```
12   6   5  13  14  14  16  15
11  10   8   5   8  11  14  14
 9   8   3   4   7  12  18  19
10   7   4   2  10  12  13  17
16   9  13  13  16  19  19  17
12  10  14  15  18  18  16  14
11   8  10  12  14  13  14  15
 8   6   3   7   9  11  12  12
```

8. Is the histogram equalization operation idempotent? That is, is performing histogram equalization *twice* the same as doing it just once?
9. Apply histogram equalization to the indices of the image `emu.png`.
10. Create a dark image with

```
>> c = imread('cameraman.png');
>> [x,map] = gray2ind(c);
```
MATLAB/Octave

The matrix `x`, when viewed, will appear as a very dark version of the cameraman image. Apply histogram equalization to it and compare the result with the original image.
11. Using either `c` and `ch or s` and `sh` from Section 4.3, enter the command

```
>> figure,plot(c,ch,'.'),grid on
```
MATLAB/Octave

or

```
In :  plt.plot(c,ch,'.'),plt.grid('on'),plt.axis('image')
```
Python

What are you seeing here?
12. Experiment with some other grayscale images.
13. Using LUTs, and following the example given in Section 4.4, write a simpler function for performing piecewise stretching than the function described in Section 4.3.

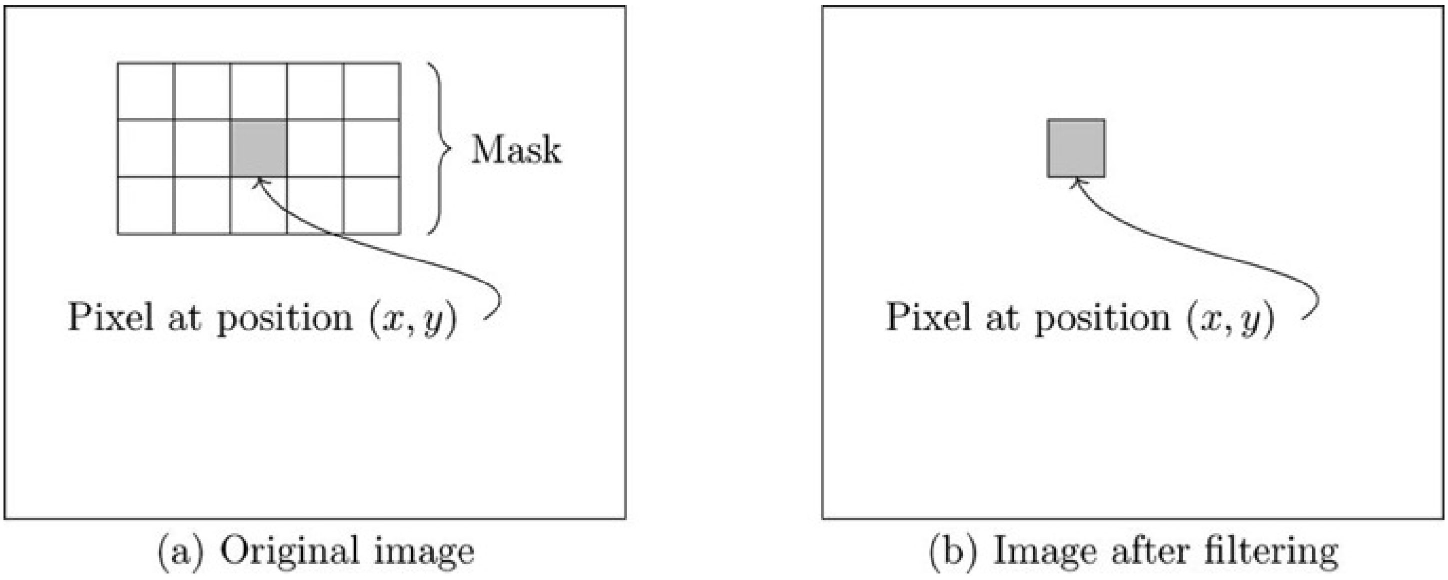
5 Neighborhood Processing

5.1 Introduction

We have seen in Chapter 4 that an image can be modified by applying a particular function to each pixel value. Neighborhood processing may be considered an extension of this, where a function is applied to a neighborhood of each pixel.

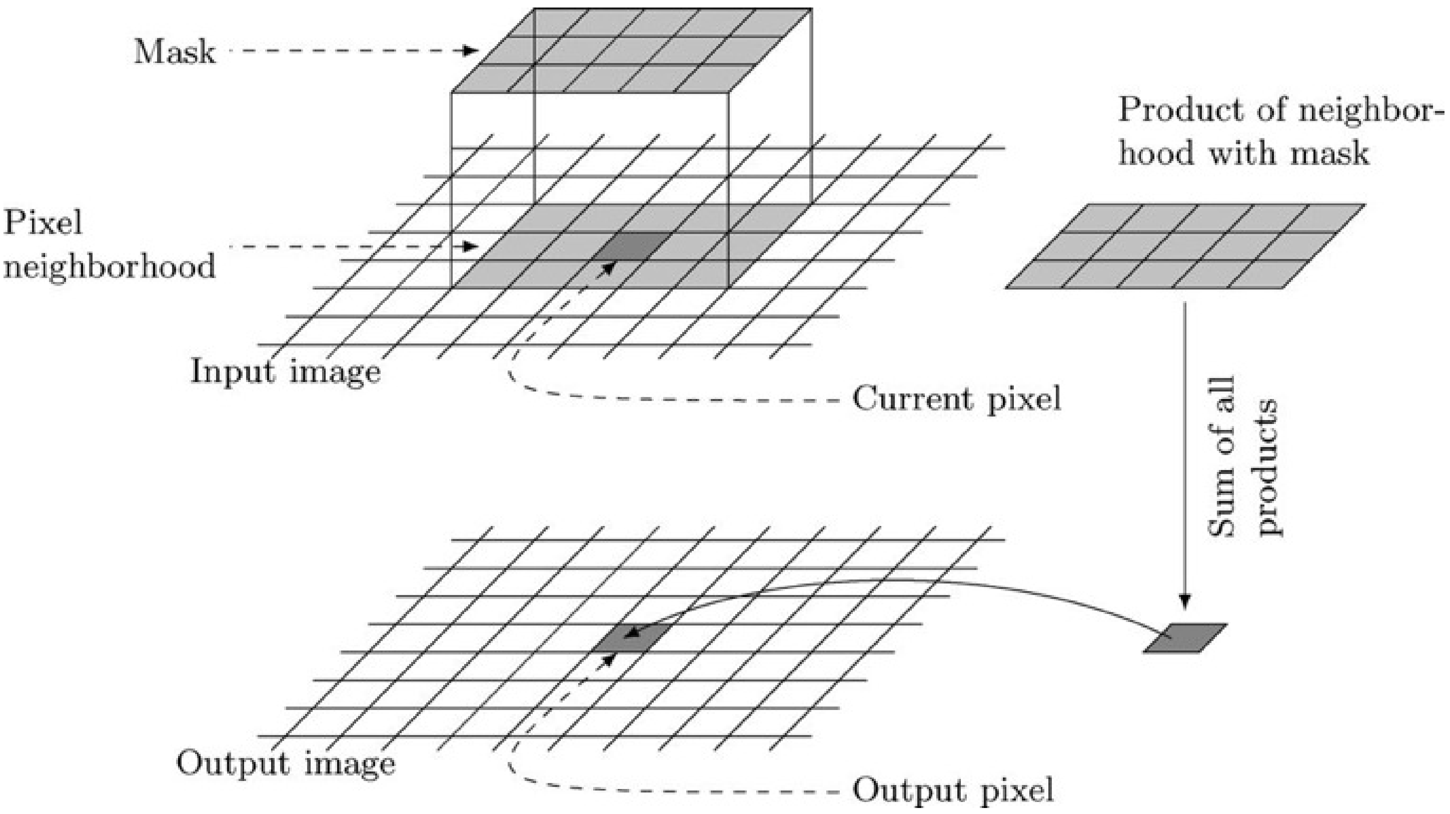
The idea is to move a “mask”: a rectangle (usually with sides of odd length) or other shape over the given image. As we do this, we create a new image whose pixels have gray values calculated from the gray values under the mask, as shown in Figure 5.1. The combination of mask and function is called a *filter*. If the function by which the new gray value is calculated is a linear function of all the gray values in the mask, then the filter is called a *linear filter*.

Figure 5.1: Using a spatial mask on an image


Mask
Pixel at position (x, y)
(a) Original image
Pixel at position (x, y)
(b) Image after filtering

A linear filter can be implemented by multiplying all elements in the mask by corresponding elements in the neighborhood, and adding up all these products. Suppose we have a 3 × 5 mask as illustrated in Figure 5.1. Suppose that the mask values are given by:

Figure 5.2: Performing linear spatial filtering



$m(-1,-2)$	$m(-1,-1)$	$m(-1,0)$	$m(-1,1)$	$m(-1,2)$
$m(0,-2)$	$m(0,-1)$	$m(0,0)$	$m(0,1)$	$m(0,2)$
$m(1,-2)$	$m(1,-1)$	$m(1,0)$	$m(1,1)$	$m(1,2)$

and that corresponding pixel values are

$p(i-1,j-2)$	$p(i-1,j-1)$	$p(i-1,j)$	$p(i-1,j+1)$	$p(i-1,j+2)$
$p(i,j-2)$	$p(i,j{-}1)$	$p(i,j)$	$p(i,j+1)$	$p(i,j+2)$
$p(i+1,j-2)$	$p(i{+}1,j-1)$	$p(i+1,j)$	$p(i+1,j+1)$	$p(i+1,j+2)$

We now multiply and add:

$$\sum_{s=-1}^{1}\sum_{t=-2}^{2} m(s,t)p(i+s,j+t).$$

A diagram illustrating the process for performing spatial filtering is given in Figure 5.2.

Spatial filtering thus requires three steps:

1. Position the mask over the current pixel
2. Form all products of filter elements with the corresponding elements of the neighborhood
3. Add up all the products

This must be repeated for every pixel in the image.

Allied to spatial filtering is spatial *convolution*. The method for performing a convolution is the same as that for filtering, except that the filter must be rotated by 180° before multiplying and adding. Using the $m(i,j)$ and $p(i,j)$ notation as before, the output of a convolution with a 3×5 mask for a single pixel is

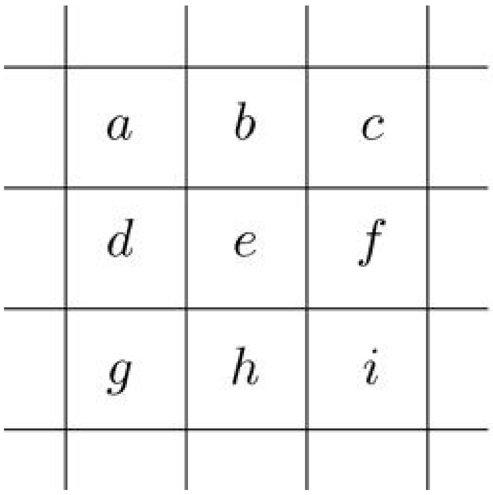
$$\sum_{s=-1}^{1}\sum_{t=-2}^{2} m(-s,-t)p(i+s,j+t).$$

Note the negative signs on the indices of m. The same result can be achieved with

$$\sum_{s=-1}^{1}\sum_{t=-2}^{2} m(s,t)p(i-s+j-t).$$

Here we have rotated the *image* pixels by 180°; this does not of course affect the result. The importance of convolution will become apparent when we investigate the Fourier transform, and the convolution theorem. Note also that in practice, most filter masks are rotationally symmetric, so that spatial filtering and spatial convolution will produce the same output. Filtering as described above, where the filter is *not* rotated by 180°, is also called *correlation*.

An example: One important linear filter is to use a 3 × 3 mask and take the average of all nine values within the mask. This value becomes the gray value of the corresponding pixel in the new image. This operation may be described as follows:



$$\longrightarrow \tfrac{1}{9}(a+b+c+d+e+f+g+h+i)$$

where *e* is the gray value of the current pixel in the original image, and the average is the gray value of the corresponding pixel in the new image.

To apply this to an image, consider the 5 × 5 "image" obtained by multiplying a magic square by 10. In MATLAB/Octave:

```
>> x=uint8(10*magic(5))
x =

   170   240    10    80   150
   230    50    70   140   160
    40    60   130   200   220
   100   120   190   210    30
   110   180   250    20    90
```

MATLAB/Octave

and in Python:

```
In:  x = 10*uint8(array([[17,24,1,8,15],[23,5,7,14,16],\
...: [4,6,13,20,22],[10,12,19,21,3],[11,18,25,2,9]]))
In: x
Out:
array([[170, 240,  10,  80, 150],
       [230,  50,  70, 140, 160],
       [ 40,  60, 130, 200, 220],
       [100, 120, 190, 210,  30],
       [110, 180, 250,  20,  90]], dtype=uint8)
```
Python

We may regard this array as being made of nine overlapping 3 × 3 neighborhoods. The output of our working will thus consist only of nine values. We shall see later how to obtain 25 values in the output.

Consider the top left 3 × 3 neighborhood of our image x:

$$\begin{array}{|ccc|cc}
\cline{1-3}
170 & 240 & 10 & 80 & 150 \\
230 & 50 & 70 & 140 & 160 \\
40 & 60 & 130 & 200 & 220 \\
\cline{1-3}
\multicolumn{1}{c}{100} & 120 & \multicolumn{1}{c}{190} & 210 & 30 \\
\multicolumn{1}{c}{110} & 180 & \multicolumn{1}{c}{250} & 20 & 90
\end{array}$$

Now we take the average of all these values in MATLAB/Octave as

```
>> mean2(x(1:3,1:3))

ans =

  111.1111
```
MATLAB/Octave

or in Python as

```
In : x[0:3,0:3].mean()
Out: 111.11111111111111
```
Python

which can be rounded to 111. Now we can move to the second neighborhood:

$$\begin{array}{c|ccc|c}
\cline{2-4}
170 & 240 & 10 & 80 & 150 \\
230 & 50 & 70 & 140 & 160 \\
40 & 60 & 130 & 200 & 220 \\
\cline{2-4}
100 & 120 & 190 & 210 & 30 \\
110 & 180 & 250 & 20 & 90
\end{array}$$

and take its average in MATLAB/Octave:

```
>> mean2(x(1:3,2:4))

ans =

  108.8889
```

MATLAB/Octave

and in Python:

```
In : x[0:3,1:4].mean()
Out: 108.88888888888889
```

Python

and this can be rounded either down to 108, or to the nearest integer 109. If we continue in this manner, the following output is obtained:

```
111.1111  108.8889  128.8889
110.0000  130.0000  150.0000
131.1111  151.1111  148.8889
```

This array is the result of filtering x with the 3 × 3 averaging filter.

5.2 Notation

It is convenient to describe a linear filter simply in terms of the coefficients of all the gray values of pixels within the mask. This can be written as a matrix.

The averaging filter above, for example, could have its output written as

$$\frac{1}{9}a+\frac{1}{9}b+\frac{1}{9}c+\frac{1}{9}d+\frac{1}{9}e+\frac{1}{9}f+\frac{1}{9}g+\frac{1}{9}h+\frac{1}{9}i$$

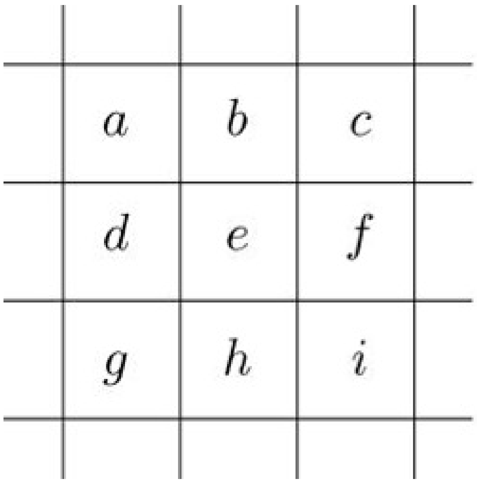
and so this filter can be described by the matrix

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

An example: The filter

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

would operate on gray values as

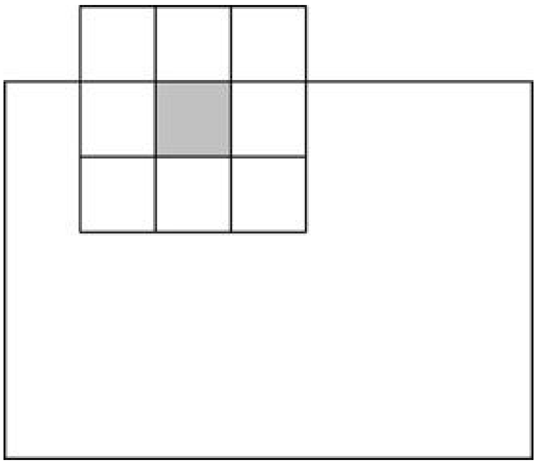


$$\longrightarrow a - 2b + c - 2d + 4e - 2f + g - 2h + i$$

Borders of the Image

There is an obvious problem in applying a filter—what happens at the border of the image, where the mask partly falls outside the image? In such a case, as illustrated in Figure 5.3 there will be a lack of gray values to use in the filter function.

Figure 5.3: A mask at the edge of an image



There are a number of different approaches to dealing with this problem:

Ignore the edges. That is, the mask is only applied to those pixels in the image for which the mask will lie fully within the image. This means all pixels except for those along the borders, and results in an output image that is smaller than the original. If the mask is very large, a significant amount of information may be lost by this method.

We applied this method in our example above.

“Pad” with zeros. We assume that all necessary values outside the image are zero. This gives us all values to work with, and will return an output image of the same size as the original, but may have the effect of introducing unwanted artifacts (for example, dark borders) around the image.

