15 Noise

Noise is the term applied to artifacts in the image that inhibit the ability to detect the targets. Noise can come in many forms, although the term is most commonly used for variations in intensity from sources other than the objects in the image. This chapter will review some types of noise and the methods by which the noise can be reduced.

15.1 RANDOM NOISE

Random noise is the addition of random values as in

$$\mathbf{b}[\vec{x}] = \mathbf{a}[\vec{x}] + \mathbf{n}[\vec{x}], \tag{15.1}$$

where $\mathbf{n}[\vec{x}]$ is an array that has the same frame size as the original image $\mathbf{a}[\vec{x}]$. The range of the values in $\mathbf{n}[\vec{x}]$ may depend on the image collecting system and/or the values of the associated pixels in $\mathbf{a}[\vec{x}]$. In the early days of couple-charged device (CCD) cameras, a bright intensity on one pixel tended to bleed over into several pixels in the same row or column. Thus, the noise was also dependent on pixel intensities of the surrounding pixels.

Gaussian noise is similar in theory except that the values in $\mathbf{n}[\vec{x}]$ are governed by a Gaussian distribution. The values are then random but within the guidelines of a distribution, and so the effect has a similar grainy appearance to that of random noise.

Consider the image shown in Figure 15.1, which consists of a student's lab work as they were tasked to use their smart phone in an pendulum experiment [18]. Of interest in the image is the laptop and table which are mostly smooth in the original image.

Code 15.1 adds noise to an original image, which is loaded as `data`. Line 5 shows that the max value of any pixel is 255, which is needed into order to properly scale the noise. In this case, 10% noise is added and thus the random array in line 5 is multiplied by 25.6. The noise is added in line 7. Replacing the **ranf** function with the **normal** generates Gaussian noise.

A portion of grayscale version of the original image is shown in Figure 15.2(a), and Figure 15.2(b) shows the same portion with noise added. The second image is grainier, which is particularly noticeable in the regions that were originally smooth. Increasing the percentage of noise increases the grainy nature of the image.

One simple method of reducing noise is to smooth the image. This works because the noise of one pixel is independent of the noise in the neighboring pixels. Thus, the average of the noise over a region should be close to a constant. Since $\mathbf{a}[\vec{x}] + k$ does not change the appearance of the image, the smoothing operation reduces the noise. Smoothing is very similar to averaging and so the graininess is reduced. However, smoothing also deteriorates the edges of the objects in the image.

The process of smoothing is simply,

$$\mathbf{b}[\vec{x}] = \mathcal{S}_{\alpha=2}\mathbf{a}[\vec{x}], \tag{15.2}$$

where α controls the amount of smoothing, and in this example $\alpha = 2$. Code 15.2 shows one method of smoothing in Python using the **cspline2d** function. The second argument to the function is the smoothing parameter. The result of the smoothing operation is shown in Figure 15.2(c). As seen, the graininess is greatly reduced but the edges are slightly more blurred.

15.2 SALT AND PEPPER NOISE

Salt and pepper noise describes the noise in which some pixels are completely white or completely black. This was common in the early days of CCD cameras as some pixels would fail. In this case,

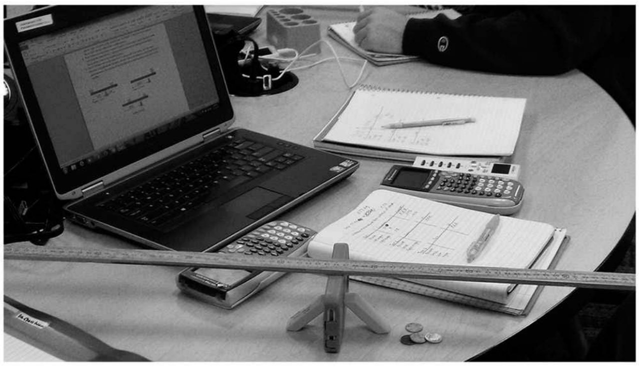

Figure 15.1 An original image.

