- 9. Open up the image `engineer.png`. Experiment with applying the Fourier Transform to this image and the following filters:

    (a) Ideal filters (both low and high pass)
    (b) Butterworth filters
    (c) Gaussian filters

What is the smallest radius of a low pass ideal filter for which the face is still recognizable?

- 10. If you have access to a digital camera, or a scanner, produce a digital image of the face of somebody you know, and perform the same calculations as in the previous question.

8 Image Restoration

8.1 Introduction

Image restoration concerns the removal or reduction of degradations that have occurred during the acquisition of the image. Such degradations may include *noise*, which are errors in the pixel values, or optical effects such as out of focus blurring, or blurring due to camera motion. We shall see that some restoration techniques can be performed very successfully using neighborhood operations, while others require the use of frequency domain processes. Image restoration remains one of the most important areas of image processing, but in this chapter the emphasis will be on the techniques for dealing with restoration, rather than with the degradations themselves, or the properties of electronic equipment that give rise to image degradation.

A Model of Image Degradation

In the spatial domain, we might have an image $f(x, y)$, and a spatial filter $h(x, y)$ for which convolution with the image results in some form of degradation. For example, if $h(x, y)$ consists of a single line of ones, the result of the convolution will be a motion blur in the direction of the line. Thus, we may write

$$g(x, y) = f(x, y) * h(x, y)$$

for the degraded image, where the symbol * represents spatial filtering. However, this is not all. We must consider noise, which can be modeled as an additive function to the convolution. Thus, if $n(x, y)$ represents random errors which may occur, we have as our degraded image:

$$g(x, y) = f(x, y) * h(x, y) + n(x, y).$$

We can perform the same operations in the frequency domain, where convolution is replaced by multiplication, and addition remains as addition because of the linearity of the Fourier transform. Thus,

$$G(i, j) = F(i, j)H(i, j) + N(i, j)$$

represents a general image degradation, where of course F, H, and N are the Fourier transforms of f, h, and n, respectively.

If we knew the values of H and N we could recover F by writing the above equation as

$$F(i,j) = (G(i,j) - N(i,j))/H(i,j).$$

However, as we shall see, this approach may not be practical. Even though we may have some statistical information about the noise, we will not know the value of $n(x, y)$ or $N(i, j)$ for all, or even any, values. As well, dividing by $H(i, j)$ will cause difficulties if there are values that are close to, or equal to, zero.

8.2 Noise

We may define noise to be any degradation in the image signal, caused by external disturbance. If an image is being sent electronically from one place to another, via satellite or wireless transmission, or through networked cable, we may expect errors to occur in the image signal. These errors will appear on the image output in different ways depending on the type of disturbance in the signal. Usually we know what type of errors to expect, and hence the type of noise on the image; hence we can choose the most appropriate method for reducing the effects. Cleaning an image corrupted by noise is thus an important area of *image restoration*.

In this chapter we will investigate some of the standard noise forms and the different methods of eliminating or reducing their effects on the image.

We will look at four different noise types, and how they appear on an image.

Salt and Pepper Noise

Also called *impulse noise, shot noise*, or binary noise. This degradation can be caused by sharp, sudden disturbances in the image signal; its appearance is randomly scattered white or black (or both) pixels over the image.

To demonstrate its appearance, we will work with the image `gull.png` which we will make grayscale before processing:

```
>> g = rgb2gray(imread('gull.png'));
```

MATLAB/Octave

In Python there is also an `rgb2gray` function, in the `color` module of `skimage`:

```
In :  g = co.rgb2gray(io.imread('gull.png'))
```

Python

To add salt and pepper noise in MATLAB or Octave, we use the MATLAB function imnoise, which takes a number of different parameters. To add salt and pepper noise:

```
>> gsp = imnoise(g,'salt_and_pepper')
```

MATLAB/Octave

The amount of noise added defaults to 10%; to add more or less noise we include an optional parameter, being a value between 0 and 1 indicating the fraction of pixels to be corrupted. Thus, for example

```
>> imnoise(g,'salt_and_pepper',0.2);
```

MATLAB/Octave

would produce an image with 20% of its pixels corrupted by salt and pepper noise.

In Python, the method `noise.random_noise` from the `util` module of `skimage` provides this functionality:

```
In :  import skimage.util.noise as noise
In :  gn = noise.random_noise(g,mode='s&p')
In :  gn2 = noise.random_noise(g,mode='s&p',amount=0.2)
```

Python

The gull image is shown in Figure 8.1(a) and the image with noise is shown in Figure 8.1(b).

Figure 8.1: Noise on an image

(a) Original image (b) With added salt and pepper noise

Gaussian Noise

Gaussian noise is an idealized form of *white noise*, which is caused by random fluctuations in the signal. We can observe white noise by watching a television that is slightly mistuned to a particular channel. Gaussian noise is white noise which is normally distributed. If the image is represented as I, and the Gaussian noise by N, then we can model a noisy image by simply adding the two:

$$I + N.$$

Here we may assume that I is a matrix whose elements are the pixel values of our image, and N is a matrix whose elements are normally distributed. It can be shown that this is an appropriate model for noise. The effect can again be demonstrated by the noise adding functions:

```
>> gg = inoise(g,'gaussian');
```

MATLAB/Octave

or

```
In :  gg = noise.random_noise(g,'gaussian')
```

Python

As with salt and pepper noise, the `"gaussian"` parameter also can take optional values, giving the mean and variance of the noise. The default values are 0 and 0.01, and the result is shown in Figure 8.2(a).

Speckle Noise

Whereas Gaussian noise can be modeled by random values *added* to an image; *speckle noise* (or more simply just *speckle*) can be modeled by random values *multiplied* by pixel values, hence it is also called *multiplicative noise*. Speckle noise is a major problem in some radar applications. As before, the noise adding functions can do speckle:

```
>> gs = imnoise(g,'speckle');
```

MATLAB/Octave

and

```
In :  gs = noise.random_noise(g,'speckle')
```

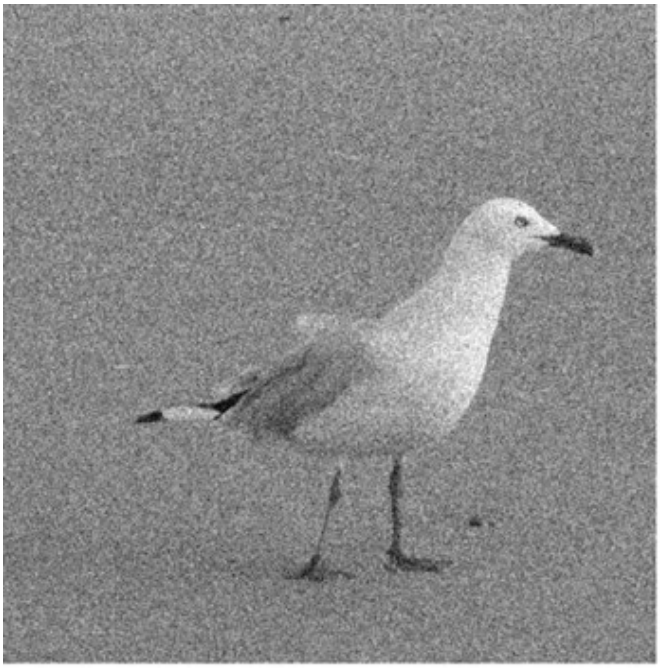
Python

and the result is shown in Figure 8.2(b). Speckle noise is implemented as

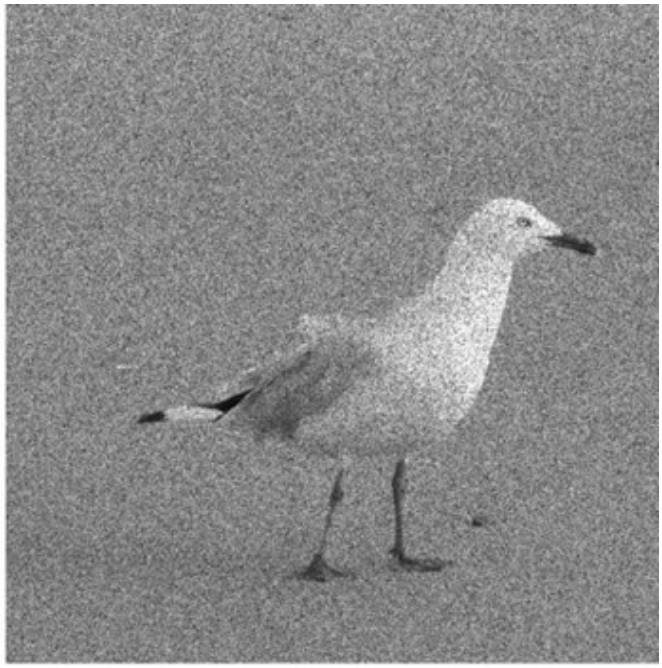
$$I(1+N)$$

where I is the image matrix, and N consists of normally distributed values with a default mean of zero. An optional parameter gives the variance of N; its default value in MATLAB or Octave is 0.04 and in Python is 0.01.

Figure 8.2: The gull image corrupted by Gaussian and speckle noise



(a) Gaussian noise



(b) Speckle noise

Although Gaussian noise and speckle noise appear superficially similar, they are formed by two totally different methods, and, as we shall see, require different approaches for their removal.

