(a) The appearance of the image?
(b) The elements of the image matrix?

4. What happens if you apply that function to the cameraman image?
5. Experiment with reducing spatial resolution of the following images:

(a) `cameraman.png`
(b) The grayscale emu image
(c) `blocks.png`
(d) `buffalo.png`

In each case, note the point at which the image becomes unrecognizable.
6. Experiment with reducing the quantization levels of the images in the previous question. Note the point at which the image becomes seriously degraded. Is this the same for all images, or can some images stand lower levels of quantization than others? Check your hypothesis with some other grayscale images.
7. Look at a grayscale photograph in a newspaper with a magnifying glass. Describe the colors you see.
8. Show that the 2 × 2 dither matrix D provides appropriate results on areas of unchanging gray. Find the results of $D > G$ when G is a 2 × 2 matrix of values (a) 50, (b) 100, (c) 150, (d) 200.
9. What are the necessary properties of D to obtain the appropriate patterns for the different input gray levels?
10. How do quantization levels effect the result of dithering? Use `gray2ind` to display a grayscale image with fewer grayscales, and apply dithering to the result.
11. Apply each of Floyd-Steinberg, Jarvis-Judice-Ninke, and Stucki error diffusion to the images in Question 5. Which of the images looks best? Which error-diffusion method seems to produce the best results? Can you isolate what aspects of an image will render it most suitable for error-diffusion?

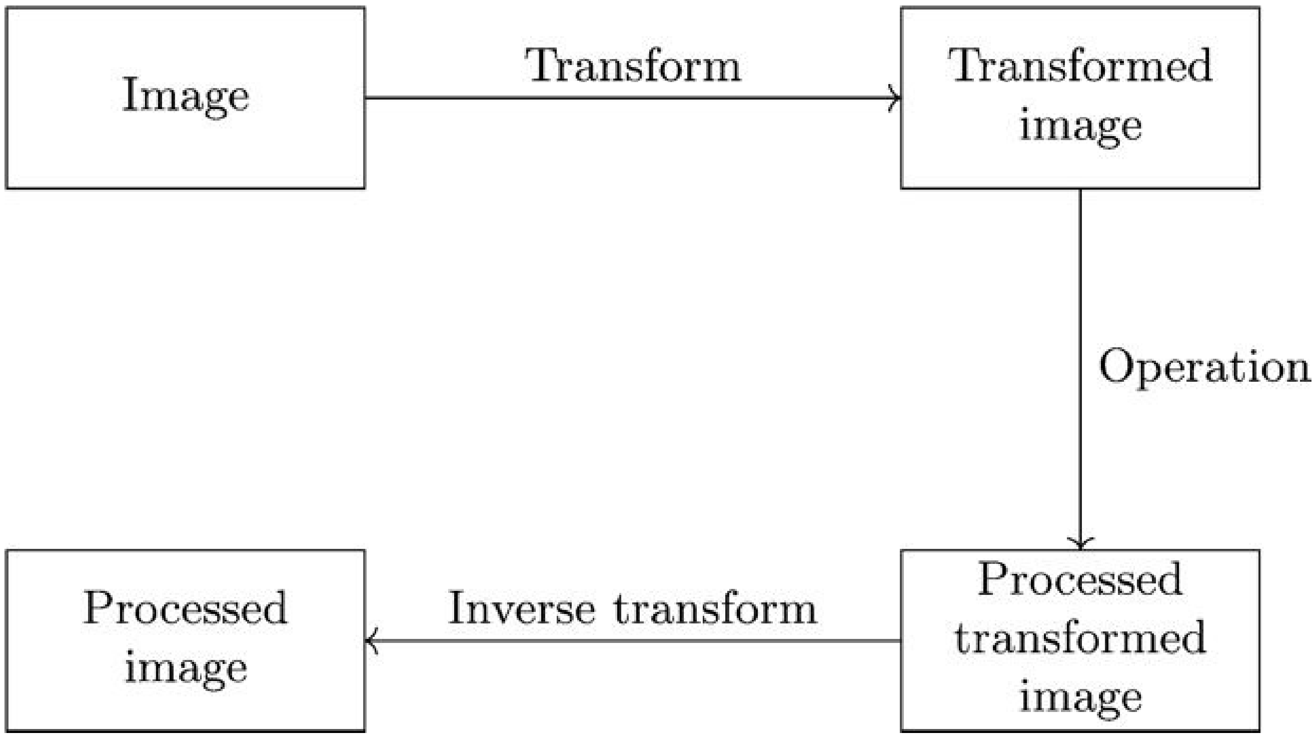
4 Point Processing

4.1 Introduction

Any image processing operation transforms the values of the pixels. However, image processing operations may be divided into three classes based on the information required to perform the transformation. From the most complex to the simplest, they are:

1. **Transforms.** A "transform" represents the pixel values in some other, but equivalent form. Transforms allow for some very efficient and powerful algorithms, as we shall see later on. We may consider that in using a transform, the entire image is processed as a single large block. This may be illustrated by the diagram shown in Figure 4.1.

Figure 4.1: Schema for transform processing



2. **Neighborhood processing.** To change the gray level of a given pixel, we need only know the value of the gray levels in a small neighborhood of pixels around the given pixel.
3. **Point operations.** A pixel's gray value is changed without any knowledge of its surrounds.

Although point operations are the simplest, they contain some of the most powerful and widely used of all image processing operations. They are especially useful in image *preprocessing*, where an image is required to be modified before the main job is attempted.

4.2 Arithmetic Operations

These operations act by applying a simple function

$$y = f(x)$$

to each gray value in the image. Thus, $f(x)$ is a function which maps the range 0 ... 255 onto itself. Simple functions include adding or subtracting a constant value to each pixel:

$$y = x \pm C$$

or multiplying each pixel by a constant:

$$y = Cx.$$

In each case, the output will need to be adjusted in order to ensure that the results are integers in the 0 ... 255 range. We have seen that MATLAB and Octave work by "clipping" the values by setting:

$$y \leftarrow \begin{cases} 255 & \text{if } y > 255, \\ 0 & \text{if } y < 0. \end{cases}$$

and Python works by dividing an overflow by 256 and returning the remainder. We can see this by considering a list of `uint8` values in each system, first in MATLAB/Octave

```
>> x = uint8(0:32:255)
ans =
   0   32   64   96  128  160  192  224

>> x + 100
ans =
   100  132  164  196  228  255  255  255

>> x - 100
ans =
   0    0    0    0   28   60   92  124
```

MATLAB/Octave

and then in Python with `arange`, which produces a list of values as an array:

```
In : x = uint8(array(arange(0,255,32)));x
Out: array([  0,  32,  64,  96, 128, 160, 192, 224], dtype=uint8)

In : x + 100
Out: array([100, 132, 164, 196, 228,   4,  36,  68], dtype=uint8)

In : x - 100
Out: array([156, 188, 220, 252,  28,  60,  92, 124], dtype=uint8)
```

Python

We can obtain an understanding of how these operations affect an image by plotting $y = f(x)$. Figure 4.2 shows the result of adding or subtracting 64 from each pixel in the image. Notice that when we add 128, all gray values of 127 or greater will be mapped to 255. And

when we subtract 128, all gray values of 128 or less will be mapped to 0. By looking at these graphs, we observe that in general adding a constant will lighten an image, and subtracting a constant will darken it.