Code 15.1 Adding random noise

```
>>> import imageio
>>> import numpy as np
>>> data = imageio.imread( 'data/altroom.png', as_gray=True)
>>> data.max()
255.0
>>> noise = (np.random.ranf( data.shape )-0.5) * 25.6
>>> sdata = data + noise
```

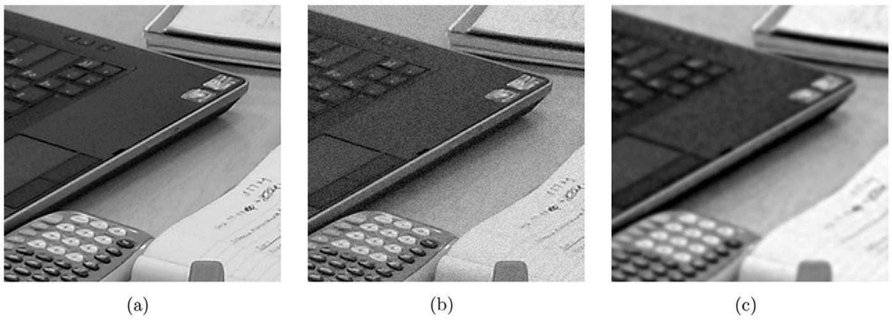

Figure 15.2 (a) A portion of the original image, (b) the image with 10% noise, and (c) the image after smoothing.

most of the pixels have the original values and some have a value of either 0 or 255. Smoothing will not be as effective here as the average noise about any pixel is not zero.

Creating the salt noise is shown in the first few lines in Code 15.3. Line 1 creates an array of random numbers that have values between 0 and 1. However, these are compared to a small factor such as 0.01. Thus, in this case, 1% of the pixels will be changed. Line 2 creates a new array `bdata`, which has the salt noise. The first term, `(1-r)*data`, maintains the original values for the 99% of

Code 15.2 Smoothing in Python

```
>>> import scipy.signal as ss
>>> bdata = ss.cspline2d( sdata,2 )
```

Code 15.3 Salt noise

```
>>> r = np.random.ranf( data.shape )<0.01
>>> bdata = (1-r)*data + r*255
>>> cdata = nd.grey_erosion(bdata,2)
>>> ddata = nd.grey_dilation(cdata,2)
```

the unaffected pixels. The array `r` is binary valued with 1% of the pixels set to 1. Thus, `1-r`, has 99% of the values set to 1.

One method of reducing this type of noise is

$$\mathbf{d}[\vec{x}] = \lhd_2 \rhd_2 \mathbf{b}[\vec{x}], \tag{15.3}$$

where $\rhd$ and $\lhd$ are the erosion and dilation operators (see Section 6.5). The erosion operator replaces the large value with the lowest neighbor value. This removes the salt noise, but it also makes bright regions slightly smaller as the perimeters also decay. Therefore, a dilation is applied to regrow bright regions to their original size. Since the salt noise was completely eliminated by the erosion, it does not return with the dilation.

Figure 15.3(a) shows the image after the salt noise is applied, and Figure 15.3(b) shows the image after the erosion and dilation. As seen, the salt noise is removed, and all of the objects are the original size.

This method works well but does have a small disadvantage. Edges in the output image that are not vertical or horizontal may have slightly exaggerated stair step borders. For the case of pepper noise, the dilation operator is applied first to fill in the black pixels with the brightest neighbor. To shrink bright areas back to their original size, the erosion operator is subsequently applied.

Salt and pepper noise is removed by

$$\mathbf{d}[\vec{x}] = \lhd_2 \rhd_4 \lhd_2 \mathbf{b}[\vec{x}]. \tag{15.4}$$

If the erosion operator is applied first, then the salt pixels are removed but the pepper pixels are larger. Thus, the erosion operator needs to have twice the extent as the original. A final erosion operator is then applied to get all shapes back to their original size.

The process could also apply the dilation first followed by the doubled erosion and final dilation. If the number of salt and pepper pixels is the same, then either method produces a similar result. If either salt or pepper is significantly dominant in the image, then it should be removed first.

