CHAPTER 3

Upsampling and Downsampling Images

Changing the size of an image is an important operation that contains a number of unexpected difficulties. In this chapter, the aspect ratio of the image will be fixed. If you increase the number of pixels, you must supply intermediate pixel values. If you decrease the number of pixels you lose information. Unless you smooth the image before doing so, the lost information will very likely produce a variety of unpleasant effects.

3.1 UPSAMPLING AND IMAGE INTERPOLATION

To *upsample* an image you increase the number of pixels in a grid. Some cases are easy. To go from, say a 100×100 image to a 200×200 image, you could simply replace each pixel with a 2×2 block of pixels, each having the same value as the original. This isn't a particularly good strategy, and the resulting images tend to look "blocky" (try it!). But upsampling by a factor that isn't an integer is more tricky. Consider going from 100×100 to 127×127. One way to do this is to duplicate 27 rows, then duplicate 27 columns in the result; to do so requires determining which columns to duplicate.

3.1.1 Inverse Warping and Interpolation

Alternatively, you might consider scanning the source (smaller - $\mathcal{S}$) image and, for each pixel, determining where it goes in the target (larger - $\mathcal{T}$) image. But there are more pixels in the target than in the source, so this approach must lead to holes in the predicted image. The correct alternative is to scan the target image and, for each pixel, determine what value it should receive. This is known as *inverse warping*. In the example, the i, j'th location of $\mathcal{T}$ must get the value of the $i/1.27$, $j/1.27$'th location of $\mathcal{S}$. In fact, most values requred are at locations that are not integer values.

An *interpolate* is a function that (a) must have the same value as the original image at the original integer grid points (b) can be evaluated at any point rather than just the integer grid points. Write $\mathcal{I}(x, y)$ for an interpolate of an image $\mathcal{I}$.

The simplest interpolate is *nearest neighbors* – take the value at the integer point closest to the location whose value you want. Break ties by rounding up, so you would use the value at $2, 2$ if you wanted the value at $1.5, 1.5$. As Figure 3.1 shows, this strategy has problems – the upsampled image looks blocky.

3.1.2 An Interpolation Framework

There are many different ways to interpolate. Write $b(u, v)$ for a function that is one at the origin (so $b(0, 0) = 1$) and is zero at every other integer grid point. There

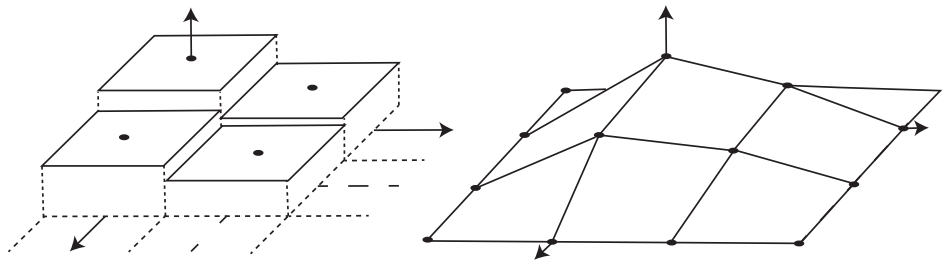

FIGURE 3.1: *On the* **left**, *a function interpolating a* 2×2 *image using nearest neighbors. The dashed lines pass through grid points, and the dotted lines are halfway between grid points. The function is zero away from the four boxes shown. Image values are shown as filled circles. On the* **right**, *a bilinear interpolate of the same data.*

are many such functions. For the moment, choose one. Then

$$\mathcal{I}(x,y) = \sum_{ij} \mathcal{I}_{ij} b(x-i, y-j)$$

will be an interpolate (check you know why).

For nearest neighbors, define

$$b_{nn}(u,v) = \begin{cases} 1 & \text{for } -1/2 \leq u < 1/2 \text{ and } -1/2 \leq u < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

This fitted function looks like a collection of boxes, and is not continuous (Figure 3.1; **exercises** ()). Most widely used is *bilinear interpolation*. For this, construct a function

$$b_{bi}(u,v) = \begin{cases} (1-u)(1-v) & \text{for } 0 < u \leq 1 \text{ and } 0 < v \leq 1 \\ (1+u)(1-v) & \text{for } -1 \leq u \leq 0 \text{ and } 0 < v \leq 1 \\ (1+u)(1+v) & \text{for } -1 \leq u \leq 0 \text{ and } -1 \leq v \leq 0 \\ (1-u)(1+v) & \text{for } 0 < u \leq 1 \text{ and } -1 \leq v \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

which is continuous, and again has the convenient property that $b_{bi}(0,0) = 1$, but $b_{bi} = 0$ for every other grid point (it looks a bit like a hat, Figure 3.1). The interpolate is

$$\mathcal{I}(x,y) = \sum_{i,j} \mathcal{I}_{ij} b_{bi}(x-i, y-j).$$

and it is a simple exercise to show that it has the properties required for an interpolate.

The construction above is a good way to think about interpolation (and can be used to build more complicated interpolates, **exercises**), but it is not the best way to evaluate a bilinear interpolate.