Figure 4.2: Adding and subtracting a constant in MATLAB and Octave

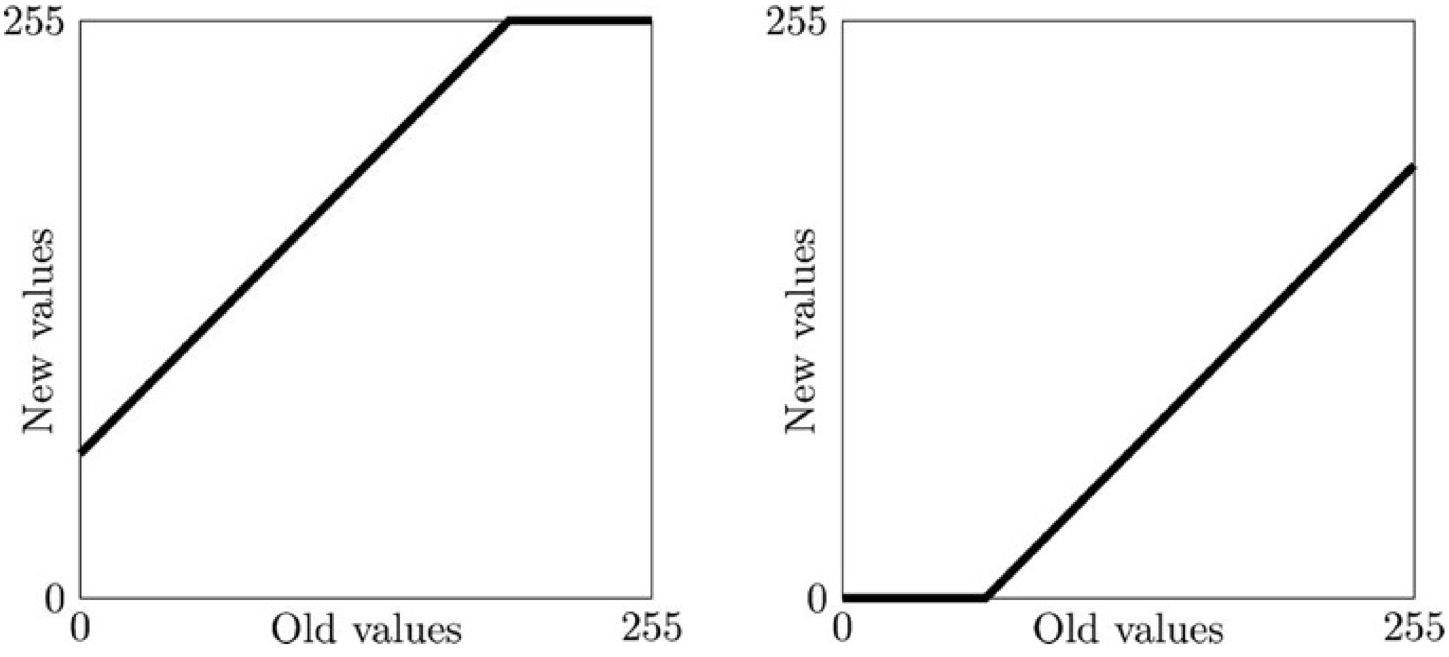
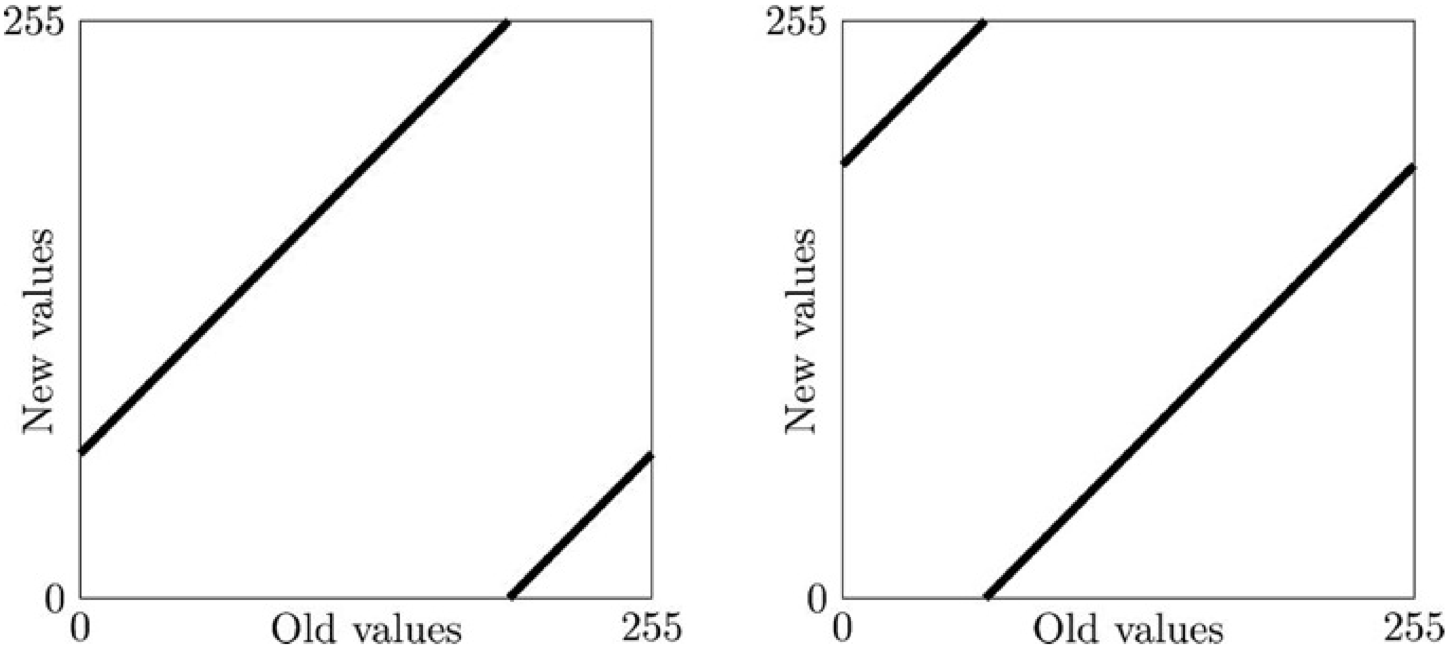


Figure 4.3 shows the results in Python.

Figure 4.3: Adding and subtracting a constant in Python



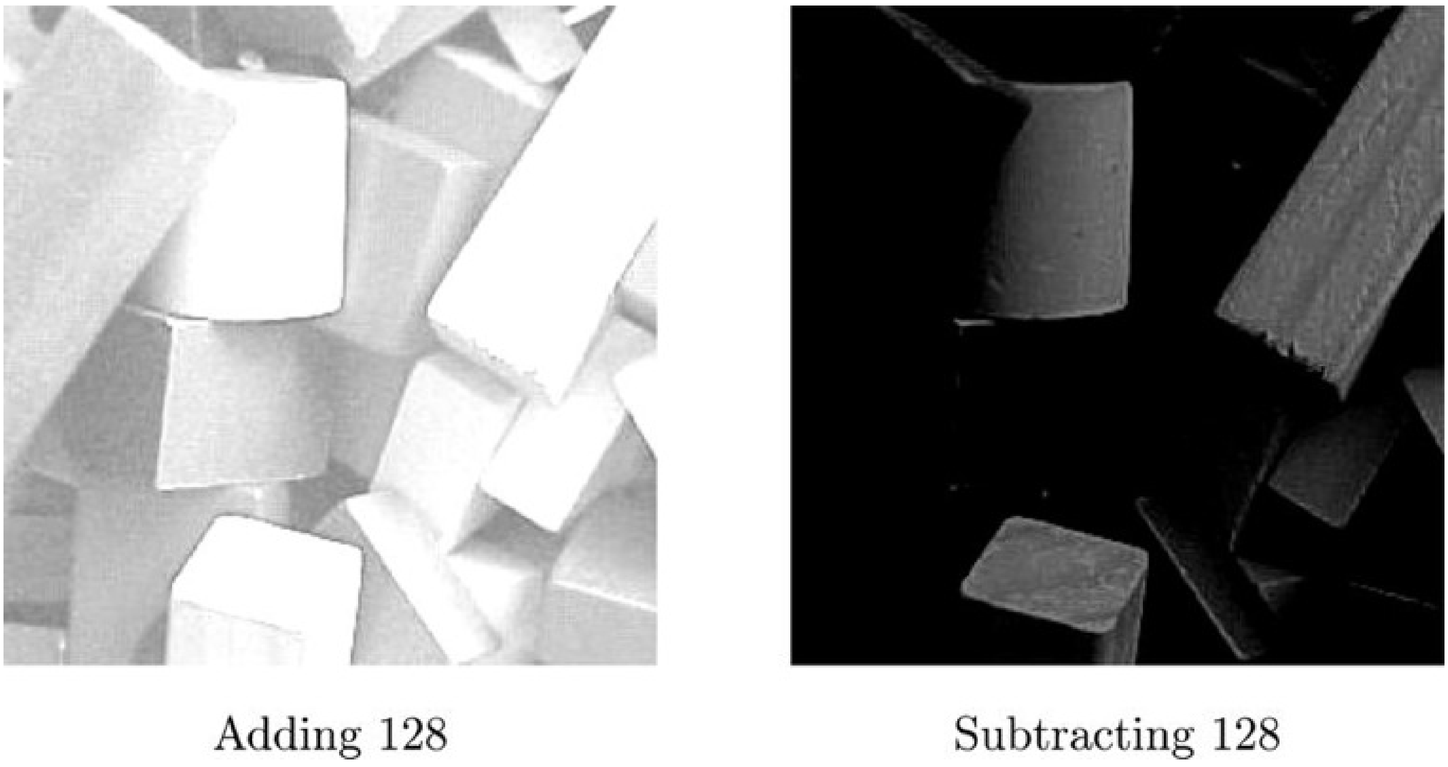
We can test this on the “blocks” image `blocks.png`, which we can see in Figure 1.4. We start by reading the image in:

```
>> b = imread('blocks.png');
>> class(b)
ans =
    uint8
```

MATLAB/Octave

The point of the second command `class` is to find the numeric data type of `b`; it is `uint8`. This tells us that adding and subtracting constants will result in automatic clipping:

Figure 4.4: Arithmetic operations on an image: adding or subtracting a constant



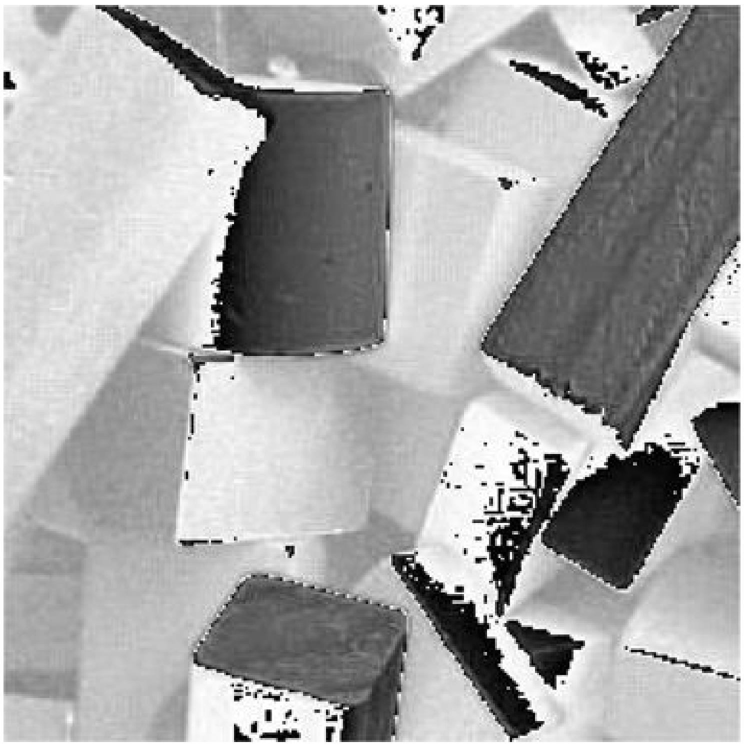
```
>> imshow(b+128)
>> figure, imshow(b-128)
```
MATLAB/Octave

and the results are seen in Figure 4.4. Because of Python's arithmetic, the results of

```
In : b = io.imread('blocks.png');
In : io.imshow(b+128)
```
Python

will not be a brightened image: it is shown in Figure 4.5.