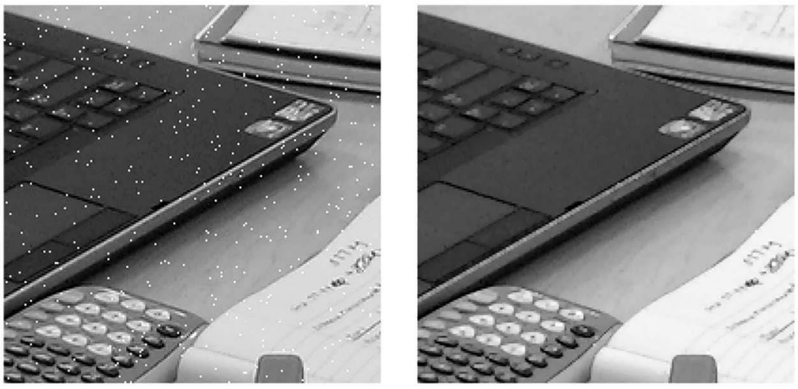

Figure 15.3 (a) After adding white pixels and (b) after erosion and dilation.

15.3 CAMERA NOISE

Early CCD cameras had issues that infused a lot of noise into the images. Decades of improvements though have eliminated many issues. However, some types of noise are still evident.

A lot of processing occurs in the capture of a digital image. For example, in some digital cameras, a signal is sent from the camera to the object when the shutter button is pressed. The returned signal helps with determining the focus and the bias and scale of the detected intensities. The image received is not actually the raw data. Modern cameras will even perform color correction, blur correction, face finding, and more.

15.4 COLORED NOISE

The term *colored noise* is applied for random noise in the Fourier plane. For an RGB image, this random noise can appear as blotchy colors. The reason is that some of the noise is applied to the lower frequencies, which correspond to larger regions in the input image.

Consider the original image shown in Figure 15.4(a). The remaining paint on the house is white. The same image under the influence of colored noise is shown in Figure 15.4(b).

Regions of the image have changes in color due to the random alteration of the original frequencies. This process is described by

$$\mathbf{b}[\vec{x}] = \mathfrak{F}^{-1} \frac{1}{1+\alpha \mathbf{n}[\vec{x}]} \mathfrak{F} \mathbf{a}[\vec{x}]. \tag{15.5}$$

The process is shown in Code 15.4.

15.5 COMPARISON OF NOISE REMOVAL SYSTEMS

This section will compare some of the common methods applied to random noise. The target image is shown in Figure 15.5(a). It is chosen because it has smooth regions (the road) and textured regions

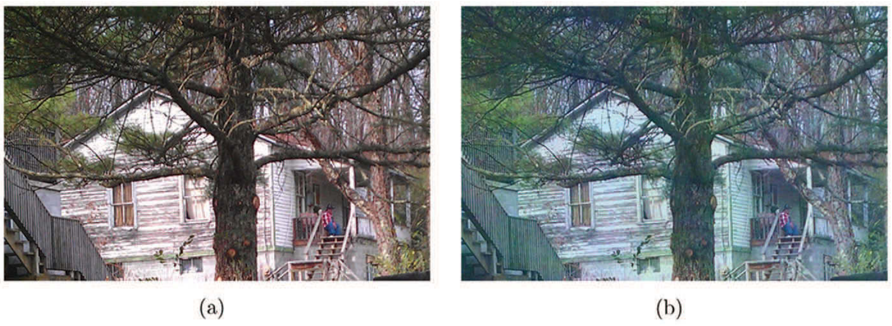

Figure 15.4 (a) An original image, and (b) the image after colored noise is applied.

Code 15.4 Applying colored noise

```
>>> bdata = np.zeros((V,H,3),complex)
>>> for i in range( 3 ):
        r = np.random.ranf( (V,H) )
        fdata = ft.fft2( data[:,:,i] )
        fdata = fdata/(1 + 1*r)
        bdata[:,:,i] = ft.ifft2( fdata )
```