Periodic Noise

If the image signal is subject to a periodic, rather than a random disturbance, we might obtain an image corrupted by *periodic noise*. The effect is of bars over the image. Neither function `imnoise` or `random_noise` has a periodic option, but it is quite easy to create such noise, by adding a periodic matrix (using a trigonometric function) to the image. In MATLAB or Octave:

```
>> [rs,cs] = size(g);
>> [x,y] = meshgrid(1:rs,1:cs);
>> p = sin(x/3+y/5)+1;
>> gp = (2*im2double(g)+p/2)/3;
```

MATLAB/Octave

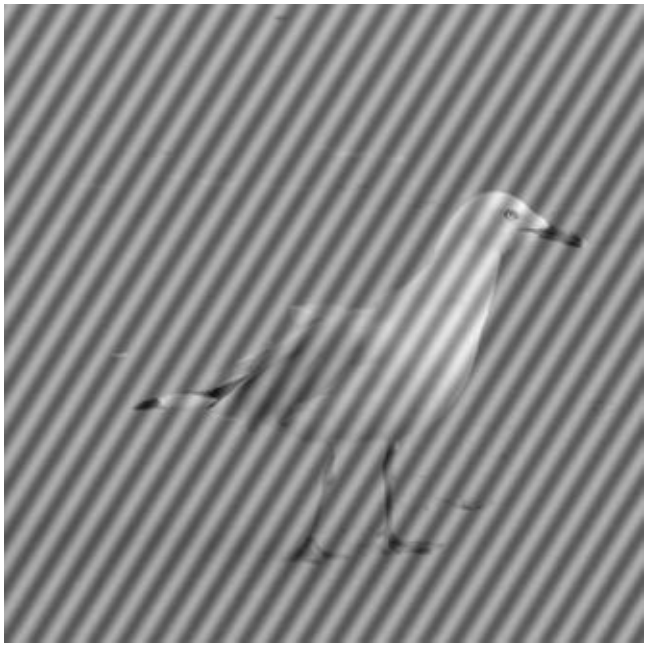
and in Python:

```
In :  r,c = g.shape
In :  x,y = np.mgrid(0:r,0:c].astype('float32')
In :  p = np.sin(x/3+y/3)+1.0
In :  gp = (2*skimage.util.img_as_float(g)+p/2)/3
```

Python

and the resulting image is shown in Figure 8.3.

Figure 8.3: The gull image corrupted by periodic noise



Salt and pepper noise, Gaussian noise, and speckle noise can all be cleaned by using spatial filtering techniques. Periodic noise, however, requires the use of frequency domain filtering. This is because whereas the other forms of noise can be modeled as *local* degradations, periodic noise is a *global* effect.

8.3 Cleaning Salt and Pepper Noise

Low Pass Filtering

Given that pixels corrupted by salt and pepper noise are high frequency components of an image, we should expect a low pass filter should reduce them. So we might try filtering with an average filter:

```
>> a3 = fspecial('average');
>> g3 = imfilter(gsp,a3);
```

MATLAB/Octave

or

```
In :  import numpy.ndimage as ndi
In :  g3 = ndi.uniform_filter(gsp,3)
```

Python

and the result is shown in Figure 8.4(a). Notice, however, that the noise is not so much removed as "smeared" over the image; the result is not noticeably "better" than the noisy image. The effect is even more pronounced if we use a larger averaging filter:

```
>> a7 = fspecial('average',7);
>> g7 = imfilter(gsp,a7);
```

MATLAB/Octave

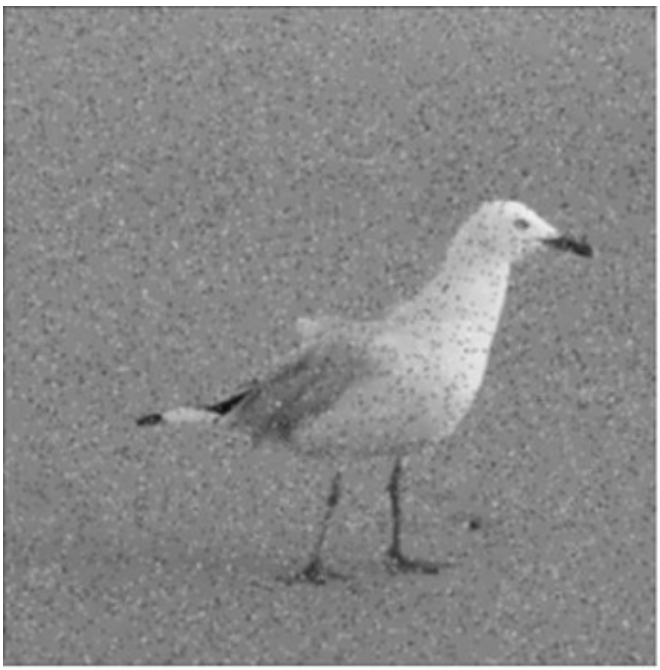
or

```
In : c7 = ndi.uniform_filter(csp,7)
```

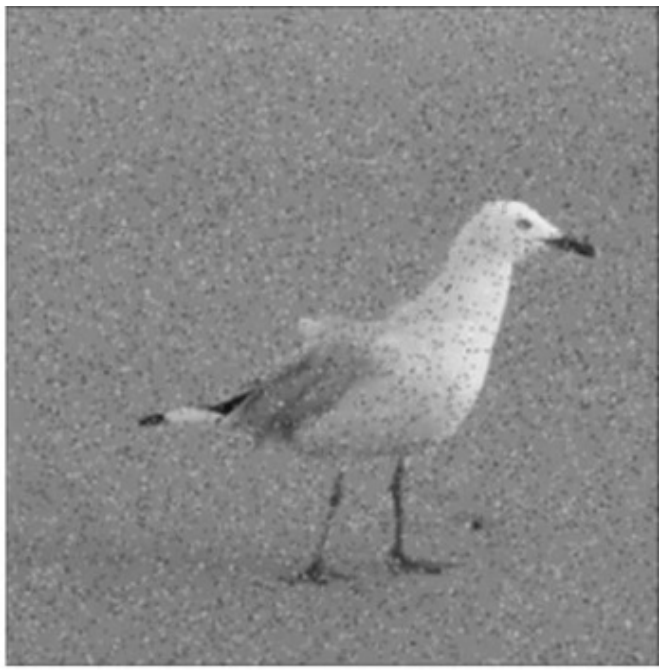
Python

and the result is shown in Figure 8.4(b).

Figure 8.4: Attempting to clean salt and pepper noise with average filtering



(a) 3×3 averaging



(b) 7×7 averaging

Median Filtering

Median filtering seems almost tailor-made for removal of salt and pepper noise. Recall that the *median* of a numeric list is the middle value when the elements of the set are sorted. If there are an even number of values, the median is defined to be the mean of the middle two. A median filter is an example of a non-linear spatial filter; using a 3×3 mask, the output value is the median of the values in the mask. For example:

50	65	52
63	255	58
61	60	57

$\longrightarrow$ 50 52 57 58 **60** 61 63 65 255 $\longrightarrow$ 60

The operation of obtaining the median means that very large or very small values—noisy values—will end up at the top or bottom of the sorted list. Thus, the median will in general replace a noisy value with one closer to its surroundings.

In MATLAB, median filtering is implemented by the `medfilt2` function:

```
>> gm3 = medfilt2(gsp);
```
MATLAB/Octave

and in Python by the `median_filter` method in the `ndimage` module of `numpy`:

```
In :  gm3 = ndi.median_filter(gsp,3)
```
Python

and the result is shown in Figure 8.5. The result is a vast improvement on using averaging filters. As with most functions, an optional parameter can be provided: in this case, a two element vector giving the size of the mask to be used.

Figure 8.5: Cleaning salt and pepper noise with a median filter

If we corrupt more pixels with noise:

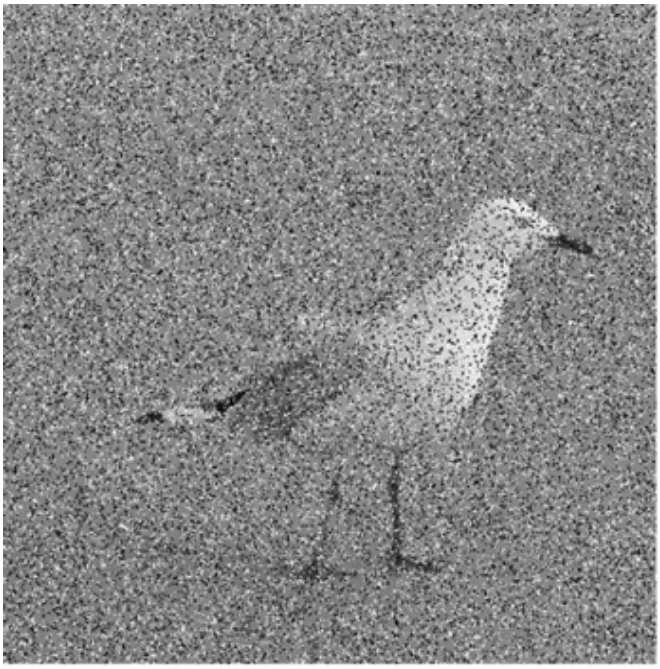
```
>> gsp2 = imnoise(g,'salt␣&␣pepper',0.2);
```
MATLAB/Octave

then `medfilt2` still does a remarkably good job, as shown in Figure 8.6. To remove noise completely, we can either try a second application of the 3×3 median filter, the result of which is shown in Figure 8.7(a) or try a 5×5 median filter on the original noisy image:

```
>> gm5 = medfilt2(gsp2,[5,5]);
```
MATLAB/Octave

the result of which is shown in Figure 8.7(b).