Figure 4.5: Adding 128 to an image in Python



In order to obtain the same results as given by MATLAB and Octave, first we need to use floating point arithmetic, and clip the output with `np.clip`:

```
In :  bf = float64(b)
In :  b1 = uint8(np.clip(bf+128,0,255))
In :  b2 = uint8(np.clip(bf-128,0,255))
```

Python

Then `b1` and `b2` will be the same as the images shown in Figure 4.4.

We can also perform lightening or darkening of an image by multiplication; Figure 4.6 shows some examples of functions that will have these effects. Again, these functions assume clipping. All these images can be viewed with `imshow`; they are shown in Figure 4.7.

Figure 4.6: Using multiplication and division

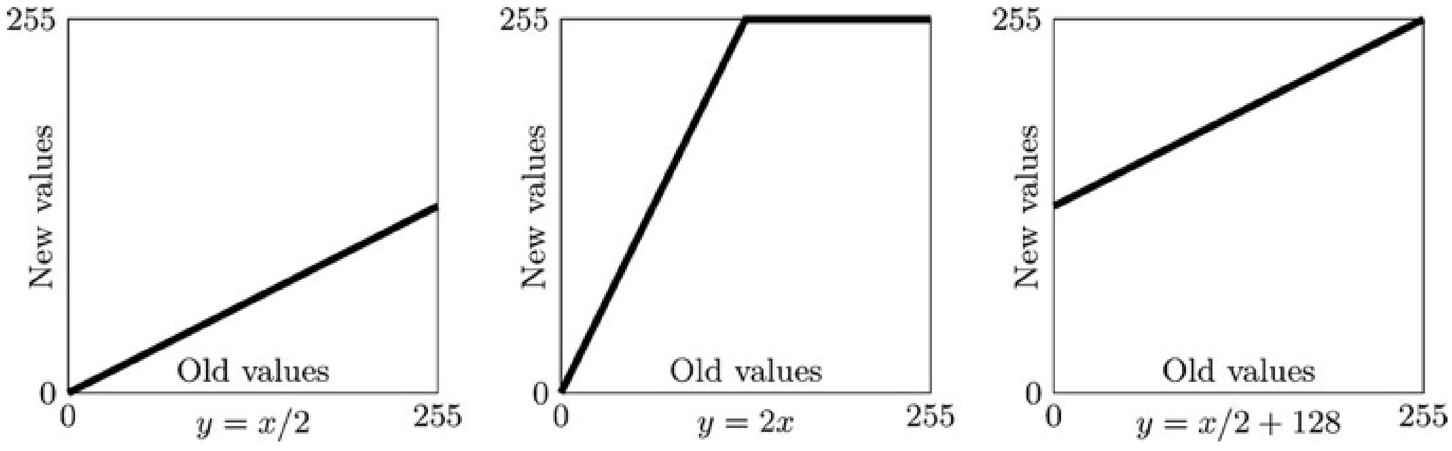
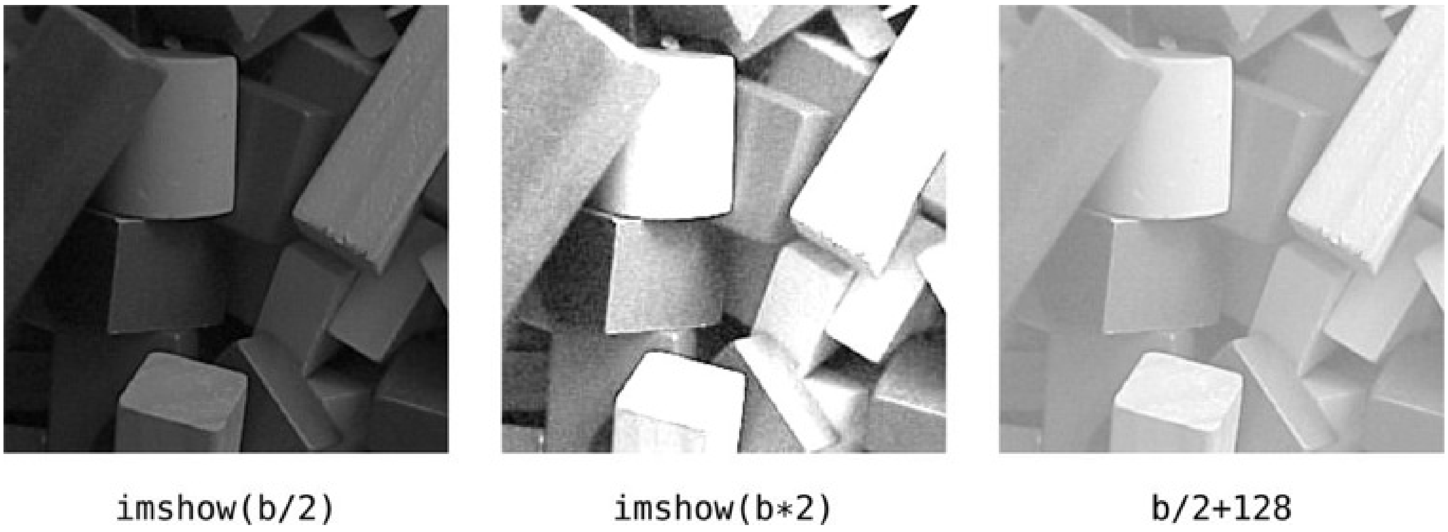


Figure 4.7: Arithmetic operations on an image: multiplication and division



Compare the results of darkening with division by two and subtraction by 128. Note that the division result, although darker than the original, is still quite clear, whereas a lot of information was lost by the subtraction process. This is because in image `b2` all pixels with gray values 128 or less have become zero.

A similar loss of information has occurred by adding 128, and so a better result is obtained by `b/2+128`.

Complements

The *complement* of a grayscale image is its photographic negative. If an image matrix `m` is of type `uint8` and so its gray values are in the range 0 to 255, we can obtain its negative with the command

```
>> 255 - m
```

MATLAB/Octave

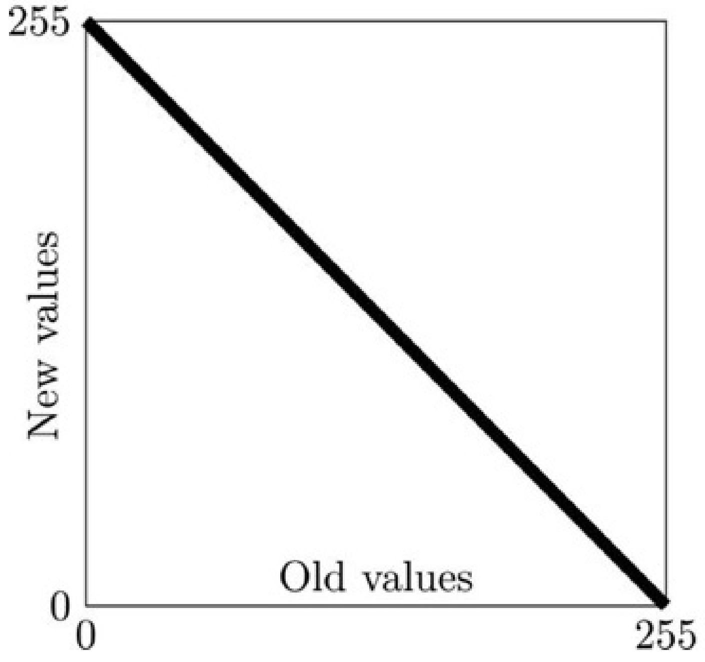
(and the same in Python). If the image is binary, we can use

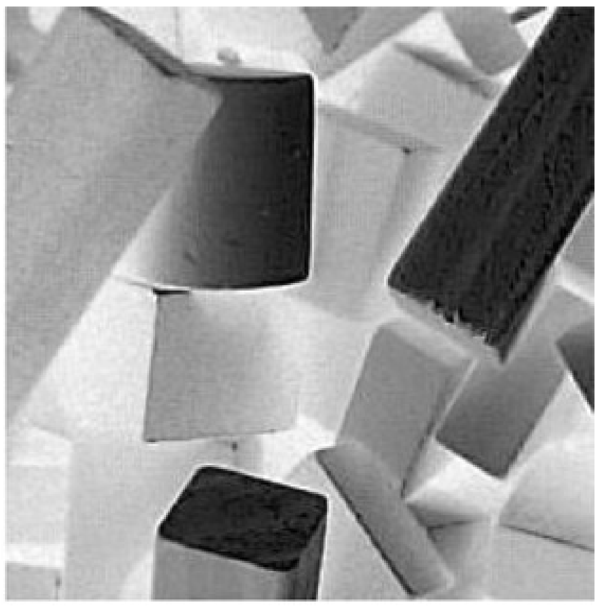
```
>> ~m
```

MATLAB/Octave

which also works for boolean arrays in Python.