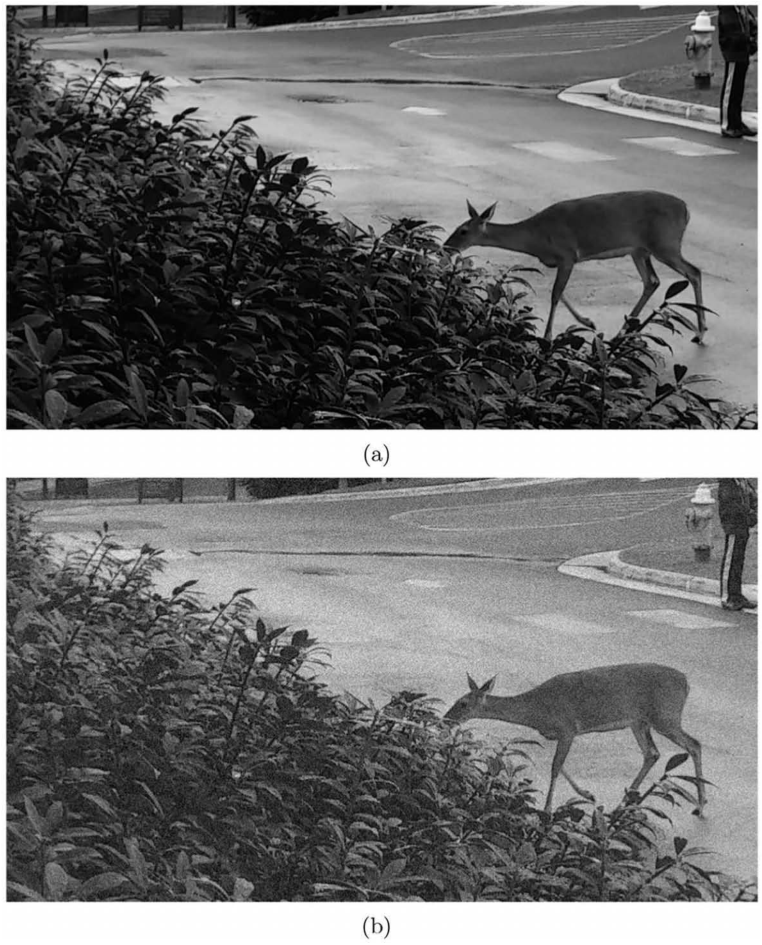

Figure 15.5 (a) The original image and the (b) after the noisehas been applied.

(the leaves). Noise is added via the **AddNoise** function shown in Code 15.5 and the result is shown in Figure 15.5(b).

15.5.1 SMOOTHING

The application of smoothing to remove noise was presented in Equation (15.2) and Code 15.2. The result is shown in Figure 15.6, and as seen, the objects are a little blurrier and the noise has not been completely removed.

Code 15.5 The **AddNoise** function

```
# deer.py
def AddNoise( fname ):
    adata = imageio.imread(fname,as_gray=True)
    noise = np.random.ranf( adata.shape )-0.5
    bdata = adata + 84*noise # 33 percent
    return bdata
```

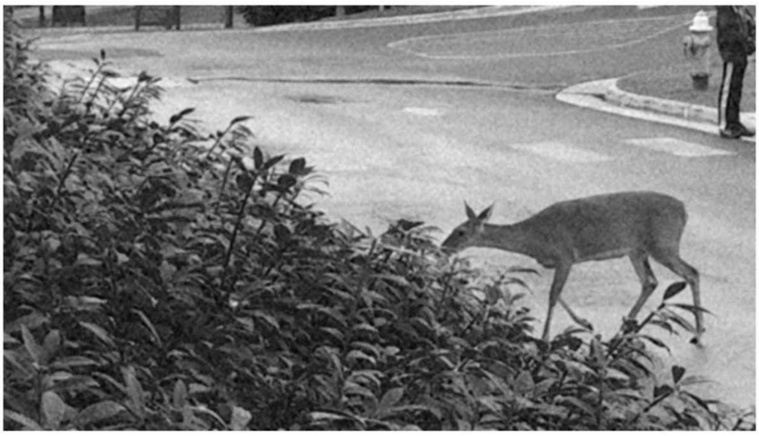

Figure 15.6 The deer image after smoothing.

15.5.2 LOW-PASS FILTERING

Since the noise is random, the noise in consecutive pixels is independent. This also means that the noise is in the highest frequencies. Thus, a low-pass filter can be applied to remove the highest of frequencies. This type of filtering was discussed in Section 11.1.1.

The grayscale input is defined as $\mathbf{b}[\vec{x}]$. The mask is an image that is the same frame size as the input with a solid circle centered in the frame. The radius of the circle is r, which the user can define to suit their needs. The radius is dependent on the frame size, and the qualities that are desired.

The process is defined as

$$\mathbf{d}[\vec{x}] = \Re \mathfrak{F}^{-1} X \left(\mathbf{o}_r[\vec{x}] \times X \mathfrak{F} \mathbf{b}[\vec{x}] \right). \tag{15.6}$$

Code 15.6 displays the function **Lopass** , which performs these functions. The mask is created in line 4 and multiplied by the swapped, Fourier transform in line 6. The image is returned back to the image space after another swap, and finally, the real components are returned. The result using $r = 128$ is shown in Figure 15.7.

