(a) A binary image,
(b) An indexed color image,
(c) A true color image.

7. The following shows the hexadecimal dump of a PNG file:

```
000000  8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
000010  0000 012c 0000 00f6 0800 0000 0049 c4e5  ...,.........I..
000020  5400 0000 0774 494d 4507 d209 1314 1f0c  T....tIME.......
000030  035d c49d 0000 0027 7445 5874 436f 7079  .]....'tEXtCopy
```

Determine the height and width of this image (in pixels), and whether it is a grayscale or color image.

- 8. Repeat the previous question with this BMP image:

```
000000  424d 3603 0000 0000 0000 3600 0000 2800  BM6.......6...(.
000010  0000 1000 0000 1000 0000 0100 1800 0000  ................
000020  0000 0003 0000 c40e 0000 c40e 0000 0000  ................
000030  0000 0000 0000 e1f5 ffe1 f5ff e1f5 ffe1  ................
```

2

http://www.imagemagick.org/

3 Image Display

3.1 Introduction

We have touched briefly in Chapter 2 on image display. In this chapter, we investigate this matter in more detail. We look more deeply at the use of the image display functions in our systems, and show how spatial resolution and quantization can affect the display and appearance of an image. In particular, we look at image quality, and how that may be affected by various image attributes. Quality is of course a highly subjective matter: no two people will agree precisely as to the quality of different images. However, for human vision in general, images are preferred to be sharp and detailed. This is a consequence of two properties of an image: its spatial resolution, and its quantization.

An image may be represented as a matrix of the gray values of its pixels. The problem here is to display that matrix on the computer screen. There are many factors that will effect the display; they include:

1. Ambient lighting
2. The monitor type and settings
3. The graphics card
4. Monitor resolution

The same image may appear very different when viewed on a dull CRT monitor or on a bright LCD monitor. The resolution can also affect the display of an image; a higher resolution may result in the image

taking up less physical area on the screen, but this may be counteracted by a loss in the color depth: the monitor may only be able to display 24-bit color at low resolutions. If the monitor is bathed in bright light (sunlight, for example), the display of the image may be compromised. Furthermore, the individual's own visual system will affect the appearance of an image: the same image, viewed by two people, may appear to have different characteristics to each person. For our purpose, we shall assume that the computer setup is as optimal as possible, and the monitor is able to accurately reproduce the necessary gray values or colors in any image.

3.2 The `imshow` Function

Grayscale Images

MATLAB and Octave can display a grayscale image with `imshow` if the image matrix `x` is of type `uint8`, or of type `double` with all values in the range 0.0 – 1.0. They can also display images of type `uint16`, but for Octave this requires that the underlying ImageMagick library is compiled to handle 16-bit images. Then

```
>> c = imread('caribou.png');
>> cd = double(c);
>> imshow(c),figure,imshow(cd)
```

MATLAB/Octave

will display `x` as an image.

This means that any image processing that produces an output of type `double` must either be scaled to the appropriate range or converted to type `uint8` before display.

If scaling is not done, then `imshow` will treat all values greater than 1 as white, and all values lower than 0 as black. Suppose we take an image and convert it to type `double`:

The results are shown in Figure 3.1.

Figure 3.1: An attempt at data type conversion

(a) The original image (b) After conversion to type `double`

As you can see, Figure 3.1(b) doesn't look much like the original picture at all! This is because any original values that were greater than 1 are now being displayed as white. In fact, the minimum value is 21, so that *every* pixel will be displayed as white. To display the matrix `cd`, we need to scale it to the range 0–1. This is easily done simply by dividing all values by 255:

```
>> imshow(cd/255)
```

MATLAB/Octave

and the result will be the caribou image as shown in Figure 3.1(a).

We can vary the display by changing the scaling of the matrix. Results of the commands:

```
>> imshow(cd/512)
>> imshow(cd/128)
```

MATLAB/Octave

are shown in Figure 3.2.

Figure 3.2: Scaling by dividing an image matrix by a scalar

(a) The matrix `cd` divided by 512

(b) The matrix `cd` divided by 128

Dividing by 512 darkens the image, as all matrix values are now between 0 and 0.5, so that the brightest pixel in the image is a mid-gray. Dividing by 128 means that the range is 0–2, and all pixels in the range 1–2 will be displayed as white. Thus, the image has an over-exposed, washed-out appearance.

The display of the result of a command whose output is a matrix of type `double` can be greatly affected by a judicious choice of a scaling factor.

We can convert the original image to `double` more properly using the function `im2double`. This applies correct scaling so that the output values are between 0 and 1. So the commands

```
>> cd = im2double(c);
>> imshow(cd)
```

MATLAB/Octave

will produce a correct image. It is important to make the distinction between the two functions `double` and `im2double: double` changes the data type but does not change the numeric values; `im2double` changes both the numeric data type *and* the values. The exception of course is if the original image is of type `double`, in which case `im2double` does nothing. Although the command `double` is not of much use for direct image display, it can be very useful for image arithmetic. We have seen examples of this above with scaling.

Corresponding to the functions `double` and `im2double` are the functions `uint8` and `im2uint8`. If we take our image `cd` of type `double`, properly scaled so that all elements are between 0 and 1, we can convert it back to an image of type `uint8` in two ways:

```
>> c2 = uint8(255*cd);
>> c3 = im2uint8(cd);
```

MATLAB/Octave

Use of `im2uint8` is to be preferred; it takes other data types as input, and always returns a correct result.