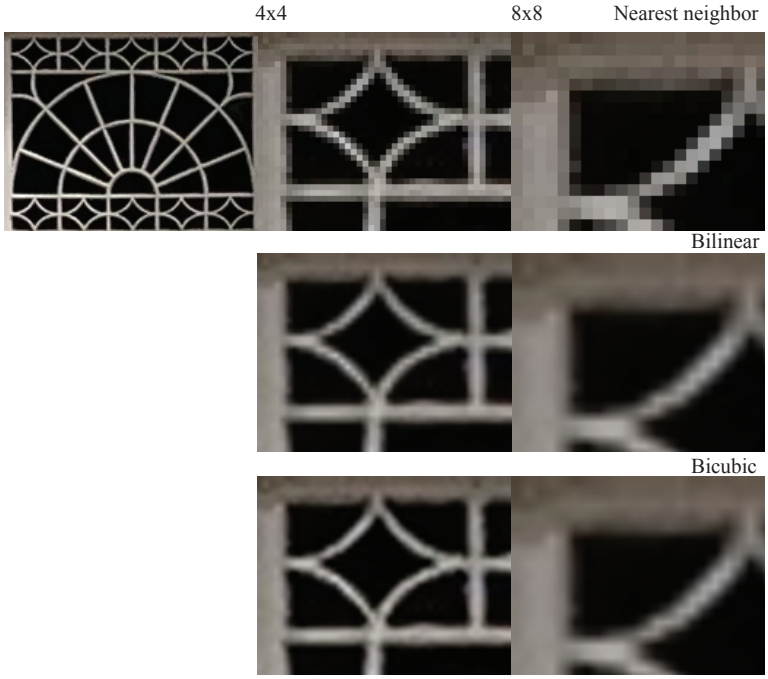

FIGURE 3.2: *The choice of interpolate when upsampling can make a real difference.* **Top left** *shows a detail from a picture. I have upsampled the image, then cropped the upsamples (showing the top left corner) and zoomed them so you can see the details.* **Center column** *shows a cropped 4x4 upsample using three different interpolation methods and* **right column** *shows 8x8 upsamples by various methods. Notice the significant blockiness in nearest neighbor interpolates (***top row***). Bilinear interpolates (***second row***) are much better, and bicubic interpolates (***third row***) are different to bilinear interpolates, but not a major improvement.* Image credit: *Figure shows my photograph of a facade in Stellenbosch.*

Procedure: 3.1 *Bilinear interpolation for an image*

To find a value for $\mathcal{I}(i+\delta, j+\epsilon)$, where i and j are integers; $0 < \delta < 1$; and $0 < \epsilon < 1$, use

$$\mathcal{I}(i+\delta, j+\epsilon) = \begin{bmatrix} \mathcal{I}_{ij}(1-\delta)(1-\epsilon)+ \\ \mathcal{I}_{i+1,j}(\delta)(1-\epsilon)+ \\ \mathcal{I}_{i,j+1}(1-\delta)(\epsilon)+ \\ \mathcal{I}_{i+1,j+1}(\delta)(\epsilon) \end{bmatrix}.$$

By a little manipulation, you can show that this procedure boils down to: predict a value for $\mathcal{I}(i+\delta, j)$ using a linear interpolate; predict a value for $\mathcal{I}(i+\delta, j+1)$ using a linear interpolate; now linearly interpolate between these two to

get a value for $\mathcal{I}(i+\delta, j+\epsilon)$. Modern hardware is particularly efficient at bilinear interpolation, and any reasonable software environment will be able to do this for you, likely very fast indeed.

The choice of interpolate can make a real difference to the quality of the result (Figure 3.2). More complicated interpolation procedures are possible. In *bicubic interpolation*, the interpolate is cubic in δ and ϵ and depends on other neighboring pixels (**exercises**). 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**). Constructing more complicated interpolates is straightforward but seldom worthwhile. Another application for interpolation is demosaicing: one could interpolate, and then sample the interpolating function. The interpolation procedures above need some minor adjustments because the unknown values are at grid points (details in **exercises**).

Remember this: *Upsampling increases the size of an image. Upsample by backward warping and interpolating. APIs offer the choice between three main interpolation techniques are: nearest neighbors (quick and blocky); bilinear interpolation (quick and much better); and bicubic interpolation (somewhat slower, slightly better). The default is often nearest neighbors.*

3.2 DOWNSAMPLING AND SMOOTHING

Reducing the size of an image by a fixed factor in each dimension is *downsampling*. Downsampling an image appears to be straightforward. Just like upsampling, the correct procedure is to scan the target image and, for each pixel, determine what value it should receive using interpolation. If you downsample by an integer amount (say, a factor of 2), you don't even need to interpolate. But downsampling an image like this can produce something that represents the image very poorly indeed. To see this, take an image whose dimensions are divisible by two (or four, or eight, and so on) then halve (or quarter, and so on) the size. To do this, you can simply take every second (fourth, eighth, and so on) pixel in each direction. Figure 3.3 shows effects that occur. Fine details can disappear or worse turn into coarse details.

3.2.1 Aliasing: Errors Caused by Downsampling

Figure 3.4 sketches a partial explanation for these effects. If there are too few samples, patterns in the image can fall between the samples. The general term for the kind of errors seen here is *aliasing*. In Chapter 22.3, we will be much more precise about these issues. As Figure 3.4 illustrates, the key question is how many samples you draw compared to how much detail there is in the function you are sampling. The figure suggests a rough explanation for what is going wrong when

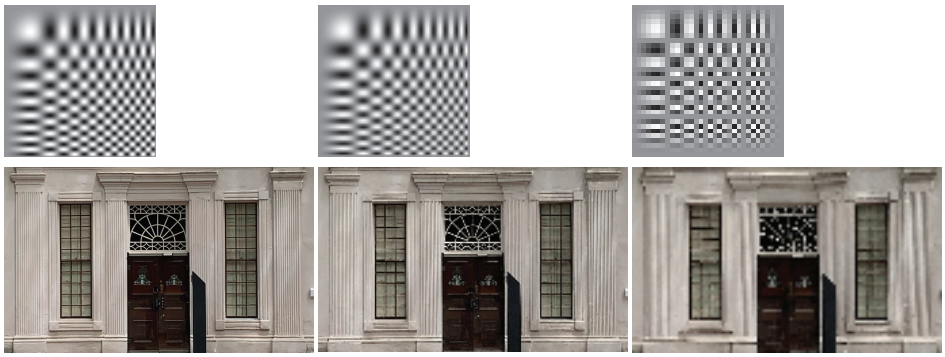

FIGURE 3.3: *Downsampling by just taking every k'th pixel in each direction reliably leads to problems. The* **top row** *shows some effects on a stylized image, and the* **bottom row** *shows results on a real photograph. The* **left** *image is the original;* **center** *is a downsampled image obtained by taking every 4'th pixel, then printing the image with larger pixels;* **right** *the original downsampled by taking every 8'th pixel. Notice how detail is lost in the resampling process. For the stylized image, some small boxes disappear (look on the edges of the image); others turn into large boxes (lower right quarter of the downsampled by 8 image). For the real image, notice the behavior of the details in the window above the door, and on either side of the door.* Image credit: *Figure shows my photograph of a facade in Stellenbosch.*