15.5.3 EROSION AND DILATION

Consecutive erosion and dilation operators are very useful for removing salt and pepper noise, but as this example demonstrates, this process is less effective on removing random noise. The process is described in Equation (15.4).

Code 15.7 shows the **ErosionDilation** function, which applies the erosion and dilation operators. The result is shown in Figure 15.8, and as seen, the noise was not significantly reduced.

Code 15.6 The **Lopass** function

```
1 # deer.py
2 def Lopass( bdata ):
3     V,H = bdata.shape
4     circ = mgc.Circle( (V,H), (V/2,H/2), 128 )
5     fbdata = ft.fftshift( ft.fft2( bdata ) )
6     fddata = fbdata * circ
7     ddata = ft.ifft2( ft.ifftshift( fddata ))
8     return ddata.real
```

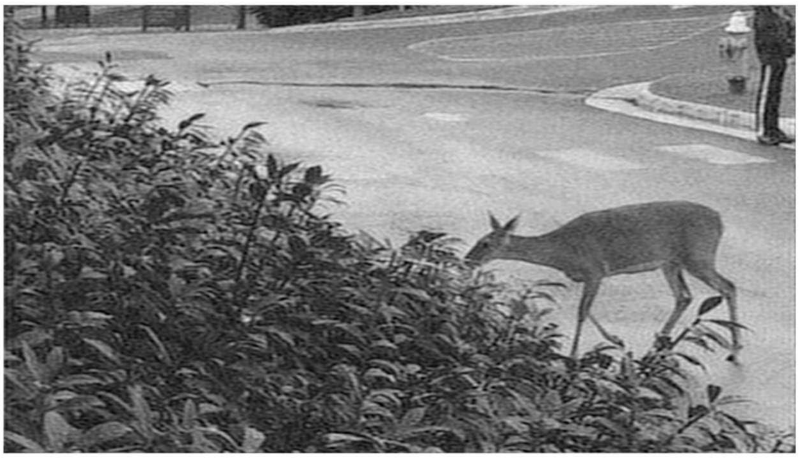

Figure 15.7 The deer image after applying a low-pass filter.

Code 15.7 The **ErosionDilation** function

```
# deer.py
def ErosionDilation( bdata ):
    b1 = nd.grey_erosion( bdata, 2 )
    b2 = nd.grey_dilation( b1, 4 )
    b3 = nd.grey_erosion( b2, 2 )
    return b3
```

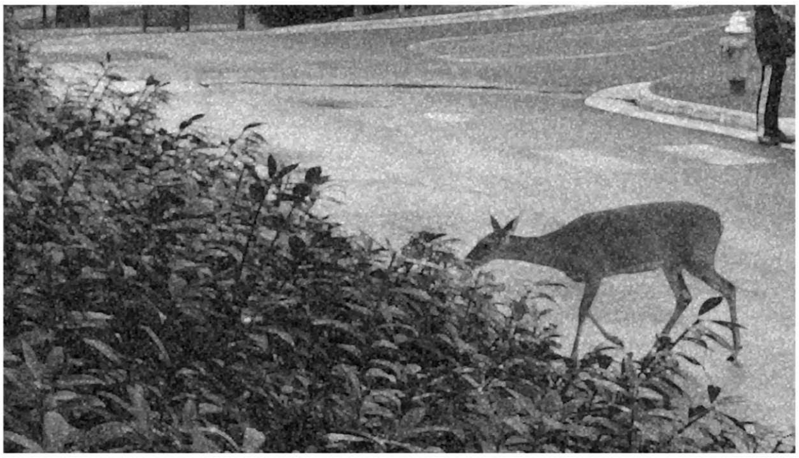

Figure 15.8 The deer image after the application of erosion and dilation operators.

