CHAPTER 9

# Denoising Images using Distant Pixels

Points:

TODO: Figure images have patches that match quite well

sec: you can inpaint missing pixels by matching;

matches might come from far away;

TODO: Figure showing incremental inpainting of text

TODO: The size of the window matters for incremental inpainting TODO: The order of the window matters for incremental inpainting

You can inpaint with patches as well as pixels, though you have to worry about boundaries

you can use these procedures to synthesize texture (the hole is on the outside of the image)

Sec: this yields non-local means; and also the bilateral filter; using SSD might be a bit shakey for denoising; matching norm can have interesting effects.

sec: patches match across images as well; this introduces huge issues of scale

You can denoise by smoothing because pixels tend to be like their neighbors. The autoencoders I have described denoise by computing a description of a neighborhood of pixels – usually called an *image patch* or *patch* – that is robust to noise, then reconstructing the patch from this description. It follows that patches tend to have quite stylized appearance – most small arrays are *not* patches. This chapter introduces a new and very important property of images: similar patches tend to appear rather often in an image. This property is extremely powerful, because it means that if a patch at some location is degraded, there is very likely another version of that patch at another location that is not. In turn, you can look far away from the pixel you are denoising to find useful information about what it looks like.

Figure 9.1 illustrates this essential property. I selected five locations in an image at random. I took the $5 \times 5$ patches centered at each location, then found the top 20 matching patches in the image. The best matches are very good. Some very good matches are to patches that are far away from the original location. It should be clear that the size of the patch (very often, referred to as the *scale*) you try to match has strong effects on this property. Experience of images will tell you that, if the patch is $1 \times 1$, there will be a large number of matches (which is slightly surprising **exercises** ). If the patch is very large, there must be few matches. But patches of moderate scale find many matches. Figure **??** shows matches varying by scale.

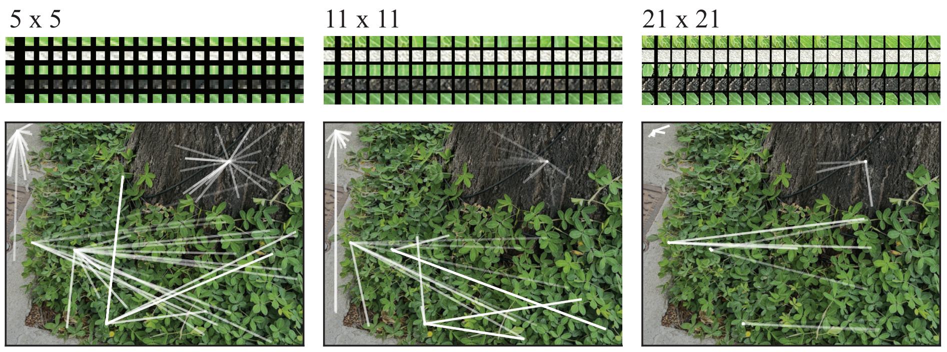

FIGURE 9.1: *Images are made up of patches chosen from quite a small vocabulary, and so any one patch in an image tends to match a number of other patches quite well.* **Top row:** *shows five image patches in detail, selected from the image below at random, and their matching patches for three different patch sizes. The* **left** *column in each shows the patch, and the* **other columns** *show 20 matching patches, found in the image, in order of SSD distance (smallest and so best matching to the left). Notice that there are many patches that are very like a given patch.* **Bottom row** *shows the matches on top of the image. The center of each of the fans of line segments is the query patch (letters key the patch to the details), and each line segment joins a patch to a matching patch. The lines are brighter for small distances, and fainter for large distances. A thicker line occurs when two or more matching patches are close to one another. Notice how matching patches can be quite far away from the query patch (long lines) and how some patches repeat often (many bright lines) whereas others have few matches (many faint lines). Notice also that smaller patches have more widely distributed matches (broad fan, few thick lines) and larger patches tend to have matches that are close together.* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

## 9.1 INPAINTING MISSING PIXELS BY MATCHING

### 9.1.1 Replacing Knocked-out Pixels

Imagine you have an image where some pixels have been set to zero (knocked out) but all others are reliable. The problem of dealing with knocked out pixels is known as *inpainting*. Because real pixels are never zero, you know which pixels are noise. For the moment, assume that the noise pixels are scattered and are selected randomly (how does not really matter), and are moderately rare. Denoising this image requires estimating the true value of the knocked out pixels. You should immediately think of applying a median filter (Section **??**), which certainly applies to this example.