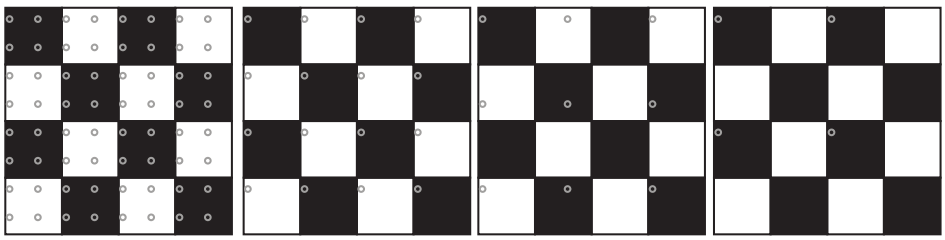

FIGURE 3.4: *A visualization of how sampling problems arise. The underlying image is a checkerboard, and the sample value is the value at the center of each of the small gray circles. The checkerboards on the* **left** *and* **center left** *illustrates a sampling procedure that appears to be successful. Whether it is or not depends on some details that we will deal with later – but the count of checks will be correct in each case. The sampling procedures shown on the* **center right** *and* **right** *are unequivocally unsuccessful. The samples suggest that there are fewer checks than there are in the original patterns. This illustrates two important phenomena: first, a successful sampling scheme must sample data often enough; and second, unsuccessful sampling schemes cause high-frequency information to appear as lower-frequency information. For example, on the* **right**, *the sampling procedure represents a checkerboard as a single dark region.*

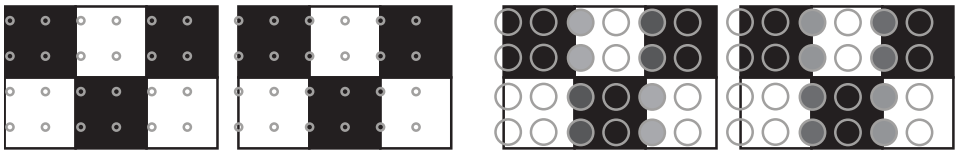

FIGURE 3.5: *Averaging can improve the representation produced by sampling.* **Left** *a small checkerboard pattern sampled on a grid. Each sample is the value at the center of the small gray circle. Though this will correctly represent the board in this configuration, a small shift in the board with respect to the samples will result in a dramatic shift in the representation (***center left***). **Center right** indicates what happens when you obtain a sample by averaging in a window. The gray level in the center of the window is my estimate of the average. Notice the edges of the checks are now blurred, but the change when the board moves (***far right***) is much less dramatic. The representation is somewhat improved.*

one subsamples an image. Samples might be poorly aligned with the underlying data, and so misrepresent it.

3.2.2 Smoothing

The downsampler needs to compute a value for the target image at i, j. This location corresponds to the location u, v in the source image (so, for example, in downsampling by two, $u = 2i$ and $v = 2j$). Call the point u, v the query point. Using the (possibly interpolated) value of the source image at this location may not be a particularly good idea, because there might be an important detail close to, but not at, the query point. An alternative is to use an average of the source image function about the query point.

In the easiest case, downsample an image by a factor of two. At every second pixel location in each direction, compute (say) an average of the $(2k+1) \times (2k+1)$ window of pixels centered at that location and report that average rather than the pixel value. A simple argument suggests that this should help: now the value of the pixel in the subsampled image is affected by its neighbors in the original image, so details that were missed by just taking every second pixel have a chance to appear in the result. Figure 3.5 is a picture of this argument.

3.2.3 Downsampling by Two with Gaussian Smoothing

As Figures 3.5 and 3.7 show, just averaging nearby values helps, because small structures that might otherwise have been missed will contribute to the downsampled image. But if the window is, say, a 5x5 window, structures that are two grid points away from the query point will have the same effect as structures that are one grid point away. As Figure 3.6 illustrated, this means that, for example, sharp edges will be misrepresented by the samples. This can be fixed by weighting the average, so that points near the sample point have a higher weight than points far from the sample point. The weighted average is formed as above, but the i, j'th pixel in $\mathcal{N}$ is now the weighted average of a $(2k-1) \times (2k-1)$ window of pixels in

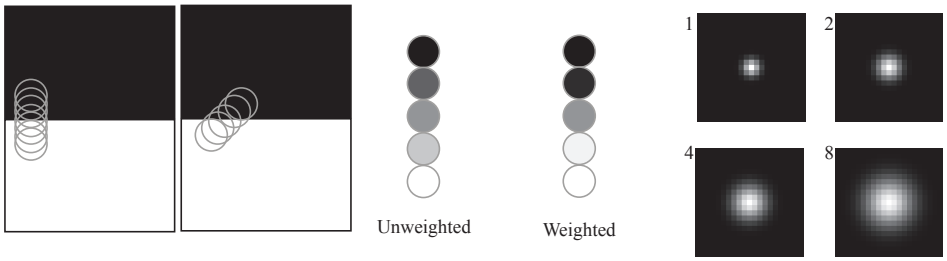

FIGURE 3.6: *Sampling with a weighted average makes significant changes in the representation. On the* **left**, *a horizontal edge in an image, to be sampled using averages. The circles show the support of the average at each sample for part of a column of samples. These overlap, so showing all sample points makes for a confusing figure.* **Center left** *shows one support per column for a set of rows of samples crossing the edge.* **Center** *shows the unweighted average within these supports; the gradient is fairly close to linear (it would be linear if the supports were squares). This slow gradient suggests that the edge is rather smoother than it really is.* **Center right** *the average within each circle is now weighted so that the center of the circle has a fairly large weight, and the weight decreases for pixels further from the center. Notice that now the representation has improved somewhat, as the gradient is sharper and is about in the right place.* **Far right** *shows four choices of weighting functions, each a gaussian of different σ (the number to the top left). Larger σ values mean that pixels far from the center contribute (and so edges are smoother); smaller σ values allow edges to be sharper, but may result in aliasing effects.*

$\mathcal{S}$, centered on i, j.