Figure 4.8: Image complementation: `255-b`

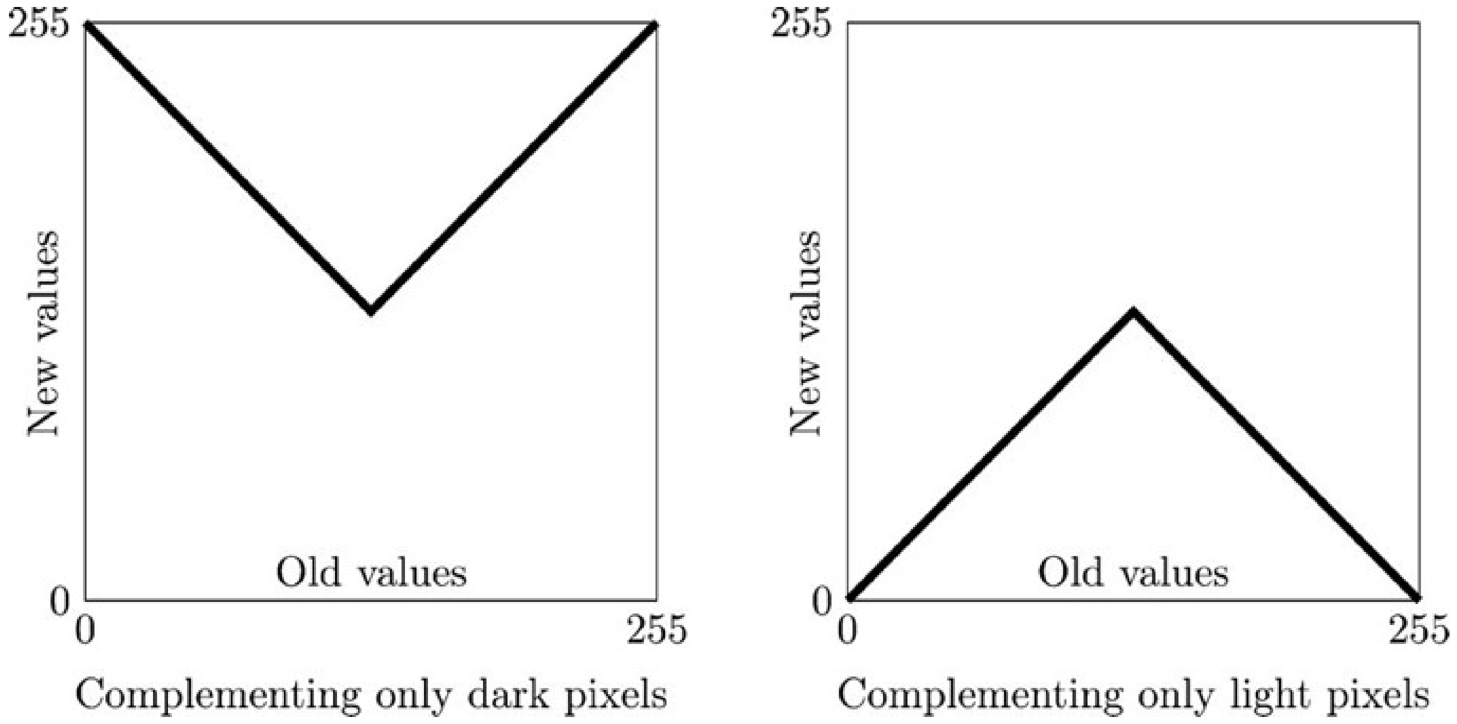




Interesting special effects can be obtained by complementing only *part* of the image; for example by taking the complement of pixels of gray value 128 or less, and leaving other pixels untouched. Or, we could take the complement of pixels that are 128 or greater and leave other pixels untouched. Figure 4.9 shows these functions.

The effect of these functions is called *solarization*, and will be discussed further in Chapter 16.

Figure 4.9: Part complementation



4.3 Histograms

Given a grayscale image, its *histogram* consists of the histogram of its gray levels; that is, a graph indicating the number of times each gray level occurs in the image. We can infer a great deal about the appearance of an image from its histogram, as the following examples indicate:

- In a dark image, the gray levels (and hence the histogram) would be clustered at the lower end.
- In a uniformly bright image, the gray levels would be clustered at the upper end.
- In a well-contrasted image, the gray levels would be well spread out over much of the range.

We can view the histogram of an image in MATLAB by using the `imhist` function:

```
>> c = imread('chickens.png');
>> imshow(c),figure,imhist(c),axis tight
```

MATLAB/Octave

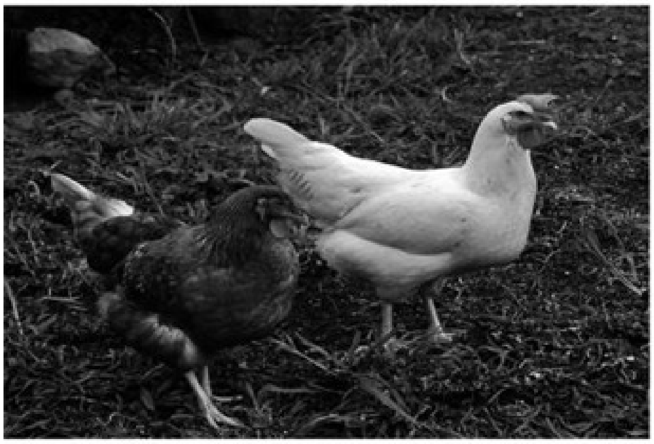
(the `axis tight` command ensures the axes of the histogram are automatically scaled to fit all the values in). In Python, the commands are:

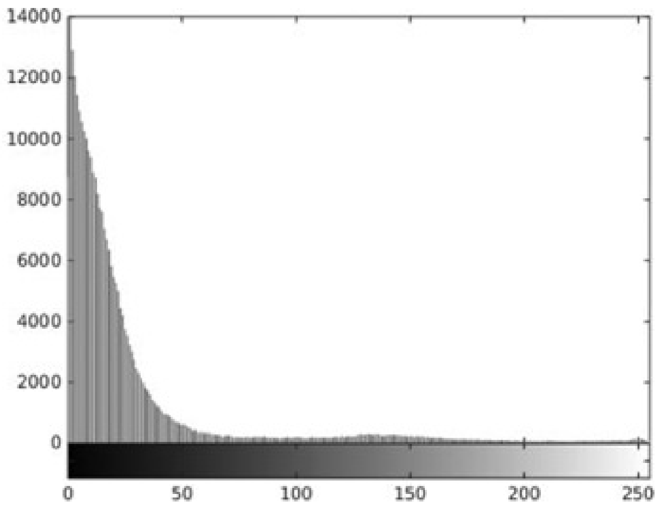
```
In :  c = io.imread('chickens.png')
In :  io.imshow(c)
In :  f = figure(); f.show(plt.hist(c.flatten(),bins=256))
```

Python

The result is shown in Figure 4.10. Since most of the gray values are all clustered

Figure 4.10: The image `chickens.png` and its histogram





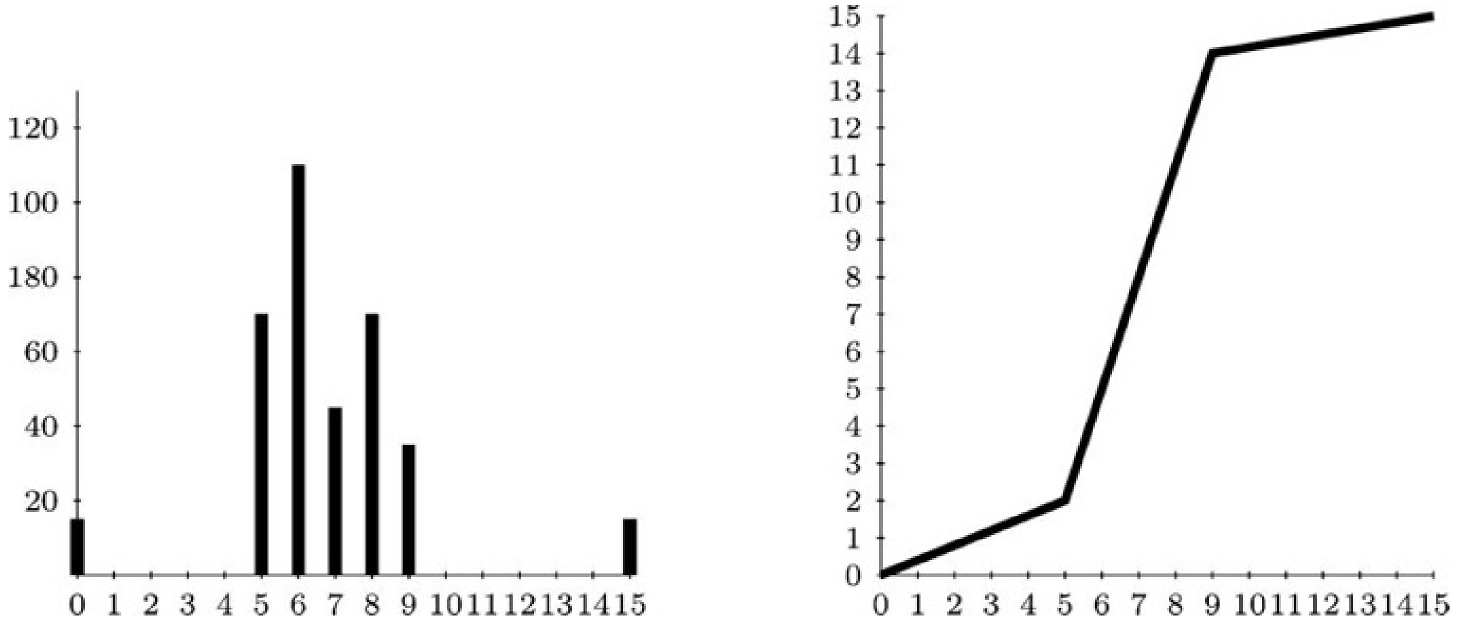
together at the left of the histogram, we would expect the image to be dark and poorly contrasted, as indeed it is.