Figure 8.6: Using a 3 × 3 median filter on more noise



(a) 20% salt and pepper noise

(b) After median filtering

Figure 8.7: Cleaning 20% salt and pepper noise with median filtering

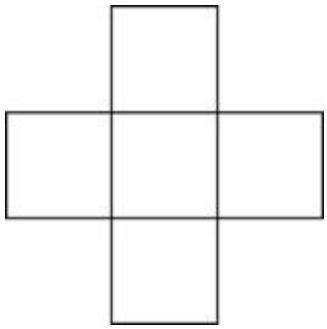
(a) Using `medfilt2` twice

(b) Using a 5 × 5 median filter

Rank-Order Filtering

Median filtering is a special case of a more general process called *rank-order filtering*. Rather than take the median of a list, the n-th value of the ordered list is chosen, for some predetermined value of n. Thus, median filtering using a 3 × 3 mask is equivalent to rank-order filtering with $n = 5$. Similarly, median filtering using a 5 × 5 mask is equivalent to rank-order filtering with $n = 13$. MATLAB and Octave implement rank-order filtering with the `ordfilt2` function; in fact the procedure for `medfilt2` is really

just a wrapper for a procedure which calls `ordfilt2`. There is only one reason for using rank-order filtering instead of median filtering, and that is that it allows the use of median filters over non-rectangular masks. For example, if we decided to use as a mask a 3 × 3 cross shape:



then the median would be the third of these values when sorted. The MATLAB/Octave command to do this is

```
>> co = ordfilt2(gsp,3,[0 1 0;1 1 1;0 1 0]);
```

MATLAB/Octave

In general, the second argument of `ordfilt2` gives the value of the ordered set to take, and the third element gives the *domain*; the non-zero values of which specify the mask. If we wish to use a cross with size and width 5 (so containing nine elements), we can use:

```
>> ordfilt2(gsp,5,[0 0 1 0 0;0 0 1 0 0;1 1 1 1 1;0 0 1 0 0;0 0 1 0 0])
```

MATLAB/Octave

In Python non-rectangular masks are handled with the `median_filter` method itself, by entering the mask (known in this context as the *footprint*) to be used:

```
In :  cross = array([[0,1,0],[1,1,1],[0,1,0]])
In :  co = ndi.median_filter(gsp,footprint=cross)
```

Python

When using `median_filter` there is no need to indicate which number to choose as output, as the output will be the median value of the elements underneath the 1's of the footprint.

An Outlier Method

Applying the median filter can in general be a slow operation: each pixel requires the sorting of at least nine values.1 To overcome this difficulty, Pratt [35] has proposed the use of cleaning salt and pepper noise by treating noisy pixels as *outliers*; that is, pixels whose gray values are significantly different from those of their neighbors. This leads to the following approach for noise cleaning:

1. Choose a threshold value D.
2. For a given pixel, compare its value p with the mean m of the values of its eight neighbors.

3. If $|p - m| > D$, then classify the pixel as noisy, otherwise not.
4. If the pixel is noisy, replace its value with m; otherwise leave its value unchanged.

There is no MATLAB function for doing this, but it is very easy to write one. First, we can calculate the average of a pixel's eight neighbors by convolving with the linear filter

$$\frac{1}{8}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.125 & 0.125 & 0.125 \\ 0.125 & 0 & 0.125 \\ 0.125 & 0.125 & 0.125 \end{bmatrix}$$

We can then produce a matrix r consisting of 1's at only those places where the difference of the original and the filter are greater than D; that is, where pixels are noisy. Then $1 - r$ will consist of ones at only those places where pixels are not noisy. Multiplying r by the filter replaces noisy values with averages; multiplying $1 - r$ with original values gives the rest of the output.

In MATLAB or Octave, it is simpler to use arrays of type `double`. With $D = 0.5$, the steps to compute this method are:

```
>> gsp = im2double(gsp);
>> av = [1 1 1;1 0 1;1 1 1]/8
>> gspa = imfilter(gsp,av);
>> D = 0.5
>> r = abs(gsp-gspa)>D;
>> imshow(r.*gspa+(1-r).*gsp)
```

MATLAB/Octave

and in Python, given that the output of `random_noise` is a floating point array, the steps are:

```
In :  av = array([[1,1,1],[1,0,1],[1,1,1]])/8.0
In :  gspa = ndi.convolve(gsp,av)
In :  D = 0.5
In :  r = (abs(gsp-gspa)>D)*1.0
In :  io.imshow(r*gspa+(1-r)*gsp)
```

Python

An immediate problem with the outlier method is that is it not completely automatic—the threshold D must be chosen. An appropriate way to use the outlier method is to apply it with several different thresholds, and choose the value that provides the best results.

Even with experimenting with different values of D, this method does not provide as good a result as using a median filter: the affect of the noise will be lessened, but there are still noise "artifacts" over the image. If we choose $D = 0.35$, we obtain the image in Figure 8.8(b), which still has some noise artifacts, although in different places. We can see that a lower value of D tends to remove noise from dark areas, and a higher value of D tends to remove noise from light areas. For this particular image, $D = 0.2$ seems to produce an acceptable result, although not quite as good as median filtering.

Clearly using an appropriate value of D is essential for cleaning salt and pepper noise by this method. If D is too small, then too many "non-noisy" pixels will be classified as noisy, and their values changed to the average of their neighbors. This will result in a blurring effect, similar to that obtained by using an averaging filter. If D is chosen to be too large, then not enough noisy pixels will be classified as noisy, and there will be little change in the output.

The outlier method is not particularly suitable for cleaning large amounts of noise; for such situations, the median filter is to be preferred. The outlier method may thus be considered as a "quick and dirty" method for cleaning salt and pepper noise when the median filter proves too slow.

A further method for cleaning salt and pepper noise will be discussed in Chapter 10.

1

In fact, this is not the case with any of our systems, all of which use a highly optimized method. Nonetheless, we introduce a different method to show that there are other ways of cleaning salt and pepper noise.

8.4 Cleaning Gaussian Noise

Image Averaging

It may sometimes happen that instead of just *one* image corrupted with Gaussian noise, we have many different copies of it. An example is satellite imaging: if a satellite passes over the same spot many times, taking pictures, there will be many different images of the

same place. Another example is in microscopy, where many different images of the same object may be taken. In such a case a very simple approach to cleaning Gaussian noise is to simply take the average—the mean—of all the images.

Figure 8.8: Applying the outlier method to 10% salt and pepper noise

(a) $D = 0.2$

(b) $D = 0.6$

To see why this works, suppose we have 100 copies of an image, each with noise; then the i-th noisy image will be:

$$M + N_i$$

where M is the matrix of original values, and N_i is a matrix of normally distributed random values with mean 0. We can find the mean M' of these images by the usual add and divide method:

$$\begin{aligned} M' &= \frac{1}{100}\sum_{i=1}^{100}(M + N_i) \\ &= \frac{1}{100}\sum_{i=1}^{100} M + \frac{1}{100}\sum_{i=1}^{100} N_i \\ &= M + \frac{1}{100}\sum_{i=1}^{100} N_i \end{aligned}$$

Since N_i is normally distributed with mean 0, it can be readily shown that the mean of all the N_i's will be close to zero—the greater the number of N_i's, the closer to zero. Thus,

$$M' \approx M$$

and the approximation is closer for larger number of images $M + N_i$.

We can demonstrate this with the cat image. We first need to create different versions with Gaussian noise, and then take the average of them. We shall create 10 versions. One way is to create an empty three-dimensional array of depth 10, and fill each "level" with a noisy image, which we assume to be of type `double`:

```
>> t = zeros([size(g),10]);
>> for i=1:10 t(:,:,i)=imnoise(g,'gaussian'); end
```

MATLAB/Octave

Note here that the `"gaussian"` option of `imnoise` calls the random number generator `randn`, which creates normally distributed random numbers. Each time `randn` is called, it creates a *different* sequence of numbers. So we may be sure that all levels in our three-dimensional array do indeed contain different images. Now we can take the average:

```
>> ta = mean(t,3);
```

MATLAB/Octave

The optional parameter 3 here indicates that we are taking the mean along the *third* dimension of our array.

In Python the commands are:

```
In :  x,y = c.shape
In :  t = zeros((x,y,10))
In :  for i in range(10):
...:      t[:,:,i] = noise.random_noise(c,'gaussian')
...:
In :  ta = np.mean(t,3)
```

Python

The result is shown in Figure 8.9(a). This is not quite clear, but is a vast improvement on the noisy image of Figure 8.2(a). An even better result is obtained by taking the average of 100 images; this can be done by replacing 10 with 100 in the commands above, and the result is shown in Figure 8.9(b). Note that this method only works if the Gaussian noise has mean 0.

Figure 8.9: Image averaging to remove Gaussian noise

(a) 10 images

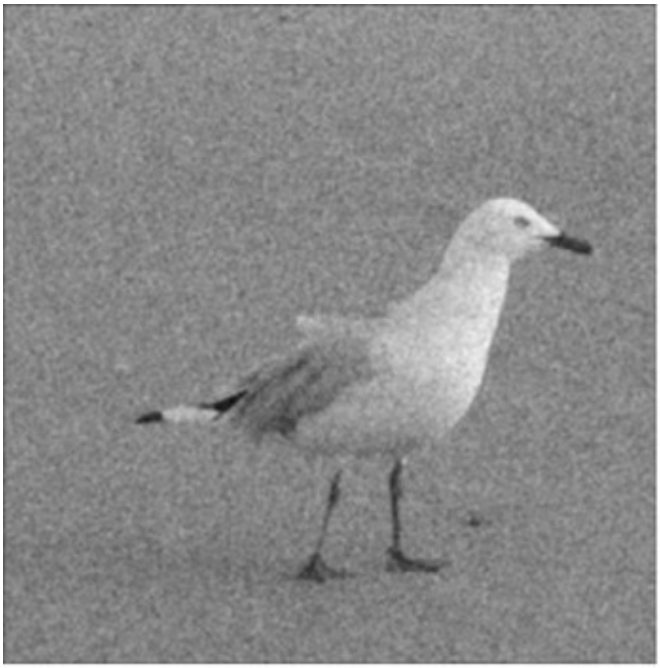
(b) 100 images

Average Filtering

If the Gaussian noise has mean 0, then we would expect that an average filter would average the noise to 0. The larger the size of the filter mask, the closer to zero. Unfortunately, averaging tends to blur an image, as we have seen in Chapter 5. However, if we are prepared to trade off blurring for noise reduction, then we can reduce noise significantly by this method.

Figure 8.10 shows the results of averaging with different sized masks. The results are not really particularly pleasing; although there has been some noise reduction, the “smeary” nature of the resulting images is unattractive.