15.5.4 MEDIAN FILTER

The *median filter* is a method that computes the local average for each pixel. It is very similar to smoothing. The Python *scipy.ndimage* module does contain a median filter function, so implementation is easy. Code 15.8 shows the application of this filter. The second argument controls the extent of the averaging. A single value uses a square extent but it is also possible to define a rectangular or

Code 15.8 Applying a median filter

```
>>> import scipy.ndimage as nd
>>> cdata = nd.median_filter(bdata,4 )
```

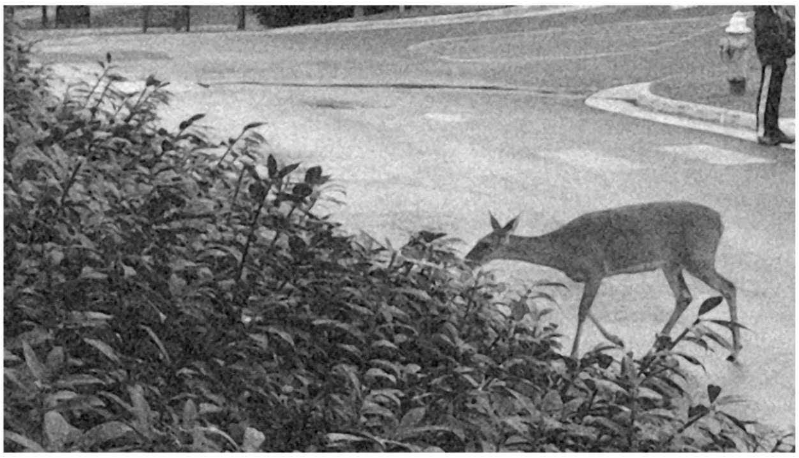

Figure 15.9 The deer image after the application of a median filter.

other shape for the region that is used to compute the average. Users should consult the *scipy* manual for a full display of the options that are available. The result of this filter is shown in Figure 15.9.

15.5.5 WIENER FILTER

The *Wiener Filter* attempts to minimize the noise in the Fourier space. The theory starts with the alteration of the original input by both a point spread function, $\mathbf{h}[\vec{x}]$, and additive noise, $\mathbf{n}[\vec{x}]$. Given the original image $\mathbf{a}[\vec{x}]$, the image received by the detector is modeled as

$$\mathbf{b}[\vec{x}] = \mathbf{h}[\vec{x}] \otimes \mathbf{a}[\vec{x}] + \mathbf{n}[\vec{x}], \tag{15.7}$$

where $\mathbf{h}[\vec{x}]$ is a point spread function perhaps a very thin Gaussian function. The $\mathbf{b}[\vec{x}]$ is the additive noise. Consider the filter $\mathbf{g}[\vec{\omega}]$, that satisfies

$$\mathfrak{F}\mathbf{a}[\vec{x}] = \mathbf{g}[\vec{\omega}] \times \mathfrak{F}\mathbf{b}[\vec{x}]. \tag{15.8}$$

The filter that performs this optimization is defined as

$$\mathbf{g}[\vec{\omega}] = \frac{\mathfrak{F}\mathbf{h}[\vec{x}]}{|\mathfrak{F}\mathbf{h}[\vec{x}]|^2 + \mathbf{f}[\vec{\omega}]/S}, \tag{15.9}$$

where $\mathbf{f}[\vec{\omega}]$ is the power spectrum of the input and S is the power spectrum of the estimated noise. However, for random noise, this value is just a scalar defined as $S = VH\sigma^2$, where V and H are the vertical and horizontal frame size and σ^2 is the standard deviation of the pixels in the received image. The power spectrum is

$$\mathbf{f}[\vec{\omega}] = |\mathfrak{F}\mathbf{a}[\vec{x}]|^2. \tag{15.10}$$

The *scipy.signal* module contains the **wiener** function as displayed in Code 15.9. Again, there are multiple options for the user to employ and consulting the *scipy* manual is informative. The result is shown in Figure 15.10 and as seen does a better job than some of the other filters.