Given a poorly contrasted image, we would like to enhance its contrast by spreading out its histogram. There are two ways of doing this.

Histogram Stretching (Contrast Stretching)

Suppose we have an image with the histogram shown in Figure 4.11, associated with a table of the numbers n_i of gray values:

Figure 4.11: A histogram of a poorly contrasted image and a stretching function



Gray level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	70	110	45	70	35	0	0	0	0	0	15

(with *n* = 360, as before.) We can stretch the gray levels in the center of the range out by applying the piecewise linear function shown at the right in Figure 4.11. This function has the effect of stretching the gray levels 5–9 to gray levels 2–14 according to the equation:

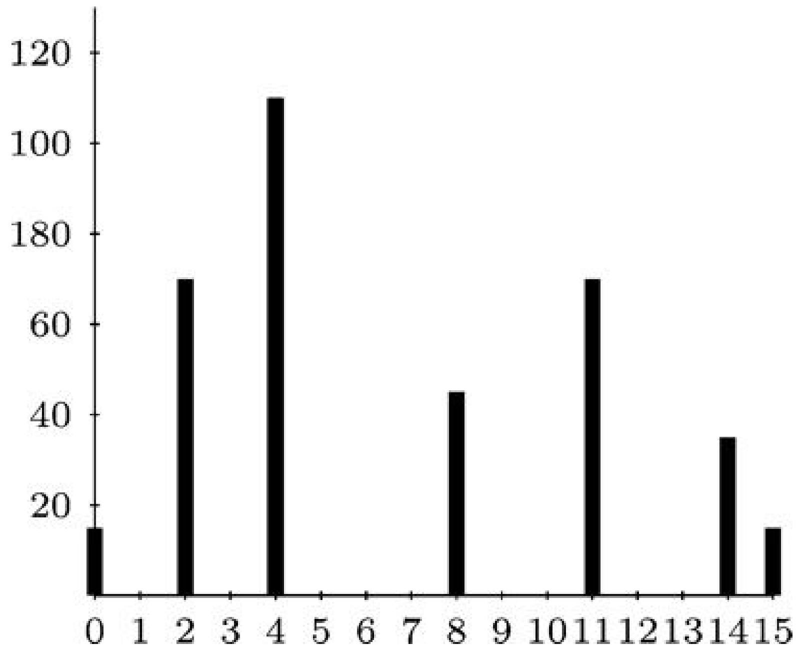
$$j = \frac{14-2}{9-5}(i-5)+2$$

where *i* is the original gray level and *j* its result after the transformation. Gray levels outside this range are either left alone (as in this case) or transformed according to the linear functions at the ends of the graph above. This yields:

i	5	6	7	8	9
j	2	5	8	11	14

and the corresponding histogram given in Figure 4.12:

Figure 4.12: Histogram after stretching



which indicates an image with greater contrast than the original.

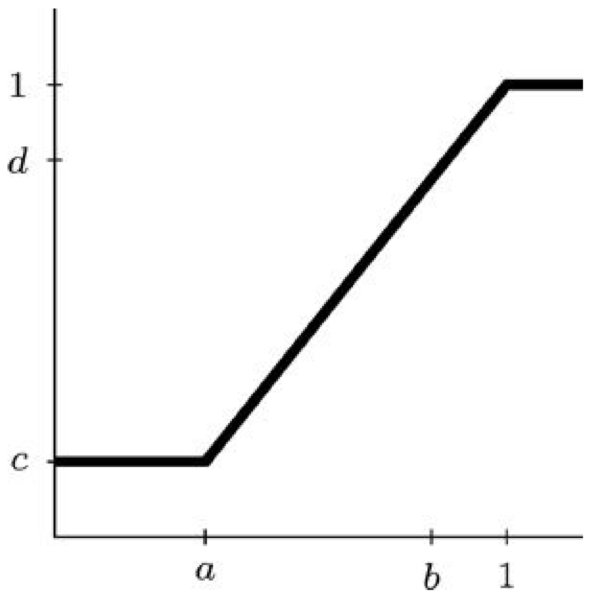
MATLAB/Octave: Use of `imadjust`.

To perform histogram stretching in MATLAB or Octave the `imadjust` function may be used. In its simplest incarnation, the command

```
imadjust(im,[a,b],[c,d])
```

stretches the image according to the function shown in Figure 4.13. Since `imadjust` is

Figure 4.13: The stretching function given by `imadjust`



designed to work equally well on images of type `double`, `uint8`, or `uint16`, the values of *a, b, c*, and *d* must be between 0 and 1; the function automatically converts the image (if needed) to be of type `double`.

Note that `imadjust` does not work quite in the same way as shown in Figure 4.11. Pixel values less than *a* are all converted to *c*, and pixel values greater than *b* are all converted to *d*. If either of `[a, b]` or `[c, d]` are chosen to be `[0, 1]`, the abbreviation `[]` may be used. Thus, for example, the command

```
>> imadjust(im,[],[])
```

MATLAB/Octave

does nothing, and the command

```
>> imadjust(im,[],[1,0])
```

MATLAB/Octave

inverts the gray values of the image, to produce a result similar to a photographic negative.

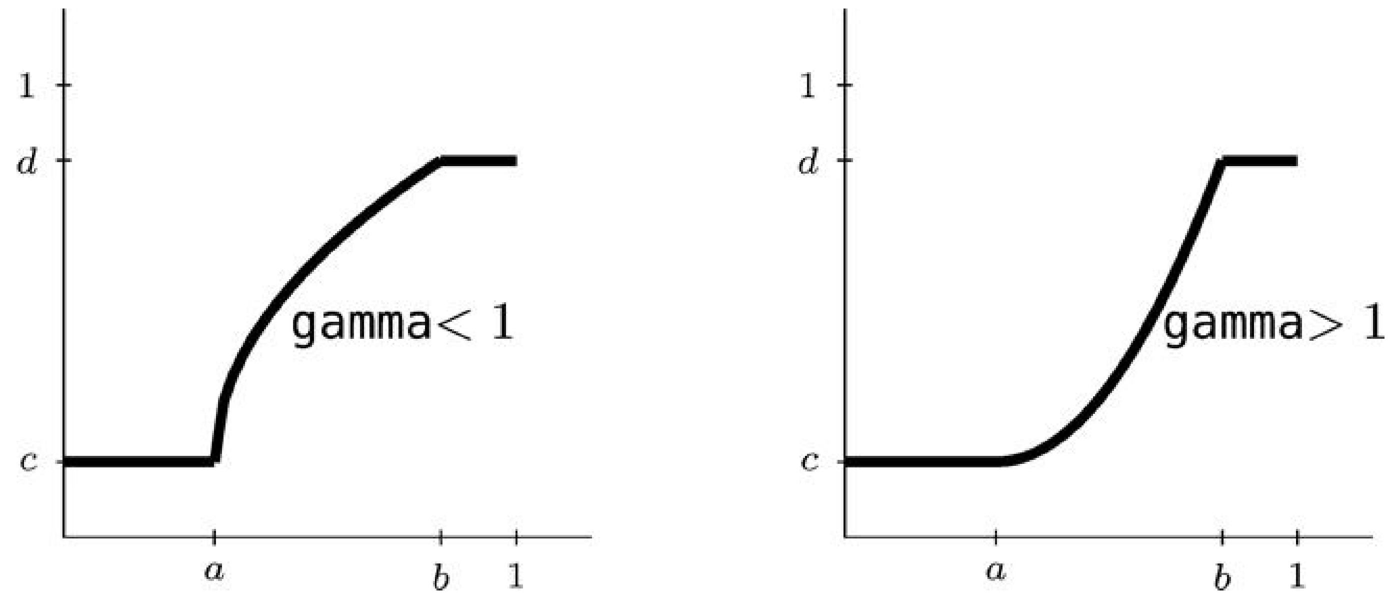
The `imadjust` function has one other optional parameter: the *gamma* value, which describes the shape of the function between the coordinates (*a, c*) and (*b, d*). If `gamma` is equal to 1, which is the default, then a linear mapping is used, as shown in Figure 4.13. However, values less than one produce a function that is concave downward, as shown on the left in Figure 4.14, and values greater than one produce a figure that is concave upward, as shown on the right in Figure 4.14.