Python is less prescriptive about values in an image. The `io.imshow` method will automatically scale the image for display. So, for example:

```
In :  c = io.imread('caribou.png')
In :  cd1 = c.astype(float)
In :  cd2 = util.img_as_float(c)
```

Python

all produce images that are equally displayable with `io.imshow`. Note that the data types and values are all different:

```
In :  c.dtype, cd1.dtype, cd2.dtype
Out:  (dtype('uint8'), dtype('float64'), dtype('float64'))
```

Python

To see the numeric values, just check the minima and maxima of each array:

```
In : c.min(), c.max()
Out: (21, 254)
In : cd1.min(), cd1.max()
Out: (21.0, 254.0)
In : cd2.min(), cd2.max()
Out: (0.0823529411764705 87, 0.99607843137254903)
```

Python

This means that multiplying and dividing an image by a fixed value will not affect the display, as the result will be scaled. In order to obtain the effects of darkening and lightening, the default scaling, which uses the maximum and minimum values of the image matrix, can be adjusted by the use of two parameters: `vmin`, which gives the minimum gray level to be viewed, and `vmax`, which gives the maximum gray level.

For example, to obtain a dark caribou image:

```
In :  io.imshow(cd1/2,vmin=0,vmax=255)
```

Python

and to obtain a light, washed out image:

```
In :  io.imshow(cd1*2,vmin=0,vmax=255)
```

Python

Without the `vmin` and `vmax` values given, the image would be automatically scaled to look like the original.

Note that MATLAB/Octave and Python differ in their handling of arithmetic on uint8 numbers. For example:

```
>> a = uint8([0 80;160 240]);
>> (a-10)*2
ans =

    0   80
  160  255
```

MATLAB/Octave

MATLAB and Octave *clip* the output, so that values greater than 255 are set equal to 255, and values less than 0 are set equal to zero. But in Python the result is different:

```
In :  a = array([[0,80],[160,240]]).astype(uint8)
In :  (a-10)*2
Out:
array([[236, 140],
       [ 44, 204]], dtype=uint8)
```

Python

Python works *modulo* 256: numbers outside the range 0–255 are divided by 256 and the remainder given.

Binary images

Recall that a binary image will have only two values: 0 and 1. MATLAB and Octave do not have a `binary` data type as such, but they do have a `logical` flag, where `uint8` values as 0 and 1 can be interpreted as logical data. The logical flag will be set by the use of relational operations such as `==, <`, or `>` or any other operations that provide a yes/no answer. For example, suppose we take the caribou matrix and create a new matrix with

```
>> cl = c>120;
```

MATLAB/Octave

(we will see more of this type of operation in Chapter 4.) If we now check all of our variables with `whos`, the output will include the line:

```
>> whos c cl
  Name        Size              Bytes  Class      Attributes

  c         256x256             65536  uint8
  cl        256x256             65536  logical
```

MATLAB

The Octave output is slightly different:

```
   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        c         256x256                    65536  uint8
        cl        256x256                    65536  logical
```

Octave

This means that the command

```
>> imshow(cl)
```

MATLAB/Octave

will display the matrix as a binary image; the result is shown in Figure 3.3.

Suppose we remove the logical flag from `cl`; this can be done by a simple command:

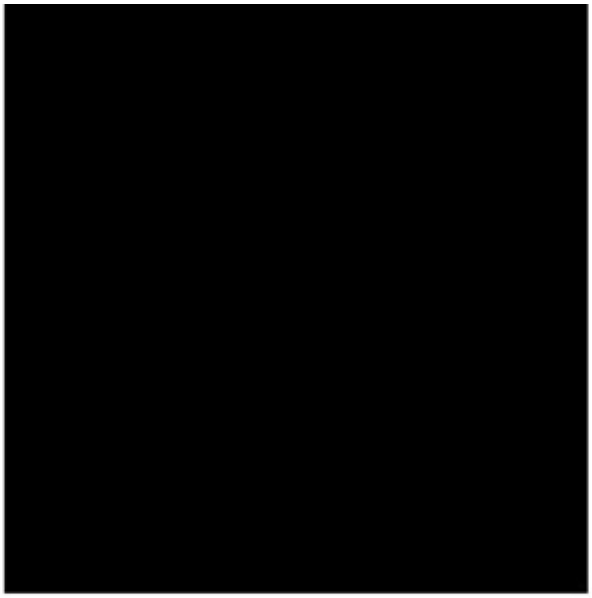
```
>> clu = uint8(cl);
```

MATLAB/Octave

Now the output of `whos` will include the line:

Figure 3.3: Making the image binary

(a) The caribou image turned binary



(b) After conversion to type `uint8`

```
   clu      256x256        65536  uint8
```

MATLAB/Octave

If we now try to display this matrix with `imshow`, we obtain the result shown in Figure 3.3(b). A very disappointing image! But this is to be expected; in a matrix of type `uint8`, white is 255, 0 is black, and 1 is a very dark gray which is indistinguishable from black.

To get back to a viewable image, we can either turn the logical flag back on, and the view the result:

```
>> imshow(logical(clu))
```

MATLAB/Octave

or simply convert to type `double`:

```
>> imshow(double(clu))
```

MATLAB/Octave

Both these commands will produce the image seen in Figure 3.3.

Python Python does have a boolean type:

```
In :  cl = c>120
In :  cl.dtype
Out:  dtype('bool')
```

Python

and the elements of the matrix `cl` are either `True` or `False`. This is still quite displayable with `io.imshow`. This matrix can be turned into a numeric matrix with any of:

```
In :  sk.util.img_as_ubyte(cl)
In :  uint8(cl*255)
In :  float16(cl*1)
```

Python

A numeric matrix `x` can be made boolean by

```
In :  x.astype(bool)
In :  util.img_as_bool(x)
```

Python

3.3 Bit Planes

Grayscale images can be transformed into a sequence of binary images by breaking them up into their *bit-planes*. If we consider the gray value of each pixel of an 8-bit image as an 8-bit binary word, then the 0th bit plane consists of the last bit of each gray value. Since this bit has the least effect in terms of the magnitude of the value, it is called the *least significant bit*, and the plane consisting of those bits the *least significant bit plane*. Similarly the 7th bit plane consists of the first bit in each value. This bit has the greatest effect in terms of the magnitude of the value, so it is called the *most significant bit*, and the plane consisting of those bits the *most significant bit plane*.