Repeat the image. This means tiling the image in all directions, so that the mask always lies over image values. Since values might thus change across edges, this method may introduce unwanted artifacts in the result.

Reflect the image. This is similar to repeating, except that the image is reflected across all of its borders. This will ensure that there are no extra sudden changes introduced at the borders.

The last two methods are illustrated in Figure 5.4.

We can also see this by looking at the image which increases from top left to bottom right:

```
 20   40   60   80  100
 40   60   80  100  120
 60   80  100  120  140
 80  100  120  140  160
100  120  140  160  180
```

The results of filtering with a 3 × 3 averaging filter with zero padding, reflection, and repetition are shown in Figure 5.5.

Note that with repetition the upper left and bottom right values are “wrong”; this is because they have taken as extra values pixels that are very different. With zero-padding the upper left value has the correct size, but the bottom right values are far smaller than they should be, as they have been convolved with zeros. With reflection the values are all the correct size.

Figure 5.4: Repeating an image for filtering at its borders

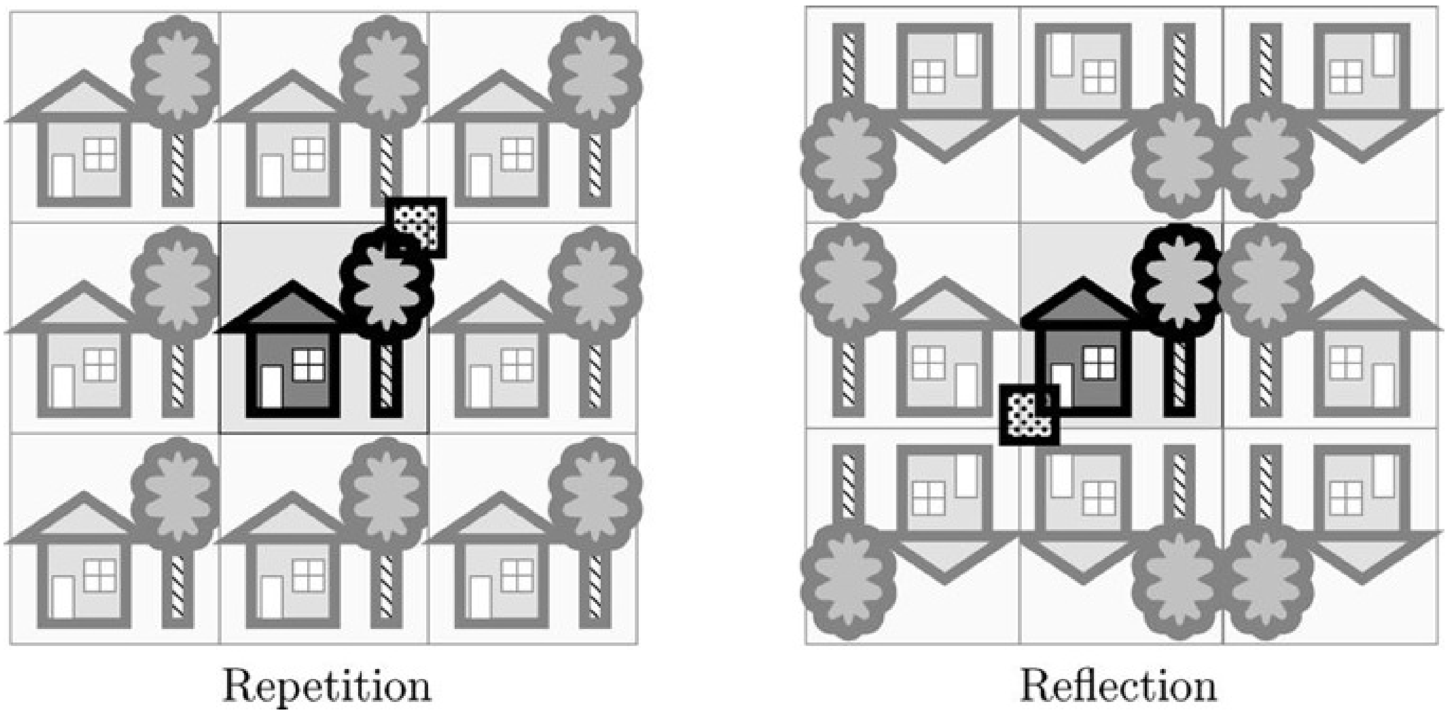


Figure 5.5: The result of filtering with different modes

```
17   33   46   59   44 |  86   73   93  113   99 |  32   46   66   86   99
33   60   80  100   73 |  73   60   80  100   86 |  46   60   80  100  113
46   80  100  120   86 |  93   80  100  120  106 |  66   80  100  120  133
60  100  120  140  100 | 113  100  120  140  126 |  86  100  120  140  153
44   73   86   99   71 |  99   86  106  126  112 |  99  113  133  153  166
    Zero-padding       |       Repetition        |       Reflection
```

5.3 Filtering in MATLAB and Octave

The `imfilter` function does the job of linear filtering for us; its use is

```
imfilter(image,filter,...)
```

and the result is a matrix of the same data type as the original image. There are other parameters for controlling the behavior of the filtering. Pixels at the boundary of the image can be managed by padding with zeros; this being the default method, but there are several other options:

Extra parameter	Implements
`'symmetric'`	Filtering with reflection
`'circular'`	Filtering with tiling repetition
`'replication'`	Filtering by repeating the border elements.

Extra parameter	Implements
`'full'`	Padding with zero, and applying the filter at all places on and around the image where the mask intersects the image matrix.

The last option returns a result that is larger than the original:

```
>> imfilter(x,a,'full')

ans =

   19   46   47   37   27   26   17
   44   77   86   66   68   59   34
   49   88  111  109  129  106   59
   41   67  110  130  150  107   46
   28   68  131  151  149   86   38
   23   57  106  108   88   39   13
   12   32   60   50   40   12   10
```

MATLAB/Octave

The central 5 × 5 values of the last operation are the same values as provided by the command `imfilter(x,a)` and with no extra parameters.

There is no single “best” approach; the method must be dictated by the problem at hand, by the filter being used, and by the result required.

We can create our filters by hand or by using the `fspecial` function; this has many options which makes for easy creation of many different filters. We shall use the average option, which produces `averaging` filters of given size; thus,

```
>> fspecial('average',[5,7])
```

MATLAB/Octave

will return an averaging filter of size 5 × 7; more simply

```
>> fspecial('average',11)
```

MATLAB/Octave

will return an averaging filter of size 11 × 11. If we leave out the final number or vector, the 3 × 3 averaging filter is returned.

For example, suppose we apply the 3 × 3 averaging filter to an image as follows:

```
>> c = imread('cameraman.png');
>> f1 = fspecial('average');
>> cf1 = imfilter(c,f1);
```

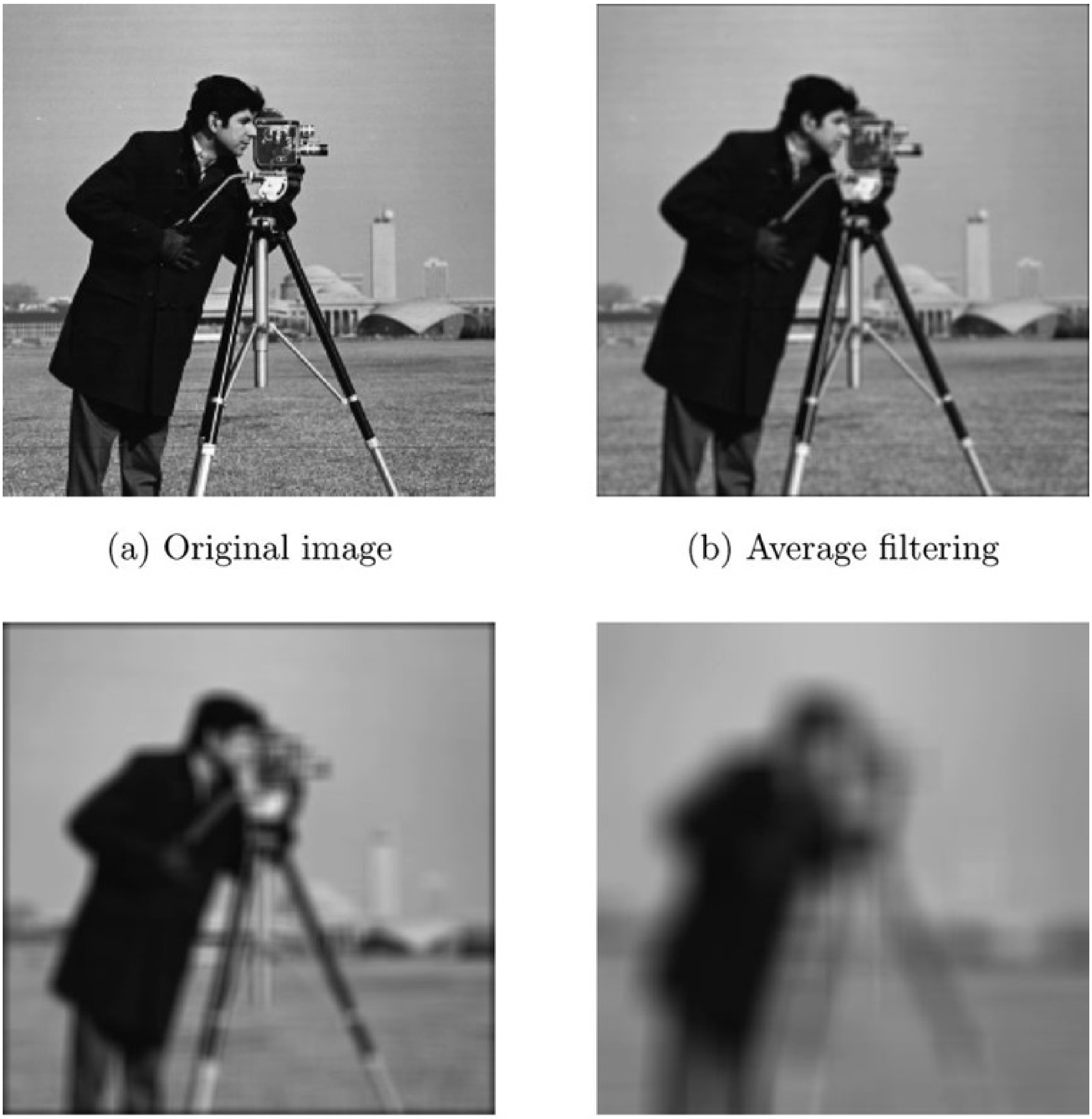
MATLAB/Octave

The averaging filter blurs the image; the edges in particular are less distinct than in the original. The image can be further blurred by using an averaging filter of larger size. This is shown in Figure 5.6(c), where a 9 × 9 averaging filter has been used, and in Figure 5.6(d), where a 25 × 25 averaging filter has been used.

Notice how the default zero padding used at the edges has resulted in a dark border appearing around the image (c). This would be especially noticeable when a large filter is being used. Any of the above options can be used instead; image (d) was created with the `symmetric` option.

The resulting image after these filters may appear to be much "worse" than the original. However, applying a blurring filter to reduce detail in an image may the perfect operation for autonomous machine recognition, or if we are only concentrating on the "gross" aspects of the image: numbers of objects or amount of dark and light areas. In such cases, too much detail may obscure the outcome.

Figure 5.6: Average filtering



(a) Original image

(b) Average filtering

(c) Using a 9×9 filter

(d) Using a 25×25 filter

Separable Filters

Some filters can be implemented by the successive application of two simpler filters. For example, since

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \frac{1}{3}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix}\frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$$

the 3 × 3 averaging filter can be implemented by first applying a 3 × 1 averaging filter, and then applying a 1 × 3 averaging filter to the result. The 3 × 3 averaging filter is thus *separable* into two smaller filters. Separability can result in great time savings. Suppose an n x n filter is separable into two filters of size n x 1 and 1 x n. The application of an n x n filter requires n^2 multiplications, and $n^2 - 1$ additions for each pixel in the image. But the application of an n x 1 filter only requires n multiplications and $n - 1$ additions. Since this must be done twice, the total number of multiplications and additions are $2n$ and $2n - 2$, respectively. If n is large the savings in efficiency can be dramatic.

All averaging filters are separable; another separable filter is the Laplacian

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}.$$

We will also consider other examples.

5.4 Filtering in Python

There is no Python equivalent to the `fspecial` and `imfilter` commands, but there are a number of commands for applying either a generic filter or a specific filter.

Linear filtering can be performed by using `convolve` or `correlate` from the `ndimage` module of the library, or `generic_filter`.

For example, we may apply 3 × 3 averaging to the 5 × 5 magic square array:

```
In:  import scipy.ndimage as ndi
In:  x = uint8(array([[17,24,1,8,15],[23,5,7,14,16],[4,6,13,20,22],\
     [10,12,19,21,3],[11,18,25,2,9]])*10)
In:  a = ones((3,3))/9
In:  ndi.convolve(x,a,mode='constant')
Out:
array([[ 76,  85,  65,  67,  58],
       [ 87, 111, 108, 128, 105],
       [ 66, 109, 130, 150, 106],
       [ 67, 131, 151, 148,  85],
       [ 56, 105, 107,  87,  38]], dtype=uint8)
                                                              Python
```

Note that this result is the same as that obtained with MATLAB's `imfilter(x,a)` command, and Python also automatically converts the result to the same data type as the input; here as an unsigned 8-bit array. The `mode` parameter, here set to `'constant'`, tells the function that the image is to be padded with constant values, the default value of which is zero, although other values can be specified with the `cval` parameter.

So to obtain, for example, the image in Figure 5.6(c), you could use the following Python commands:

```
In:  c = io.imread('cameraman.png')
In:  cf = ndi.convolve(c,ones((9,9))/81,mode='constant')
In:  io.imshow(cfs)
```
Python

Alternately, you could use the built in `uniform_filter` function:

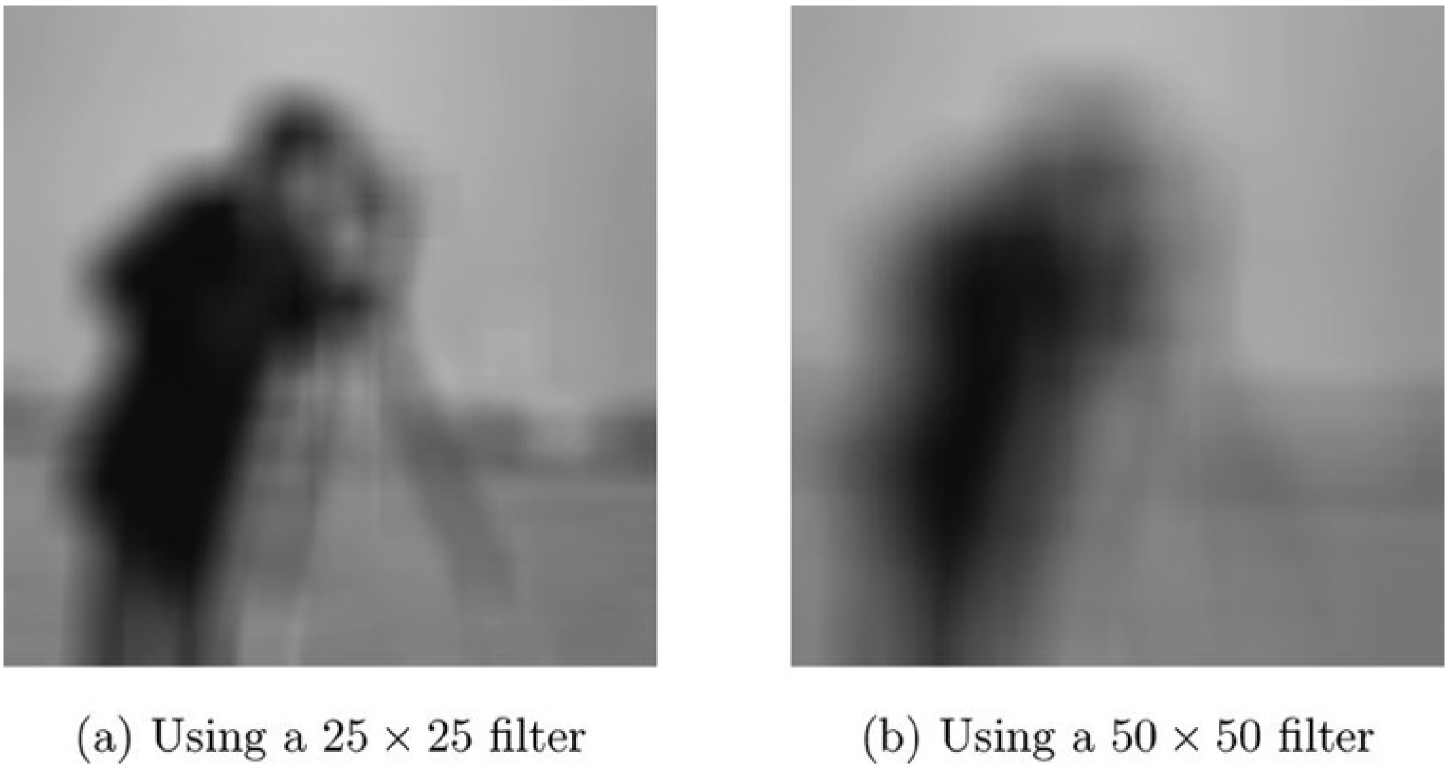
```
In:  cf = ndi.uniform_filter(c,[9,9],mode='constant')
```
Python

Python differs from MATLAB and Octave here in that the default behavior of spatial convolution at the edges of the image is to reflect the image in all its edges. This will ameliorate the dark borders seen in Figure 5.6. So with leaving the `mode` parameter in its default setting, the commands

```
In:  cf = ndi.uniform_filter(c,25)
In:  cf2 = ndi.uniform_filter(c,50)
```
Python

will produce the images shown in Figure 5.7.