Code 15.9 Applying a Wiener filter

```
>>> import scipy.signal as ss
>>> cdata = ss.wiener( bdata )
```

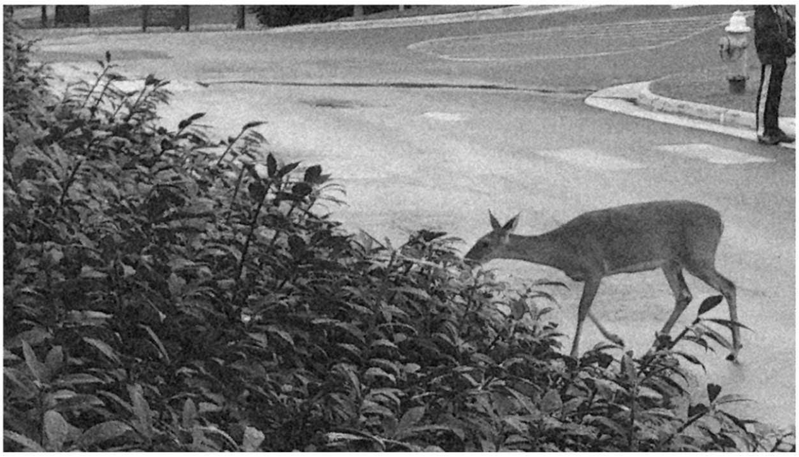

Figure 15.10 The deer image after the application of a Wiener filter.

15.6 OTHER TYPES OF NOISE

The work in this chapter was mostly applied to random noise. However, there are other types of noise: one example is shown in Figure 11.7. As the netting is in the way of the ball field, this is also a type of noise. *Structured noise* is a pattern that interferes with the image. The example in Section 11.4 removes these artifacts through Fourier filtering. This requires that the structure of the noise be known so that the correct frequencies can be removed.

Clutter is usually the presence of other objects in the image that interfere with the presentation of the target. For example, if the target is an image of a vehicle and there is a tree between the viewer and the target, then it interferes with the viewing of the target. The car is then presented to the viewer as two distinct segments separated by significant presence of the tree.

Glint is the bright reflection of the illuminating source. An example is shown in Figure 15.11, which shows two balloons in a room with several light sources. Several bright spots are seen on the balloons due to the reflection of the illuminating source. This reflection is so bright that the camera can no longer record the color of the balloon material in that region.

Shadows also provide a source of noise. A vehicle driving along a tree-lined lane will have several shadow patterns. The problem can be severe as the intensity of the car changes drastically between regions in the shade and in the direct light. If edge enhancing algorithms are applied to this type of image, then there are edges that arise from the boundary of the shadows rather than the objects in the image. One possible method of alleviating this issue is to convert the image to a different color space (HSV, YUV, etc.) in order to separate intensity from the hue. However, if the range of values between shaded and brightly lit areas is severe, then the hue viewed in the image will also be affected by this type of noise.

15.7 SUMMARY

Noise is a general term that accounts for artifacts that disrupt the purity of an image. Random noise adds a grainy texture to the image. An image with random pixels incorrectly set to fully ON or OFF

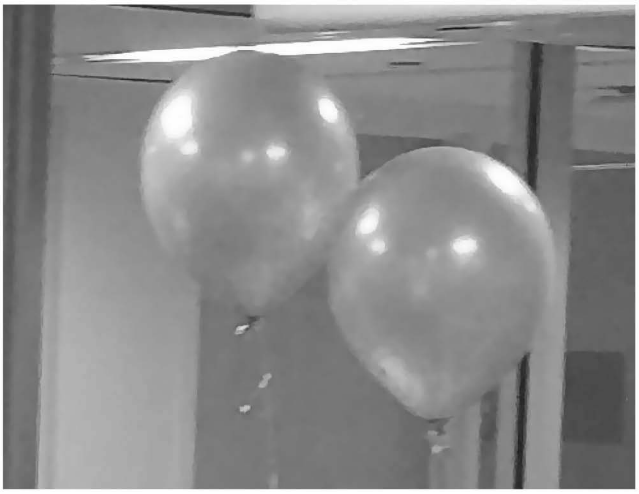

Figure 15.11 Reflecting balloons.

has salt and pepper noise. Colored noise are random fluctuations in the frequencies of the image. Several other types of noise exist as well. This chapter reviewed some of the types of noise and a few popular methods used to remove the noise.