If we have a grayscale image of type `uint8`:

```
>> c = imread('cameraman.png');
```

MATLAB/Octave

then its bit planes can be accessed by the `bitget` function. In general, the function call

```
>> bitget(x,n)
```

MATLAB/Octave

will isolate the *n*-th rightmost bit from every element in `x`

All bitplanes can be simultaneously displayed using `subplot`, which places graphic objects such as images or plots on a rectangular grid in a single figure, but in order to minimize the distance between the images we can use the `position` parameter, starting off by creating the *x* and *y* positions for each subplot:

```
>> posx = [0 1 2 0 1 2 0 1]/3
>> posy = [2 2 2 1 1 1 0 0]/3
```

MATLAB/Octave

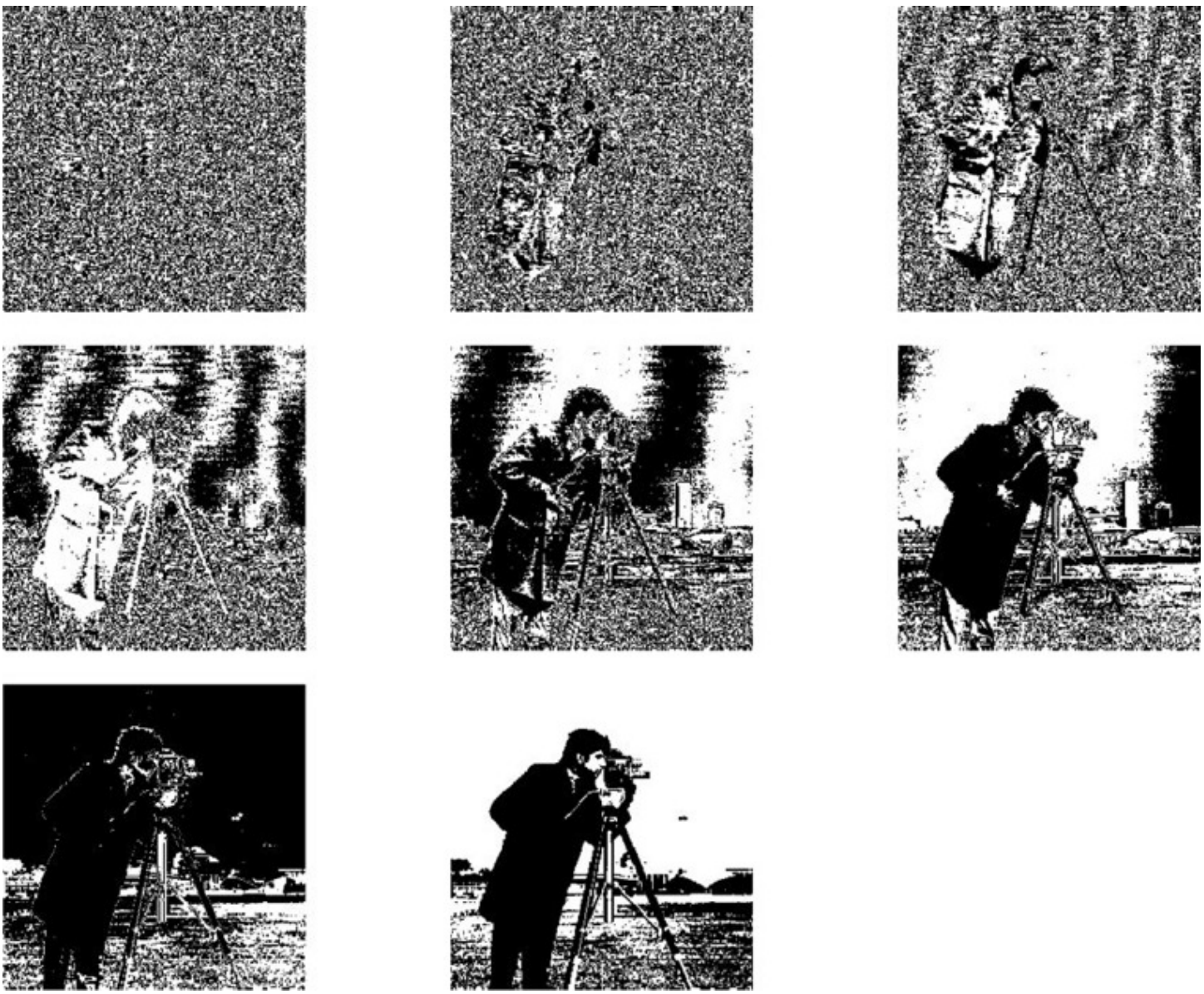
Then they can be plotted as:

```
>> for i = 1:8
>    subplot("position",[posx(i),posy(i),0.3,0.3]),
>    imshow(logical(bitget(c,i))),axis image
>  end
```

MATLAB/Octave

The result is shown in Figure 3.4. Note that the least significant bit plane, at top left, is to all intents and purposes a random array, and that as the index value of the bit plane increases, more of the image appears. The most significant bit plane, which is at the bottom center, is actually a `threshold` of the image at level 128:

Figure 3.4: The bit planes of an 8-bit grayscale image



```
>> ct = c>127;
>> b8 = logical(bitget(c,8));
>> isequal(ct,b8)

ans =

     1
```

MATLAB/Octave

We shall discuss thresholding in Chapter 9.