Here is an alternative strategy. Look at the patch around a noise pixel (the target patch). There are no other noise pixels in this patch, at least for the moment. Images are quite repetitive in structure, meaning that there is likely another patch in the image that matches this one. Now find a pool of patches that match the target well enough. The center pixel in each of these patches is a good estimate

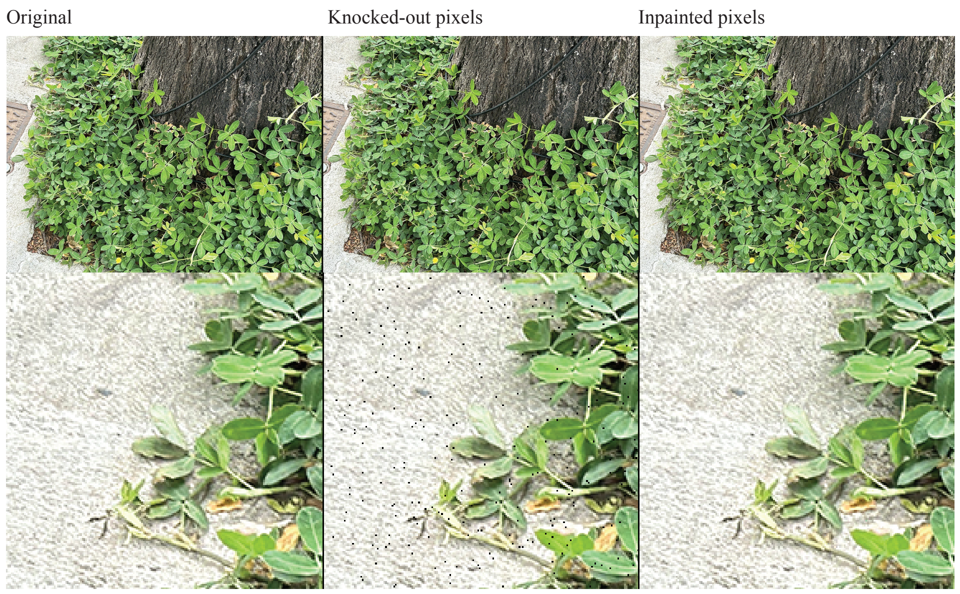

FIGURE 9.2: *Inpainting occasional missing pixels by matching patches is very successful.* **Top row** *shows images;* **bottom row** *shows detail. For reference, the original image is on the* **left**; *just under* 1% *of the pixels in this image have been set to zero (locations chosen uniformly at random) to produce the* **center** *image; the image on the right has been reconstructed by finding the closest* $5 \times 5$ *patch that matches the patch surrounding the knocked out pixel (but doesn't have a knocked out pixel in it), then replacing the knocked out pixel with the center of the patch. Look for problems by finding a black pixel in the center detail image that is replaced by an implausible pixel in the right image (a fruitless search!).* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

of the value of the knocked out pixel. The knocked out pixel can then be replaced either by summarizing these center pixels using a mean, or choosing randomly among them. Alternatively, you could take the center pixel from the best matching patch.

The details are straightforward. Compute the goodness of the match with the sum of squared differences. There should always be at least one matching patch, otherwise you can't obtain the value of the missing pixel. This means the pool of matches that are good enough should always contain the best match. Build this pool from the best match, together with the top $k$ matches that are better than some threshold. Leave out the center pixel when computing the SSD, and match only to patches without a missing pixel, otherwise you might replace a knocked out pixel with another one. Figure 9.2 shows an example. This procedure works for blocks of pixels that have been knocked out as well, with minimal changes. Figure 9.4 shows an example where $3 \times 3$ blocks of pixels have been knocked out.

