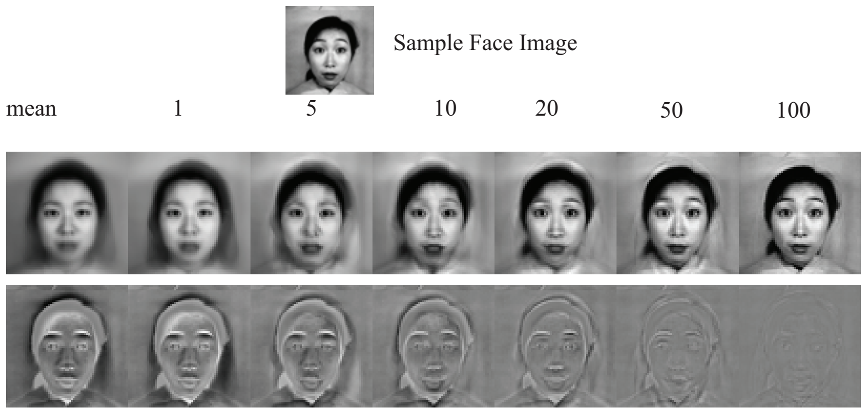

FIGURE 2.17: *Approximating a face image by the mean and some principal components; notice how good the approximation becomes with relatively few components.*