We can recover and display the original image by multiplying each bitplane by an appropriate power of two and adding them; each scaled bitplane is added to an empty array which is initiated with the useful `zeros` function:

```
>> y = zeros(size(c));
>> for i = 1:8
>    y = y + 2^(i-1)*double(bitget(c,i));
>>   end
```

MATLAB/Octave

This new image `y` is the cameraman image:

```
>> isequal(c,uint8(y))
ans = 1
```

MATLAB/Octave

Python does not have a `bitget` function, however the equivalent result can be obtained by `bitshifting`: taking the binary string corresponding to a grayscale, and shifting it successively to the right, so the rightmost bits successively "drop off." Python uses the "»" notation for bitshifting:

```
In :  n = uint8(175)
In :  for i in range(8):
...:      print bin(n>>i)
...:
0b10101111
0b1010111
0b101011
0b10101
0b1010
0b101
0b10
0b1
```

Python

The rightmost bits can be obtained by using the modulo operator:

```
In :  for i in range(8):
...:     print (n>>i)%2
...:
1
1
1
1
0
1
0
1
```

Python

So the following generates and displays all bit planes using the `pyplot` module of the `matplotlib` library:

```
In :  import matplotlib.pyplot as plt
In :  c = io.imread('cameraman.png')
In :  bps = [(c>>i)%2 for i in range(8)]
In :  for i in range(8):
...:      plt.subplot(3,3,i+1)
...:      io.imshow(bps[i])
...:      plt.axis('off')
```

Python

and the bit planes can be reassembled with

```
In :  z = sum(bps[i]*2**i for i in range(8))
In :  (c==z).all()
Out:  True
```

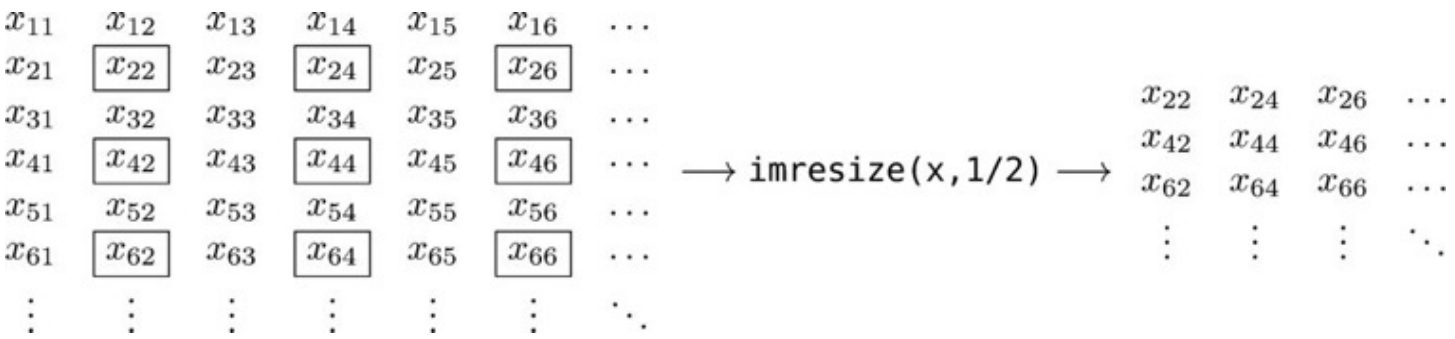
Python

3.4 Spatial Resolution

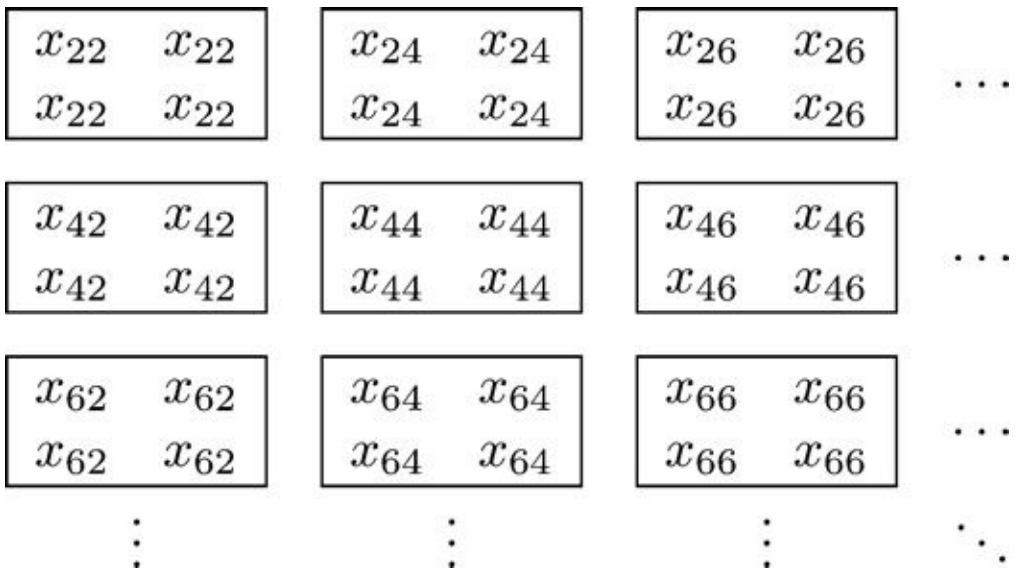
Spatial resolution is the density of pixels over the image: the greater the spatial resolution, the more pixels are used to display the image. We can experiment with spatial resolution with MATLAB's `imresize` function. Suppose we have a 256 × 256 8-bit grayscale image saved to the matrix `x`. Then the command

```
imresize(x,1/2);
```

will halve the size of the image. It does this by taking out every other row and every other column, thus leaving only those matrix elements whose row and column indices are even:



If we apply `imresize` to the result with the parameter `2` and the method "`nearest`," rather than `1/2`, all the pixels are repeated to produce an image with the same size as the original, but with half the resolution in each direction:



The effective resolution of this new image is only 128 × 128. We can do all this in one line:

```
x2=imresize(imresize(x,1/2),2,'nearest');
```

By changing the parameters of `imresize`, we can change the effective resolution of the image to smaller amounts:

Command	Effective resolution
`imresize(imresize(x,1/4),4,'nearest');`	64 × 64
`imresize(imresize(x,1/8),8,'nearest');`	32 × 32
`imresize(imresize(x,1/16),16,'nearest');`	16 × 16
`imresize(imresize(x,1/32),32,'nearest');`	8 × 8

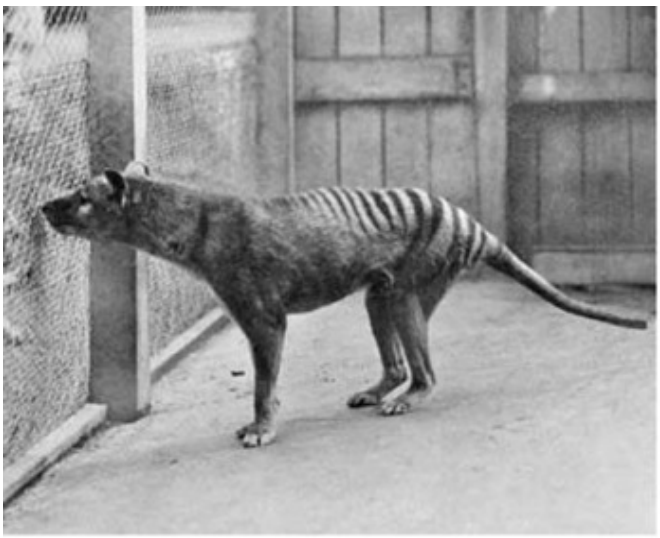
To see the effects of these commands, suppose we apply them to the image `thylacine.png`:1

```
>>  x = imread('thylacine.png;
>> for i = 1:4
>    subplot(2,2,i),imshow(imresize(imresize(x,1/(2^(i+1))),2^(i+1),'
     nearest'))
>    end
```

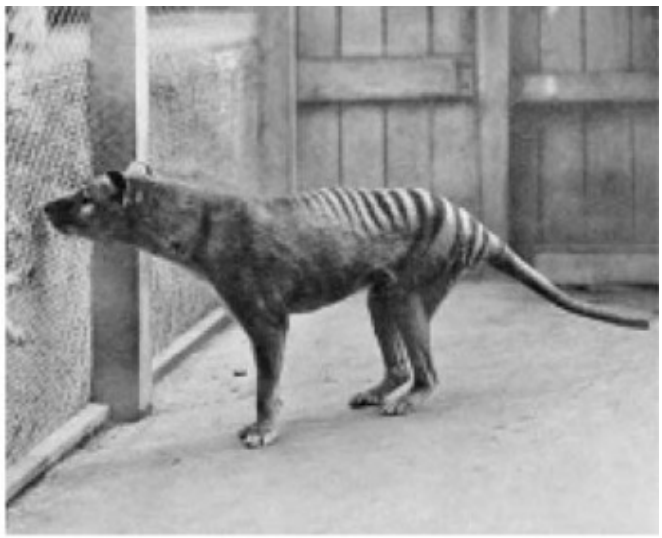
MATLAB/Octave

Since this image has size 320 × 400, the effective resolutions will be these dimensions divided by powers of two.

Figure 3.5: Reducing resolution of an image

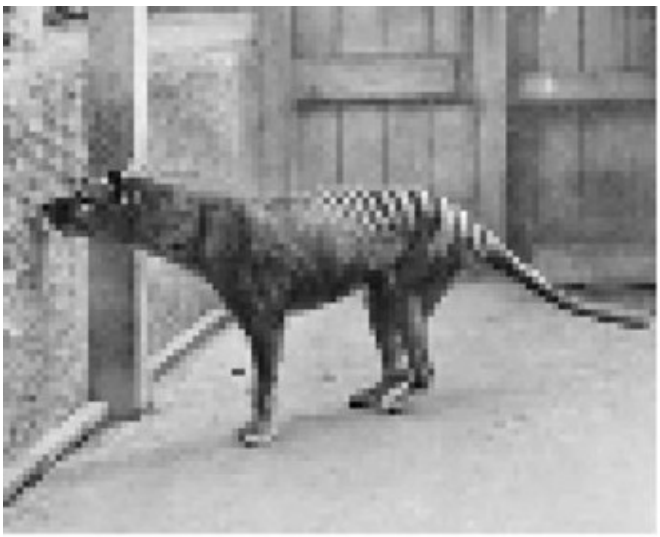


(a) The original image

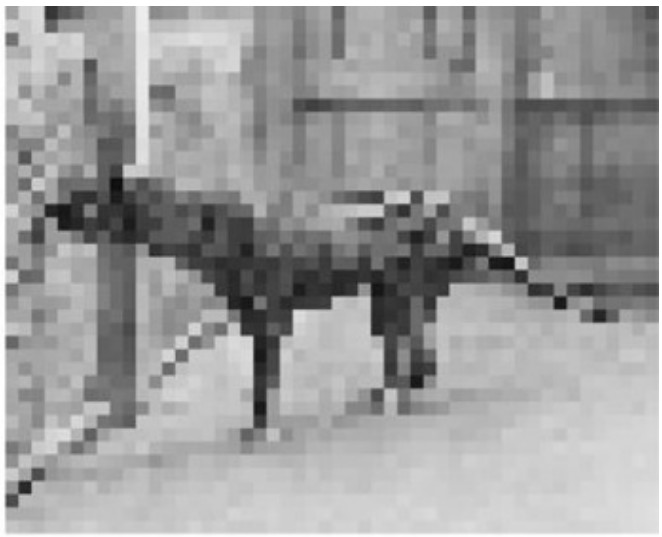


(b) at 160 × 200 resolution

Figure 3.6: Further reducing the resolution of an image



(a) At 80 × 100 resolution



(b) At 40 × 50 resolution

The effects of increasing blockiness or *pixelization* become quite pronounced as the resolution decreases; even at 160 × 200 resolution fine detail, such as the wire mesh of the enclosure, are less clear, and at 80 ×