The function used is a slight variation on the standard line between two points:

$$y = \left(\frac{x-a}{b-a}\right)^{\gamma}(d-c)+c.$$

Use of the gamma value alone can be enough to substantially change the appearance of the image. For example, with the chickens image:

Figure 4.14: The `imadjust` function with `gamma` not equal to 1

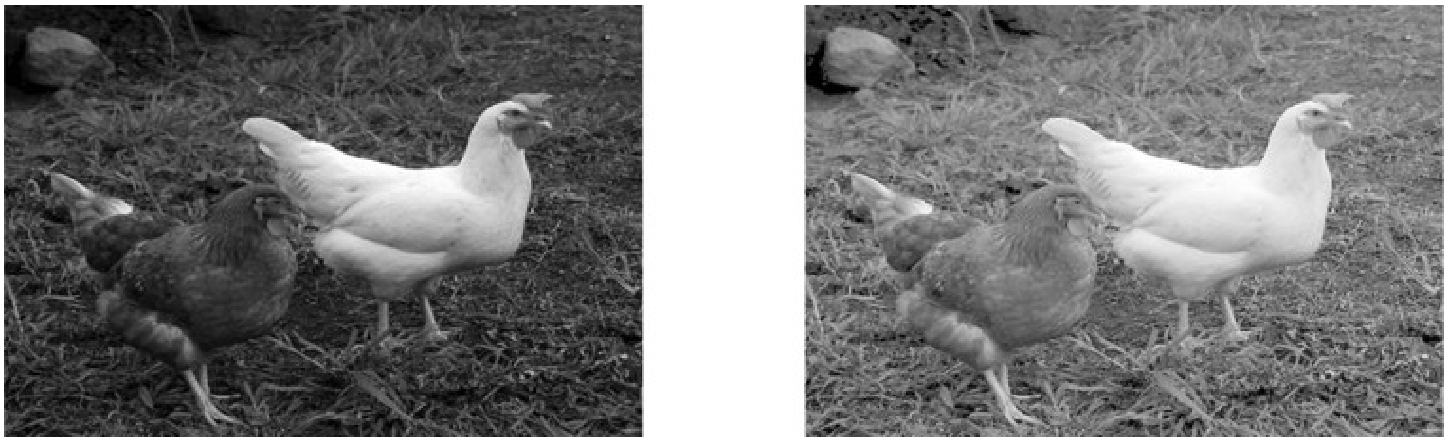


```
>> ca1 = imadjust(t,[],[],0.5);
>> ca2 = imadjust(t,[],[],0.25);
>> imshow(ca1),figure,imshow(ca2)
```

MATLAB/Octave

produces the result shown in Figure 4.15.

Figure 4.15: The chickens image with different adjustments with the gamma value



Both results show background details that are hard to determine in the original image. Finding the correct gamma value for a particular image will require some trial and error.

The `imadjust` stretching function can be viewed with the `plot` function. For example,

```
>> plot(c,ca1,'.'),axis tight
```

MATLAB/Octave

produces the plot shown in Figure 4.16. Since `p` and `ph` are matrices that contain the original values and the values after the `imadjust` function, the `plot` function simply plots them, using dots to do it.

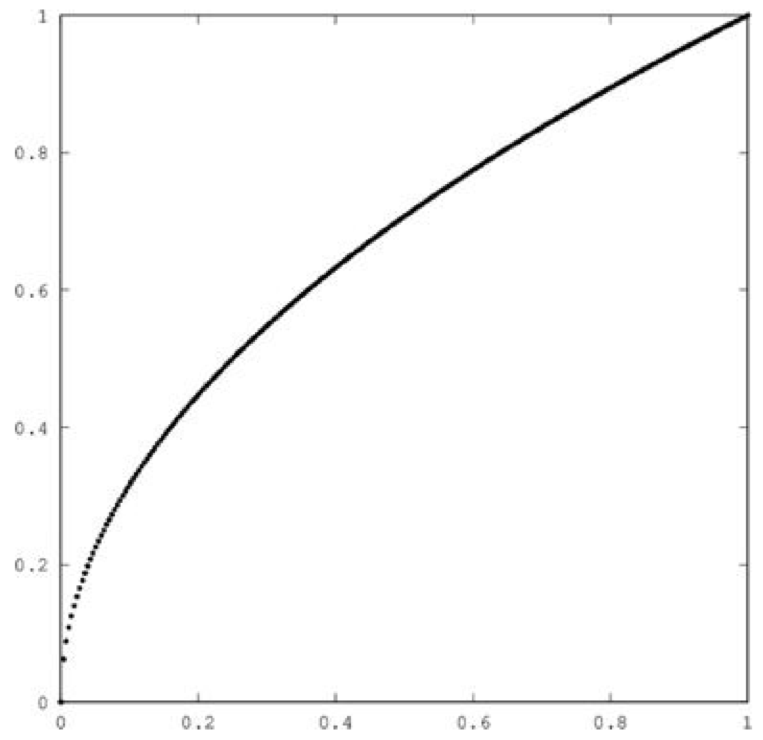
Adjustment in Python

Adjustment methods are held in the `exposure` module of `skimage`. Adjustment with gamma can be achieved with:

```
In :  import skimage.exposure as ex
In :  c = io.imread('chickens.png')
In :  ca1 = ex.adjust_gamma(c,0.5)
```

Python

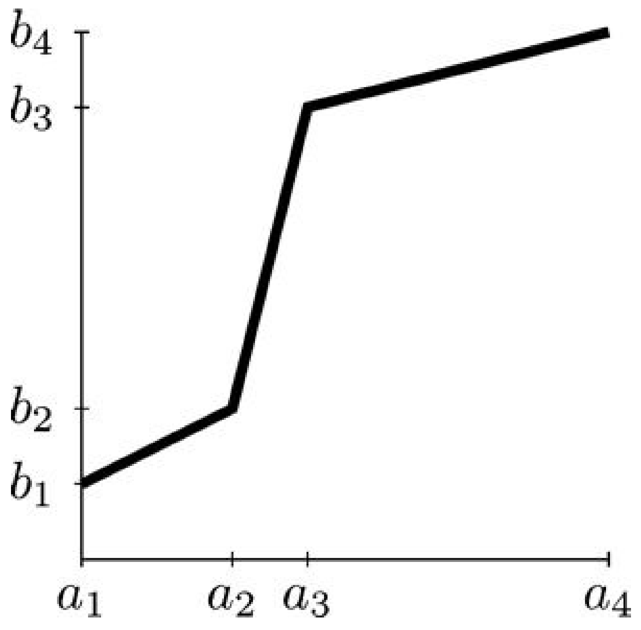
Figure 4.16: The function used in Figure 4.15



A Piecewise Linear Stretching Function

We can easily write our own function to perform piecewise linear stretching as shown in Figure 4.17. This is easily implemented by first specifying the points (a_i, b_i) involved and

Figure 4.17: A piecewise linear stretching function



then using the one-dimensional interpolation function `interp1` to join them. In this case, we might have:

```
>> a = [0 100 150 255]
>> b = [40 100 220 255]
>> lin = interp1(a,b,0:255,'linear');
```

MATLAB/Octave

Then it can be applied to the image with

```
>> cl = uint8(lin(c+1));
```

MATLAB/Octave

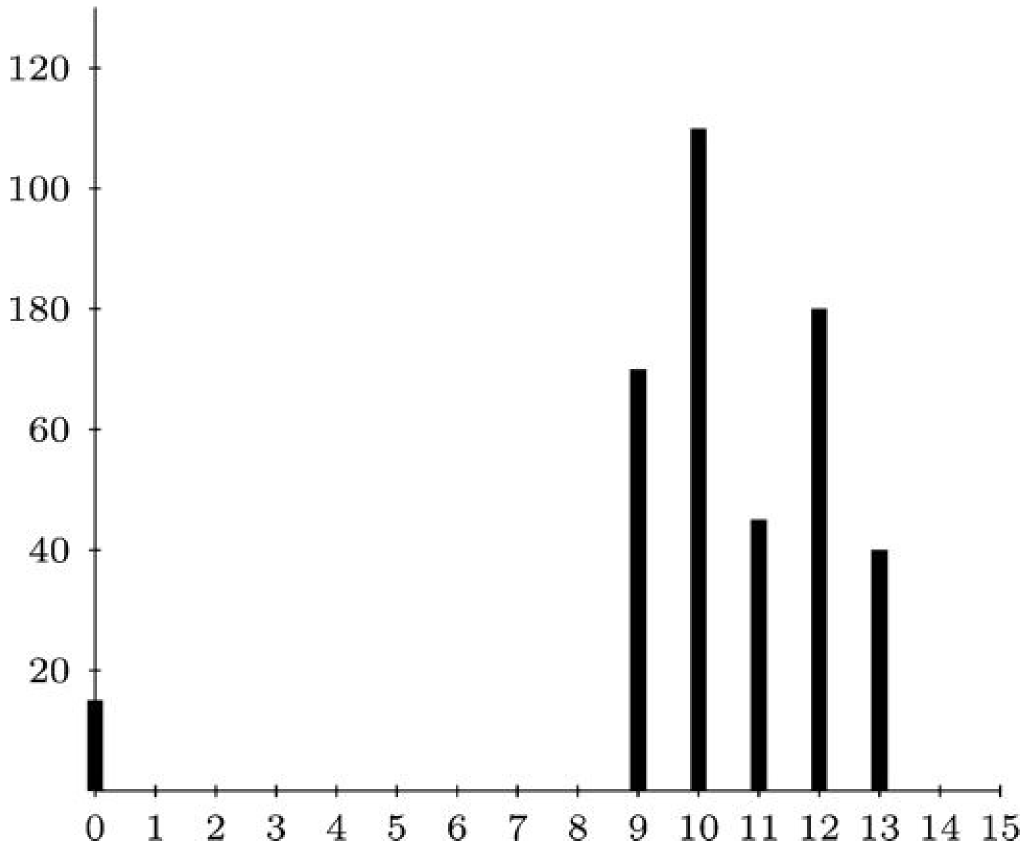
Python also provides easy interpolation:

```
In :  a = array([0, 100, 150, 255])
In :  b = array([40, 100, 220, 255])
In :  lin = np.interp(range(256),a,b)
```

Python

Then it can be applied to the image with

Figure 4.18: Another histogram indicating poor contrast



```
>> cl = uint8(lin[c])
```

Python

Histogram Equalization

The trouble with any of the above methods of histogram stretching is that they require user input. Sometimes a better approach is provided by *histogram equalization*, which is an entirely automatic procedure. The idea is to change the histogram to one that is uniform so that every bar on the histogram is of the same height, or in other words each gray level in the image occurs with the same frequency. In practice, this is generally not possible, although as we shall see the result of histogram equalization provides very good results.