Figure 5.7: Average filtering in Python


(a) Using a 25×25 filter (b) Using a 50×50 filter

5.5 Frequencies; Low and High Pass Filters

It will be convenient to have some standard terminology by which we can discuss the effects a filter will have on an image, and to be able to choose the most appropriate filter for a given image processing task. One important aspect of an image which enables us to do this is the notion of `frequencies`. Roughly speaking, the frequencies of an image are a measure of the amount by which gray values change with distance. This concept will be given a more formal setting in Chapter 7. *High frequency components* are characterized by large changes in gray values over small distances; examples of high frequency components are edges and noise. *Low frequency components*, on the other hand, are parts of the image characterized by little change in the gray values. These may include backgrounds or skin textures. We then say that a filter is a

> **high pass filter** if it "passes over" the high frequency components, and reduces or eliminates low frequency components
> **low pass filter** if it "passes over" the low frequency components, and reduces or eliminates high frequency components

For example, the 3 × 3 averaging filter is a low pass filter, as it tends to blur edges. The filter

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

is a high pass filter.

We note that the sum of the coefficients (that is, the sum of all e elements in the matrix), in the high pass filter is zero. This means that in a low frequency part of an image, where the gray values are similar, the result of using this filter is that the corresponding gray values in the new image will be close to zero. To see this, consider a 4 × 4 block of similar values pixels, and apply the above high pass filter to the central four:

$$\begin{array}{|c|c|c|c|} \hline 150 & 152 & 148 & 149 \\ \hline 147 & 152 & 151 & 150 \\ \hline 152 & 148 & 149 & 151 \\ \hline 151 & 149 & 150 & 148 \\ \hline \end{array} \longrightarrow \begin{array}{|r|r|} \hline 11 & 6 \\ \hline -13 & -5 \\ \hline \end{array}$$

The resulting values are close to zero, which is the expected result of applying a high pass filter to a low frequency component. We shall see how to deal with negative values later.

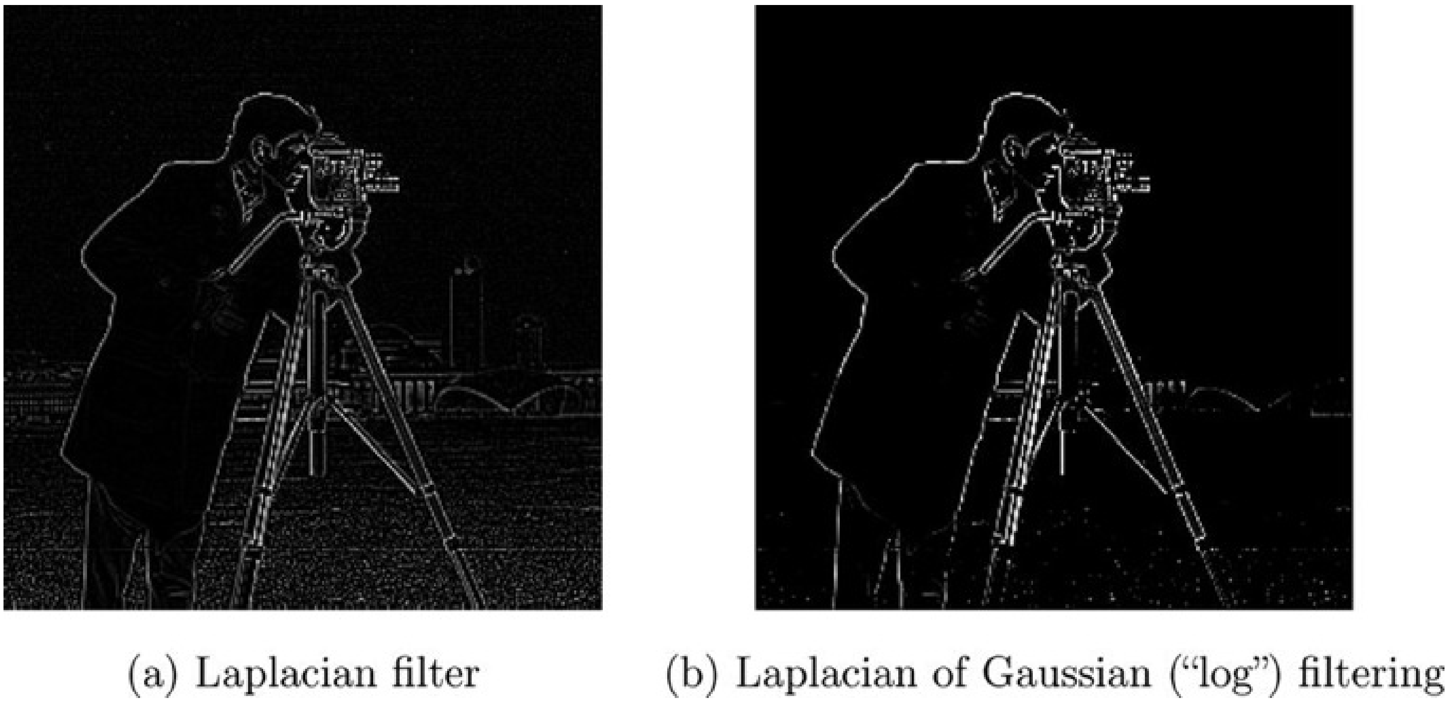
High pass filters are of particular value in edge detection and edge enhancement (of which we shall see more in Chapter 9). But we can provide a sneak preview, using the cameraman image.

```
>> f=fspecial('laplacian')

f =

    0.1667    0.6667    0.1667
    0.6667   -3.3333    0.6667
    0.1667    0.6667    0.1667

>> cf=imfilter(c,f,'symmetric');
>> f1=fspecial('log')

f1 =

    0.0448    0.0468    0.0564    0.0468    0.0448
    0.0468    0.3167    0.7146    0.3167    0.0468
    0.0564    0.7146   -4.9048    0.7146    0.0564
    0.0468    0.3167    0.7146    0.3167    0.0468
    0.0448    0.0468    0.0564    0.0468    0.0448

>> cf1=imfilter(c,f1,'symmetric');
```

MATLAB/Octave

The images are shown in Figure 5.8. Image (a) is the result of the Laplacian filter; image (b) shows the result of the Laplacian of Gaussian ("log") filter. We discuss Gaussian filters in Section 5.6.

Figure 5.8: High pass filtering


(a) Laplacian filter (b) Laplacian of Gaussian ("log") filtering

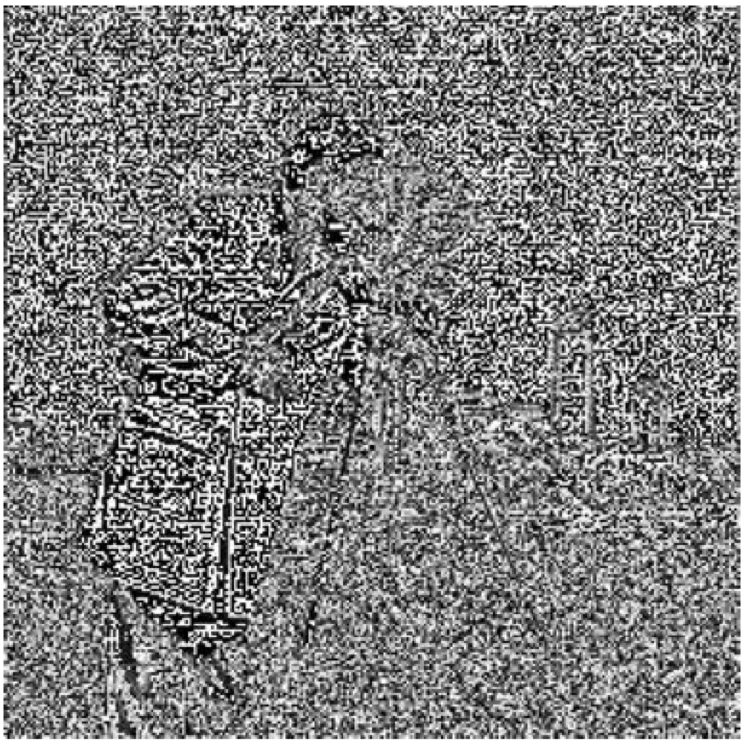
In each case, the sum of all the filter elements is zero.

Note that both MATLAB and Octave in fact do more than merely apply the Laplacian or LoG filters to the image; the results in Figure 5.8 have been "cleaned up" from the raw filtering. We can see this by applying the Laplacian filter in Python:

```
In:  f = array([[1,4,1],[4,-20,4],[1,4,1]])
In:  cf = ndi.convolve(c,f)
In:  io.imshow(cf)
```

Python

Figure 5.9: Laplacian filtering without any extra processing



of which the result is shown in Figure 5.9. We shall see in Chapter 9 how Python can be used to obtain results similar to those in Figure 5.8.

Values Outside the Range 0–255

We have seen that for image display, we would like the gray values of the pixels to lie between 0 and 255. However, the result of applying a linear filter may be values that lie outside this range. We may consider ways of dealing with values outside of this "displayable" range.

Make negative values positive. This will certainly deal with negative values, but not with values greater than 255. Hence, this can only be used in specific circumstances; for example, when there are only a few negative values, and when these values are themselves close to zero.

Clip values. We apply the following thresholding type operation to the gray values x produced by the filter to obtain a displayable value y:

$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 255 \\ 255 & \text{if } x > 255 \end{cases}$$

This will produce an image with all pixel values in the required range, but is not suitable if there are many gray values outside the 0–255 range; in particular, if the gray values are equally spread over a larger range. In such a case, this operation will tend to destroy the results of the filter.

Scaling transformation. Suppose the lowest gray value produced by the filter if g_L and the highest value is g_H. We can transform all values in the range $g_L - g_H$ to the range 0–255 by the linear transformation illustrated below: Since the gradient of the line is $255/(g_H - g_L)$ we can write the equation of the line as $y = 255\frac{x - g_L}{g_H - g_L}$ and applying this transformation to all gray levels *x* produced by the filter will result (after any necessary rounding) in an image that can be displayed.

As an example, let's apply the high pass filter given in Section 5.5 to the cameraman image:

```
>> f2 = [1 -2 1;-2 4 -2;1 -2 1];
>> cf2 = imfilter(double(c),f2);
```

MATLAB/Octave

We have to convert the image to type "double" before the filtering, or else the result will be an automatically scaled image of type `uint8`. The maximum and minimum values of the matrix `cf2` are 593 and −541, respectively. The `mat2gray` function automatically scales the matrix elements to displayable values; for any matrix *M*, it applies a linear transformation to its elements, with the lowest value mapping to 0.0, and the highest value mapping to 1.0. This means the output of `mat2gray` is always of type `double`. The function also requires that the input type is `double`.

```
>> figure,imshow(mat2gray(cf2));
```

MATLAB/Octave

To do this by hand, so to speak, applying the linear transformation above, we can use:

```
>> maxcf2 = max(cf2(:));
>> mincf2 = min(cf2(:));
>> cf2g = (cf2-mincf2)/(maxcf2-mncf2);
```

MATLAB/Octave

The result will be a matrix of type `double`, with entries in the range 0.0–1.0. This can be be viewed with `imshow`. We can make it a `uint8` image by multiplying by 255 first. The result can be seen in Figure 5.10.

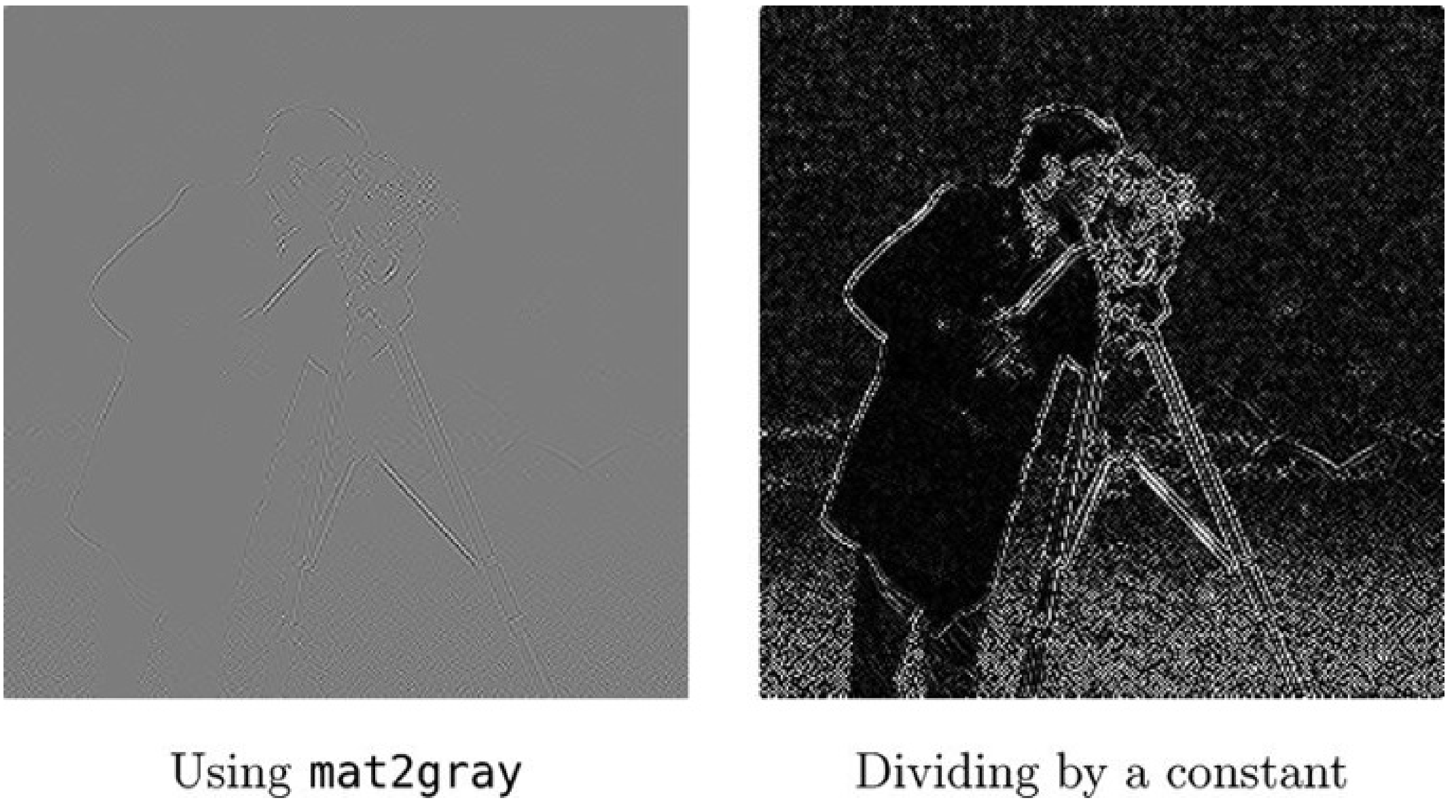
Sometimes a better result can be obtained by dividing an output of type `double` by a constant before displaying it:

```
>> figure,imshow(cf2/60)
```

MATLAB/Octave

and this is also shown in Figure 5.10.

Figure 5.10: Using a high pass filter and displaying the result



High pass filters are often used for edge detection. These can be seen quite clearly in the right-hand image of Figure 5.10.

In Python, as we have seen, the `convolve` operation returns a `uint8` array as output if the image is of type `uint8`. To apply a linear transformation, we need to start with an output of type `float32`:

```
In:  cf2 = ndi.convolve(float32(c),f,mode='constant')
In:  maxcf2 = cf2.max()
In:  mincf2 = cf1.min()
In:  cf2f = (cf2-mincf2)/(maxcf2-mincf2)
In:  io.imshow(cf2f)
```

Python

Note that Python's `imshow` commands automatically scale the image to full brightness range. To switch off that behavior the parameters `vmax` and `vmin` will give the "true" range, as opposed to the displayed range:

```
In:  io.imshow(cf2/60,vmax=1.0,vmin=0.0)
```

Python

These last two `imshow` commands will produce the same images as shown in Figure 5.10.

5.6 Gaussian Filters

We have seen some examples of linear filters so far: the averaging filter and a high pass filter. The `fspecial` function can produce many different filters for use with the `imfilter` function; we shall look at a particularly important filter here.

Gaussian filters are a class of low pass filters, all based on the Gaussian probability distribution function

$$f(x) = e^{-\frac{x^2}{2\sigma^2}}$$

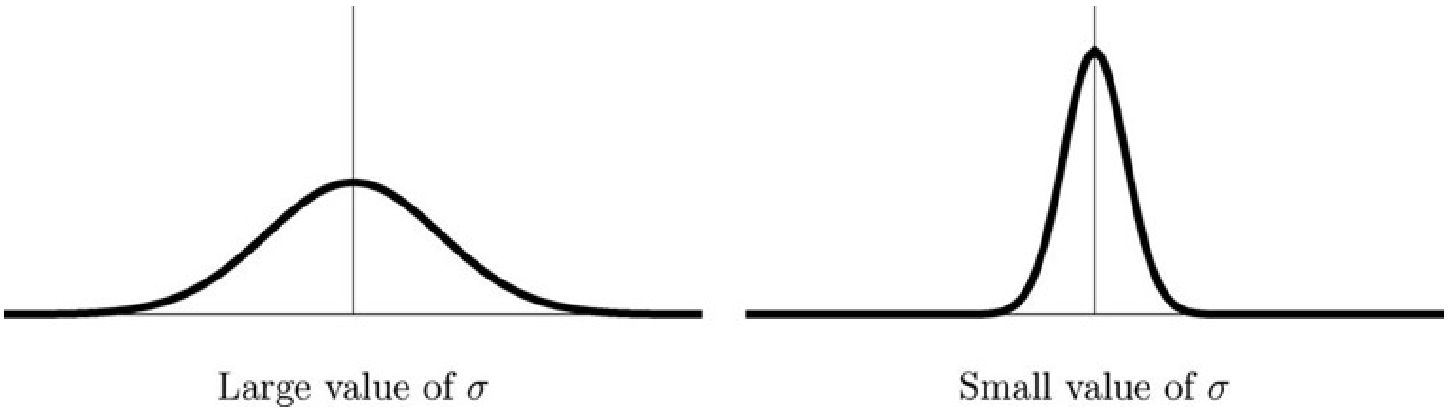
where σ is the standard deviation: a large value of σ produces to a flatter curve, and a small value leads to a "pointier" curve. Figure 5.11 shows examples of such one-dimensional Gaussians. Gaussian filters are important for a number of reasons:

1. They are mathematically very "well behaved"; in particular the Fourier transform (see Chapter 7) of a Gaussian filter is another Gaussian.
2. They are rotationally symmetric, and so are very good starting points for some edge detection algorithms (see Chapter 9),
3. They are *separable*; in that a Gaussian filter may be applied by first applying a one-dimension Gaussian in the *x* direction, followed by another in the *y* direction. This can lead to very fast implementations.
4. The convolution of two Gaussians is another Gaussian.