Figure 8.10: Using averaging filtering to remove Gaussian noise



(a) 4×3 averaging

(b) 5×5 averaging

Adaptive Filtering

Adaptive filters are a class of filters that change their characteristics according to the values of the grayscales under the mask; they may act more like median filters, or more like average filters, depending on their position within the image. Such a filter can be used to clean Gaussian noise by using local statistical properties of the values under the mask.

One such filter is the *minimum mean-square error filter*; this is a non-linear spatial filter; and as with all spatial filters, it is implemented by applying a function to the gray values under the mask.

Since we are dealing with additive noise, our noisy image M' can be written as

$$M' = M + N$$

where M is the original correct image, and N is the noise; which we assume to be normally distributed with mean 0. However, within our mask, the mean may not be zero; suppose the mean is m_f, and the variance in the mask is σ_f^2. Suppose also that the variance of the noise over the entire image is known to be σ_g^2. Then the output value can be calculated as

$$m_f + \frac{\sigma_f^2}{\sigma_f^2 + \sigma_g^2}(g - m_f)$$

where g is the current value of the pixel in the noisy image. Note that if the local variance σ_f^2 is high, then the fraction will be close to 1, and the output close to the original image value g. This is appropriate, as high variance implies high detail such as edges, which should be preserved. Conversely, if the local variance is low, such as in a background area of the image, the fraction is close to zero, and the value returned is close to the mean value m_f. See Lim [29] for details.

Another version of this filter [52] has output defined by

$$g - \frac{\sigma_g^2}{\sigma_f^2}(g - m_f)$$

and again the filter returns a value close to either g or m_f depending on whether the local variance is high or low.

In practice, m_f can be calculated by simply taking the mean of all gray values under the mask, and σ_f^2 by calculating the variance of all gray values under the mask. The value σ_g^2 may not necessarily be known, so a slight variant of the first filter may be used:

$$m_f + \frac{\max\{0, \sigma_f^2 - n\}}{\max\{\sigma_f^2, n\}}(g - m_f)$$

where n is the computed noise variance, and is calculated by taking the mean of all values of σ_f^2 over the entire image. This particular filter is implemented in MATLAB with the function `wiener2`. The name reflects the fact that this filter attempts to minimize the square of the difference between the input and output images; such filters are in general known as *Wiener filters*. However, Wiener filters are more usually applied in the frequency domain; see Section 8.7.

Suppose we take the noisy image `cg` shown in Figure 8.2(a), and attempt to clean this image with adaptive filtering. We will use the `wiener2` function, which can take an optional parameter indicating the size of the mask to be used. The default size is 3 × 3. We shall create four images:

```
>> gw1 = wiener2(gg);
>> gw2 = wiener2(gg,[5,5]);
>> gw3 = wiener2(gg,[7,7]);
>> gw4 = wiener2(gg,[9,9]);
```

MATLAB/Octave

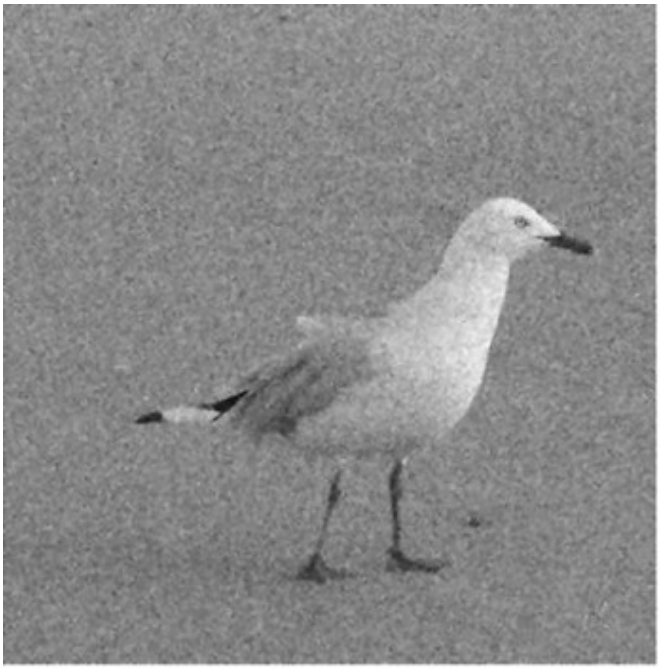
In Python the Wiener filter is implemented by the `wiener` method of the `signal` module of `scipy`:

```
In :  from scipy.signal import wiener
In :  cw1 = wiener(cg,[3,3])
```

Python

and the other commands similarly. The results are shown in Figure 8.11. Being a low pass filter, adaptive filtering does tend to blur edges and high frequency components of the image. But it does a far better job than using a low pass blurring filter.

Figure 8.11: Examples of adaptive filtering to remove Gaussian noise



(a) 3×3 filtering

(b) 5×5 filtering

(c) 7×7 filtering

(d) 9×9 filtering

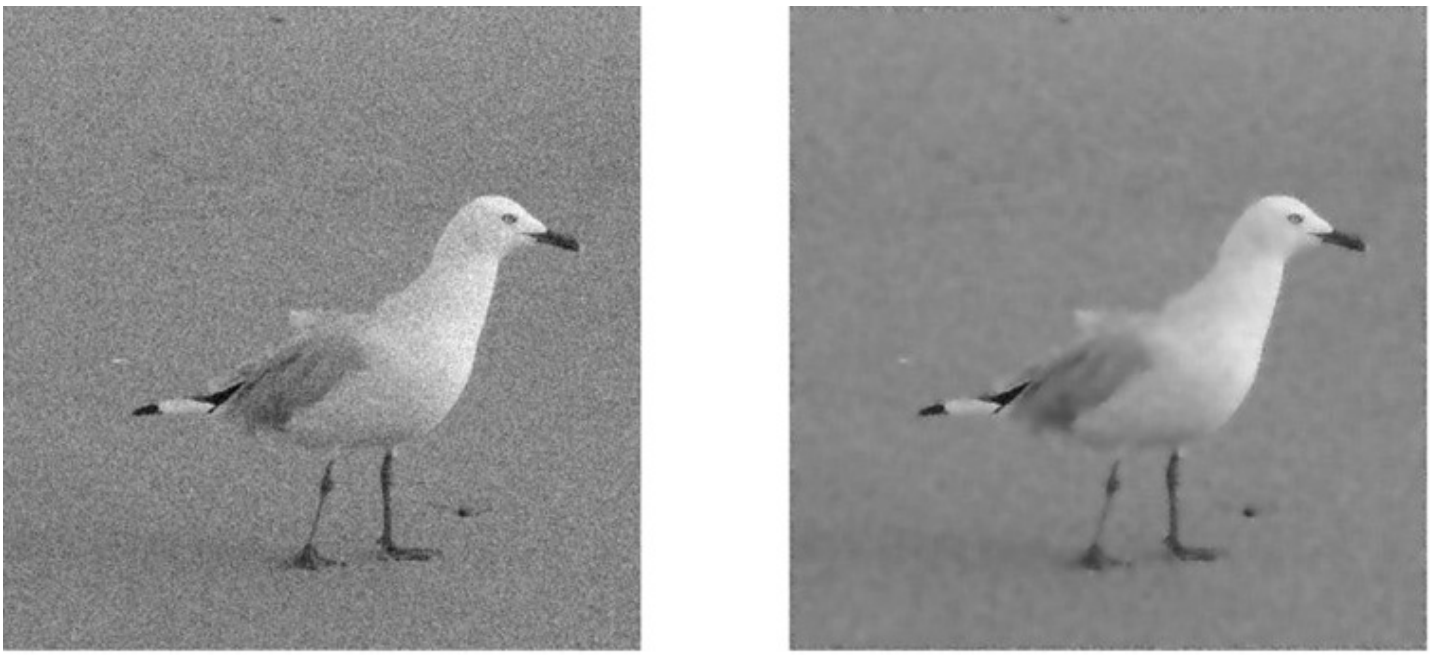
We can achieve very good results for noise where the variance is not as high as that in our current image.

```
>> gg2 = imnoise(g,'gaussian',0,0.005);
>> imshow(gg2)
>> gg2w = wiener2(im2double(gg2),[7,7]);
>> figure,imshow(gg2w)
```

MATLAB/Octave

The image and its appearance after adaptive filtering is shown in Figure 8.12. The result is a great improvement over the original noisy image. Notice in each case that there may be some blurring of the background, but the edges are preserved well, as predicted by our analysis of the adaptive filter formulas above.