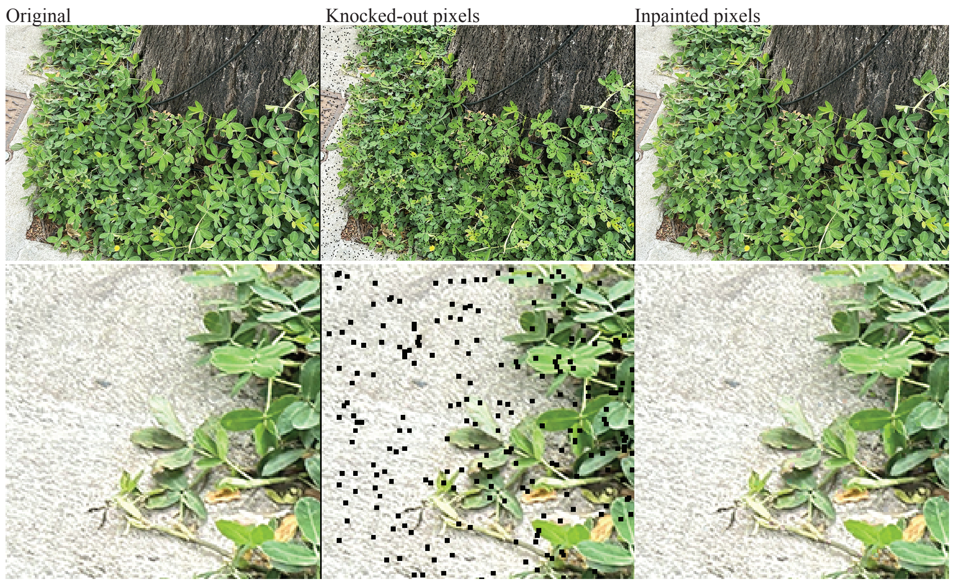

FIGURE 9.3: *Inpainting still works when a lot of pixels have been knocked out, as long as you are careful about how you match.* **Top row** *shows images;* **bottom row** *shows detail. For reference, the original image is on the* **left***; just under* 6% *of the pixels in this image have been set to zero. The locations were chosen uniformly at random, and pixels were knocked out in* $3 \times 3$ *blocks to produce the* **center** *image; the image on the right has been reconstructed by finding the closest* $7 \times 7$ *patch that matches the patch surrounding the knocked out block (but doesn't have a knocked out pixel in it), then replacing the knocked out block with the center of the patch. Look for problems by finding a black block in the center detail image that is replaced by an implausible pixel in the right image (a fruitless search!).* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

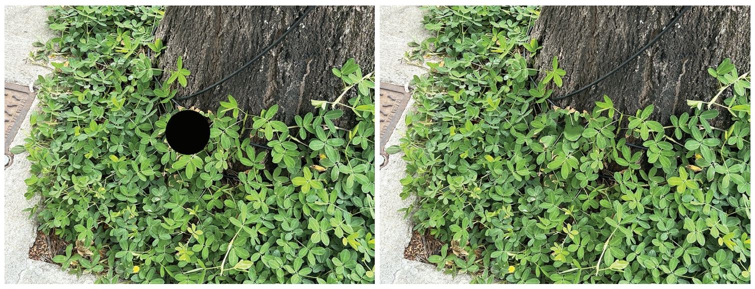

FIGURE 9.4: *Incremental inpainting can fill in large holes. On the* **top left***, an image with a large hole in it;* **top right** *shows the inpainted image, using* $11 \times 11$ *patches and a radial order. Alternatives appear in Figure* **??**. Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