A two-dimensional Gaussian function is given by

$$f(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Figure 5.11: One-dimensional Gaussians



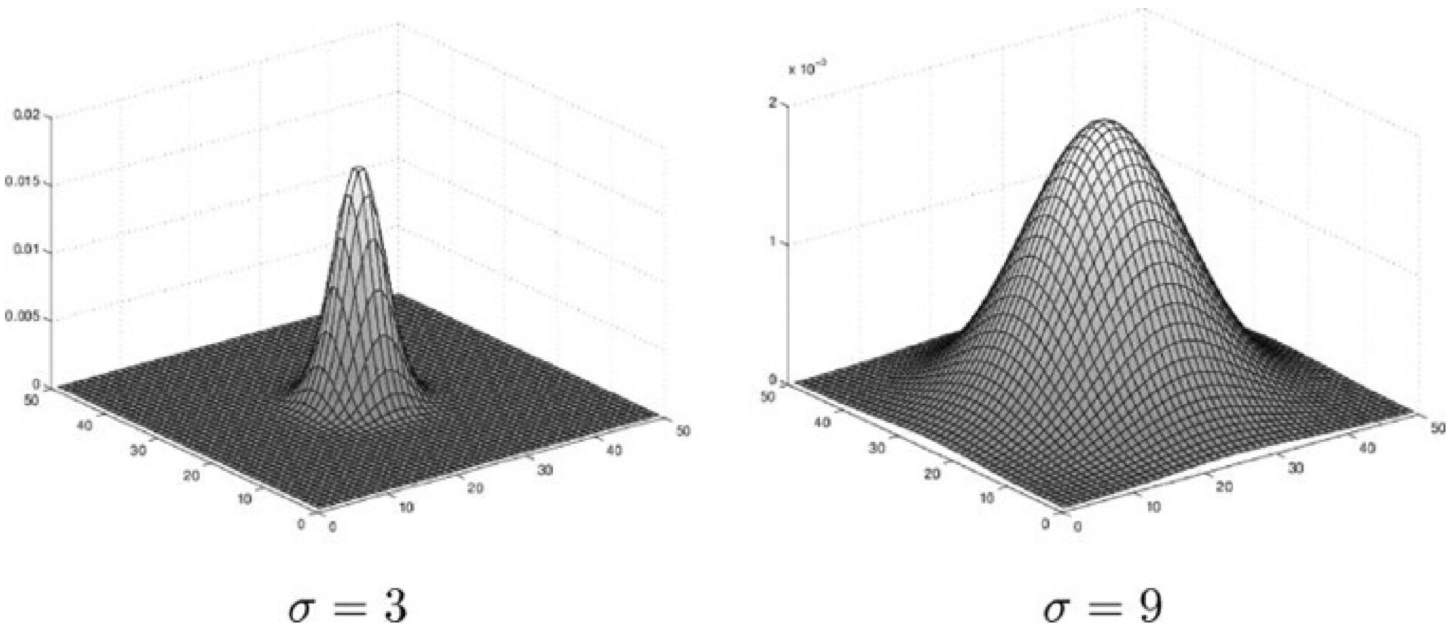
The command `fspecial('gaussian')` produces a discrete version of this function. We can draw pictures of this with the `surf` function, and to ensure a nice smooth result, we shall create a large filter (size 50 × 50) with different standard deviations.

```
>> a = 50; s = 3;
>> g = fspecial('gaussian',[a a],s);
>> surf(1:a,1:a,g)
>> s = 9;
>> g2 = fspecial('gaussian',[a a],s);
>> figure,surf(1:a,1:a,g2)
```

MATLAB/Octave

The surfaces are shown in Figure 5.12.

Figure 5.12: Two-dimensional Gaussians.



Gaussian filters have a blurring effect which looks very similar to that produced by neighborhood averaging. Let's experiment with the cameraman image, and some different Gaussian filters.

```
>> g1 = fspecial('gaussian',[5,5]);
>> g1 = fspecial('gaussian',[5,5],2);
>> g1 = fspecial('gaussian',[11,11],1);
>> g1 = fspecial('gaussian',[11,11],5);
```

MATLAB/Octave

The final parameter is the standard deviation; which if not given, defaults to 0.5. The second parameter (which is also optional), gives the size of the filter; the default is 3 × 3. If the filter is to be square, as in all the previous examples, we can just give a single number in each case.

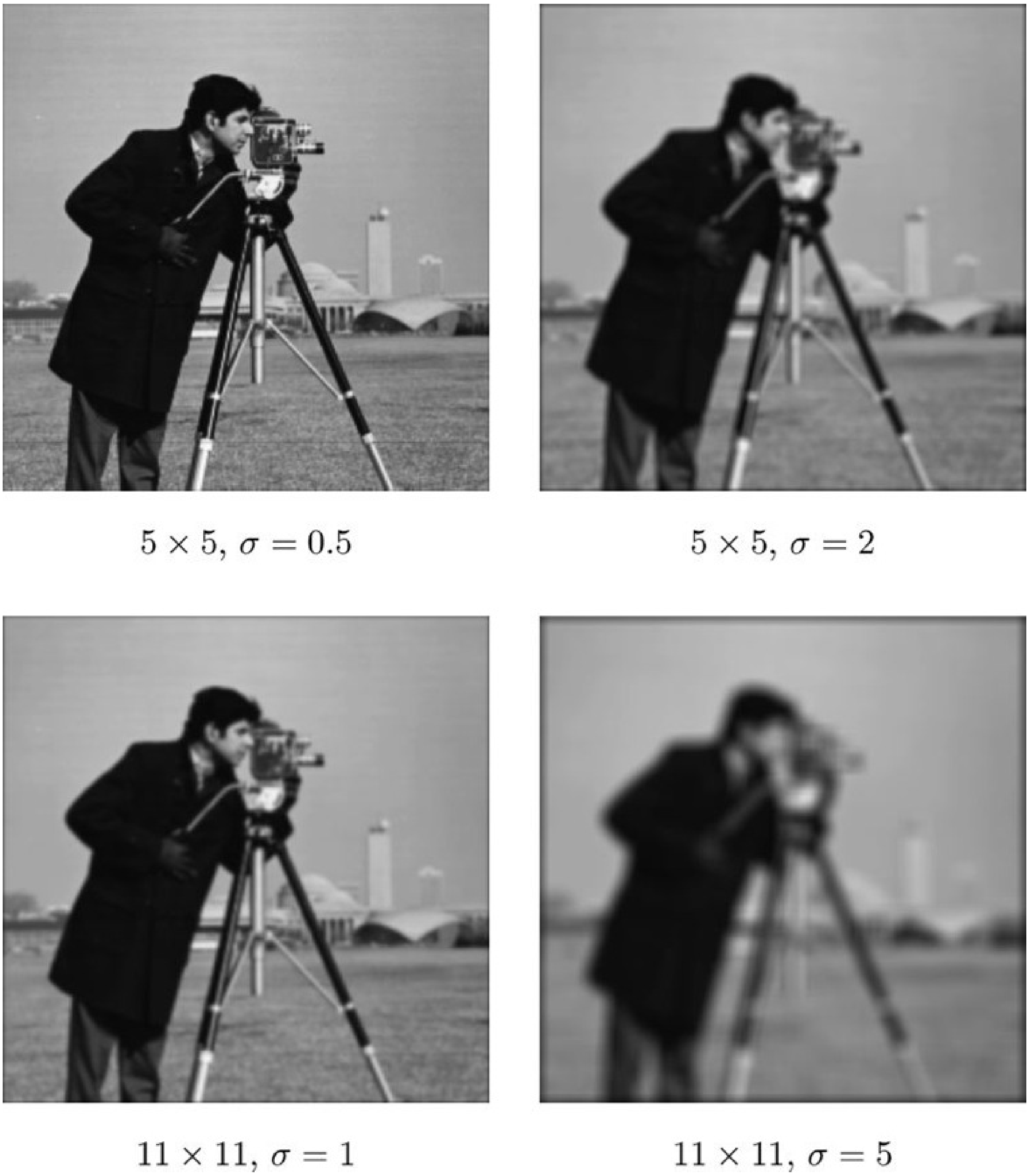
Now we can apply the filter to the cameraman image matrix `c` and view the result.

```
>> imshow(imfilter(c,g1))
>> figure,imshow(imfilter(c,g2))
>> figure,imshow(imfilter(c,g3,'symmetric'))
>> figure,imshow(imfilter(c,g4,'symmetric'))
```

MATLAB/Octave

As for the averaging filters earlier, the `symmetric` option is used to prevent the dark borders which would arise from low pass filtering using zero padding. The results are shown in Figure 5.13. Thus, to obtain a spread out blurring effect, we need a large standard deviation.

Figure 5.13: Effects of different Gaussian filters on an image


$5 \times 5,\ \sigma = 0.5$
$5 \times 5,\ \sigma = 2$
$11 \times 11,\ \sigma = 1$
$11 \times 11,\ \sigma = 5$

In fact, if we let the standard deviation grow large without bound, we obtain the averaging filters as limiting values. For example:

```
>> fspecial('gaussian',3,100)

ans =

    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
```

MATLAB/Octave

and we have the 3 × 3 averaging filter.

Although the results of Gaussian blurring and averaging look similar, the Gaussian filter has some elegant mathematical properties that make it particularly suitable for blurring.

In Python, the command `gaussian_filter` can be used, which takes as parameters the standard variation, and also a `truncate` parameter, which gives the number of standard deviations at which the filter should be cut off. This means that the images in Figure 5.13 can be obtained with:

```
In:  cg1 = ndi.gaussian_filter(c,0.5,truncate=4.5)
In:  cg2 = ndi.gaussian_filter(c,2,truncate=1)
In:  cg3 = ndi.gaussian_filter(c,1,truncate=5)
In:  cg4 = ndi.gaussian_filter(c,5,truncate=1)
```

Python

As with the uniform filter, the default behavior is to reflect the image in its edges, which is the same result as the `symmetric` option used in MATLAB/Octave. To see the similarity between MATLAB/Octave and Python, we can apply a Gaussian filter in Python to an image consisting of all zeros except for a central one. This will produce the same output as MATLAB's `fspecial` function:

```
>> fspecial('gaussian',[5,5],1)
ans =

   0.0029690   0.0133062   0.0219382   0.0133062   0.0029690
   0.0133062   0.0596343   0.0983203   0.0596343   0.0133062
   0.0219382   0.0983203   0.1621028   0.0983203   0.0219382
   0.0133062   0.0596343   0.0983203   0.0596343   0.0133062
   0.0029690   0.0133062   0.0219382   0.0133062   0.0029690
```

MATLAB/Octave

and the Python equivalent:

```
In:  x = ones((5,5)); x(2,2)=1
In:  ndi.gaussian_filter(x,1,truncate=2).round(7)
Out:
array([[ 0.002969 ,  0.0133062,  0.0219382,  0.0133062,  0.002969 ],
       [ 0.0133062,  0.0596343,  0.0983203,  0.0596343,  0.0133062],
       [ 0.0219382,  0.0983203,  0.1621028,  0.0983203,  0.0219382],
       [ 0.0133062,  0.0596343,  0.0983203,  0.0596343,  0.0133062],
       [ 0.002969 ,  0.0133062,  0.0219382,  0.0133062,  0.002969 ]])
```
Python

Other filters will be discussed in future chapters; also check the documentation for `fspecial` for other filters.

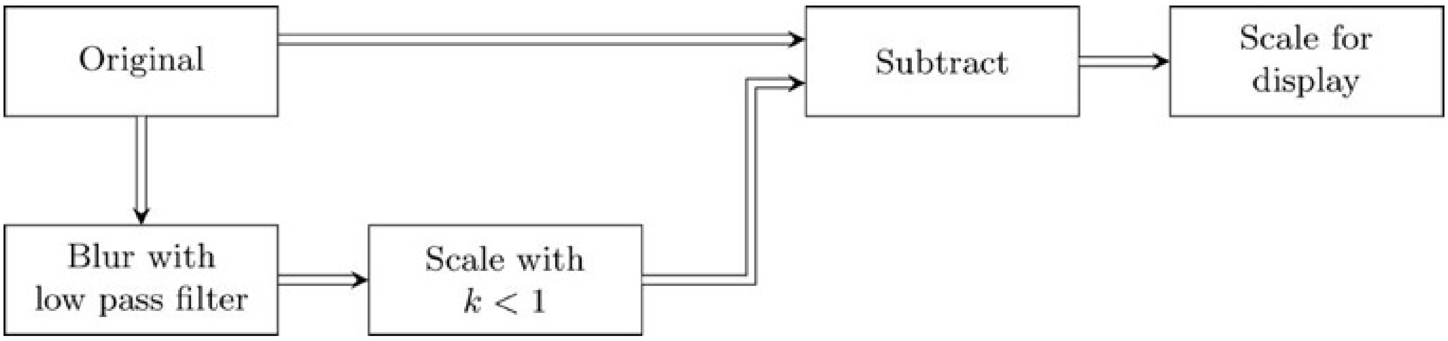
5.7 Edge Sharpening

Spatial filtering can be used to make edges in an image slightly sharper and crisper, which generally results in an image more pleasing to the human eye. The operation is variously called "edge enhancement," "edge crispening," or "unsharp masking." This last term comes from the printing industry.

Unsharp Masking

The idea of unsharp masking is to subtract a scaled "unsharp" version of the image from the original. In practice, we can achieve this affect by subtracting a scaled blurred image from the original. The schema for unsharp masking is shown in Figure 5.14.

Figure 5.14: Schema for unsharp masking



In fact, all these steps can be built into the one filter. Starting with the filter

$$id = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

which is an "identity" filter (in that applying it to an image leaves the image unchanged), all the steps in Figure 5.14 could be implemented by the filter

$$u = s(id - \frac{1}{k}a)$$

where *s* is the scaling factor. To ensure the output image has the same levels of brightness as the original, we need to ensure that the sum of all the elements of *u* is 1, that is, that

$$s(1 - \frac{1}{k}) = 1$$

or that

$$s = \frac{k}{k-1}$$

Suppose for example we let *k* = 1.5. Then *s* = 3 and so the filter is

$$3\begin{bmatrix}0 & 0 & 0\\0 & 1 & 0\\0 & 0 & 0\end{bmatrix} - 2\left(\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}\right) = \frac{1}{9}\begin{bmatrix}-2 & -2 & -2\\-2 & 25 & -2\\-2 & -2 & -2\end{bmatrix}$$

An alternative derivation is to write the equation relating *s* and *k* as

$$s - \frac{s}{k} = 1.$$

If *s*/*k* = *t*, say, then *s* = *t* + 1. So if *s*/*k* = 2/3, then *s* = 5/3 and so the filter is

$$\frac{5}{3}\begin{bmatrix}0 & 0 & 0\\0 & 1 & 0\\0 & 0 & 0\end{bmatrix} - \frac{2}{3}\left(\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}\right) = \frac{1}{27}\begin{bmatrix}-2 & -2 & -2\\-2 & 43 & -2\\-2 & -2 & -2\end{bmatrix}$$

For example, consider an image of an iconic Australian animal:

```
>> k = imread('koala.png');
```

MATLAB/Octave

The first filter above can be constructed as

```
>> id = [0 0 0;0 1 0;0 0 0];
>> f = fspecial('average');
>> u = 3*id - 2*f
u =

  -0.22222  -0.22222  -0.22222
  -0.22222   2.77778  -0.22222
  -0.22222  -0.22222  -0.22222
```

MATLAB/Octave

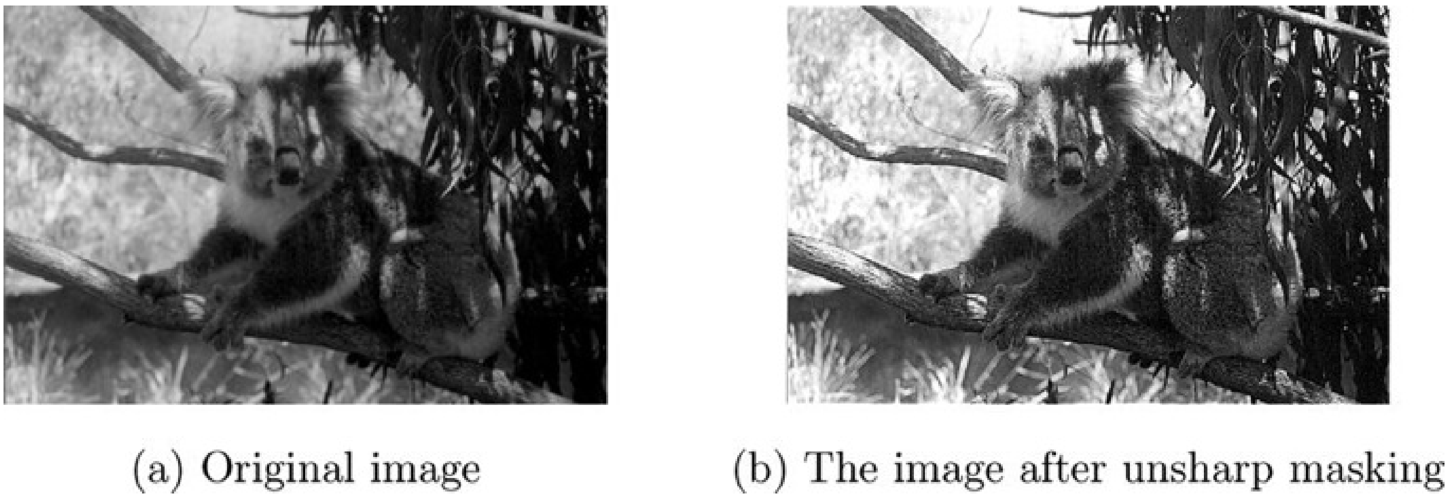
and applied to the image in the usual way:

```
>> ku = imfilter(k,u);
>> imshow(ku)
```

MATLAB/Octave

The image k is shown in Figure 5.15(a), then the result of unsharp masking is given in Figure 5.15(b). The result appears to be a better image than the original; the edges are crisper and more clearly defined.