Figure 8.12: Using adaptive filtering to remove Gaussian noise with low variance



(a) Image with variance 0.005 (b) Result after adaptive filtering

8.5 Removal of Periodic Noise

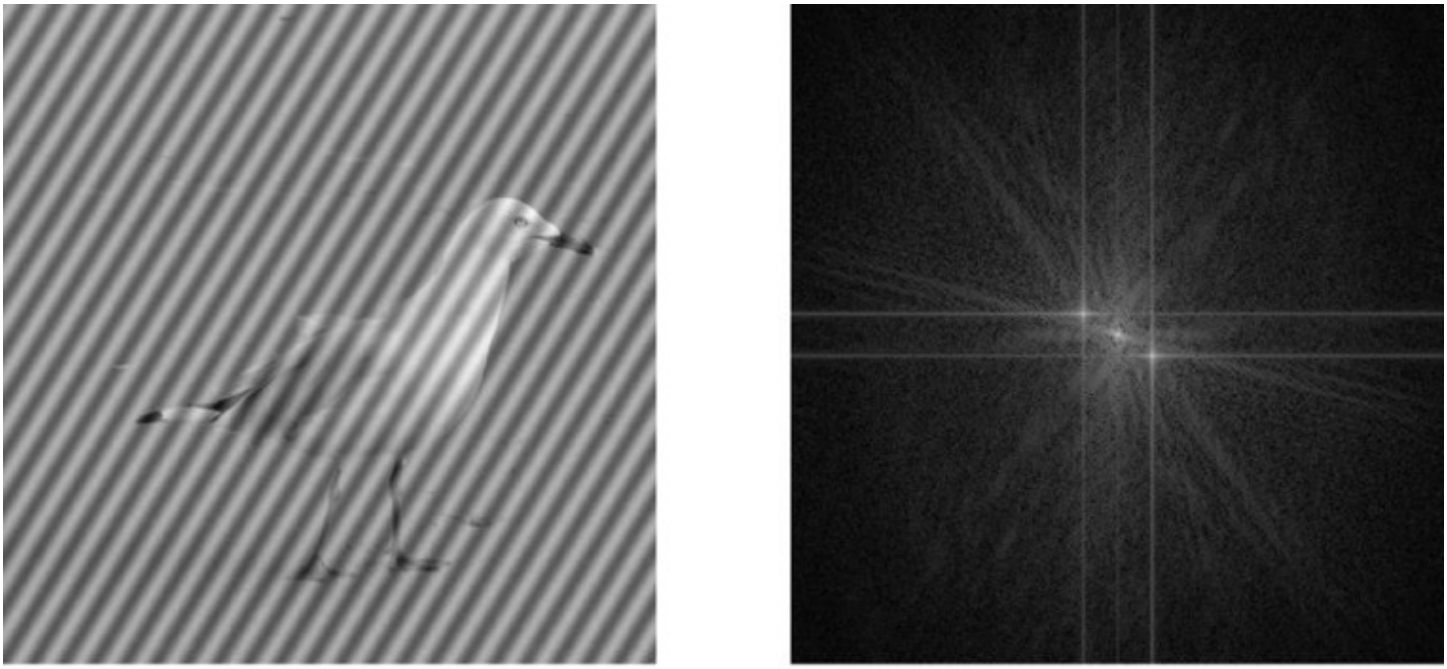
Periodic noise may occur if the imaging equipment (the acquisition or networking hardware) is subject to electronic disturbance of a repeating nature, such as may be caused by an electric motor. We have seen in Section 8.2 that periodic noise can be simulated by overlaying an image with a trigonometric function. We used

```
>> p = sin(x/3+y/5)+1;
>> gp = (2*im2double(c)+p/2)/3;
```

MATLAB/Octave

where `g` is our gull image. The first line simply creates a sine function, and adjusts its output to be in the range 0–2. The last line first adjusts the gull image to be in the range 0–2; adds the sine function to it, and divides by 3 to produce a matrix of type `double` with all elements in the range 0.0–1.0. This can be viewed directly with `imshow`, and it is shown in Figure 8.13(a). We can produce its shifted DFT and this is shown in Figure 8.3(b). The extra two "spikes" away from the center correspond to the noise just added. In general, the tighter the period of the noise, the further from the center the two spikes will be. This is because a small period corresponds to a high frequency (large change over a small distance), and is therefore further away from the center of the shifted transform.

Figure 8.13: The gull image (a) with periodic noise and (b) its transform



We will now remove these extra spikes, and invert the result. The first step is to find their position. Since they will have the largest maximum value after the DC coefficient, which is at position (201, 201), we simply have to find the maxima of all absolute values except the DC coefficient. This will be easier to manage if we turn the absolute values of the FFT into an array of type `uint8`. Note that we make use of the `find` function which returns all indices in an array corresponding to elements satisfying a certain condition; in this case, they are equal to the maximum:

```
>> gf = fftshift(fft2(gp));
>> imshow(mat2gray(log(1+abs(gf))))
>> gf2 = im2uint8(mat2gray(abs(gf)));
>> gf2(201,201) = 0;
>> [i,j] = find(gf2==max(gf2(:)))
i =

   188
   214

j =

   180
   222
```

MATLAB/Octave

Check that each point is the same distance from the center by computing the square of the distance of each point from (201, 201):

```
>> (i-201).^2+(j-201).^2

   610
   610
```

MATLAB/Octave

With Python, the above operations could be performed as follows, where to find array elements satisfying a condition we use the `where` function:

```
In :  from numpy.fft import ifft2,fft2,fftshift
In :  gf = fftshift(fft2(gp))
In :  temp = ex.rescale_intensity(abs(gf),out_range=(0,1))
In :  gf2 = ut.img_as_ubyte(temp)
In :  gf2[200,200] = 0
In :  i,j = np.where(gf2 == gf2.max())
In :  i,j
Out:  (array([179, 221]), array([187, 213]))
In :  (i-200)**2+(j-200)**2
Out:  array([610, 610])
```

Python

There are two methods we can use to eliminate the spikes; we shall look at both of them.

Band reject filtering. A band reject filter eliminates a particular band: that is, an annulus (a ring) surrounding the center. If the annulus is very thin, then the filter is sometimes called a *notch filter*. In our example, we need to create a filter that rejects the band at about $\sqrt{610}$ from the center. In MATLAB/Octave:

```
>> z = sqrt((x-201).^2+(y-201).^2);
>> d = sqrt(610);
>> k = 1;
>> br = (z < floor(d-k) | z > ceil(d+k));
```

MATLAB/Octave

or in Python:

```
In :  z = np.sqrt((x-200)**2+(y-200)**2)
In :  k = 1
In :  d = sqrt(610.0)
In :  br = (z< np.floor(d-k)) | (z>np.ceil(d+k))
```

Python

This particular ring will have a thickness large enough to cover the spikes. Then as before, we multiply this by the transform:

```
>> cfr = cf.*br;
```

MATLAB/Octave

or with

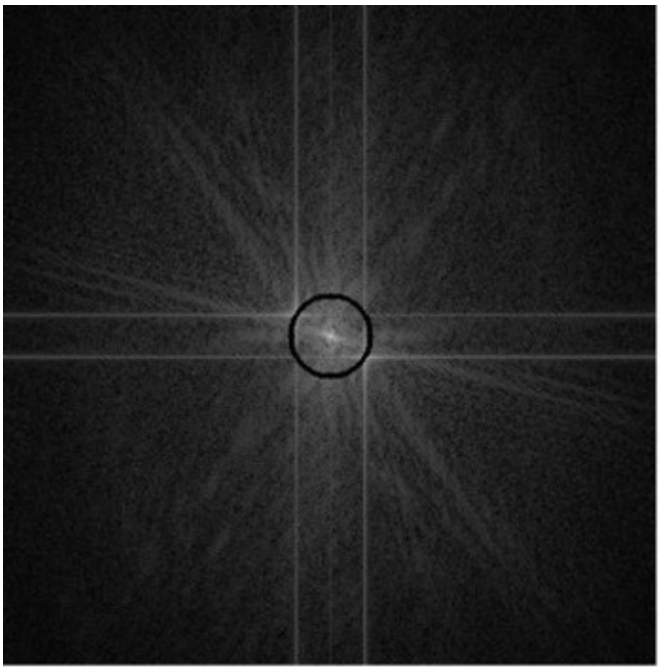
```
In :  cfr = cf*br
```

Python

and finally apply the inverse transform and display its absolute value. This final result is shown in Figure 8.14(a). The result is that the spikes have been blocked out by this filter. Taking the inverse transform produces the image shown in Figure 8.14(b). Note that not all the noise has gone, but a significant amount has, especially in the center of the image. More noise can be removed by using a larger band: Figure 8.15 shows the results using $k = 2$ and $k = 5$.

Criss-cross filtering. Here the rows and columns of the transform that contain the spikes are set to zero, which can easily be done using the i and **j** values from above:

Figure 8.14: Removing periodic noise with a band-reject filter



(a) A band-reject filter

(b) After inversion

Figure 8.15: Removing periodic noise with wider band-reject filters

(a) Band-reject filter with $k = 2$

(b) Band-reject filter with $k = 5$

```
>> gf2 = gf;
>> gf2(i,:) = 0;
>> gf2(:,j) = 0;
```

MATLAB/Octave

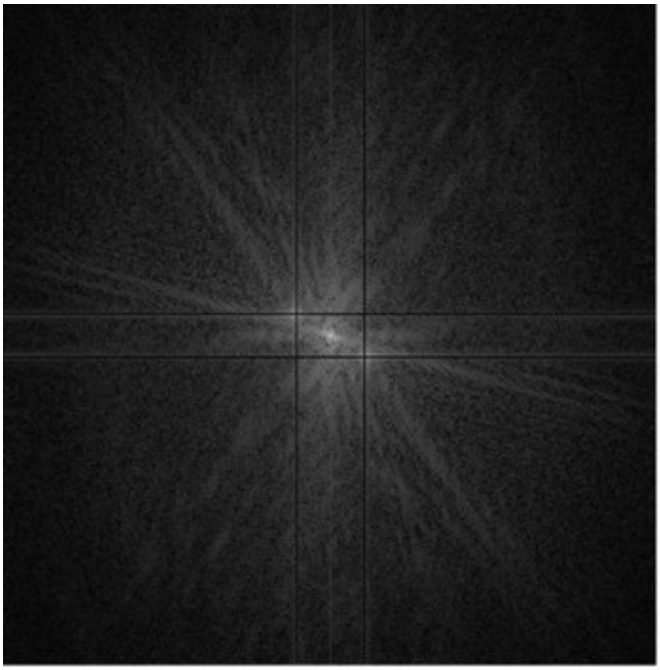
or

```
In :  cf2 = np.copy(cf)
In :  cf2[i,:] = 0
In :  cf2[:,j] = 0
```

Python

and the result after applying the inverse is shown in Figure 8.16(a). The image after inversion is shown in Figure 8.16(b). As before, much of the noise in the center has been removed. Making more rows and columns of the transform zero would result in a larger reduction of noise.