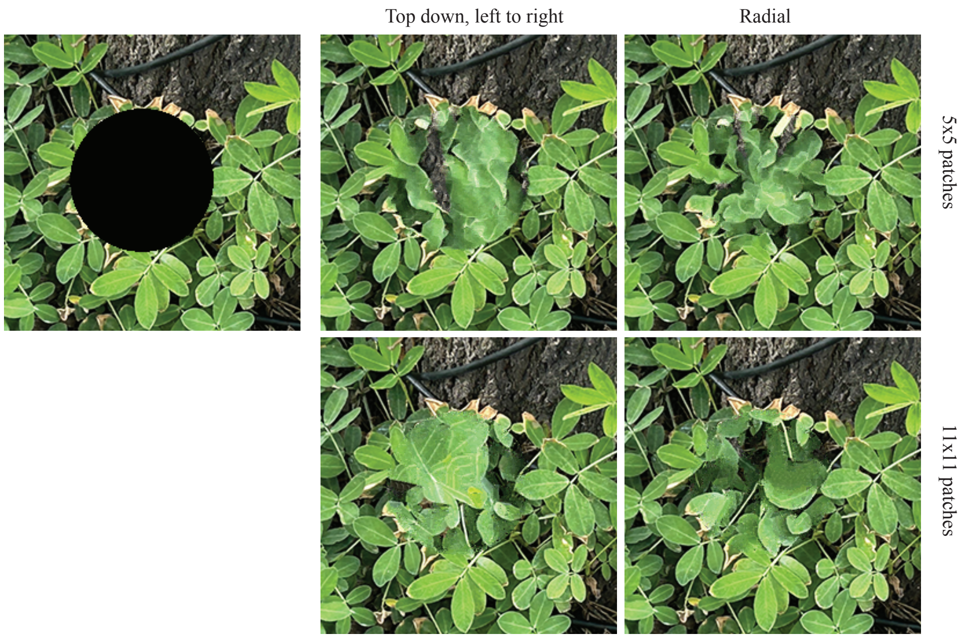

FIGURE 9.5: *The size of the patch you use in matching and the order in which you inpaint pixels have significant effects on the results of incremental inpainting. This figure shows a detail of Figure 9.4, with the blob inpainted in four different ways: patch size differs from row to row, order from column to column. Details in main text.* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

### 9.1.2 Incremental Inpainting

Now imagine that the process that knocks out pixels doesn't just choose pixels at random, but has some some kind of spatial structure. For example, you might have an image with writing on it, and want to replace the writing. Alternatively, the image might have one more more large holes in it.

The pixel inpainting procedure above will work, but some details need to change. When isolated pixels are knocked out, you expect that the patch around the pixel is known. If the image has a large hole in it, this no longer applies. Fixing a pixel requires you have at least some known pixel values close to it. Choose such a pixel, and match the patch using the known pixels only. You can do this with a mask that zeros the contribution of knocked out pixels to the SSD. This produces a pool of matches. Now estimate the value of the pixel using this pool. For the moment, choose the center of the best match. Place this value in the image, and you now have an image with a slightly smaller hole in it, so you should be able to find more candidate pixels for replacing. In this *incremental reconstruction* approach, the order in which you visit pixels and the size of the patch becomes important and can quite strongly affect the result.

As Figure 9.4 shows, really quite large holes in images can be fixed quite satisfactorily in this way. The scale of the patch and the order of inpainting matters

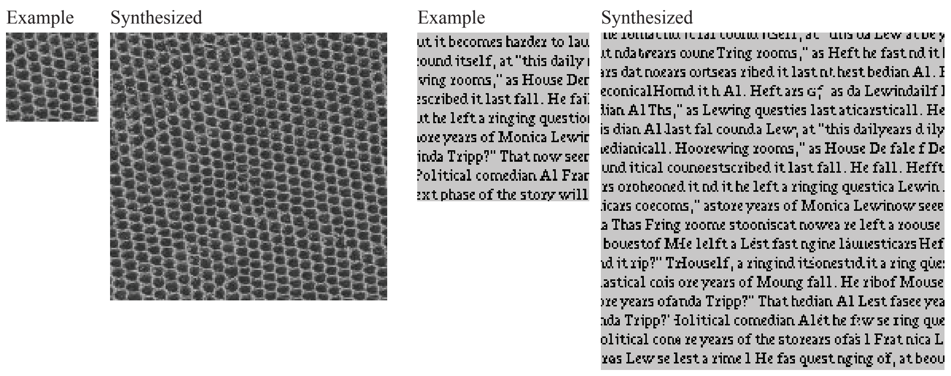

FIGURE 9.6: *Synthesizing a texture image from a small example using incremental reconstruction, and works for textures where the repeated structure isn't obvious. The case on the* **left** *is a texture that is quite periodic, but with random deformations. Notice how realistic the synthesized texture seems to be, and look for repeated cells (they are hard to find). The case on the* **right** *shows a small piece of text from a document that was well known in 1999, together with a synthesized image expanding that example. Text is hard, because although it does have structure (lines), it isn't periodic. Notice how the synthesized text is almost readable.* Image credit: *Elements of Figure 3 from "Texture Synthesis by Non-parametric Sampling", IEEE International Conference on Computer Vision, Corfu, Greece, September 1999.* **No permission yet**

a lot (Figure 9.5). Top down, left to right order means the pixels are ordered by vertical coordinate, then by horizontal coordinate. This tends to produce somewhat disordered inpaintings, because some pixels with all-inpainted neighbors are inpainted before others with known neighbors. Radial order means that pixels on the edge of the blob are filled in first, so pixels with more known neighbors are inpainted first. This tends to preserve structure. Notice also the effect of the scale of the patch. Matching larger patches tends to reproduce more long-scale structure, as you should expect. For example, the regions inpainted with larger patches appear to have leaves and stems in them.