Suppose our image has L different gray levels 0, 1, 2, ... $L - 1$, and that gray level i occurs n_i times in the image. Suppose also that the total number of pixels in the image is n so that $n_0 + n_1 + n_2 + \cdots + n_{L-1} = n$. To transform the gray levels to obtain a better contrasted image, we change gray level i to

$$\left(\frac{n_0 + n_1 + \cdots + n_i}{n}\right)(L-1).$$

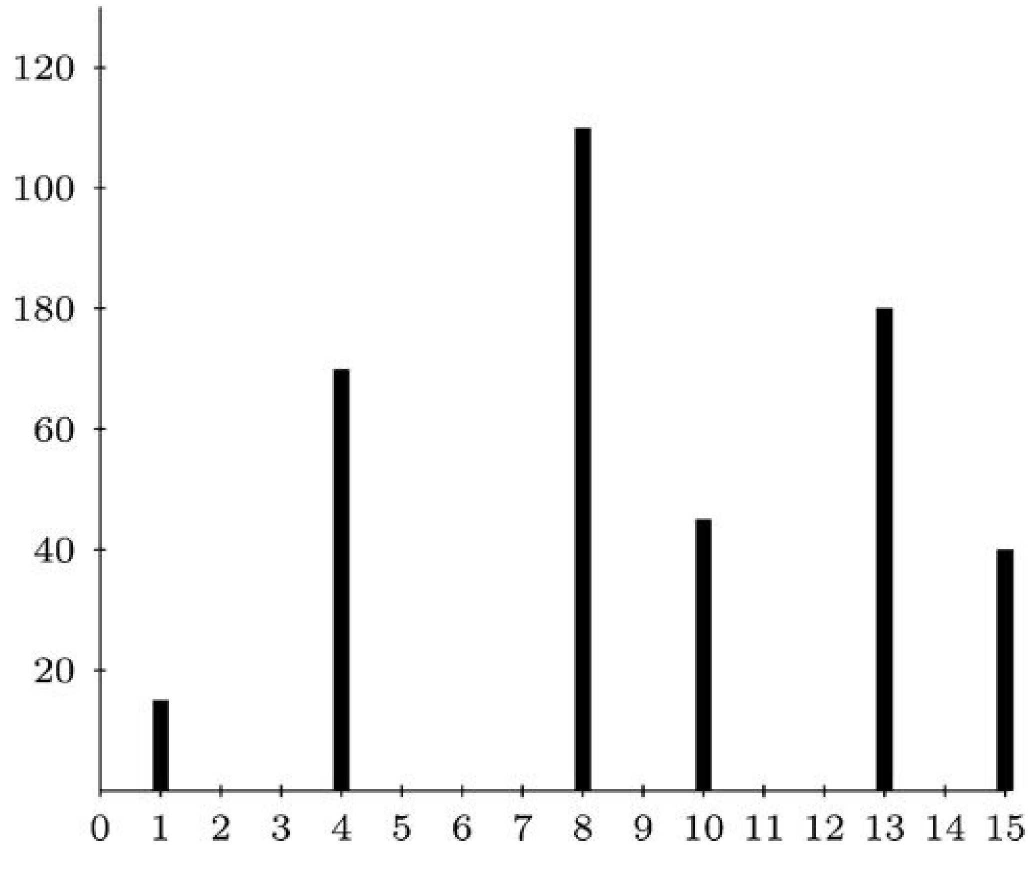
and this number is rounded to the nearest integer.

An Example

Suppose a 4-bit grayscale image has the histogram shown in Figure 4.18 associated with a table of the numbers n_i of gray values:

Gray level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	0	0	0	0	70	110	45	80	40	0	0

Figure 4.19: The histogram of Figure 4.18 after equalization



(with $n = 360$.) We would expect this image to be uniformly bright, with a few dark dots on it. To equalize this histogram, we form running totals of the n_i, and multiply each by 15/360 = 1/24:

Gray level *i*	n_i	$\sum n_i$	**(1/24)**$\sum n_i$	**Rounded value**

0	15	15	0.63	1
1	0	15	0.63	1
2	0	15	0.63	1
3	0	15	0.63	1
4	0	15	0.63	1
5	0	15	0.63	1
6	0	15	0.63	1
7	0	15	0.63	1
8	0	15	0.63	1
9	70	85	3.65	4
10	110	195	8.13	8
11	45	240	10	10
12	80	320	13.33	13
13	40	360	15	15
14	0	360	15	15
15	0	360	15	15

We now have the following transformation of gray values, obtained by reading off the first and last columns in the above table:

Original gray level i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Final gray level j	1	1	1	1	1	1	1	1	1	4	8	10	13	15	15	15

and the histogram of the *j* values is shown in Figure 4.19. This is far more spread out than the original histogram, and so the resulting image should exhibit greater contrast.

To apply histogram equalization in MATLAB or Octave, use the `histeq` function; for example:

```
>> c = imread('chickens.png');
>> ch = histeq(c);
>> imshow(ch),figure,imhist(ch),axis tight
```

MATLAB/Octave

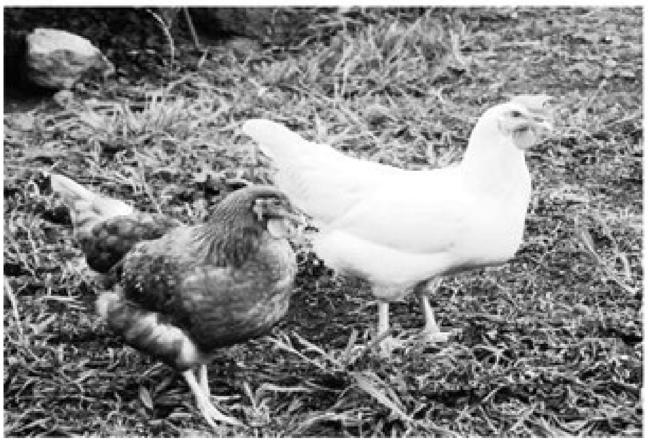
Python supplies the `equalize_hist` method in the `exposure` module:

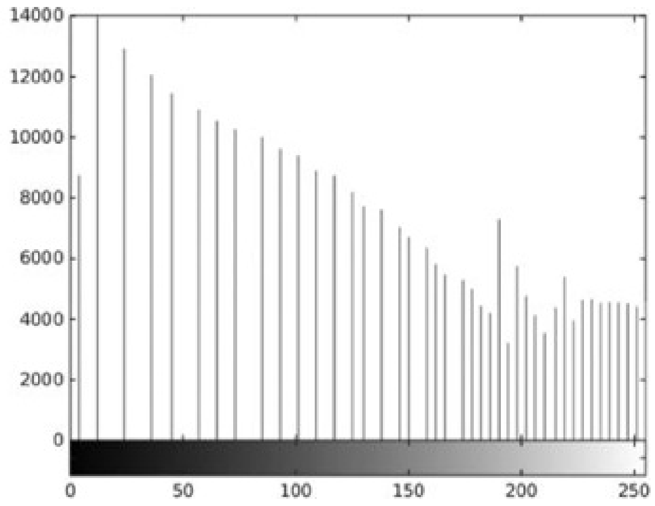
```
In :  c = io.imread('chickens.png')
In :  import sk.exposure as ex
In :  ch = ex.equalize_hist(c)
In :  f = figure(); f.show(plt.hist(ch.flatten(),bins=256))
```

Python

applies histogram equalization to the chickens image, and produces the resulting histogram. These results are shown in Figure 4.20. Notice the far greater spread of the histogram. This

Figure 4.20: The histogram of Figure 4.10 after equalization





corresponds to the greater increase of contrast in the image.

We give one more example, that of a very dark image. For example, consider an underexposed image of a sunset:

```
>> s = imread('sunset.png');
>> imshow(s)
```

MATLAB/Octave

It can be seen both from the image and the histogram shown in Figure 4.21 that the matrix `e` contains mainly low values, and even without seeing the image first we can infer from its histogram that it will appear very dark when displayed. We can display this matrix and its histogram with the usual commands:

```
>> sh = imhist(s);
>> imhow(sh),figure,imhist(sh),axis auto
```