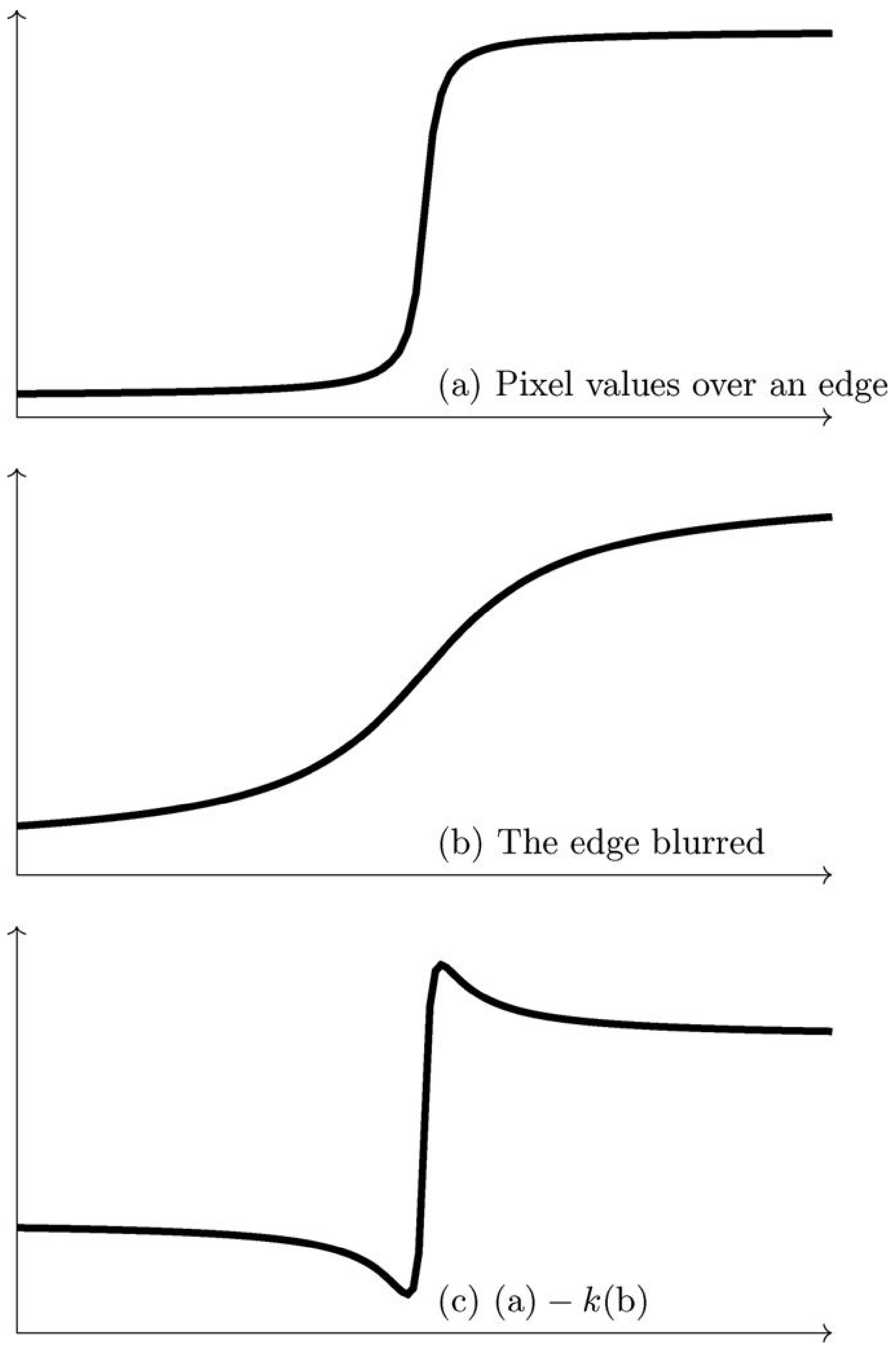
Figure 5.15: An example of unsharp masking



(a) Original image (b) The image after unsharp masking

The same effect can be seen in Python, where we convolve with the image converted to floats, so as not to lose any information in the arithmetic:

Figure 5.16: Unsharp masking



```
In:  k = io.imread('koala.png')
In:  u = array([[-2,-2,-2],[-2,25,-2],[-2,-2,-2]])/9.0
In:  ku = ndi.convolve(k.astype(float),u)
In:  io.imshow(ku/255,vmax=1.0,vmin=0.0)
```

Python

To see why this works, we may consider the function of gray values as we travel across an edge, as shown in Figure 5.16.

As a scaled blur is subtracted from the original, the result is that the edge is enhanced, as shown in graph (c) of Figure 5.16.

We can in fact perform the filtering and subtracting operation in one command, using the linearity of the filter, and that the 3 × 3 filter

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

is the "identity filter."

Hence, unsharp masking can be implemented by a filter of the form

$$f = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{k} \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

where *k* is a constant chosen to provide the best result. Alternatively, the unsharp masking filter may be defined as

$$f = k \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

so that we are in effect subtracting a blur from a scaled version of the original; the scaling factor may also be split between the identity and blurring filters.

The `unsharp` option of `fspecial` produces such filters; the filter created has the form

$$\frac{1}{\alpha+1} \begin{bmatrix} -\alpha & \alpha-1 & -\alpha \\ \alpha-1 & \alpha+5 & \alpha-1 \\ -\alpha & \alpha-1 & -\alpha \end{bmatrix}$$

where α is an optional parameter which defaults to 0.2. If α = 0.5, the filter is

$$\frac{1}{3} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 11 & -1 \\ -1 & -1 & -1 \end{bmatrix} = 4 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - 3 \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Figure 5.17 was created using the MATLAB commands

```
>> p = imread('pelicans.png');
>> u = fspecial('unsharp',0.5);
>> pu = imfilter(p,u);
>> imshow(p),figure,imshow(pu)
```

MATLAB/Octave

Figure 5.17(b), appears much sharper and “cleaner” than the original. Notice in particular the rocks and trees in the background, and the ripples on the water.

Although we have used averaging filters above, we can in fact use any low pass filter for unsharp masking.

Exactly the same affect can be produced in Python; starting by creating an `unsharp` function to produce the filter:

```
In :  def unsharp(alpha=0.2):
...:      A1 = array([[-1,1,-1],[1,1,1],[-1,1,-1]])
...:      A2 = array([[0,-1,0],[-1,5,-1],[0,-1,0]])
...:      return (alpha*A1+A2)/(alpha+1)
```

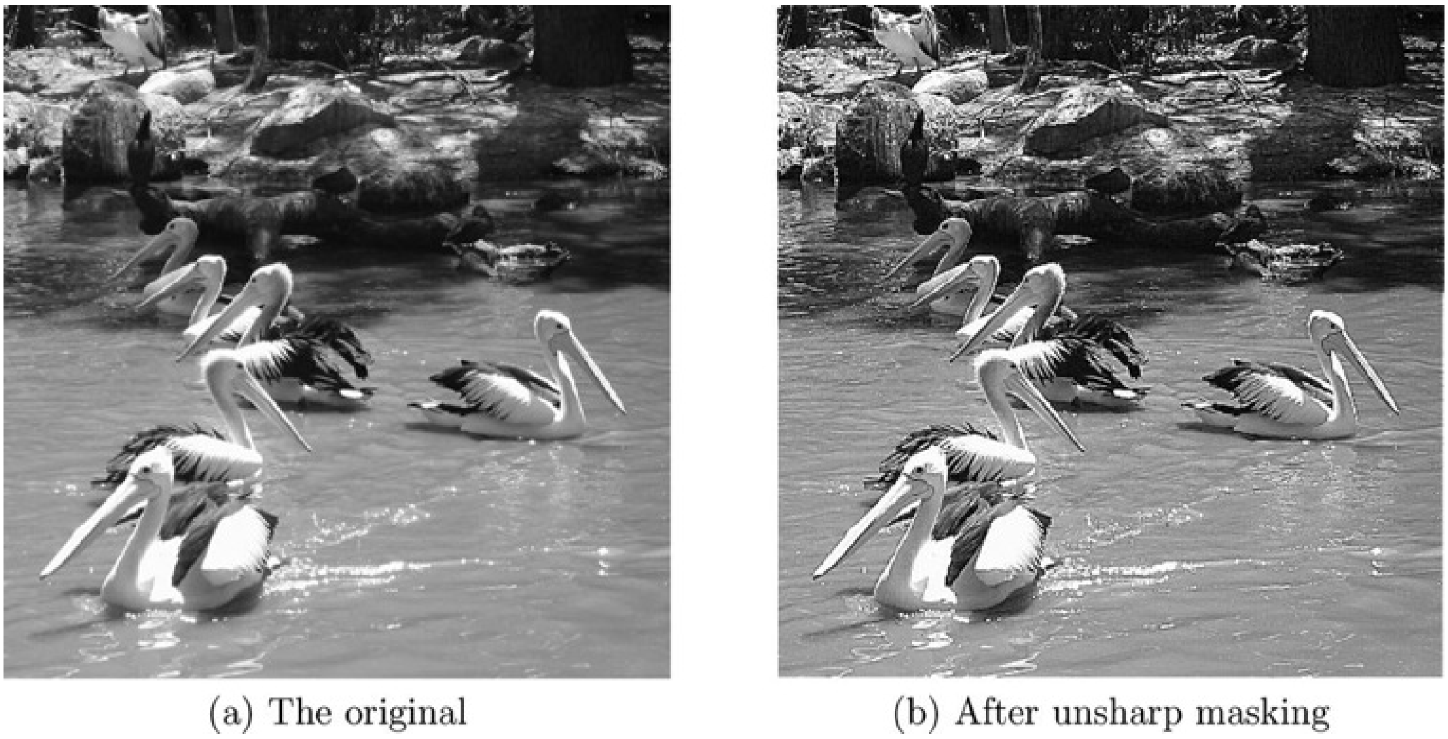
Python

This can be applied with convolve:

```
In:  p = io.imread('pelicans.png')
In:  u = unsharp(0.5)
In:  pu = ndi.convolve(p.astype(float),u)
In:  io.imshow(pu/255,vmax=1.0,vmin=0.0)
```

Python

Figure 5.17: Edge enhancement with unsharp masking


(a) The original (b) After unsharp masking

High Boost Filtering

Allied to unsharp masking filters are the *high boost* filters, which are obtained by

$$\frac{1}{9}\begin{bmatrix}-1 & -1 & -1\\ -1 & z & -1\\ -1 & -1 & -1\end{bmatrix}$$

where A is an "amplification factor." If $A = 1$, then the high boost filter becomes an ordinary high pass filter. If we take as the low pass filter the 3×3 averaging filter, then a high boost filter will have the form

```
>> f = [-1 -1 -1;-1 11 -1;-1 -1 -1]/9;
>> xf = imfilter(f,x);
>> imshow(xf/80)
```

MATLAB/Octave

where $z > 8$. If we put $z = 11$, we obtain a filtering very similar to the unsharp filter above, except for a scaling factor. Thus, the commands:

$$\begin{aligned}\text{high boost} &= A(\text{original}) - (\text{low pass})\\ &= A(\text{original}) - ((\text{original}) - (\text{high pass}))\\ &= (A-1)(\text{original}) + (\text{high pass}).\end{aligned}$$

will produce an image similar to that in Figure 5.15. The value 80 was obtained by trial and error to produce an image with similar intensity to the original.

We can also write the high boost formula above as

$$\begin{aligned}\text{high boost} &= A(\text{original}) - (\text{low pass})\\ &= A(\text{original}) - ((\text{original}) - (\text{high pass}))\\ &= (A-1)(\text{original}) + (\text{high pass}).\end{aligned}$$

Best results for high boost filtering are obtained if we multiply the equation by a factor w so that the filter values sum to 1; this requires

$$wA - w = 1$$

or

$$w = \frac{1}{A-1}.$$

So a general unsharp masking formula is

$$\frac{A}{A-1}(\text{original}) - \frac{1}{A-1}(\text{low pass}).$$

Another version of this formula is

$$\frac{A}{2A-1}(\text{original}) - \frac{1-A}{2A-1}(\text{low pass})$$

where for best results A is taken so that

$$\frac{3}{5} \leq A \leq \frac{5}{6}.$$

If we take $A = 3/5$, the formula becomes

$$\frac{3/5}{2(3/5)-1}(\text{original}) - \frac{1-(3/5)}{2(3/5)-1}(\text{low pass}) = 3(\text{original}) - 2(\text{low pass})$$

If we take $A = 5/6$ we obtain

$$\frac{5}{4}(\text{original}) - \frac{1}{4}(\text{low pass})$$

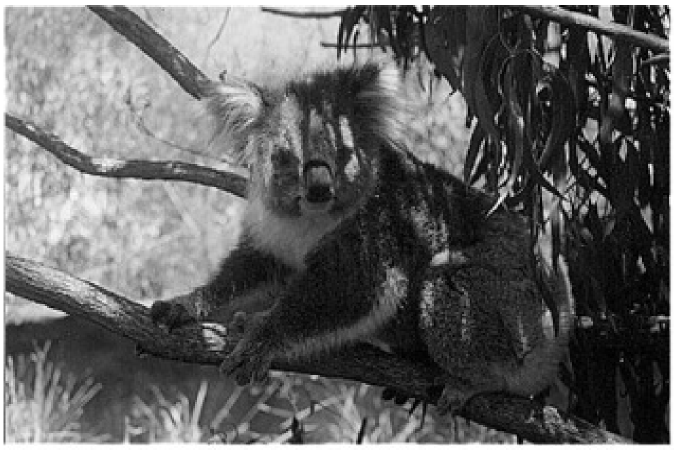
Using the identity and averaging filters, we can obtain high boost filters by:

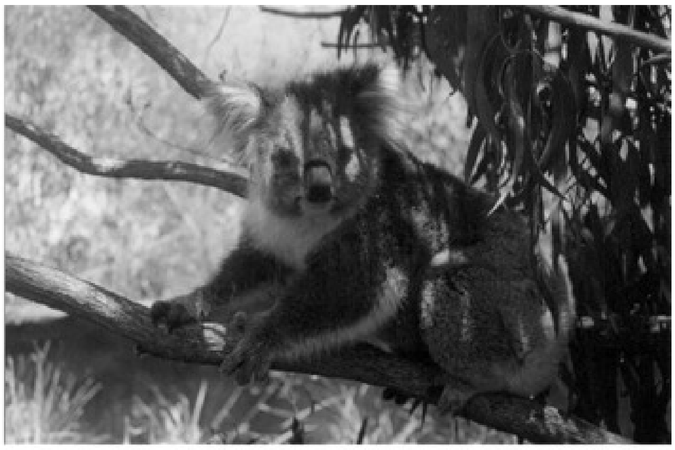
```
>> id=[0 0 0;0 1 0;0 0 0];
>> f=fspecial('average');
>> hb1=3*id-2*f

hb1 =

   -0.2222   -0.2222   -0.2222
   -0.2222    2.7778   -0.2222
   -0.2222   -0.2222   -0.2222

>> hb2=1.25*id-0.25*f

hb2 =

   -0.0278   -0.0278   -0.0278
   -0.0278    1.2222   -0.0278
   -0.0278   -0.0278   -0.0278
```
MATLAB/Octave

If each of the filters `hb1` and `hb2` are applied to an image with `imfilter`, the result will have enhanced edges. The images in Figure 5.18 show these results; Figure 5.18(a) was obtained with

```
>> k1 = imfilter(k,hb1);
>> imshow(k1)
```
MATLAB/Octave

Figure 5.18: High boost filtering


(a) High boost filtering with hb1


(b) High boost filtering with hb2

and Figure 5.18(b) similarly.

With Python, these images could be obtained easily as:

```
In:  k = im2fl(io.imread('koala.png'))
In:  kf = ndi.uniform_filter(k,3)
In:  hb1 = 3*k - 2*kf
In:  hb2 = 1.25*k - 0.25*kf
In:  subplot(121),io.imshow(hb1,vmax=1.0,vmin=0.0)
In:  subplot(122),io.imshow(hb2,vmax=1.0,vmin=0.0)
```
Python

Of the two filters, `hb1` appears to produce the best result; `hb2` produces an image not much crisper than the original.

5.8 Non-Linear Filters

Linear filters, as we have seen in the previous sections, are easy to describe, and can be applied very quickly and efficiently.

A *non-linear filter* is obtained by a non-linear function of the grayscale values in the mask. Simple examples are the *maximum filter*, which has as its output the maximum value under the mask, and the corresponding *minimum filter*, which has as its output the minimum value under the mask.