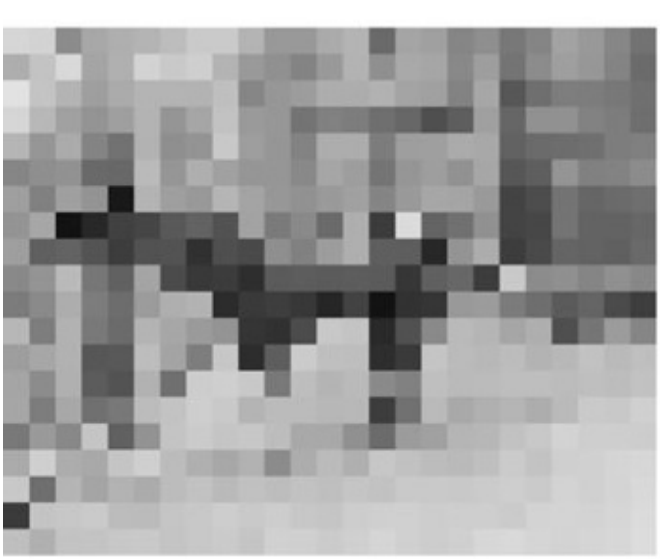
100 all edges are now quite blocky. At 40 × 50, the image is barely recognizable, and at 20 × 25 and 10 × 12 the image becomes unrecognizable.

Python again is similar to MATLAB and Octave. To change the spatial resolution, use the `rescale` method from the `skimage.transform` module:

```
In :  import skimage.transform as tr
In :  x4 = tr.rescale(tr.rescale(x,0.25),4,order=0)
```

Python

Figure 3.7: Reducing the resolution of an image even more



(a) At 20 × 25 resolution

(b) At 10 × 12 resolution

The "`order`" parameter indicates the method of interpolation used; 0 corresponds to nearest neighbor interpolation.

1

This is a famous image showing the last known *thylacine*, or Tasmanian Tiger, a now extinct carnivorous marsupial, in the Hobart Zoo in 1933.

3.5 Quantization and Dithering

Quantization refers to the number of grayscales used to represent the image. As we have seen, most images will have 256 grayscales, which is more than enough for the needs of human vision. However, there are circumstances in which it may be more practical to represent the image with fewer grayscales. One simple way to do this is by *uniform quantization*: to represent an image with only n grayscales, we divide the range of grayscales into n equal (or nearly equal) ranges, and map the ranges to the values 0 to $n - 1$. For example, if $n = 4$, we map grayscales to output values as follows:

Original values	Output value

0–63	0
64–127	1
128–191	2
192–255	3

and the values 0, 1, 2, 3 may need to be scaled for display. This mapping can be shown graphically, as in Figure 3.8.

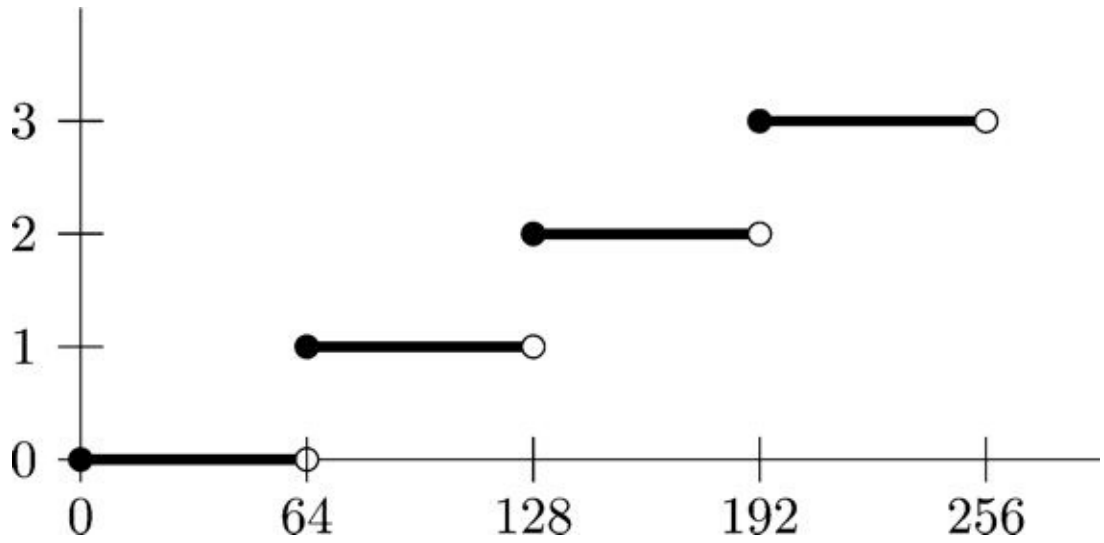
To perform such a mapping in MATLAB, we can perform the following operations, supposing `x` to be a matrix of type `uint8`:

```
>> q = (f/64)*64
```

MATLAB/Octave

Since `f` is of type `uint8`, dividing by 64 will also produce values of type `uint8`; so any fractional parts will be removed. For example:

Figure 3.8: A mapping for uniform quantization



```
>> y = uint8(reshape(0:16:16*15,4,4))
y =

    0   64  128  192
   16   80  144  208
   32   96  160  224
   48  112  176  240
>> z = y/64
z =

    0    1    2    3
    0    1    2    3
    1    2    3    4
    1    2    3    4
>> z*64
ans =

    0   64  128  192
    0   64  128  192
   64  128  192  255
   64  128  192  255
```

MATLAB/Octave

The original array has now been quantized to only five different values. Given a `uint8` array `x`, and a value `v`, then in general the command

```
>> (x/v)*v
```

MATLAB/Octave

will produce a maximum of $256/v+1$ different possible grayscales. So with $v = 64$ as above, we would expect $256/64 + 1 = 5$ possible output grayscales.