A traditional weighting scheme is given by a one parameter family of functions, derived from the normal distribution and widely called *gaussians*. The parameter σ is sometimes called the *scale* and more usually called the *sigma* of the weights. For downsampling by two, $\sigma = 1$ or $\sigma = 1.5$ are fair choices. In a $2k + 1 \times 2k + 1$ window, where the pixels are indexed starting at 1, the weights will be:

$$k_{ij} = \frac{1}{C} e^{-\left(\frac{(i-k-1)^2+(j-k-1)^2}{2\sigma^2}\right)}$$

where C is chosen so the weights sum to one. Figure 3.6 shows examples for four different values of σ, using a 21×21 window. In this figure, the largest value of the weights is always the same so that you can see the difference in falloff. If I had not scaled the weights like this, the windows would be mostly dark – bigger gaussians have much smaller individual weights, so that all the weights sum to one. All this yields the procedure in the box.

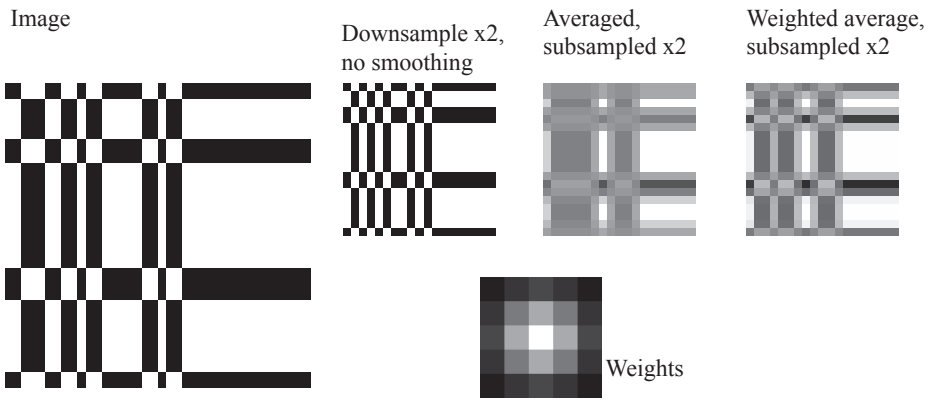

FIGURE 3.7: *The effects shown in Figure 3.6 are quite visible in images. On the* **far left**, *an image of stripes ranging from fine to coarse.* **Center left** *shows pure subsampling, with no smoothing (if you think no detail has been lost, look at the relative size of stripes).* **Center right** *shows a version of the image that has been subsampled by 2, but now the value of each sample is an average within a 5x5 window centered on the relevant pixel. Notice how the unweighted average has caused multiple lines to merge into a gray bar, and the relatively "slow" gradient of the lines, which is most obvious on the horizontal lines.* **Far right**, *the average in the sample is weighted with the set of weights show on the* **bottom right** *(these weights have been rescaled so the largest weight is light). Notice how some – though not all – of the vertical lines on the left have been resolved, and the faster gradient at the top and bottom of the horizontal lines.*

Procedure: 3.2 *Downsampling by Two with Gaussian Smoothing*

Given a source image $\mathcal{S}$, size $M \times N$, construct a target image $\mathcal{T}$, size $\text{floor}(M/2) \times \text{floor}(N/2)$. Adopt the convention that for u or v out of range, $S_{uv} = 0$. Choose k (likely 3 or 4) and σ (likely 1 or 1.5). Construct a Gaussian kernel $\mathcal{G}$ using these parameters. Now for each $1 \leq i \leq \text{floor}(M/2)$, $1 \leq j \leq \text{floor}(N/2)$, set

$$T_{ij} = \sum_{s=-k}^{s=k} \left[\sum_{t=-k}^{t=k} \left[S_{(2i+s),(2j+t)} G_{(s+k+1),(t+k+1)} \right] \right]$$

Figure 3.7 shows the considerable improvement in subsampling that can result from using a set of weights. Figure 3.8 shows an annoying feature of using unweighted averages to smooth. Unexpected fine details can appear, an effect known as *ringing*. Gaussian smoothing suppresses ringing rather well. Section **??** explains where ringing comes from.