Both the maximum and minimum filters are examples of *rank-order filters*. In such a filter, the elements under the mask are ordered, and a particular value returned as output. So if the values are given in increasing order, the minimum filter is a rank-order filter for which the *first* element is returned, and the maximum filter is a rank-order filter for which the last element is returned

For implementing a general non-linear filter in MATLAB or Octave, the function to use is `nlfilter`, which applies a filter to an image according to a pre-defined function. If the function is not already defined, we have to create an m-file that defines it.

Here are some examples; first to implement a maximum filter over a 3 × 3 neighborhood (note that the commands are slightly different for MATLAB and for Octave):

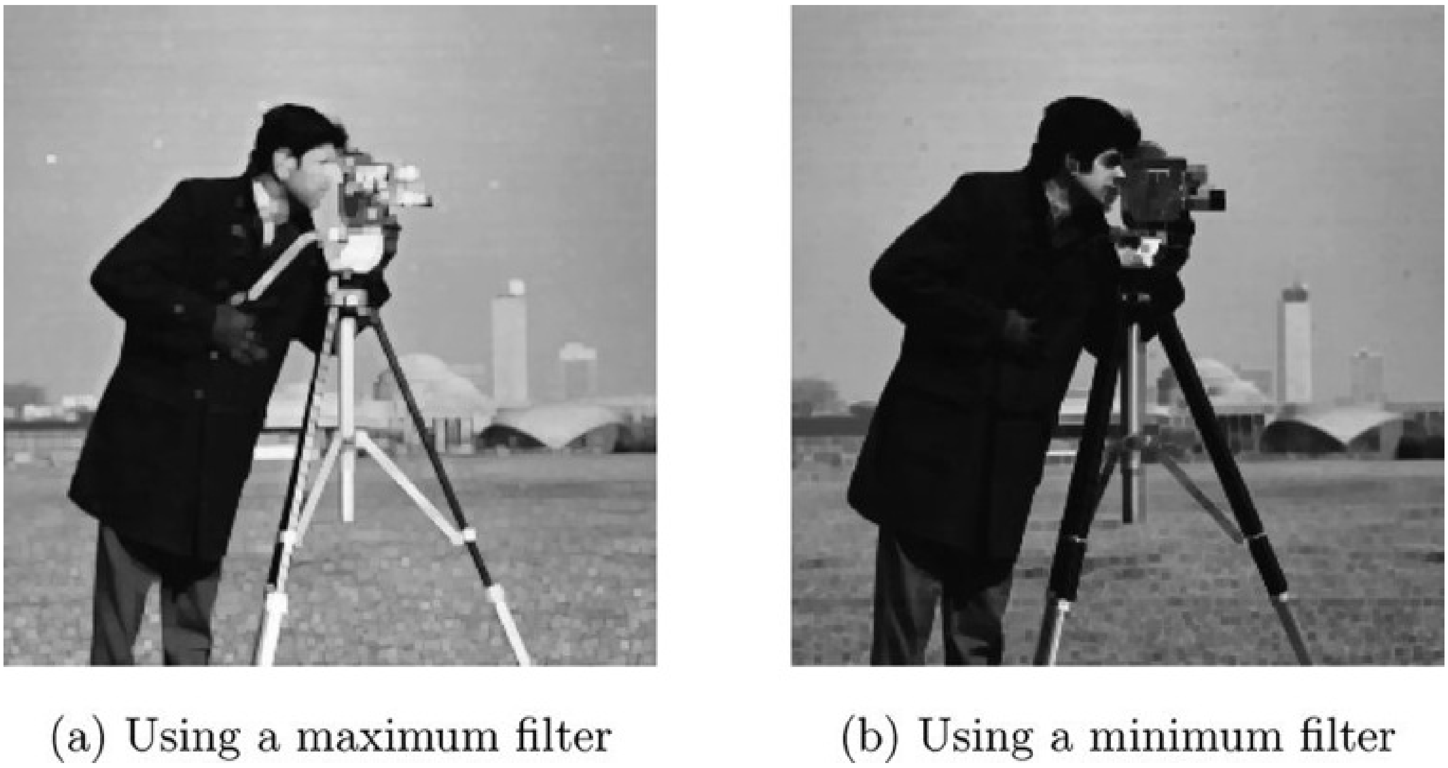
```
>> cmax = nlfilter(c,[3,3],'max(x(:))');      # This is MATLAB
>> cmax = nlfilter(c,[3,3],@(x) max(x(:)));   # This is Octave
```
MATLAB/Octave

Python has a different syntax, made easier for maxima and minima in that the `max` function is applied to the entire array, rather than just its columns:

```
In:  cmax = ndi.generic_filter(c,max,[3,3])
```
Python

The `nlfilter` function and the `generic_filter` function each require three arguments: the image matrix, the size of the filter, and the function to be applied. The function must be a matrix function that returns a scalar value. The result of these operations is shown in Figure 5.19(a). Replacing `max` with `min` in the above commands implements a minimum filter, and the result is shown in Figure 5.19(b).

Figure 5.19: Using non-linear filters


(a) Using a maximum filter (b) Using a minimum filter

Note that in each case the image has lost some sharpness, and has been brightened by the maximum filter, and darkened by the minimum filter. The `nlfilter` function is very slow; in general, there is little call for non-linear filters except for a few that are defined by their own commands. We shall investigate these in later chapters.

However, if a non-linear filter is needed in MATLAB or Octave, a faster alternative is to use the `colfilt` function, which rearranges the image into columns first. For example, to apply the maximum filter to the cameraman image, we can use

```
>> cmax = colfilt(c,[3,3],'sliding',@max);
```
MATLAB/Octave

The parameter `sliding` indicates that overlapping neighborhoods are being used (which of course is the case with filtering). This particular operation is almost instantaneous, as compared with the use of `nlfilter`.

To implement the maximum and minimum filters as rank-order filters, we may use the MATLAB/Octave function `ordfilt2`. This requires three inputs: the image, the index value of the ordered results to choose as output, and the definition of the mask. So to apply the maximum filter on a 3 × 3 mask, we use

```
>> cmax = ordfilt2(c,9,ones(3,3));
```
MATLAB/Octave

and the minimum filter can be applied with

```
>> cmin = ordfilt2(c,1,ones(3,3));
```
MATLAB/Octave

A very important rank-order filter is the *median filter*, which takes the central value of the ordered list. We could apply the median filter with

```
>> cmed = ordfilt2(c,5,ones(3,3));
```
MATLAB/Octave

However, the median filter has its own command, `medfilt2`, which we discuss in more detail in Chapter 8.

Python has maximum and minimum filters in the `scipy.ndimage` module and also in the `skimage.filter.rank` module. As an example of each:

```
In:  cmin = ndi.minimum_filter(c, size=(3,3))
In:  cmax = rk.minimum(c,ones((3,3)))
```
Python

Rank order filters are implemented by the `rank_filter` function in the `ndimage` module, and one median filter is also provided by `ndimage` as `median_filter`. Thus the two commands

```
In:  cm = ndi.median_filter(c,size=(3,3))
In:  cm2 = ndi.rank_filter(c,4,size=(3,3))
```
Python

produce the same results. (Recall that in Python arrays and lists are indexed starting with zero, so the central value in a 3 × 3 array has index value 4.) User-defined filters can be applied using `generic_filter`. For example, suppose we consider the *root-mean-square filter*, which returns the value

$$\sqrt{\frac{1}{N}\sum_{x\in M} x^2}$$

where *M* is the mask, and *N* is the number of its elements. First, this must be defined as a Python function:

```
In :  def rms(x):
...:      return sqrt(mean(x**2))
```

Python

Then it can be applied, for example, to the cameraman image `c`:

```
In : cr = ndi.generic_filter(c,rms,size=(3,3))
```

Python

Other non-linear filters are the *geometric mean filter*, which is defined as

$$\left(\prod_{(i,j)\in M} x(i,j)\right)^{1/N}$$

where as for the root-mean-square filter *M* is the filter mask, and *N* its size; and the *alpha-trimmed mean filter*, which first orders the values under the mask, trims off elements at either end of the ordered list, and takes the mean of the remainder. So, for example, if we have a 3 × 3 mask, and we order the elements as

$$x_1 \le x_2 \le x_3 \le \cdots \le x_9$$

and trim off two elements at either end, the result of the filter will be

$$(x_3 + x_4 + x_5 + x_6 + x_7)/5.$$

Both of these filters have uses for image restoration; again, see Chapter 8.

Non-linear filters are used extensively in image restoration, especially for cleaning noise; and these will be discussed in Chapter 8.

5.9 Edge-Preserving Blurring Filters

With the uniform (average) and Gaussian filters, blurring occurs across the entire image, and edges are blurred as well as backgrounds and other low frequency components. However, there is a large class of filters that blur low frequency components but keep the edges fairly sharp.

The median filter mentioned above is one such; we shall explore it in greater detail in Chapter 8. But just for a quick preview of its effects, consider median filters of size 3 × 3 and 5 × 5 applied to the cameraman image. These are shown in Figure 5.20. Even though much of the fine detail has gone, especially when using the larger filter, the edges are still sharp.

Figure 5.20: Using a median filter

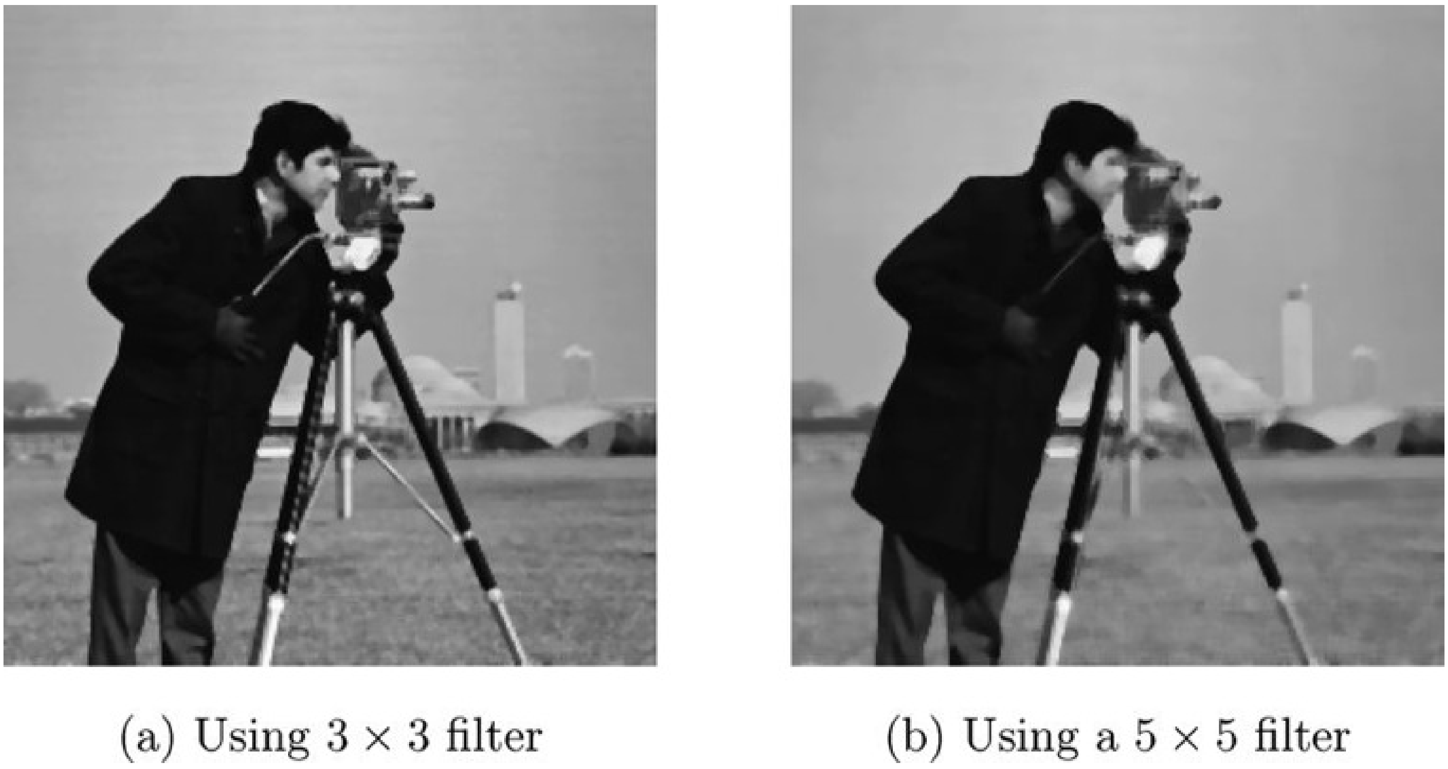
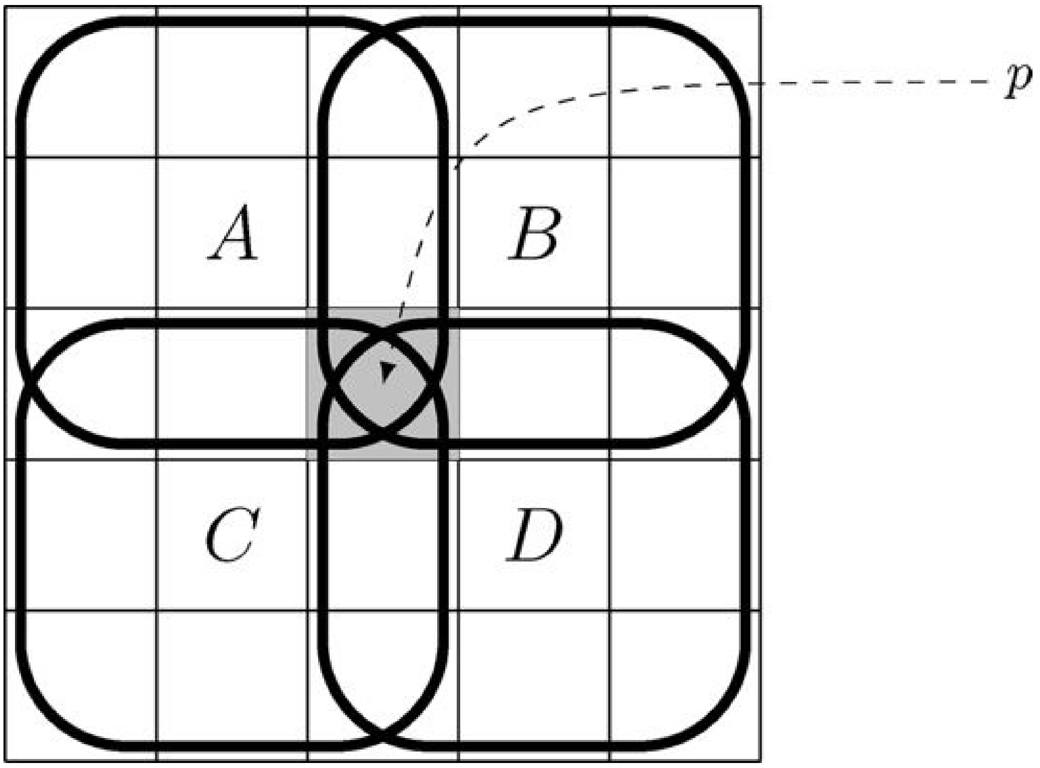


Figure 5.21: The neighborhoods used in the Kuwahara filter



Kuwahara Filters

These are a family of filters in which the variance of several regions in the neighborhood are computed, and the output is the mean of the region with the lowest variance. The simplest version takes a 5 × 5 neighborhood of a given pixel *p* looks at the four overlapping 3 × 3 neighborhoods of which it is a corner as shown in Figure 5.21.

The filter works by first computing the variances of the four neighborhoods around *p*, and then the output is the mean of the neighborhood with the lowest variance. So for example, consider the following neighborhood:

```
169  140  105  126  110
140   65  175  247   79
 40  178  240  171   37
 56   28  203   55   53
208  193   75  165  212
```

Then the variances of each of the neighborhoods can be found as

```
In:  x[0:3,0:3].var(), x[0:3,2:5].var(), x[2:5,0:3].var(), x[2:5,2:5].var()
```

Python

or as

```
>>> var(x(1:3,1:3)(:),1),var(x(1:3,3:5)(:),1),var(x(3:5,1:3)(:),1),...
    var(x(3:5,3:5)(:),1)
```

MATLAB/Octave

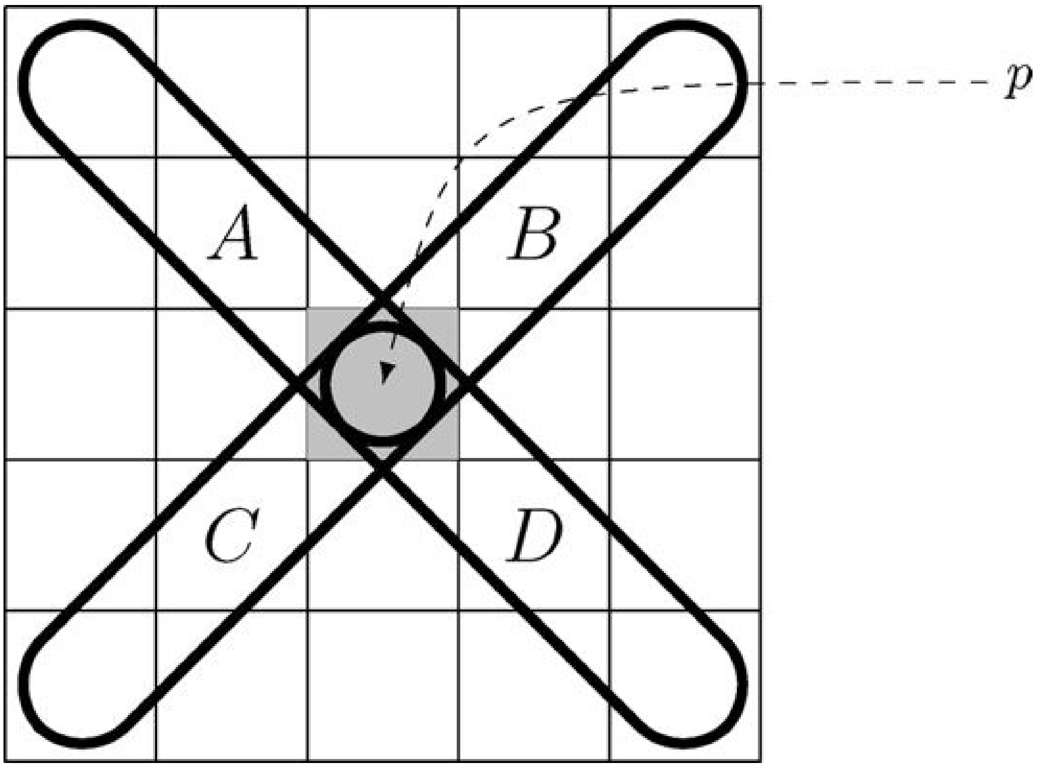
The variances of the neighborhoods *A, B, C*, and *D* will be found to be 3372.54, 4465.11, 6278.0, 5566.69, respectively. Of these four values, the variance of *A* is the smallest, so the output is the mean of *A*, which (when rounded to an integer) is 139.