Figure 8.16: Removing periodic noise with a criss-cross filter



(a) A criss-cross filter

(b) After inversion

8.6 Inverse

We have seen that we can perform filtering in the Fourier domain by multiplying the DFT of an image by the DFT of a filter: this is a direct use of the convolution theorem. We thus have

$$Y(i,j) = X(i,j)F(i,j)$$

where X is the DFT of the image; F is the DFT of the filter, and Y is the DFT of the result. If we are given Y and F, then we should be able to recover the (DFT of the) original image X simply by dividing by F:

$$X(i,j) = \frac{Y(i,j)}{F(i,j)}. \tag{8.1}$$

Suppose, for example, we take the buffalo image `buffalo.png`, and blur it using a low pass Butterworth filter. In MATLAB/Octave:

```
>> b = imread('buffalo.png');
>> bf = fftshift(fft2(b));
>> [r,c] = size(b);
>> [x,y] = meshgrid(-c/2:c/2-1,-r/2:r/2-1);
>> bworth = 1./(1+(sqrt(2)-1)*((x.^2+y.^2)/15^2).^2);
>> bw = bf.*bworth;
>> bwa = abs(ifft2(bw));
>> blur = im2uint8(mat2gray(bwa));
>> imshow(blur)
```

MATLAB/Octave

In Python:

```
In :  b = io.imread('buffalo.png')
In :  bf = fftshift(fft2(b))
In :  r,c = b.shape
In :  x,y = np.mgrid(-c/2:c/2,-r/2:r/2]
In :  bworth = 1/(1+(np.sqrt(2)-1)*((x**2+y**2)/15**2)**2)
In :  bw = bf*bworth
In :  bwa = abs(ifft2(bw))
In :  blur = ut.img_as_ubyte(ex.rescale_intensity(bwa,out_range=(0.0,1.0)))
```

Python

The result is shown on the left in Figure 8.17. We can attempt to recover the original image by dividing by the filter:

```
>> blf = fftshift(fft2(blur));
>> blfw = blf./bworth;
>> bla = abs(ifft2(blfw));
>> imshow(mat2gray(bla))
```

MATLAB/Octave

or with

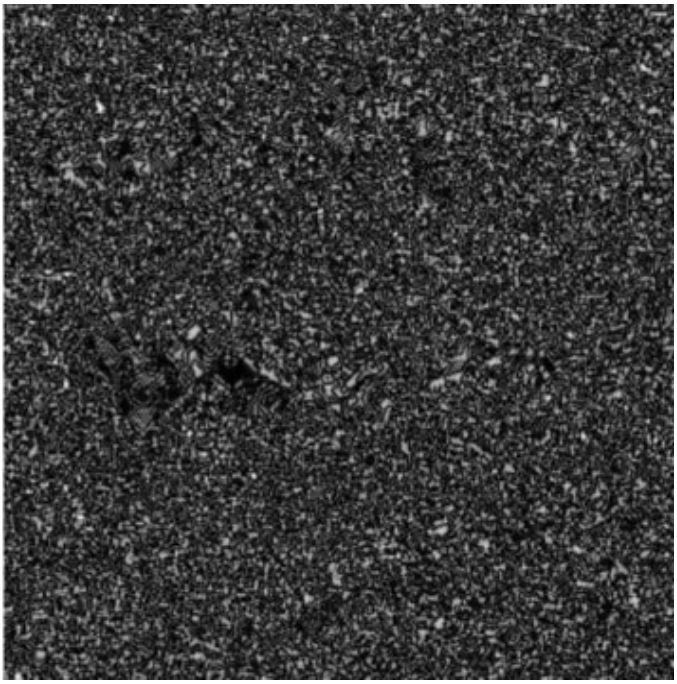
```
In :  blf = fftshift(fft2(blur))
In :  blfw = blf/bworth
In :  bla = abs(ifft2(blfw))
In :  io.imshow(bla)
```

Python

and the result is shown on the right in Figure 8.17. This is no improvement! The trouble is that some elements of the Butterworth matrix are very small, so dividing produces very large values which dominate the output. This issue can be managed in two ways:

1. Apply a low pass filter L to the division: $X(i,j) = \frac{Y(i,j)}{F(i,j)} L(i,j).$ This should eliminate very low (or zero) values.

Figure 8.17: An attempt at inverse filtering



- 2. "Constrained division": choose a threshold value d, and if $|F(i,j)| < d$, the original value is returned instead of a division: Thus:

$$X(i,j) = \begin{cases} \dfrac{Y(i,j)}{F(i,j)} & \text{if } |F(i,j)| \geq d, \\ Y(i,j) & \text{if } |F(i,j)| < d. \end{cases}$$

The first method can be implemented by multiplying a Butterworth low pass filter to the matrix `c1` above:

```
>> blf = fftshift(fft2(blur));
>> D = 40
>> bworth2 = 1./(1+(sqrt(2)-1)*((x.^2+y.^2)/D^2).^10);
>> blfb = blf./bworth.*bworth2;
>> ba = abs(ifft2(blfb));
>> imshow(mat2gray(ba))
```

MATLAB/Octave

or

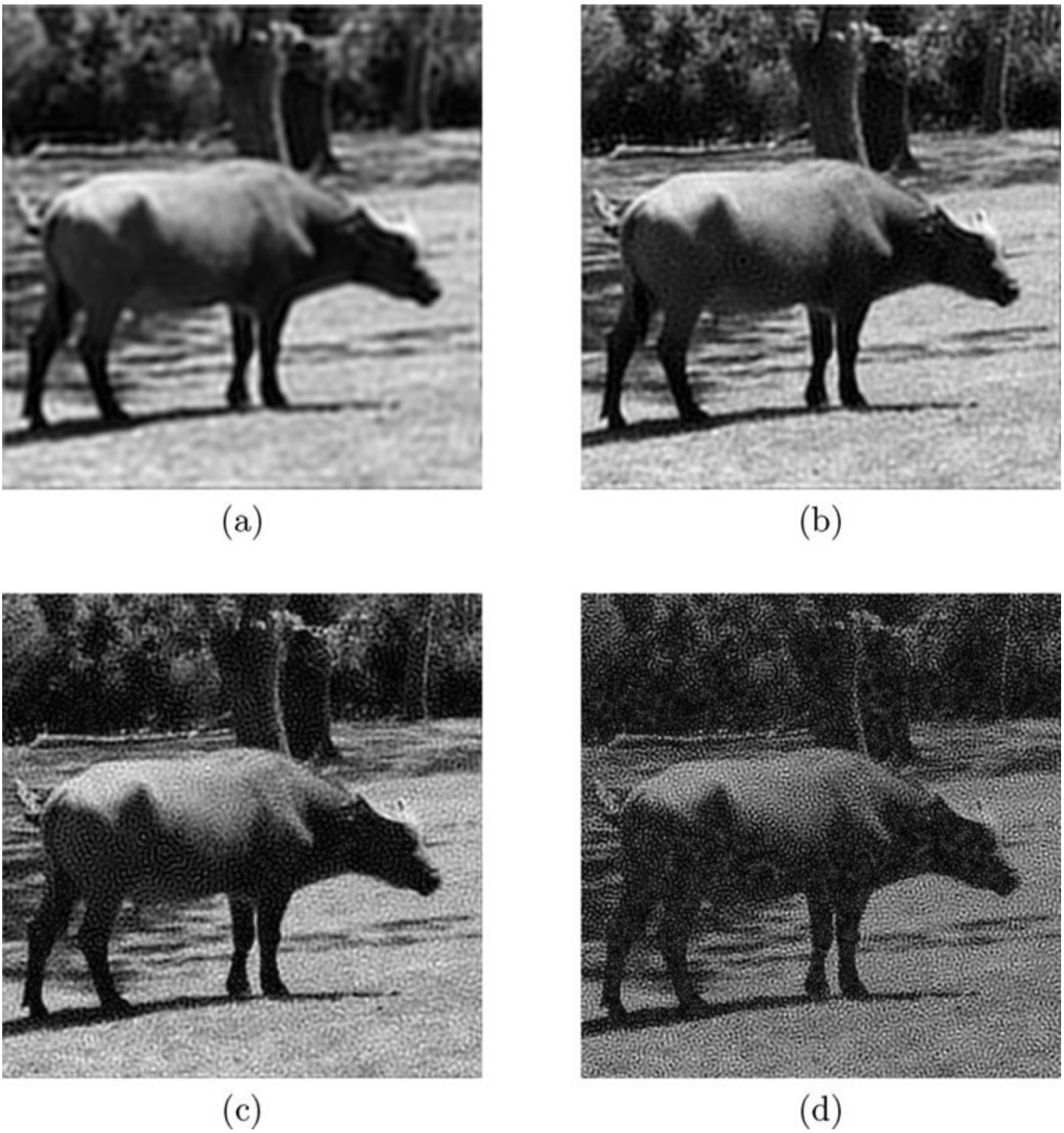
```
In :  blf = fftshift(fft2(blur))
In :  D = 40
In :  bworth2 = 1/(1+(sqrt(2)-1)*((x**2+y**2)/D**2)**10)
In :  blfb = blf/bworth*bworth2
In :  ba = abs(ifft2(blfb))
In :  io.imshow(ex.rescale_intensity(ba,out_range=(0.0,1.0)))
```

Python

Figure 8.18 shows the results obtained by using a different cutoff radius of the Butterworth filter each time: (a) uses 40 (as in the MATLAB commands just given); (b) uses 60; (c) uses 80, and (d) uses 100. It seems that using a low pass filter with a cutoff round about 60 will yield the best results. After we use larger cutoffs, the result degenerates.