Python acts similarly:

```
In :  y = uint8(16*np.reshape(range(16),(4,4)))
In :  y
Out:
array([[  0,  16,  32,  48],
       [ 64,  80,  96, 112],
       [128, 144, 160, 176],
       [192, 208, 224, 240]], dtype=uint8)

In: z = y/64; z
Out:
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]], dtype=uint8)

In : z*64
Out:
array([[  0,   0,   0,   0],
       [ 64,  64,  64,  64],
       [128, 128, 128, 128],
       [192, 192, 192, 192]], dtype=uint8)
```

Python

Note that dividing gives a different result in MATLAB/Octave and Python. In the former, the result is the closest integer (by rounding) to the value of $n/64$; in Python, the result is the floor.

If we wish to precisely specify the number of output grayscales, then we can use the `grayslice` function. Given an image matrix `x` and an integer `n`, the MATLAB command `grayslice(x, n)` produces a matrix whose values have been reduced to the values 0, 1, ..., $n - 1$. So, for example

```
>> x4 = grayslice(x,4);
```

MATLAB

will produce a `uint8` version of our image with values 0, 1, 2 and 3. Note that Octave works slightly differently from MATLAB here; Octave requires that the input image be of type `double`:

```
>> x4 = grayslice(im2double(x),4);
```

Octave

We cannot view this directly, as it will appear completely black: the four values are too close to zero to be distinguishable. We need to treat this matrix as the indices to a color map, and the color map we shall use is `gray(4)`, which produces a color map of four evenly spaced gray values between 0 (black) and 1.0 (white). In general, given an image `x` and a number of grayscales n, the commands

```
>> y = grayslice(x,n);
>> imshow(x,gray(n))
```

MATLAB/Octave

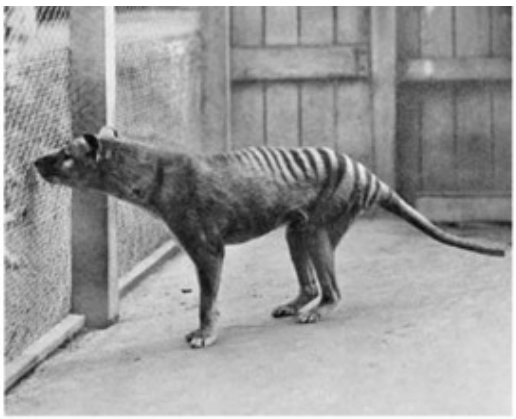
will display the quantized image. Note that if you are using Octave, you will need to ensure that `x` is of type `double`. Quantized images can be displayed in one go using `subplot`:

```
>> qs = [256,64,32,16,4,2]
>> px = [1/6,1/2,1/6,1/2,1/6,1/2]
>> py = [2/3,2/3,1/3,1/3,0,0]
>> for i = 1:6
>    subplot("position",[px[i],py[i],1/3,1/3]),imshow(grayslice(x,qs[i]),
    gray(qs[i]))
>    end
```

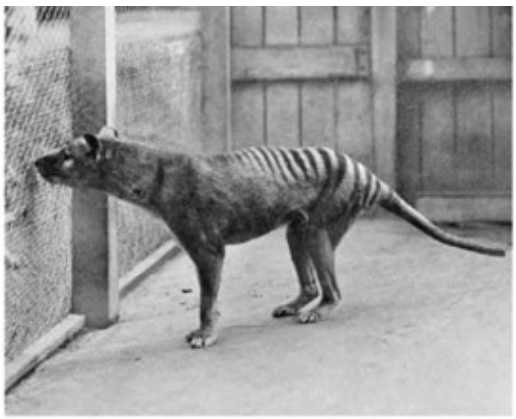
MATLAB/Octave

If we apply these commands to the thylacine image, we obtain the results shown in Figures 3.9 to 3.12.

Figure 3.9: Quantization (1)

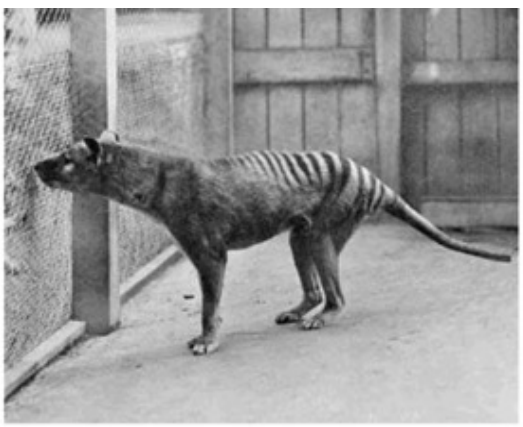


(a) The image quantized to 128 grayscales

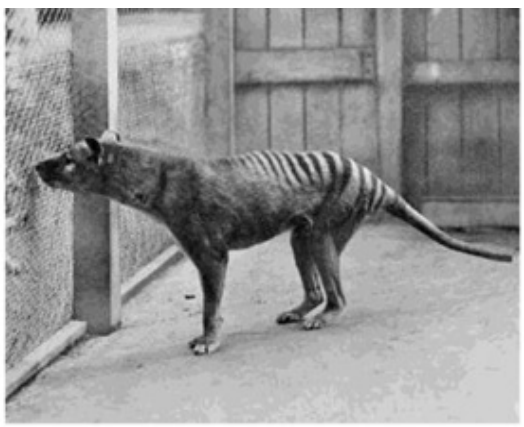


(b) The image quantized to 64 grayscales

Figure 3.10: Quantization (2)