MATLAB/Octave

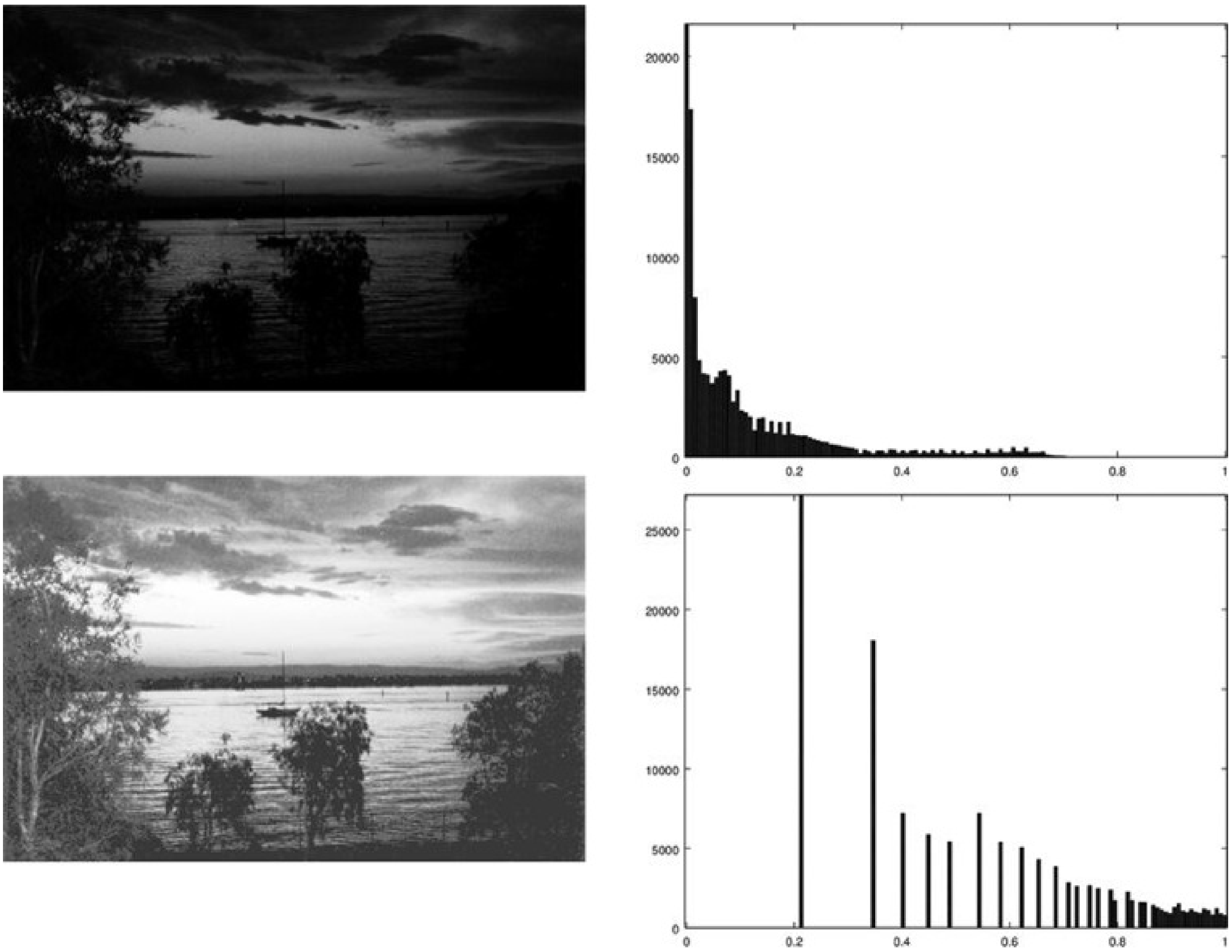
or in Python as

```
In :  s = io.imread('sunset.png')
In :  io.imshow(s)
In :  f = figure(); f.show(plt.hist(s.flatten(),bins=256))
```

Python

and improve the contrast of the image with histogram equalization, as well as displaying the resulting histogram. The results are shown in the top row of Figure 4.21.

Figure 4.21: The sunset image and its histogram, with equalization



As you see, the very dark image has a corresponding histogram heavily clustered at the lower end of the scale.

But we can apply histogram equalization to this image, and display the results:

```
>> sh = histeq(s);
>> imshow(sh),figure,imhist(sh),axis tight
```
MATLAB/Octave

or

```
In :  sh = ex.equalize_hist(s)
In :  f = figure(); f.show(plt.hist(sh.flatten(),bins=256))
```
Python

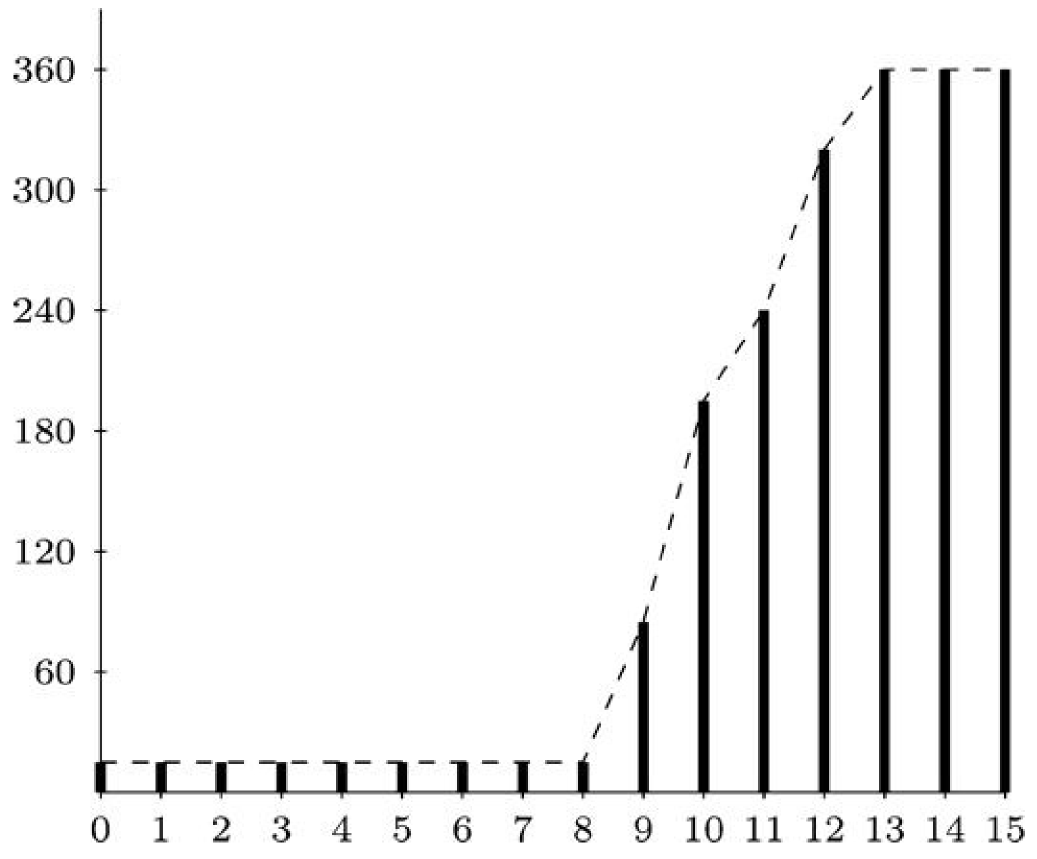
and the results are shown in the bottom row of Figure 4.21. Note that many details that are obscured in the original image are now quite clear: the trunks of the foreground trees, the ripples on the water, the clouds at the very top.

Why It Works

Consider the histogram in Figure 4.18. To apply histogram stretching, we would need to stretch out the values between gray levels 9 and 13. Thus, we would need to apply a piecewise function similar to that shown in Figure 4.11.

Let's consider the cumulative histogram, which is shown in Figure 4.22. The dashed line is simply joining the top of the histogram bars. However, it can be interpreted as an appropriate histogram stretching function. To do this, we need to scale the *y* values so that they are between 0 and 15, rather than 0 and 360. But this is precisely the method described in Section 4.3.

Figure 4.22: The cumulative histogram



As we have seen, none of the example histograms, after equalization, are uniform. This is a result of the discrete nature of the image. If we were to treat the image as a continuous function *f*(*x, y*), and the histogram as the area between different contours (see, for example, Castleman [7]), then we can treat the histogram as a probability density function. But the corresponding cumulative density function will always have a uniform histogram; see, for example, Hogg and Craig [17].

4.4 Lookup Tables

Point operations can be performed very effectively by the use of a *lookup table*, known more simply as an LUT. For operating on images of type `uint8`, such a table consists of a single array of 256 values, each value of which is an integer in the range 0 ... 255. Then our operation can be implemented by replacing each pixel value p by the corresponding value t_p in the table.

For example, the LUT corresponding to division by 2 looks like:

Index:	0	1	2	3	4	5	...	250	251	252	253	254	255
LUT:	0	0	1	1	2	2	...	125	125	126	126	127	127

This means, for example, that a pixel with value 4 will be replaced with 2; a pixel with value 253 will be replaced with value 126.

If `T` is a lookup table, and `im` is an image, then the lookup table can be applied by the simple command

`T(im)` or `T[im]`

depending on whether we are working in MATLAB/Octave or Python.

For example, suppose we wish to apply the above lookup table to the blocks image. We can create the table with

```
>> T = uint8(floor(0:255)/2);
```

MATLAB/Octave

or

```
In :  T = uint8(arange(256))/2
```

Python

apply it to the blocks image `b` with

```
>> b2 = T(b);
```

MATLAB/Octave

or

```
In :  b2 = T[b]
```
Python

The image `b2` is of type `uint8`, and so can be viewed directly with the viewing command.

The piecewise stretching function discussed above was in fact an example of the use of a lookup table.

Exercises

Image Arithmetic

1. Describe lookup tables for

   (a) Multiplication by 2
   (b) Image complements

2. Enter the following command on the blocks image `b`:

```
b2 = (b/64)*64
```

Comment on the result. Why is the result not equivalent to the original image?

3. Replace the value 64 in the previous question with 32 and 16. **Histograms**

4. Write informal code to calculate a histogram *h*[*f*] of the gray values of an image *f*[*row*][*col*].

5. The following table gives the number of pixels at each of the gray levels 0–7 in an image with those gray values only:

0	1	2	3	4	5	6	7
3244	3899	4559	2573	1428	530	101	50

Draw the histogram corresponding to these gray levels, and then perform a histogram equalization and draw the resulting histogram.

6. The following tables give the number of pixels at each of the gray levels 0–15 in an image with those gray values only. In each case, draw the histogram corresponding to these gray levels, and then perform a histogram equalization and draw the resulting histogram.

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	40	60	75	80	75	65	55	50	45	40	35	30	25	20	30

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	40	80	45	110	70	0	0	0	0	0	0	0	0	15

7. The following small image has gray values in the range 0 to 19. Compute the gray level histogram and the mapping that will equalize this histogram. Produce an 8 × 8 grid containing the gray values for the new histogram-equalized image.

```
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.