We can try the second method; to implement it, we simply make all values of the filter that are too small equal to 1:

Figure 8.18: Inverse filtering using low pass filtering to eliminate zeros




```
>> d = 0.01;
>> bw = bworth; bw(find(bw<d))=1
>> fbw = fftshift(fft2(blur))./bw;
>> w1a = abs(ifft2(w1));
>> imshow(mat2gray(w1a))
```

MATLAB/Octave

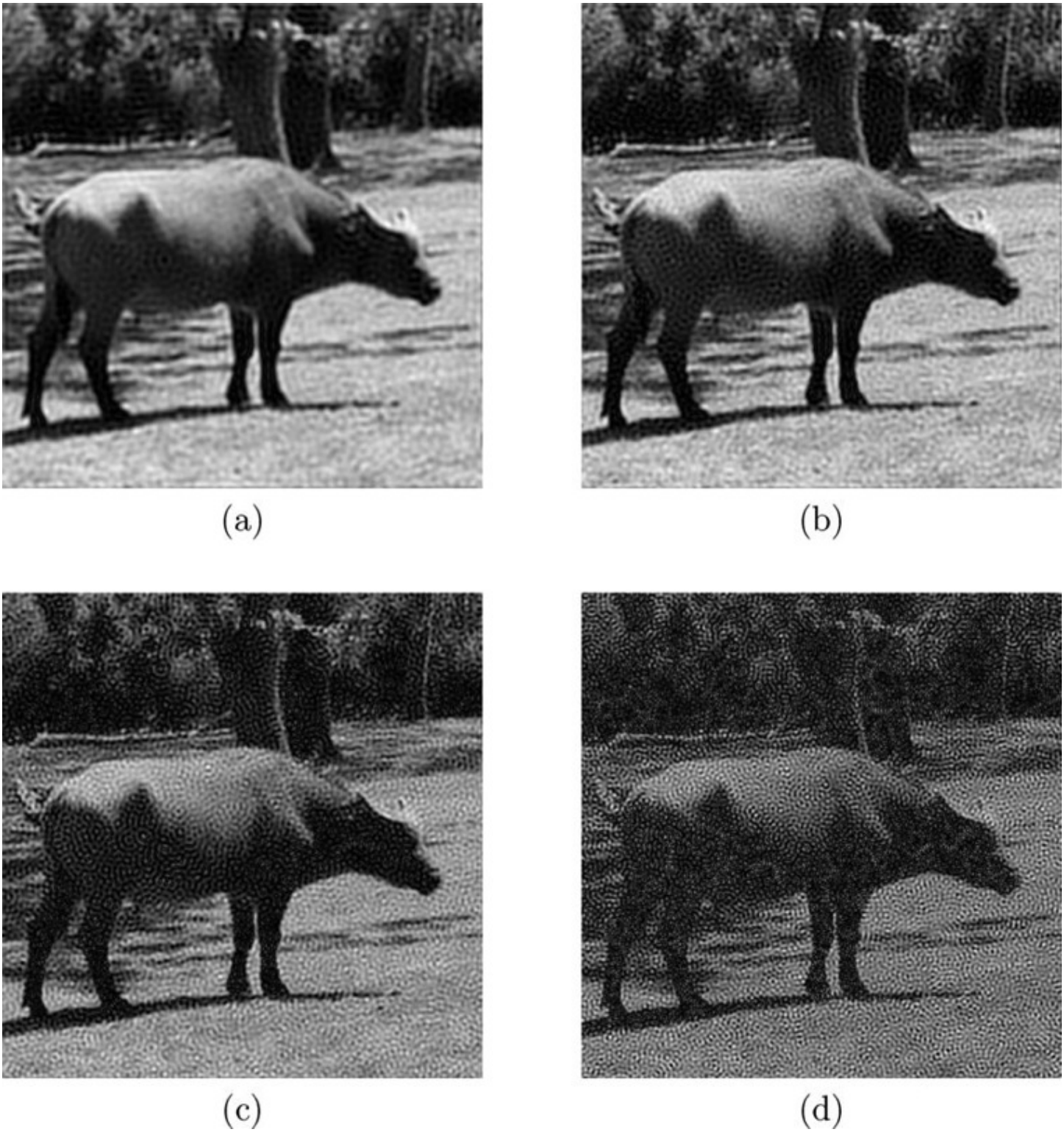
or

```
In :  d = 0.01
In :  bw = np.copy(bworth)
In :  bw[np.where(bw<d)] = 1
In :  fbw = fftshift(fft2(blur))/bw
In :  ba = abs(ifft2(fbw))
In :  io.imshow(ex.rescale_intensity(w1a,out_range=(0.0,1.0)))
```

Python

Figure 8.19 shows the results obtained by using a different cutoff radius of the Butterworth filter each time: (a) uses $d = 0.01$ (as in the MATLAB commands just given); (b) uses $d = 0.005$; (c) uses $d = 0.002$, and (d) uses $d = 0.001$. It seems that using a threshold d in the range $0.002 \leq d \leq 0.005$ produces reasonable results.

Figure 8.19: Inverse filtering using constrained division



Motion Deblurring

We can consider the removal of blur caused by motion to be a special case of inverse filtering. Start with the image `car.png` saved to `cr`, and which is shown in Figure 8.20(a). It can be blurred in MATLAB or Octave using the `motion` parameter of the `fspecial` function.

```
>> m = fspecial('motion',7,0);
>> cm = imfilter(cr,m);
```

MATLAB/Octave

In Python, such an effect can be obtained by creating a motion filter and then applying it:

```
>> m = np.ones((1,7))/7
>> cm = ndi.correlate(cr,m)
```

Python

and the result is shown as Figure 8.20(b). The result of the blur has effectively obliterated the text of the number-plate.

Figure 8.20: The result of motion blur

(a) (b)

To deblur the image, we need to divide its transform by the transform corresponding to the blur filter. So the first step is to create a matrix corresponding to the transform of the blur:

```
>> m2 = zeros(size(cr));
>> m2(1,1:7) = m;
>> mf = fft2(m2);
```

MATLAB/Octave

or

```
In :  m2 = np.zeros_like(cr)
In :  m2[1,0:7] = m
In :  mf = fft2(m2)
```

Python

The second step is dividing by this transform.

```
>> bmi = ifft2(fft2(cm)./mf);
>> fftshow(bmi,'abs')
```

MATLAB/Octave

or

```
In :  bmi = ifft2(fft2(cm)/mf)
In :  io.imshow(ex.rescale_intensity(abs(bmi),out_range=(0,1)))
```

Python

and the result is shown in Figure 8.21(a). As with inverse filtering, the result is not particularly good because the values close to zero in the matrix mf have tended to dominate the result. As above, we can constrain the division by only dividing by values that are above a certain threshold.

```
>> d = 0.02;
>> mf = fft2(m2); mf(find(abs(mf)<d)) = 1;
>> bmi = ifft2(fft2(cm)./mf);
>> imshow(mat2gray(abs(bmi))*2)
```

MATLAB/Octave

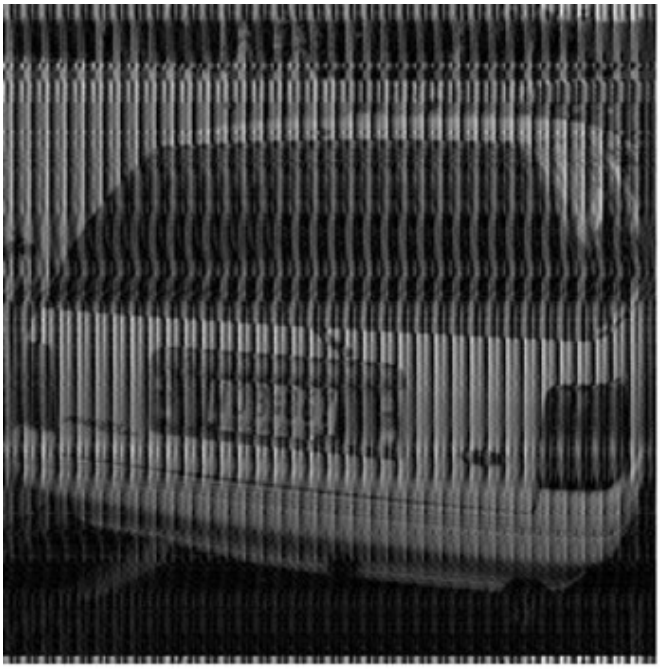
where the last multiplication by 2 just brightens the result, or

```
In :  d = 0.02
In :  mf = fft2(m2)
In :  mf[np.where(abs(mf)<d)] = 1
In :  bmi = abs(ifft2(fft2(cm)/mf))
In :  bmu = ut.img_as_ubyte(bmi/bmi.max())
In :  io.imshow(ex.rescale_intensity(bmu,in_range=(0,128)))
```

Python

where the use of a limited `in_range` scales the image to be brighter. The output is shown in Figure 8.21(b). The image is not perfect—there are still vertical artifacts—but the number plate is now quite legible.

Figure 8.21: Attempts at removing motion blur



(a) Straight division

(b) Constrained division

8.7 Wiener Filtering

As we have seen from the previous section, inverse filtering does not necessarily produce particularly pleasing results. The situation is even worse if the original image has been corrupted by noise. Here we would have an image X filtered with a filter F and corrupted by noise N. If the noise is additive (for example, Gaussian noise), then the linearity of the Fourier transform gives us

$$Y(i,j) = X(i,j)F(i,j) + N(i,j)$$

and so

$$X(i,j) = \frac{Y(i,j) - N(i,j)}{F(i,j)}$$

as we have seen in the introduction to this chapter. So not only do we have the problem of dividing by the filter, we have the problem of dealing with noise. In such a situation, the presence of noise can have a catastrophic effect on the inverse filtering: the noise can completely dominate the output, making direct inverse filtering impossible.

To introduce Wiener filtering, we shall discuss a more general question: given a degraded image M' of some original image M and a restored version R, what measure can we use to say whether our restoration has done a good job? Clearly we would like R to be as close as possible to the "correct" image M. One way of measuring the closeness of R to M is by adding the squares of all differences:

$$\sum (m_{i,j} - r_{i,j})^2$$

where the sum is taken over all pixels of R and M (which we assume to be of the same size). This sum can be taken as a measure of the closeness of R to M. If we can minimize this value, we may be sure that our

procedure has done as good a job as possible. Filters that operate on this principle of *least squares* are called *Wiener filters*. We can obtain X by

$$X(i,j) \approx \left[\frac{1}{F(i,j)} \frac{|F(i,j)|^2}{|F(i,j)|^2 + K}\right] Y(i,j) \tag{8.2}$$

where K is a constant [13]. This constant can be used to approximate the amount of noise: if the variance σ^2 of the noise is known, then $K = 2\sigma^2$ can be used. Otherwise, K can be chosen interactively (in other words, by trial and error) to yield the best result. Note that if $K = 0$, then Equation 8.2 reduces to Equation 8.1.