None of MATLAB, Octave, or Python support the Kuwahara filter directly, but it is very easy to program it. Recall that for a random variable *X*, its variance can be computed as

$$\overline{X^2} - \left(\overline{X}\right)^2$$

where $\overline{(\cdot)}$ is the mean. This means that the variances in all 3 × 3 neighborhoods of an image x can be found by first filtering x^2 with the averaging filter, then squaring the result of filtering x with the averaging filter, and subtracting them:

Figure 5.22: Alternative neighborhoods for a Kuwahara filter



```
>> cd = float(c);
>> cdm = imfilter(cd,ones(3)/9,'symmetric');
>> cd2f = imfilter(cd.^2,ones(3)/9,'symmetric');
>> cdv = cd2f - cdm.^2;
```

MATLAB/Octave

```
In:  cd = float32(c)
In:  cdm = ndi.uniform_filter(cd,(3,3))
In:  cd2f = ndi.uniform_filter(cd**2,(3,3))
In:  cdv = cd2f - cdm**2
```

Python

At this stage, the array `cdm` contains all the mean values, and `cdv` contains all the variances. At every point (i, j) in the image then:

1. Compute the list
$\text{vars} = [\texttt{cdv}(i-1, j-1), \texttt{cdv}(i-1, j+1), \texttt{cdv}(i+1, j-1), \texttt{cdv}(i+1, j+1)]$.
2. Also compute the list
$\text{means} = [\texttt{cdm}(i-1, j-1), \texttt{cdm}(i-1, j+1), \texttt{cdm}(i+1, j-1), \texttt{cdm}(i+1, j+1)]$.
3. Output the value of "means" corresponding to the lowest value of "vars."

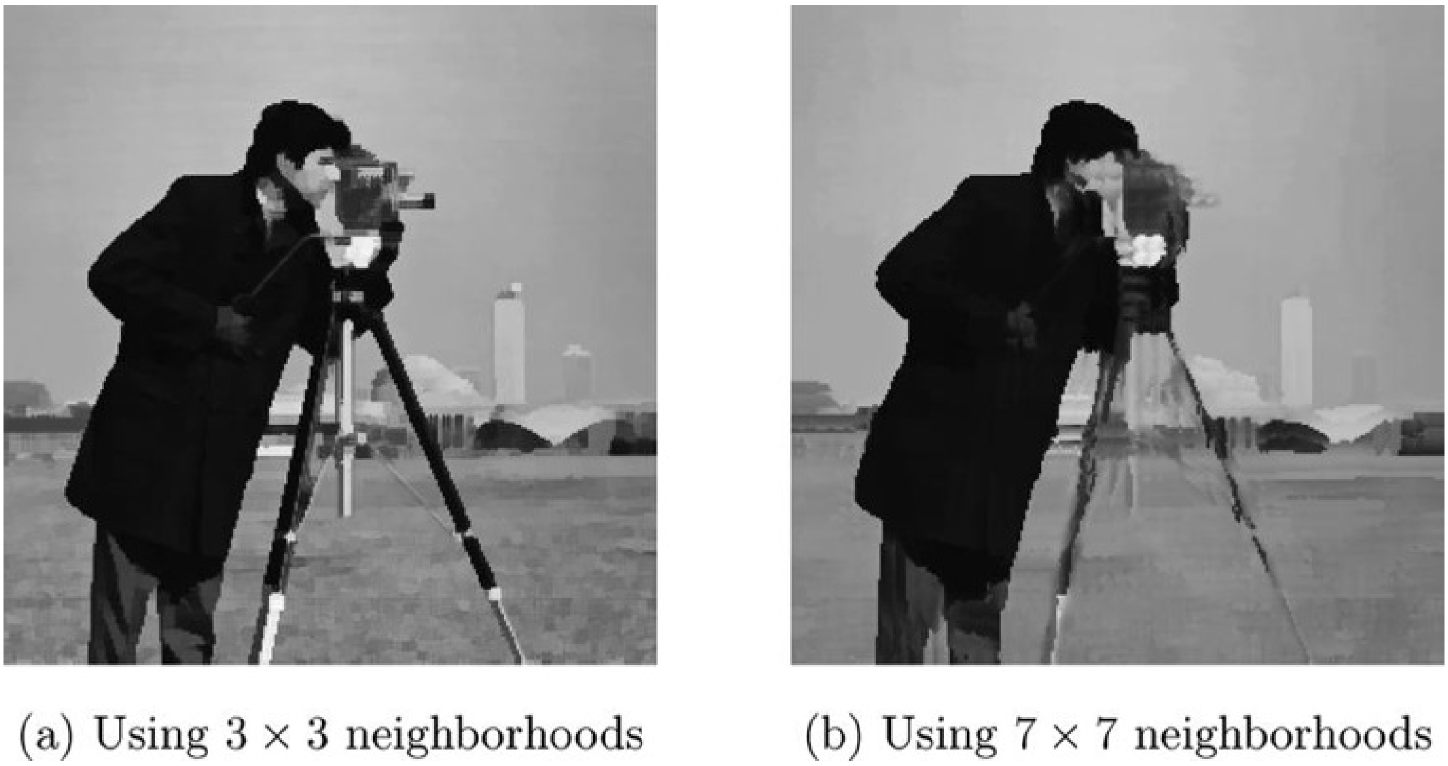
Programs are given at the end of the chapter. Note that the neighborhoods can of course be larger than 3×3, any odd square size can used, or indeed any other shape, such as shown in Figure 5.22. Figure 5.23 shows the cameraman image first with the Kuwahara filter using 3 × 3 neighborhoods and with 7 × 7 neighborhoods.

Note that even with the significant blurring with the larger filter, the edges are still remarkably sharp.

Bilateral Filters

Linear filters (and many non-linear filters) are *domain filters*; the weights attached to each neighboring pixel depend on only the *position* of those pixels. Another family of filters are the range *filters*, where the weights depend on the relative *difference* between pixel values. For example, consider the Gaussian filter $e^{(-x^2-y^2)/2}$ (so with variance $\sigma^2 = 1$) over the region $-1 \le x, y \le 1$:

Figure 5.23: Using Kuwahara filters


(a) Using 3 × 3 neighborhoods (b) Using 7 × 7 neighborhoods

$$G = \begin{matrix} 0.36788 & 0.60653 & 0.36788 \\ 0.60653 & 1.00000 & 0.60653 \\ 0.36788 & 0.60653 & 0.36788 \end{matrix}$$

Applied to the neighborhood

$$N = \begin{matrix} 0.8 & 0.1 & 0.6 \\ 0.3 & 0.5 & 0.7 \\ 0.4 & 0.9 & 0.2 \end{matrix}$$

the output is simply the sum of all the products of corresponding elements of *G* and *N*.

As a *range filter*, however, with variance σ_r^2, the output would be the Gaussian function applied to the difference of the elements of *N* with its central value:

$$N - 0.5 = \begin{matrix} 0.3 & -0.4 & 0.1 \\ -0.2 & 0.0 & 0.2 \\ -0.1 & 0.4 & -0.3 \end{matrix}$$

For each of these values v the filter consists $e^{-v^2/2\sigma_r^2}$.

Thus, a domain filter is based on the *closeness* of pixels, and a range filter is based on the *similarity* of pixels. Clearly, the larger the value of σ_r the flatter the filter, and so there will be less distinction of closeness.

The following three arrays show the results with σ_r = 0.1, 1, 10, respectively:

$\sigma_r = 0.1:$

$$\begin{matrix} 0.01111 & 0.00034 & 0.60653 \\ 0.13534 & 1.00000 & 0.13534 \\ 0.60653 & 0.00034 & 0.01111 \end{matrix}$$

$\sigma_r = 1:$

$$\begin{matrix} 0.95600 & 0.92312 & 0.99501 \\ 0.98020 & 1.00000 & 0.98020 \\ 0.99501 & 0.92312 & 0.95600 \end{matrix}$$

$\sigma_r = 10:$

$$\begin{matrix} 0.99955 & 0.99920 & 0.99995 \\ 0.99980 & 1.00000 & 0.99980 \\ 0.99995 & 0.99920 & 0.99955 \end{matrix}$$

In the last example, the values are very nearly equal, so this range filter treats all values as being (roughly) equally similar.

The idea of the *bilateral filter* is to use *both* domain and range filtering, both with Gaussian filters, and with variances σ_d^2 and σ_r^2. For each pixel in the image, a range filter is created using σ_r^2, which maps the similarity of pixels in that neighborhood. That filter is then convolved with the domain filter which uses σ_d.

By adjusting the size of the filter mask, and of the variances, it is possible to obtain varying amounts of blurring, as well as keeping the edges sharp.

At the time of writing, MATLAB's Image Processing Toolbox does not include bilateral filtering, but both Octave and Python do: the former with the `bilateral` parameter of the `imsmooth` function; the latter with the `denoise_bilateral` method in the `restoration` module of `skimage`.

However, a simple implementation can be written, and one is given at the chapter end.

Using any of these functions and applied to the cameraman image produces the results shown in Figure 5.24, where in each case w gives the size of the filter. Thus, the image in Figure 5.24(a) can be created with our function by:

```
>> cb = bilateral(c,2,2,0,2);
```

MATLAB/Octave

where the first parameter w (in this case 2) produces filters of size $2w + 1 \times 2w + 1$.

Notice that even when a large flat Gaussian is used as the domain filter, as in Figure 5.24(d), the corresponding use of an appropriate range filter keeps the edges sharp.

In Python, a bilateral filter can be applied to produce, for example, the image in Figure 5.24(b) with

```
In :  import skimage.restoration as re
In :  cb = re.denoise_bilateral(c,win_size=7,sigma_range=0.2,\
...:  sigma_spatial=10)
```

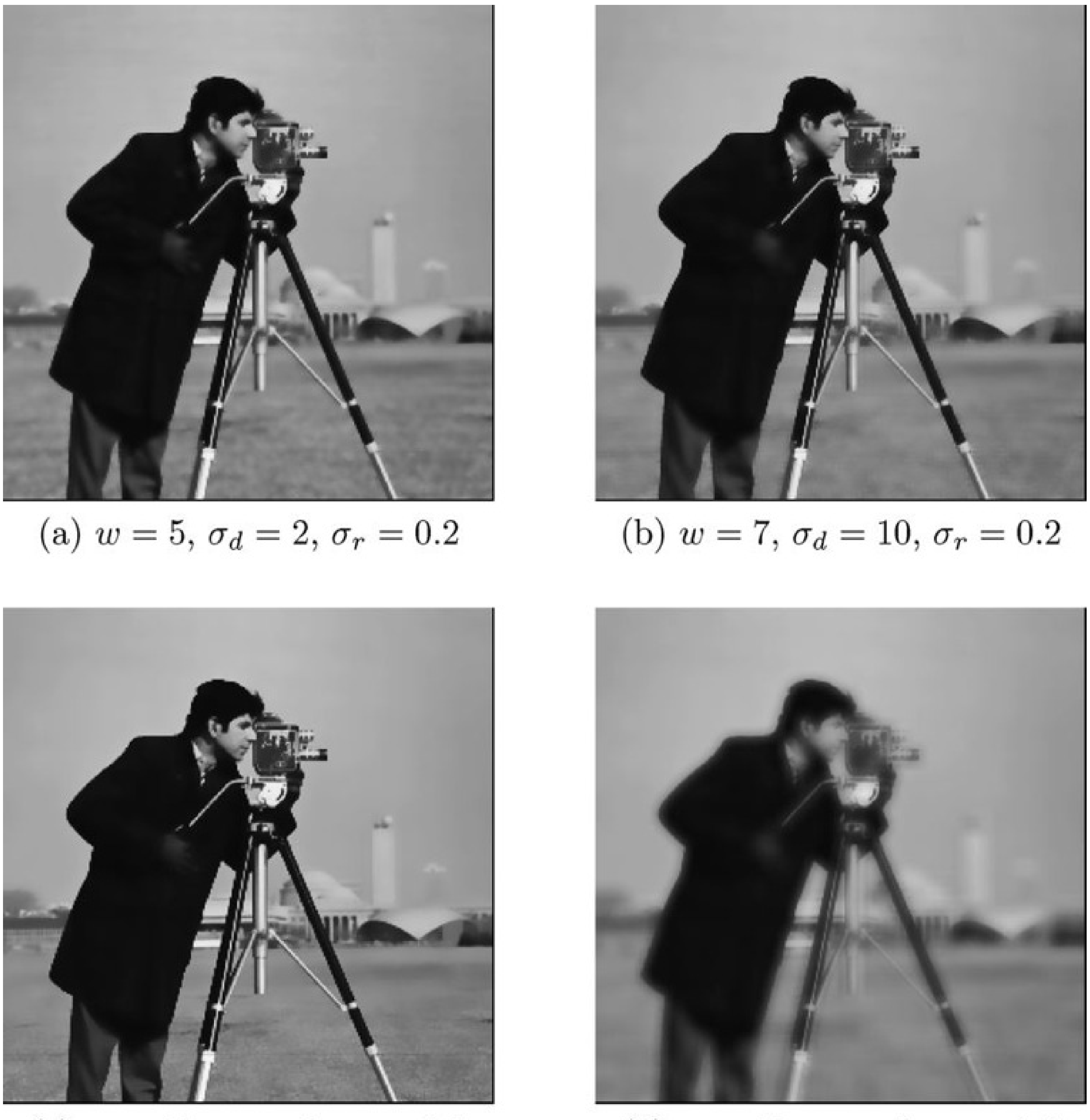
Python

The following is an example of the use of Octave's `imsmooth` function:

```
>> cb = imsmooth(c,'bilateral',sigma_d=2,sigma_r=0.1);
```

Octave

Figure 5.24: Bilateral filtering



(a) $w = 5$, $\sigma_d = 2$, $\sigma_r = 0.2$

(b) $w = 7$, $\sigma_d = 10$, $\sigma_r = 0.2$

(c) $w = 11$, $\sigma_d = 3$, $\sigma_r = 0.1$

(d) $w = 11$, $\sigma_d = 5$, $\sigma_r = 0.5$

and the window size of the filter is automatically generated from the σ values.

5.10 Region of Interest Processing

Often we may not want to apply a filter to an entire image, but only to a small region within it. A non-linear filter, for example, may be too computationally expensive to apply to the entire image, or we may only be interested in the small region. Such small regions within an image are called *regions of interest* or *ROIs*, and their processing is called *region of interest processing*.