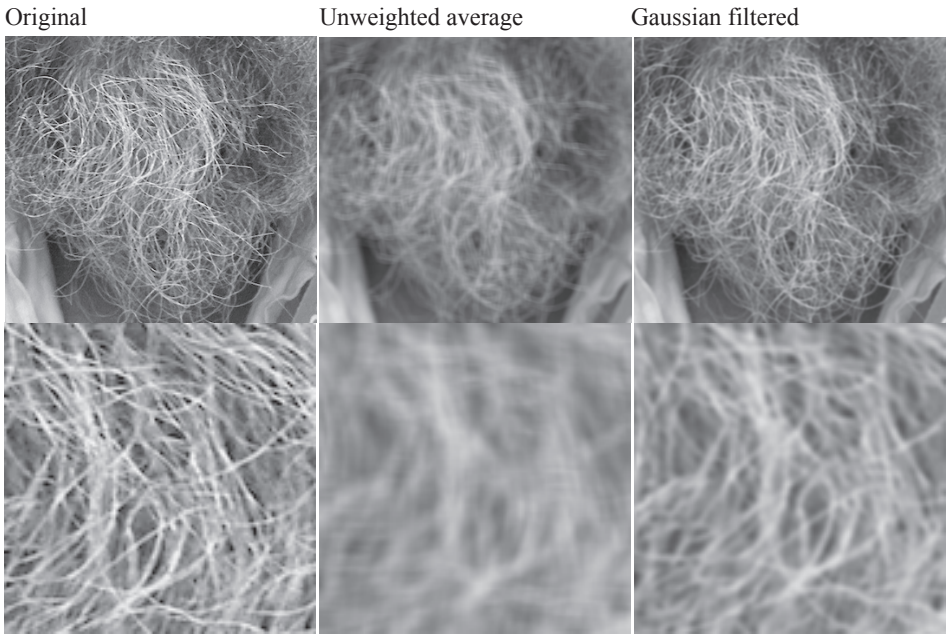

FIGURE 3.8: *The effects shown in Figure 3.7 are quite visible in real images. The* **top row** *shows:* **left** *a luxuriant beard;* **center**, *that beard smoothed with an unweighted average; and* **right**, *that beard smoothed with a Gaussian. The* **bottom row** *shows details of those images. Notice the narrow dark stripes that have appeared in the version smoothed with an unweighted average. This version appears as if it is both more smoothed than the Gaussian smoothed version, and as if it has gained some very fine details (the stripes) out of the smoothing procedure. The unweighted smoother is ringing.*

3.2.4 Downsampling by a Small Factor with Gaussian Smoothing

You wish to downsample by a small factor, so taking an $M \times N$ image to a $R \times S$ image where $2 > M/R > 1$, and N/S is very close to M/R. Doing so requires smoothing, and it is sensible to use Gaussian weights with a small σ (between 1 and 2, depending on the application). But doing so also requires interpolation, as the downsampling will require values that *aren't* on the source grid. Interpolation should strike you as likely to interact inefficiently with the weighting process. A straightforward procedure yields a pre-smoothed version of the original image, which you can then downsample using backward warping and interpolation.

Procedure: 3.3 *Downsampling an image by a small factor*

Take the source image $\mathcal{S}$, and form a new image $\mathcal{N}$ from that source. The i, j'th pixel in $\mathcal{N}$ is now a weighted average of a $(2k+1) \times (2k+1)$ window of pixels in $\mathcal{S}$, centered on i, j. Organize the weights into a small array – the *mask*, which you could obtain by evaluating the Gaussian, as above – and form a new image $\mathcal{N}$ from the original image and the mask, using the rule

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

This expression is the root of all sorts of interesting ideas (Chapter 5). There are some problems when i or j or u or v are too big or too small. Deal with these by asserting that $\mathcal{I}$ and $\mathcal{W}$ are zero for locations outside the range. Evaluate $\mathcal{N}$ on an $M \times N$ grid. Now downsample using backward warping and interpolation.

3.2.5 The Gaussian Pyramid

Now consider downsampling by a large factor. You *could* (but shouldn't) smooth with a gaussian with large σ, then downsample. This is not a good idea, because the support of the gaussian is infinite, meaning that working with a $2k+1 \times 2k+1$ window involves some truncation. As σ gets bigger, k will need to get bigger to keep this truncation reasonable, so the smoothing process will be expensive. The more efficient alternative is to smooth, downsample by two, then smooth the result and downsample *that* by two and so on, until the image size is only slightly larger than what you want. Then downsample that by a small factor.

A useful construction follows. In some applications (Section 15.1.3 and Chapter 22.3), it will be useful to have versions of an image downsampled by different factors. A *gaussian pyramid* is a collection of smoothed and downsampled representations of an image. Downsampling is usually by a factor of either two or the square root of two (so two rounds of downsampling halves the edge length of the image). The name comes from a visual analogy. If we were to stack the layers on top of each other, an inverted pyramid would result. The smallest image is the most heavily smoothed. The layers are often referred to as *coarse scale* versions of the image that forms the top layer.

Procedure: 3.4 *Building a Gaussian pyramid*

Write D_σ for the operation that smoothes an image with a gaussian of scale σ then downsamples it; U for the operation that upsamples an image; and G_k for the k'th layer of a gaussian pyramid. This notation suppresses by how much the image is downsampled, and what particular interpolation you use in upsampling, because these aren't important here. An N level gaussian pyramid then can be written as:

$$\begin{aligned} G_1 &= \mathcal{I} \\ \dots & \\ G_k &= D_\sigma(G_{k-1}) \\ \dots & \\ G_N &= D_\sigma(G_{N-1}). \end{aligned}$$

3.2.6 The Laplacian Pyramid

One thing should trouble you about the gaussian pyramid of 3.2.5. There is redundant information in the representation. Although some information is lost in downsampling and then upsampling, it isn't that much, because $U(G_k)$ looks rather a lot like G_{k-1}. This suggests using a representation where only the residual $G_k - U(G_{k+1})$ is preserved.

Procedure: 3.5 *Building a Laplacian pyramid*

Write D_σ for the operation that smoothes an image with a gaussian of scale σ then downsamples it; U for the operation that upsamples an image; and G_k for the k'th layer of a gaussian pyramid. An N level *laplacian pyramid* can be written as:

$$\begin{aligned} L_1 &= G_1 - U(D_\sigma(G_1)) \\ \dots & \\ L_k &= G_k - U(D_\sigma(G_k)) \\ \dots & \\ L_N &= G_N. \end{aligned}$$

Figure 3.10 compares Gaussian and Laplacian pyramids. Each layer of a Laplacian pyramid can be thought of as a representation of image information at a particular scale. If a pattern in the image is too small for a layer, then it will have been smoothed out; if it is too large, there will be little difference between G_k and $U(D_\sigma(G_k))$ and it will be suppressed by the subtraction.