We can easily implement Equation 8.2 in MATLAB/Octave:

```
>> K = 0.01;
>> bf = fftshift(fft2(blur));
>> w1 = wbf.*(abs(b).^2./(abs(b).^2+K)./b);  % This is the equation
>> w1a = abs(ifft2(w1));
>> imshow(mat2gray(w1a))
```

MATLAB/Octave

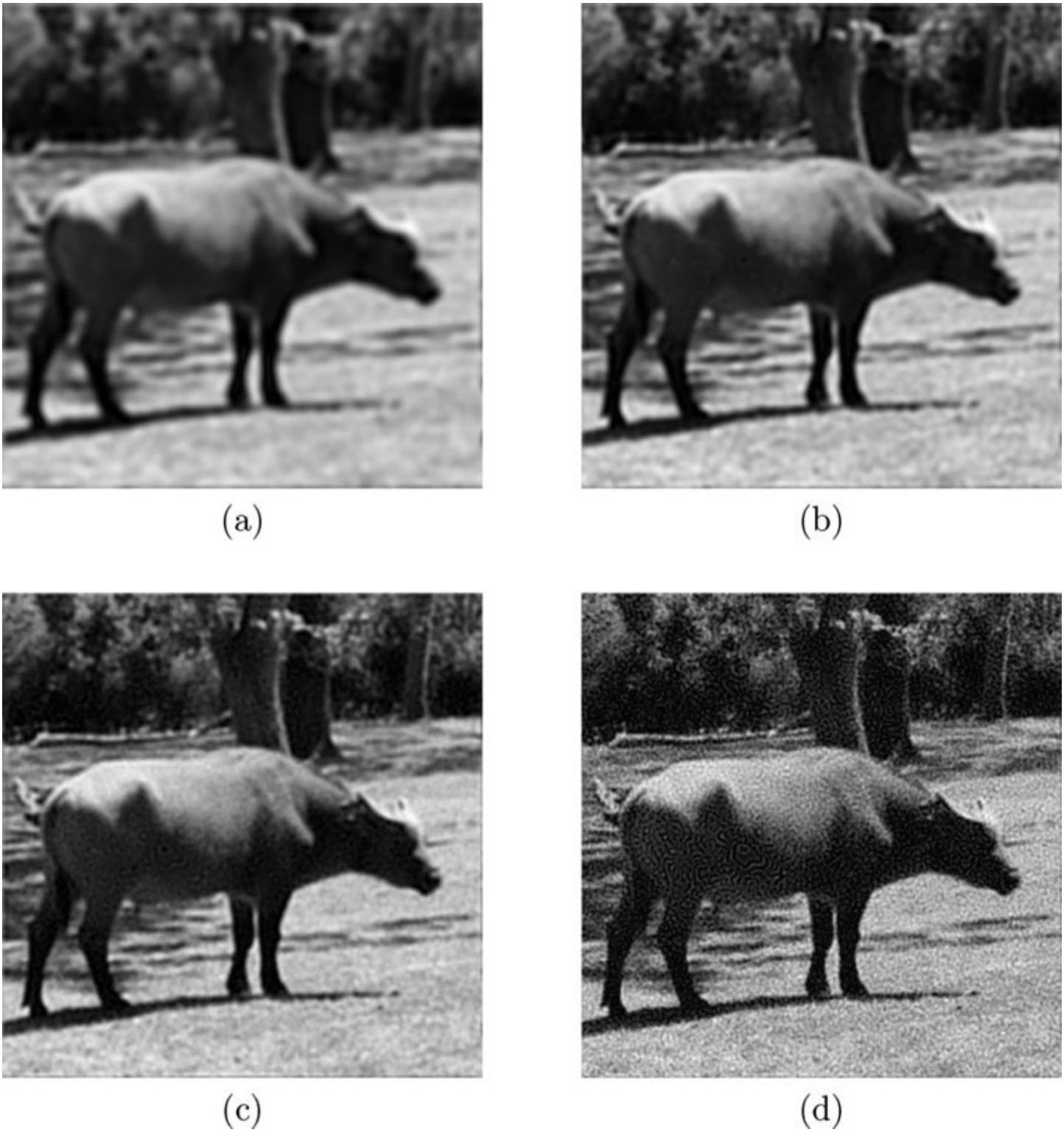
or in Python:

```
In :  K = 0.01
In :  bf = fftshift(fft2(blur))
In :  w1 = bf*(abs(bworth)**2/(abs(bworth)**2+K)/bworth)
In :  w1a = abs(ifft2(w1))
In :  io.imshow(w1a)
```

Python

The result is shown in Figure 8.22(a). Images (b), (c), and (d) in this figure show the results with K = 0.001, K = 0.0001, and K = 0.00001 respectively. Thus, as K becomes very small, noise starts to dominate the image.

Figure 8.22: Wiener filtering



Exercises

1. The following arrays represent small grayscale images. Compute the 4 × 4 image that would result in each case if the middle 16 pixels were transformed using a 3 × 3 median filter:

8	17	4	10	15	12
10	12	15	7	3	10
15	10	50	5	3	12
4	8	11	4	1	8
16	7	4	3	0	7
16	24	19	3	20	10

1	1	2	5	3	1
3	20	5	6	4	6
4	6	4	20	2	2
4	3	3	5	1	5
6	5	20	2	20	2
6	3	1	4	1	2

7	8	11	12	13	9
8	14	0	9	7	10
11	23	10	14	1	8
14	7	11	8	9	11
13	13	18	10	7	12
9	11	14	12	13	10

2. Using the same images as in Question 1, transform them by using a 3 × 3 averaging filter.
3. Use the outlier method to find noisy pixels in each of the images given in Question 1. What are the reasonable values to use for the difference between the gray value of a pixel and the average of its eight 8-neighbors?
4. Pratt [35] has proposed a "pseudo-median" filter, in order to overcome some of the speed disadvantages of the median filter. For example, given a five-element sequence {*a, b, c, d, e*}, its pseudo-median is defined as

$$\begin{aligned}\text{psmed}(a,b,c,d,e) &= \tfrac{1}{2}\max\Big[\min(a,b,c),\min(b,c,d),\min(c,d,e)\Big] \\ &+ \tfrac{1}{2}\min\Big[\max(a,b,c),\max(b,c,d),\max(c,d,e)\Big]\end{aligned}$$

So for a sequence of length 5, we take the maxima and minima of all subsequences of length three. In general, for an odd-length sequence L of length $2n + 1$, we take the maxima and minima of all subsequences of length $n + 1$. We can apply the pseudo-median to 3 × 3 neighborhoods of an image, or cross-shaped neighborhoods containing 5 pixels, or any other neighborhood with an odd number of pixels. Apply the pseudo-median to the images in Question 1, using 3 × 3 neighborhoods of each pixel.
5. Write a function to implement the pseudo-median, and apply it to the images above. Does it produce a good result?
6. Choose any grayscale image of your choice and add 5% salt and pepper noise to the image. Attempt to remove the noise with

   (a) Average filtering
   (b) Median filtering
   (c) The outlier method
   (d) Pseudo-median filtering

   Which method gives the best results?

- 7. Repeat the above question but with 10%, and then with 20% noise.
- 8. For 20% noise, compare the results with a 5 × 5 median filter, and two applications of a 3 × 3 median filter.
- 9. Add Gaussian noise to your chosen image with the following parameters:

   (a) Mean 0, variance 0.01 (the default)
   (b) Mean 0, variance 0.02
   (c) Mean 0, variance 0.05
   (d) Mean 0, variance 0.1

In each case, attempt to remove the noise with average filtering and with Wiener filtering. Can you produce satisfactory results with the last two noisy images?

- 10. Gonzalez and Woods [13] mention the use of a *midpoint filter* for cleaning Gaussian noise. This is defined by $g(x,y) = \frac{1}{2}\left(\max_{(x,y)\in B} f(x,y) + \min_{(x,y)\in B} f(x,y)\right)$ where the maximum and minimum are taken over all pixels in a neighborhood B of (x, y). Use rank-order filtering to find maxima and minima, and experiment with this approach to cleaning Gaussian noise, using different variances. Visually, how do the results compare with spatial Wiener filtering or using a blurring filter?
- 11. In Chapter 5 we defined the alpha-trimmed mean filter, and the geometric mean filter. Write functions to implement these filters, and apply them to images corrupted with Gaussian noise. How well do they compare to average filtering, image averaging, or adaptive filtering?
- 12. Add the sine waves to an image of your choice by using the same commands as above, but with the final command `s = 1+sin(x+y/1.5);` Now attempt to remove the noise using band-reject filtering or criss-cross filtering. Which one gives the best result?
- 13. For each of the following sine commands:

(a) `s = 1+sin(x/3+y/5)`
(b) `s = 1+sin(x/5+y/1.5)`
(c) `s = 1+sin(x/6+y/6)`

add the sine wave to the image as shown in the previous question, and attempt to remove the resulting periodic noise using band-reject filtering or notch filtering. Which of the three is easiest to "clean up"?

- 14. Apply a 5 × 5 blurring filter to a grayscale image of your choice. Attempt to deblur the result using inverse filtering with constrained division. Which threshold gives the best results?
- 15. Repeat the previous question using a 7 × 7 blurring filter.
- 16. Work through the motion deblurring example, experimenting with different values of the threshold. What gives the best results?

9 Image Segmentation

9.1 Introduction

Segmentation refers to the operation of partitioning an image into component parts or into separate objects. In this chapter, we shall investigate two very important topics: thresholding and edge detection.

9.2 Thresholding

Single Thresholding

A grayscale image is turned into a binary (black and white) image by first choosing a gray level T in the original image, and then turning every pixel black or white according to whether its gray value is greater than or less than T:

$$\text{A pixel becomes } \begin{cases} \text{white if its gray level is } > T, \\ \text{black if its gray level is } \leq T. \end{cases}$$