Regions of Interest in MATLAB

Before we can process a ROI, we have to define it. There are two ways: by listing the coordinates of a polygonal region; or interactively, with the mouse. For example, suppose we take part of a monkey image:

```
>> m2 = imread('monkey.png');
>> m = m2(56:281,221:412);
```

MATLAB/Octave

and attempt to isolate its head. If the image is viewed with `impixelinfo`, then the coordinates of a hexagon that enclose the head can be determined to be (60, 14), (27, 38), (14, 127), (78, 177), (130, 160) and (139, 69), as shown in Figure 5.25. We can then define a region of interest using the `roipoly` function:

```
>> xi = [60 27 14 78 130 139]
>> yi = [14 38 127 177 160 69]
>> roi=roipoly(m,yi,xi);
```

MATLAB/Octave

Note that the ROI is defined by two sets of coordinates: first the columns and then the rows, taken in order as we traverse the ROI from vertex to vertex. In general, a ROI mask will be a binary image the same size as the original image, with 1s for the ROI, and 0s elsewhere. The function `roipoly` can also be used interactively:

```
>> roi=roipoly(m);
```

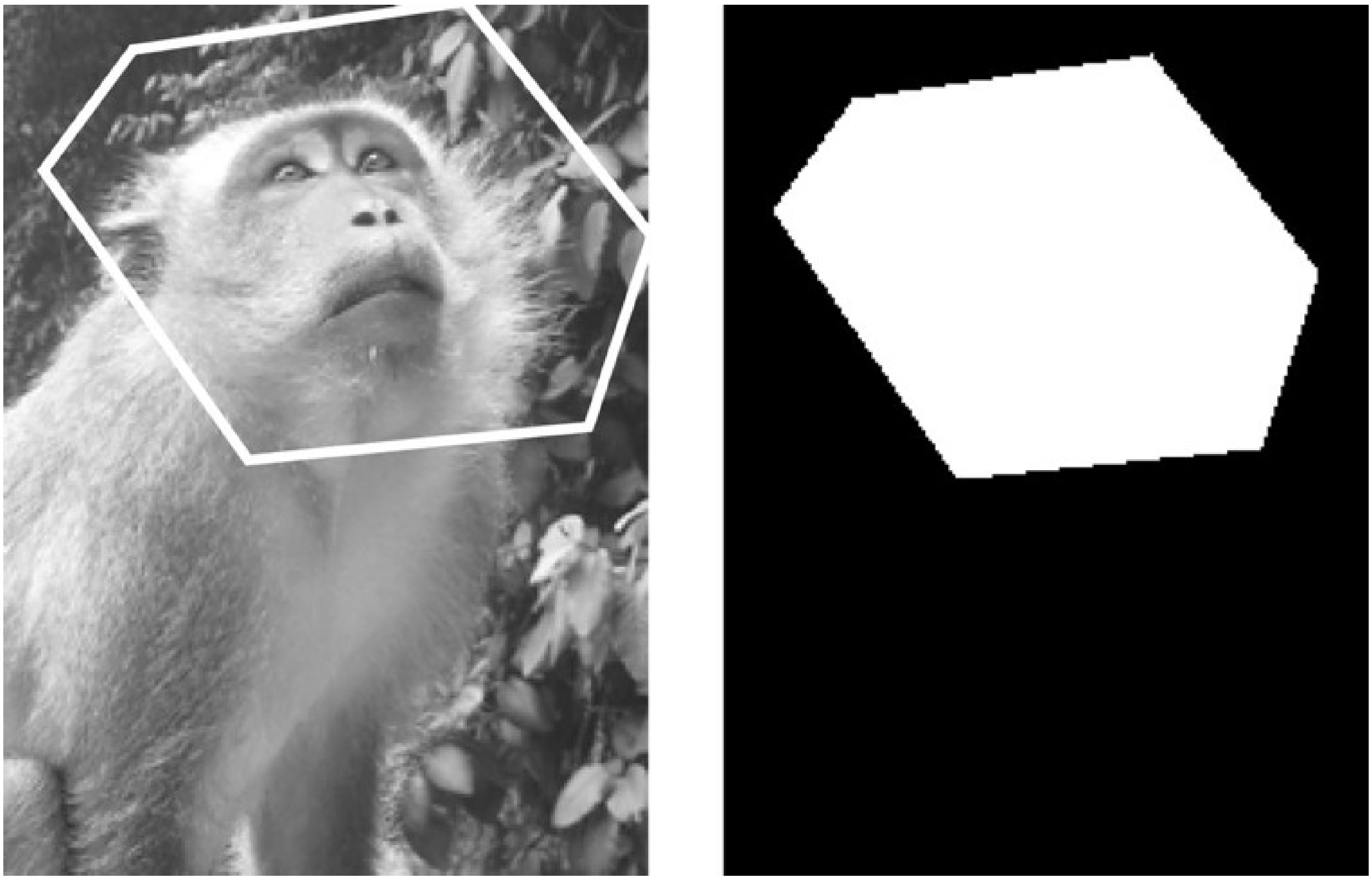
MATLAB/Octave

This will bring up the monkey image (if it isn't shown already). Vertices of the ROI can be selected with the mouse: a left click selects a new vertex, backspace or delete removes the most recently chosen vertex, and a right click finishes the selection.

Region of Interest Filtering

One of the simplest operations on a ROI is spatial filtering; this is implemented with the function `roifilt2`. With the monkey image and the ROI found above, we can experiment:

Figure 5.25: An image with an ROI, and the ROI mask



```
>> a = fspecial('average',15);
>> ma = roifilt2(a,m,roi);
>> u = fspecial('unsharp');
>> mu = roifilt2(u,m,roi);
>> l = fspecial('log');
>> ml = roifilt2(l,m,roi);
>> imshow(ma),figure,imshow(mu),figure,imshow(ml)
```

MATLAB

The images are shown in Figure 5.26.

Figure 5.26: Examples of the use of `roifilt2`

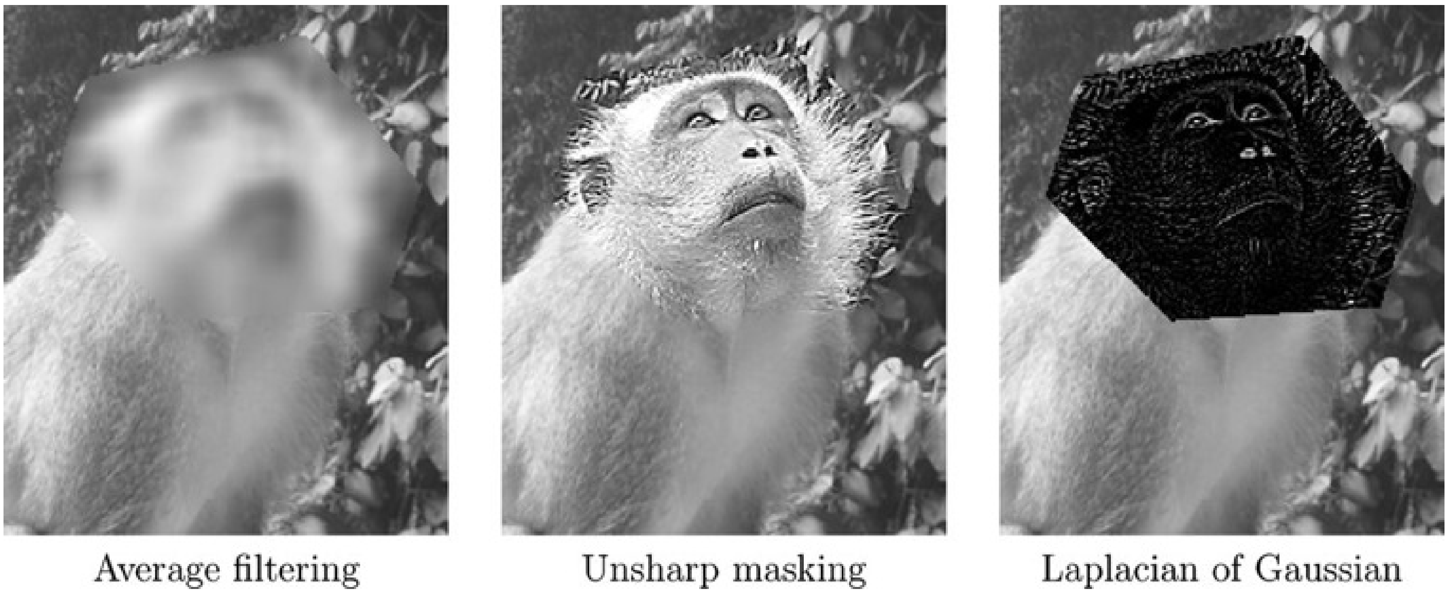
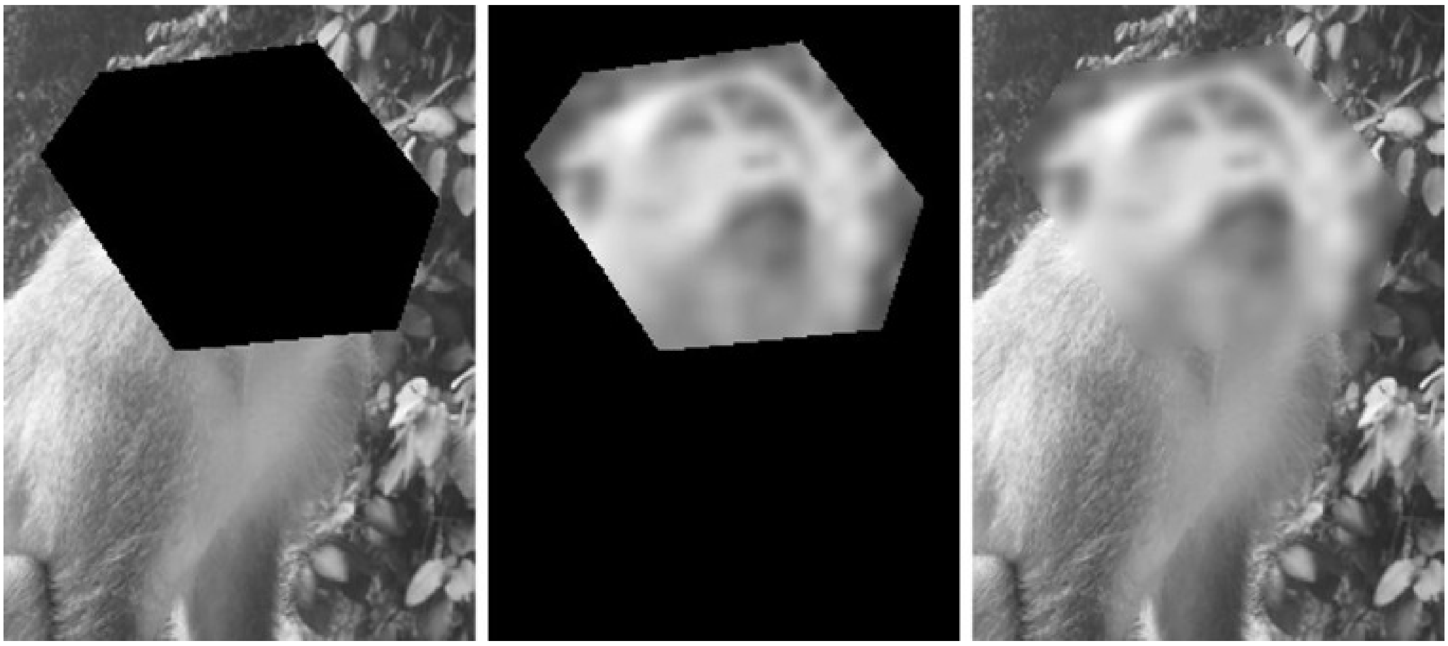


Figure 5.27: ROI filtering with a polygonal mask



Regions of Interest in Octave and Python

Neither Octave nor Python have `roipoly` or `roifilt2` commands. However, in each it is quite straightforward to create a mask defining the region of interest, and use that mask to restrict the act of filtering to the region.

With the monkey above, and the head region, suppose that the matrices for the image and the ROI are M and R, respectively. If M_f is the result of the image matrix after filtering, then

$$M_f R + M(1 - R)$$

will provide the result we want. This is shown in Figure 5.27 where the rightmost image is the sum of the other two.

In Octave, this can be achieved with:

```
>  m2 = imread('monkey.png'); m = m2(56:281,221:412);
>  [r,c] = size(m);
>  xi = [60 27 14 78 130 139]
>  yi = [14 38 127 177 160 69]
>  roi = poly2mask(yi,xi,r,c);
>  f = fspecial('gaussian',9,3);
>  mg = imfilter(m,f));
>  mr = imadd(mg.*roi,m.*~roi)
```

Octave

And in Python, using `zeros_like` which creates an array of zeros the same size as its input, as well as `polygon` from the `draw` module of `skimage`:

```
In:  m2 = io.imread('monkey.png'); m = m2[55:281,220:412]
In:  r,c = m.shape

In:  xi = np.array([60,27,14,78,130,139])
In:  yi = np.array([14,38,127,177,160,69])
In:  roi = np.zeros_like(m)
In:  r,c = polygon(yi,xi)
In:  roi[c,r] = 1

In:  mg = ut.img_as_ubyte(fl.gaussian_filter(m,g))
In:  mr = mg*roi + m*(1-roi)
```

Python

5.11 Programs

This is a simple program (which can be run in MATLAB or Octave) for bilateral filtering.

```
function out = bilateral(im,w,sigma_d,sigma_r)
  im = im2double(im);
  [r,c] = size(im);
  out = zeros(r,c);
  A = padarray(im,[w,w],'symmetric');
  G = fspecial('gaussian',2*w+1,sd);          # the domain filter
  for i = 1+w:r+w-1
      for j = 1+w:c+w-1
          R = A(i-w:i+w,j-w:j+w);             # region to be computed
          H = exp(-(R-A(i,j)).^2/(2*sr^2));   # the range filter
          F = H.*G;
          out(i-w,j-w) = sum(F(:).*R(:))/sum(F(:));
      end;
  end;
  close(h);
end
```

MATLAB/Octave

Exercises

1. The array below represents a small grayscale image. Compute the images that result when the image is convolved with each of the masks (a) to (h) shown. At the edge of the image use a restricted mask. (In other words, pad the image with zeros.)

20	20	20	10	10	10	10	10	10
20	20	20	20	20	20	20	20	10
20	20	20	10	10	10	10	20	10
20	20	10	10	10	10	10	20	10
20	10	10	10	10	10	10	20	10
10	10	10	10	20	10	10	20	10
10	10	10	10	10	10	10	10	10
20	10	20	20	10	10	10	20	20
20	10	10	20	10	10	20	10	20

(a) $\begin{matrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{matrix}$ (b) $\begin{matrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{matrix}$ (c) $\begin{matrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{matrix}$ (d) $\begin{matrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{matrix}$

(e) $\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$ (f) $\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$ (g) $\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$ (h) $\begin{matrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{matrix}$

- 2. Check your answers to the previous question with using `imfilter` (if you are using MATLAB or Octave), or `ndi.correlate` if you are using Python.
- 3. Describe what each of the masks in the previous question might be used for. If you can't do this, wait until Question 5 below.
- 4. Devise a 3 × 3 mask for an "identity filter," which causes no change in the image.
- 5. Choose an image that has a lot of fine detail, and load it. Apply all the filters listed in Question 1 to this image. Can you now see what each filter does?
- 6. Apply larger and larger averaging filters to this image. What is the smallest sized filter for which the fine detail cannot be seen?
- 7. Repeat the previous question with Gaussian filters with the following parameters:

Size	Standard deviation		
[3,3]	0.5	1	2
[7,7]	1	3	6
[11,11]	1	4	8
[21,21]	1	5	10

At what values do the fine details disappear?

- 8. Can you see any observable difference in the results of average filtering and of using a Gaussian filter?
- 9. If you are using MATLAB or Octave, read through the help page of the `fspecial` function, and apply some of the other filters to the cameraman image and to the mandrill image.
- 10. Apply different Laplacian filters to an image of your choice at and to the cameraman images. Which produces the best edge image?
- 11. Is the 3 × 3 median filter separable? That is, can this filter be implemented by a 3×1 filter followed by a 1 × 3 filter?
- 12. Repeat the above question for the maximum and minimum filters.
- 13. Apply a 3 × 3 averaging filter to the middle 9 values of the matrix

$$\begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{bmatrix}$$

and then apply another 3 × 3 averaging filter to the result. Using your answer, describe a 5 × 5 filter that has the effect of two averaging filters. Is this filter separable?

- 14. Use the appropriate commands to produce the outputs shown in Figure 5.5, starting with the diagonally increasing image

20	40	60	80	100
40	60	80	100	120
60	80	100	120	140
80	100	120	140	160
100	120	140	160	180

- 15. Display the difference between the `cmax` and `cmin` images obtained in Section 5.8. You can do this with an image subtraction. What are you seeing here? Can you account for the output of these commands?

- 16. If you are using MATLAB or Octave, then use the `tic` and `toc` timer functions to compare the use of `nlfilter` and `colfilt` functions. If you are using the ipython enhanced shell of Python, try the `%time` function
- 17. Use `colfilt` (MATLAB/Octave) or `generic_filter` (Python) to implement the geometric mean and alpha-trimmed mean filters.
- 18. If you are using MATLAB or Octave, show how to implement the root-mean-square filter.
- 19. Can unsharp masking be used to reverse the effects of blurring? Apply an unsharp masking filter after a 3 × 3 averaging filter, and describe the result.
- 20. Rewrite the Kuwahara filter as a single function that can be applied with either `colfilt` (MATLAB/Octave) or `generic_filter` (Python).

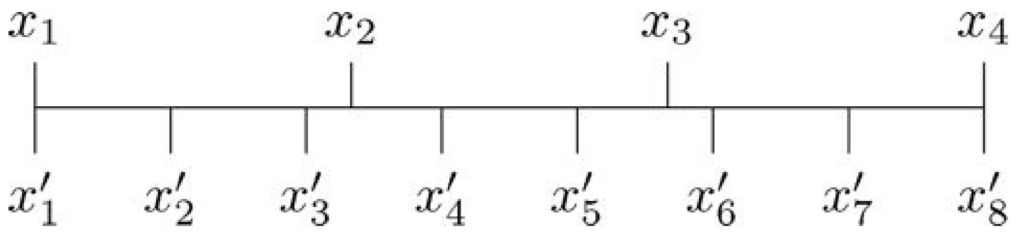
6 Image Geometry

There are many situations in which we might want to change the shape, size, or orientation of an image. We may wish to enlarge an image, to fit into a particular space, or for printing; we may wish also to reduce its size, say for inclusion on a web page. We might also wish to rotate it: maybe to adjust for an incorrect camera angle, or simply for affect. Rotation and scaling are examples of *affine transformations*, where lines are transformed to lines, and in particular parallel lines remain parallel after the transformation. Non-affine geometrical transformations include warping, which we will not consider.

6.1 Interpolation of Data

We will start with a simple problem: suppose we have a collection of 4 values, which we wish to enlarge to 8. How do we do this? To start, we have our points x_1, x_2, x_3, and x_4, which we suppose to be evenly spaced, and we have the values at those points: $f(x_1)$, $f(x_2)$, $f(x_3)$, and $f(x_4)$. Along the line $x_1 \dots x_4$ we wish to space eight points $x'_1, x'_2, \dots, x'_8$. Figure 6.1 shows how this would be done.

Figure 6.1: Replacing four points with eight



Suppose that the distance between each of the x_i points is 1; thus, the length of the line is 3. Thus, since there are seven increments from x'_i to x'_8, the distance between each two will be 3/7 ≈ 0.4286. To obtain a relationship between x and x' we draw Figure 6.1 slightly differently as shown in Figure 6.2. Then

$$x' = \frac{1}{3}(7x - 4),$$
$$x = \frac{1}{7}(3x' + 4).$$

As you see from Figure 6.1, none of the x'_i coincides exactly with an original x_j, except for the first and last. Thus we are going to have to "guess" at possible function values $f(x'_i)$. This guessing at function values is