PROBLEMS

1. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create $\mathbf{b}[\vec{x}]$ by adding 40% random noise to $\mathbf{a}[\vec{x}]$. Compute the error between the two images by

$$E = \sqrt{\sum_{\vec{x}} \frac{(\mathbf{a}[\vec{x}] - \mathbf{b}[\vec{x}])^2}{(256)^2 VH}}, \tag{15.11}$$

where V and H are the vertical and horizontal dimensions.
2. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create $\mathbf{b}[\vec{x}]$ by adding 40% random noise to $\mathbf{a}[\vec{x}]$. Compute the error using Equation (15.11).
3. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create an image $\mathbf{b}[\vec{x}]$ that is the same as $\mathbf{a}[\vec{x}]$ except that 2% of randomly selected pixels are set to the max value (255). Measure the error by Equation (15.11).
4. Using the image from problem 3 as $\mathbf{b}[\vec{x}]$, compute $\mathbf{c}[\vec{x}] = \lhd_2 \rhd_2 \mathbf{b}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{c}[\vec{x}]$ using Equation (15.11) by replacing $\mathbf{b}[\vec{x}]$ with $\mathbf{c}[\vec{x}]$.
5. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create an image $\mathbf{b}[\vec{x}]$ that is the same as $\mathbf{a}[\vec{x}]$ except that 2% of randomly selected pixels are set to the min value (0). Compute $\mathbf{c}[\vec{x}] = \rhd_2 \lhd_2 \mathbf{b}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{c}[\vec{x}]$ using Equation (15.11) by replacing $\mathbf{b}[\vec{x}]$ with $\mathbf{c}[\vec{x}]$.
6. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 1% pepper noise to $\mathbf{a}[\vec{x}]$. Compute $\mathbf{c}[\vec{x}] = \lhd_2 \rhd_4 \lhd_2 \mathbf{b}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{c}[\vec{x}]$ using Equation (15.11) by replacing $\mathbf{b}[\vec{x}]$ with $\mathbf{c}[\vec{x}]$.
7. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 3% pepper noise to $\mathbf{a}[\vec{x}]$. Compute $\mathbf{c}[\vec{x}] = \lhd_2 \rhd_4 \lhd_2 \mathbf{b}[\vec{x}]$. Compute $\mathbf{d}[\vec{x}] = \rhd_2 \lhd_4 \rhd_2 \mathbf{b}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{c}[\vec{x}]$ using Equation (15.11) by replacing $\mathbf{b}[\vec{x}]$ with $\mathbf{c}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{d}[\vec{x}]$ in the same manner. Which method of removing noise performed better?
8. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}\text{'})$. Create $\mathbf{b}[\vec{x}] = \mathbf{a}[\vec{x}] + 0.1\mathbf{q}[\vec{x}]$. Create $\mathbf{c}[\vec{x}] = \mathcal{S}_2 \mathbf{b}[\vec{x}]$. Measure the error between $\mathbf{a}[\vec{x}]$ and $\mathbf{c}[\vec{x}]$ using Equation (15.11) by replacing $\mathbf{b}[\vec{x}]$ with $\mathbf{c}[\vec{x}]$.

9. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}] = \mathbf{a}[\vec{x}] + 25 * \mathbf{q}_{\vec{w}}$. Clean $\mathbf{b}[\vec{x}]$ using a median filter.
10. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 1% pepper noise to $\mathbf{a}[\vec{x}]$. Clean $\mathbf{b}[\vec{x}]$ using a median filter.
11. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 1% pepper noise to $\mathbf{a}[\vec{x}]$. Compute
$$\mathbf{d}[\vec{x}] = \mathfrak{F}^{-1}(\mathfrak{F}\mathbf{b}[\vec{x}] \times (0.1\mathbf{q}[\vec{x}] + 0.9))$$
Clean $\mathbf{b}[\vec{x}]$ using a median filter.
12. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}] = \mathbf{a}[\vec{x}] + 25 * \mathbf{q}_{\vec{w}}$. Clean $\mathbf{b}[\vec{x}]$ using a Weiner filter.
13. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 1% pepper noise to $\mathbf{a}[\vec{x}]$. Clean $\mathbf{b}[\vec{x}]$ using a Weiner filter.
14. Use $\mathbf{a}[\vec{x}] = \mathcal{L}_L Y(\text{`}\texttt{clock.png}')$. Create $\mathbf{b}[\vec{x}]$ by adding 1% salt and 1% pepper noise to $\mathbf{a}[\vec{x}]$. Compute
$$\mathbf{d}[\vec{x}] = \mathfrak{F}^{-1}(\mathfrak{F}\mathbf{b}[\vec{x}] \times (0.1\mathbf{q}[\vec{x}] + 0.9))$$
Clean $\mathbf{d}[\vec{x}]$ using a Weiner filter.