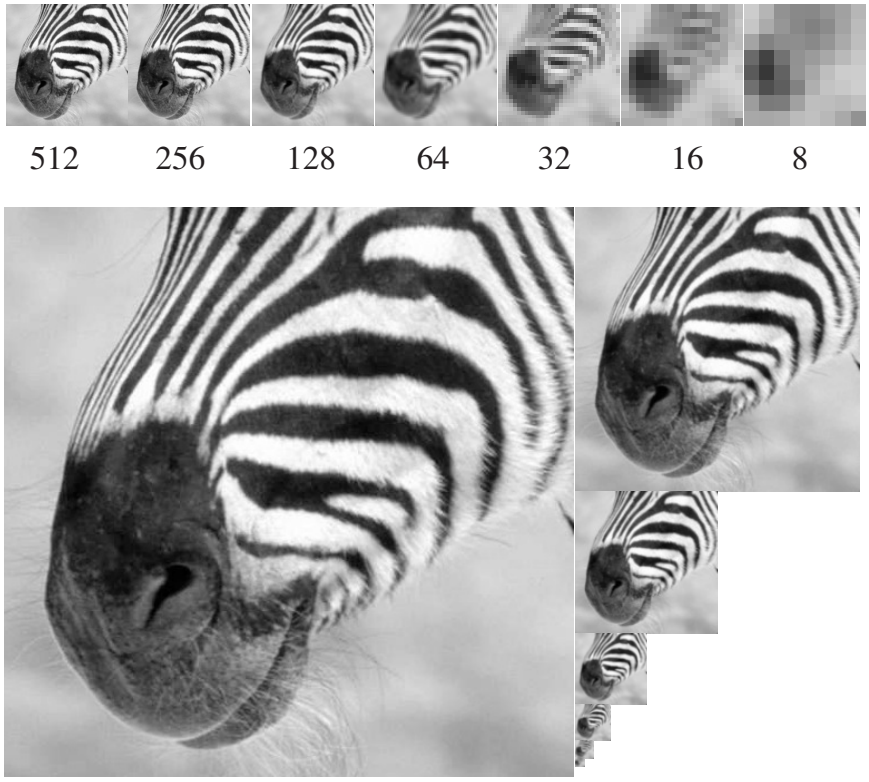

FIGURE 3.9: *A Gaussian pyramid of images running from 512x512 to 8x8. On the top row, I have shown each image at the same size (so that some have bigger pixels than others), and the lower part of the figure shows the images to scale. Notice that an 8x8 pixel block at the finest scale might contain a few hairs; at a coarser scale, it might contain an entire stripe; and at the coarsest scale, it contains the animal's muzzle.*

3.2.7 Reconstruction from Pyramids

It is easy to get an image back from a Gaussian pyramid (take the biggest layer). It is easy to get a gaussian pyramid from a laplacian pyramid, too, because $G_N = D_\sigma(G_{N-1})$.

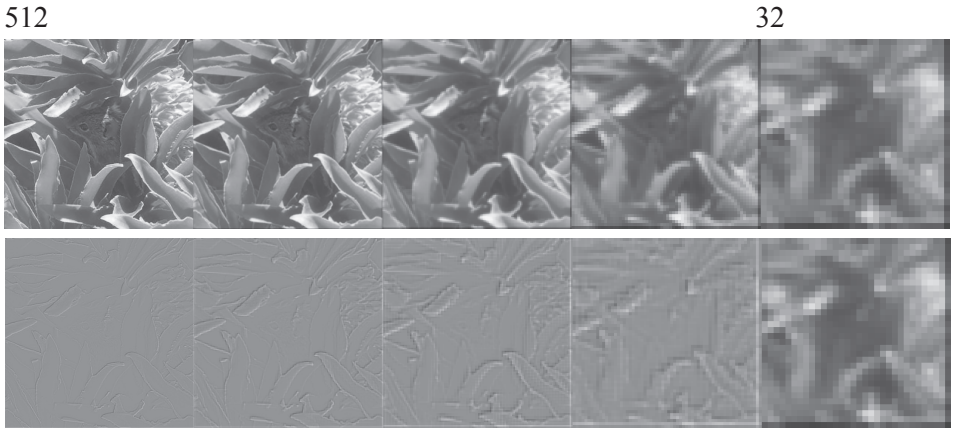

FIGURE 3.10: *A comparison of Gaussian and Laplacian pyramids.* **Top row** *shows a five layer Gaussian pyramid, and* **bottom row** *a Laplacian pyramid derived from it. Each image has been shown at the same size (so the pixels for the* 32×32 *layers are larger). The image is on a scale 0-1 (dark-light). All but the coarsest layer in the Laplacian pyramid have been shown on a scale where mid-gray is 0.5, negative numbers are dark, and positive numbers are light.* Image credit: *Figure shows my photograph of a striped mouse.*

Procedure: 3.6 *Recovering an Image from a Laplacian pyramid*

Write

$$\begin{aligned} R_1 &= w(1)L_1 + R_2 \\ &\dots \\ R_k &= w(k)L_k + R_{k+1} \dots \\ R_N &= L_N = G_N. \end{aligned}$$

If all the weights are 1, then $R_1 = \mathcal{I}$.

You can emphasize or de-emphasize some effects in the image by upweighting or downweighting the relevant scale by choosing $w(k)$. Using strongly different weights for different scales doesn't usually end well. For the example of Figure 3.11, I used weights obtained by: (a) choosing some largest scale k_x (in this case, $k_x = 3$); (b) choosing a weight α then (c) forming

$$w(k) = \left(1 + \left[\alpha \frac{\max(k_x - k, 0)}{k_x}\right]\right).$$

Figure 3.11 shows how various choices of α either sharpen or smooth the image.