### 9.1.3 Texture Synthesis by Incremental Reconstruction

Imagine you have a small texture image you would like to make larger, where the larger image should have the same texture as the original part. There are a variety of application reasons to do this. For example, you might want to apply a texture to a computer graphics model. Just tiling the texture won't work. The patches may not join up properly, and even if they do the periodic structure that results looks bad (**exercises** ). Think of the problem as a rather odd inpainting problem – rather than knocking out a block of pixels, the noise process has obscured pixels outside the image.

It is straightforward to extend the incremental inpainting procedure to make a larger texture image from a small one. Find a pixel location whose value isn't

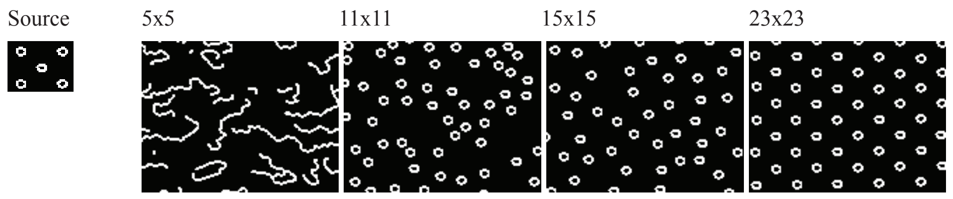

FIGURE 9.7: *The size of the patch used for texture synthesis matters a great deal. Various synthesized textures using differently sized windows and the same original example image. The smallest window cannot see the whole of a ring; the next can, but does not see that spacing is regular; a larger window can see relatively regular spacing; and a very large window simply copies the example.* Image credit: *Elements of Figure 3 from "Texture Synthesis by Non-parametric Sampling", IEEE International Conference on Computer Vision, Corfu, Greece, September 1999.* **No permission yet**

known, but where many neighbors have known values. Find matching patches in the known parts of the image, where you compute the SSD using only the known pixel values. Now choose the center pixel value from the pool of matching patches at random (rather than using the best match). The value of this pixel is now known, and you can iterate. The result is a synthesized texture image. Figure 9.7 shows an example.

It is quite important to have several patches to choose the pixel value from, and to choose at random. This prevents the texture you synthesize from being overly repetitious or even constant. The size of the patch you use also has important effects (Figure **??**)

### 9.1.4 Patches and Efficiency

## 9.2 DENOISING WITH PATCHES

Inpainting relies on a somewhat specialized model of noise. A small fraction of pixels need fixing, you which ones they are, and other pixel values are reliable. If the noise is, say, additive gaussian noise, these constraints don't apply. In turn, this means that all the pixels surrounding the pixel you want to fix may be somewhat wrong as well, so matching using their values may not be wise.

### 9.2.1 Non-Local Means

Here is one strategy to exploit the other patches. Estimate the true value of the pixel in the center of a target patch as a weighted sum of all other pixels, where the weight is big when the patch around the pixel is similar to the target patch and small when it isn't. These weights should be normalized to add up to one. The big difference between this strategy and filtering is that distant pixels can contribute if they are in comparable patches. This approach should yield a good estimate of the true value, at a ferocious cost. You need to look at every pixel in the image to estimate the true value of a single pixel, so estimating the whole image is quadratic