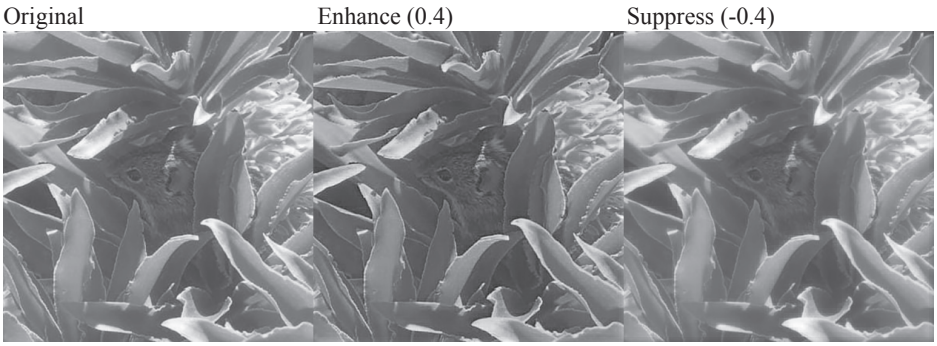

FIGURE 3.11: *Images can be reconstructed from Laplacian pyramids, and weighting components can emphasize or smooth edges. The Laplacian pyramid of Figure 3.10, reconstructed into an image using the method of Section 3.2.6, with* $\alpha = 0$ *(***left***; original image);* $\alpha = 0.4$ *(***center***; emphasizes edges); and* $\alpha = -0.4$ *(***right***; smoothes edges).*

Remember this: *Downsample by smoothing, backwards warping and interpolating. Downsampling without smoothing can create significant effects that weren't in the original. Always smooth when downsampling, and use a Gaussian unless you have very good reason not to. Gaussian pyramids represent an image at multiple scales. Laplacian pyramids contain less redundant information.*

3.3 YOU SHOULD

3.3.1 remember these facts:

3.3.2 remember these procedures:

3.3.3 be able to:

- Upsample an image without leaving holes using at least nearest neighbors or bilinear interpolation.

- Downsample an image by a small factor using Gaussian smoothing and interpolation.
- Construct a Gaussian pyramid.
- Construct a Laplacian pyramid.

EXERCISES

QUICK CHECKS

3.1. Is a nearest neighbor interpolate continuous?

3.2. Is a bilinear interpolate continuous?

3.3. Define $b_{bi}(u, v) = f(u)f(v)$ where

$$f(u) = \begin{cases} \frac{(u-1)(u-2)(u+1)(u+2)}{4} & \text{for } -2 \leq u \leq 2 \\ 0 & \text{otherwise} \end{cases}.$$

Check that this an interpolate. Is this a useful interpolate? Why not?

3.4. Define $b_{sinc}(u, v) = f(u)f(v)$ where

$$f(u) = \frac{\sin \pi u}{u}.$$

Check that this an interpolate. Why is this not a useful interpolate for reconstructing images? Remember that

$$\lim_{x \to 0} \frac{\sin x}{x} = 1.$$

3.5. Can the effects of Figure 3.3 be controlled by interpolating before downsampling? Why?

3.6. Recall the weights for Gaussian smoothing take the form

$$k_{ij} = \frac{1}{C} e^{-\left(\frac{(i-k-1)^2+(j-k-1)^2}{2\sigma^2}\right)}.$$

Assume $k > 5$. What do you expect will happen if you use

$$k_{ij} = \frac{1}{C} e^{-\left(\frac{(i-k-2)^2+(j-k-2)^2}{2\sigma^2}\right)}$$

instead?

3.7. Imagine you decide to store each intensity image as a Gaussian pyramid, downsampling by 2. What is the worst (reasonable!) case for how much more space it will take?

3.8. Imagine you decide to store each intensity image as a Laplacian pyramid, downsampling by 2. Do you expect the pyramid to take a lot more space than the original image? Why?

3.9. You wish to downsample an image by a factor of 9 in each direction. How should you do this efficiently?

3.10. The coarsest scale images of Figure 3.10 have visible dark bars on some edges. Where do these come from?

LONGER PROBLEMS

3.11. Write

$$\mathcal{I}(x, y) = \sum_{ij} \mathcal{I}_{ij} b(x - i, y - j)$$

for an interpolate.

(a) Check that this is an interpolate if $b(u, v)$ is one at the origin (so $b(0, 0) = 1$) and is zero at every other integer grid point.

(b) Define

$$b_{nn}(u,v) = \begin{cases} 1 & \text{for } -1/2 \leq u < 1/2 \text{ and } -1/2 \leq u < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

Check that this produces a nearest neighbors interpolate.

(c) Define

$$b_{bi}(u,v) = \begin{cases} (1-u)(1-v) & \text{for } 0 < u \leq 1 \text{ and } 0 < v \leq 1 \\ (1+u)(1-v) & \text{for } -1 \leq u \leq 0 \text{ and } 0 < v \leq 1 \\ (1+u)(1+v) & \text{for } -1 \leq u \leq 0 \text{ and } -1 \leq v \leq 0 \\ (1-u)(1+v) & \text{for } 0 < u \leq 1 \text{ and } -1 \leq v \leq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Check that this produces a bilinear interpolate.

(d) Show that using

$$\mathcal{I}(i+\delta, j+\epsilon) = \begin{bmatrix} \mathcal{I}_{ij}(1-\delta)(1-\epsilon) + \\ \mathcal{I}_{i+1,j}(\delta)(1-\epsilon) + \\ \mathcal{I}_{i,j+1}(1-\delta)(\epsilon) + \\ \mathcal{I}_{i+1,j+1}(\delta)(\epsilon) \end{bmatrix}.$$

to find a value for $\mathcal{I}(i+\delta, j+\epsilon)$ yields a bilinear interpolate. Here i and j are integers; $0 < \delta < 1$; and $0 < \epsilon < 1$.

(e) Show that using

$$\mathcal{I}(i+\delta, j+\epsilon) = \begin{bmatrix} \mathcal{I}_{ij}(1-\delta)(1-\epsilon) + \\ \mathcal{I}_{i+1,j}(\delta)(1-\epsilon) + \\ \mathcal{I}_{i,j+1}(1-\delta)(\epsilon) + \\ \mathcal{I}_{i+1,j+1}(\delta)(\epsilon) \end{bmatrix}.$$

to predict a bilinear interpolate boils down to: predict a value for $\mathcal{I}(i+\delta, j)$ using a linear interpolate; predict a value for $\mathcal{I}(i+\delta, j+1)$ using a linear interpolate; now linearly interpolate between these two to get a value for $\mathcal{I}(i+\delta, j+\epsilon)$.

(f) Show that a bilinear interpolate is continuous everywhere, but has discontinuities in first derivative.

(g) Show that a local maximum of a bilinear interpolate always occurs at a grid point.

3.12. A simple model for interpolation of mosaiced data has a known value at points $(2i + j \text{mod} 2, j)$ for i and j integer points. Recall $j \text{mod} 2$ is 0 if j is even and 1 if it is odd.

(a) Show that the known values lie on the grid sketched in Figure **??**.

(b) Recall the basis function for a bilinear interpolate

$$b_{bi}(u,v) = \begin{cases} (1-u)(1-v) & \text{for } 0 < u \leq 1 \text{ and } 0 < v \leq 1 \\ u(1-v) & \text{for } -1 \leq u \leq 0 \text{ and } 0 < v \leq 1 \\ uv & \text{for } -1 \leq u \leq 0 \text{ and } -1 \leq v \leq 0 \\ (1-u)v & \text{for } 0 < u \leq 1 \text{ and } -1 \leq v \leq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Show that

$$\mathcal{I}(x,y) = \sum_{i,j} \mathcal{I}_{(2i+j \text{mod} 2),j} b_{bi}(x - (2i + j \text{mod} 2), y - j).$$

is a bilinear interpolate of the mosaic.

3.13. Write

$$W(x) = \begin{cases} (2+a)|x|^3 - (3+a)|x|^2 + 1 & \text{for } |x| \leq 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2 \\ 0 & \text{otherwise} \end{cases}$$

(a) Show that $W(1) = 1$, and that $W(x) = 0$ for x any integer other than 1.
(b) Show that W is continuous everywhere.
(c) Show that W has continuous derivative everywhere.
(d) Show that $\sum_{ij} I_{ij} b_{\text{bicub}}(u, v)$ is a bicubic interpolate of an image, where $b_{\text{bicub}} = W(u)W(v)$.
(e) Recall you can use

$$\mathcal{I}(i+\delta, j+\epsilon) = \begin{bmatrix} \mathcal{I}_{ij}(1-\delta)(1-\epsilon) + \\ \mathcal{I}_{i+1,j}(\delta)(1-\epsilon) + \\ \mathcal{I}_{i,j+1}(1-\delta)(\epsilon) + \\ \mathcal{I}_{i+1,j+1}(\delta)(\epsilon) \end{bmatrix}.$$

to find a value for $\mathcal{I}(i+\delta, j+\epsilon)$ using a bilinear interpolate. Here i and j are integers; $0 < \delta < 1$; and $0 < \epsilon < 1$. What function of δ and ϵ would be used for the bicubic interpolate $\sum_{ij} I_{ij} b_{\text{bicub}}(u, v)$, where $b_{\text{bicub}} = W(u)W(v)$.
(f) What is a good choice of a? (you should write a program and do some experiments; there are ups and downs for each available choice, but $a = -0.5$ is popular).
(g) Write $\sum_{ij} I_{ij} b_{\text{bicub}}(u, v)$ for a bicubic interpolate of an image, where $b_{\text{bicub}} = W(u)W(v)$. Show the local maxima do not always occur at gridpoints.

PROGRAMMING EXERCISES

3.14. **(a)** Find a natural image that shows strong aliasing effects when downsampled by 2 without any smoothing. A good place to start is an image with many thin, high contrast lines like the image of Figure 3.3.
(b) Collect a set of at least 300 natural images from the internet (for example, you could use the Berkeley Segmentation Dataset at `https://github.com/BIDS/BSDS500`). Downsample each by 2 without smoothing, then upsample the result. Compute the mean squared difference between the original and the upsampled downsampled image.
(c) Find the 10% of images with the largest difference. Can you characterize them qualitatively?
(d) If you downsample by 4 without smoothing, then upsample by 4, does the set of images with largest difference change? Equivalently, how well does aliasing error for downsampling by 2 predict aliasing error for downsampling by 4?
(e) Use the procedure of the previous subexercise to determine whether aliasing can cause pixels to change color, and how substantial these changes can be.

3.15. Collect a set of at least 500 natural images from the internet (for example, you could use the Berkeley Segmentation Dataset at `https://github.com/BIDS/`

BSDS500). This exercise investigates how well smoothing works to suppress aliasing when you downsample by 2.

(a) Split this dataset into a training subset and a test subset. Put 80% of the images into the training set and 20% of the images into the test set. For Gaussian filters with $\sigma \in \{0.5, 1, 1.5, 2\}$ with size in $\{3, 5, 7, 9\}$, smooth the image using the filter (one from a total of 16), downsample it by 2, then upsample the result to the original size. Compute the mean squared difference between the original and the upsampled downsampled image, averaged over the training subset. Use this error to choose the "best" filter.

(b) Now use the procedure of the previous subexercise to choose the "best" filter on the test set. Is this the same filter as the "best" filter on the training set?

3.16. Gaussian pyramids: Write a function that produces a Gaussian pyramid from an image.

(a) Start with the easy case, where the image size is $2^k \times 2^k$ and each layer is half the size on edge of the previous layer.

(b) Extend your code to deal with the case where the image size is $2^k \times 2^k$ and each layer is $2^(-1/m)$ the size on edge of the previous layer. Assume $0 < m < 4$.

(c) Extend your code to deal with images of arbitrary dimension.

3.17. Laplacian pyramids: Write a function that produces a Laplacian pyramid from an image. For each case, verify that you can: (a) recover a Gaussian pyramid from your Laplacian pyramid; (b) recover the original image from your Laplacian pyramid.

(a) Start with the easy case, where the image size is $2^k \times 2^k$ and each layer is half the size on edge of the previous layer.

(b) Extend your code to deal with the case where the image size is $2^k \times 2^k$ and each layer is $2^(-1/m)$ the size on edge of the previous layer. Assume $0 < m < 4$. You need to be careful with upsampling here to ensure that an upsampled layer is the same size as the layer that was downsampled.

(c) Extend your code to deal with images of arbitrary dimension. You need to be careful with upsampling here to ensure that an upsampled layer is the same size as the layer that was downsampled.

(d) In which cases should you be able to recover the original image exactly?