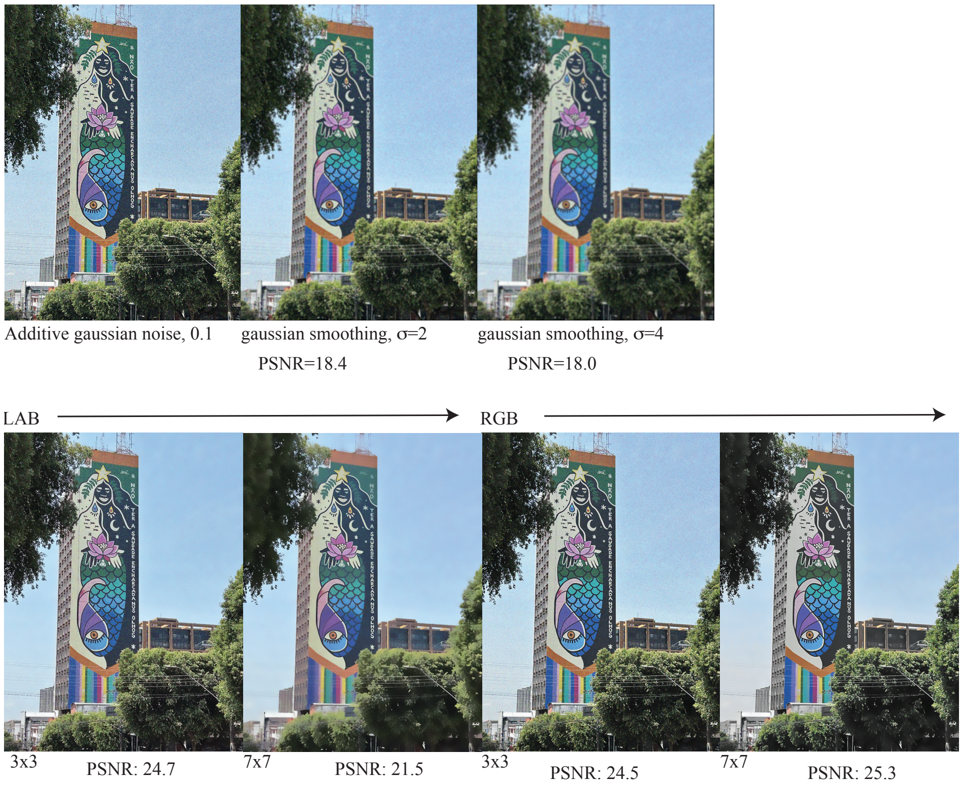

FIGURE 9.8: **Top row:** *Gaussian smoothing suppresses noise, but blurs edges, whereas non-local means preserves edges while smoothing gaussian noise (***bottom row***).* Image credit: *Figure shows my photograph of a building in downtown Manaus.*

in the number of pixels.

The approach is easily formalized. Write $K(\mathbf{p}_{ij}, \mathbf{p}_{uv})$ for a function that compares an image patch $\mathbf{p}_{ij}$ around the $i$, $j$'th pixel with the image patch around the $u$, $v$'th pixel. This function should be large when the patches are similar, and small when they are different. A useful estimate of the pixel value $\mathbf{x}_{ij}$ at $i$, $j$ is then

$$\sum_{uv \in \text{image}} \frac{K(\mathbf{p}_{ij}, \mathbf{p}_{uv})\mathbf{x}_{uv}}{\sum_{kl \in \text{image}} K(\mathbf{p}_{ij}, \mathbf{p}_{lm})}.$$

Notice that the weights sum to one. The estimate clearly depends quite strongly on the choice of $K$.

**The gaussian Kernel:** One natural choice uses SSD between patches. Write $\text{NSSD}(\mathbf{p}_{ij}, \mathbf{p}_{uv}))$ for the sum of squared differences between the two patches normalized to deal with the number of pixels in the patch (exercises), write $\sigma$ for some scale chosen to work well, note that I have suppressed the size of the patch, and use

$$K_{\text{NSSD}}(\mathbf{p}_{ij}, \mathbf{p}_{uv}) = e^{\frac{\left(-\text{NSSD}(\mathbf{p}_{ij}, \mathbf{p}_{uv})\right)}{2\sigma^2}}.$$

The method described here is sometimes known as *non-local means.* As described, it is very slow (quadratic in the number of pixels). Methods to speed it up remain difficult, and are out of scope (**exercises** ). As Figure 9.8 shows, non-local means can suppress a great deal of noise without blurring edges.

Some details are important. Using larger patches will tend to increase the computation time, and can improve denoising up to a point. If the patch is too small, there will be many matches but some noise will be preserved. Similarly, if the patch is too large, there will be too few matches to be helpful. Further, the choice of color representation has an effect (see Figure 9.8). You should expect this. RGB values are somewhat correlated (Section 41.2), while LAB values are not. This means that some SSD values computed from RGB will overstate the difference between patches (**exercises** ).

The gaussian kernel weights down patches that are different from the target patch, but pays no attention to the distance between patches. A natural extension, known as the *bilateral filter*, downweights patches based on their distance. This gives

$$K_{\text{bilat}}(\mathbf{p}_{ij}, \mathbf{p}_{uv}) = e^{\frac{\left(-\text{NSSD}(\mathbf{p}_{ij}, \mathbf{p}_{uv})\right)}{2\sigma^2}} e^{\frac{\left(-\left[(i-u)^2+(j-v)^2\right]\right)}{2\sigma_d^2}}$$

where $\sigma_d$ controls the rate at which a patches contribution falls off with distance. The bilateral filter admits significant speedups (**exercises** ).