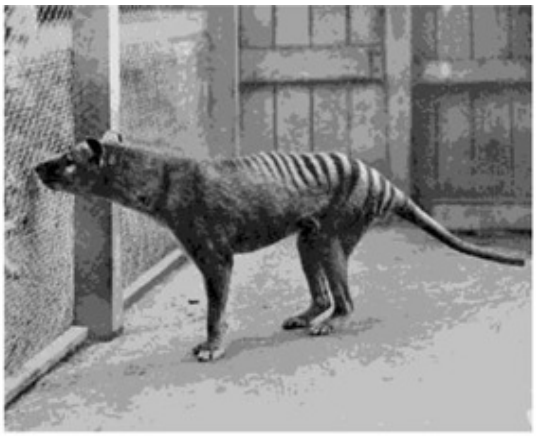
(a) The image quantized to 32 grayscales



(b) The image quantized to 16 grayscales

One immediate consequence of uniform quantization is that of “false contours,” most noticeable with fewer grayscales. For example, in Figure 3.11, we can see on the ground and rear fence that the gray texture is no longer smooth; there are observable discontinuities between different gray values. We may expect that if fewer grayscales are used, and the jumps between consecutive grayscales becomes larger, that such false contours will occur.

Figure 3.11: Quantization (3)

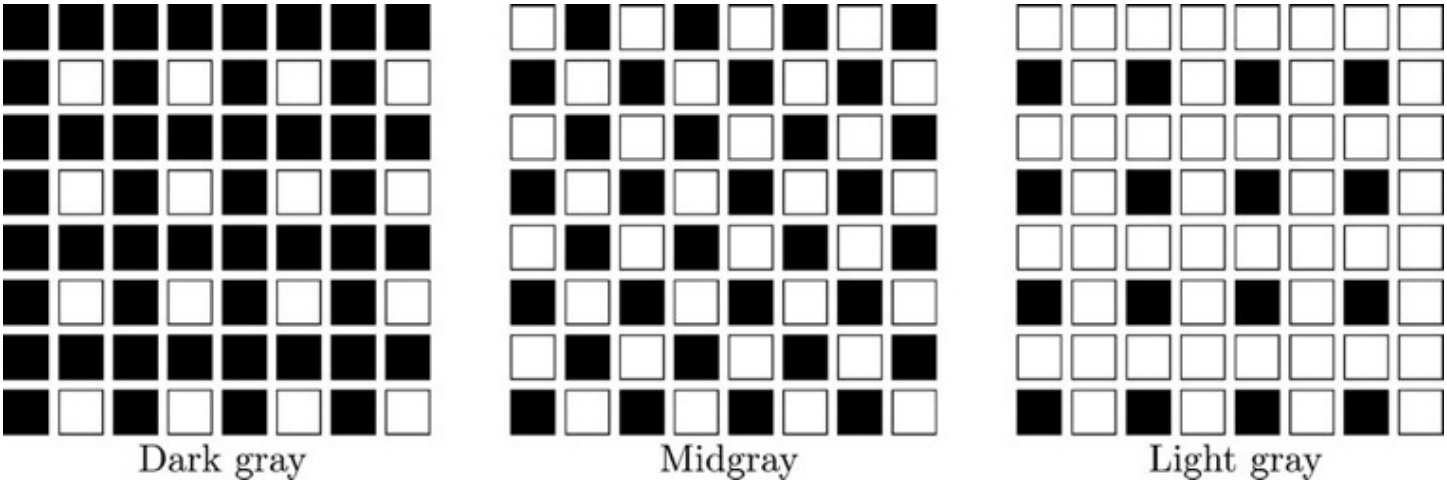


(a) The image quantized to 8 grayscales

(b) The image quantized to 4 grayscales

Figure 3.12: The image quantized to 2 grayscales

Figure 3.13: Patterns for dithering output



Dithering

Dithering, in general terms, refers to the process of reducing the number of colors in an image. For the moment, we shall be concerned only with grayscale dithering. Dithering is necessary sometimes for display, if the image must be displayed on equipment with a limited number of colors, or for printing. Newsprint, in particular, only has two scales of gray: black and white. Representing an image with only two tones is also known as *halftoning*.

One method of dealing with such false contours involves adding random values to the image before quantization. Equivalently, for quantization to 2 grayscales, we may compare the image to a random matrix r. The trick is to devise a suitable matrix so that grayscales are represented evenly in the result. For example, an area containing mid-way gray level (around 127) would have a checkerboard pattern; a darker area will have a pattern containing more black than white, and a light area will have a pattern containing more white than black. Figure 3.13 illustrates this. One standard matrix is

$$D = \begin{pmatrix} 0 & 128 \\ 192 & 64 \end{pmatrix}$$

which is repeated until it is as big as the image matrix, when the two are compared. Suppose $d(i, j)$ is the matrix obtained by replicating D. Thus, an output pixel $p(i, j)$ is defined by

$$p(i,j) = \begin{cases} 1 & \text{if } x(i,j) > d(i,j) \\ 0 & \text{if } x(i,j) \leq d(i,j) \end{cases}$$

This approach to quantization is called *dithering*, and the matrix D is an example of a *dither matrix*. Another dither matrix is given by

$$D_2 = \begin{pmatrix} 0 & 128 & 32 & 160 \\ 192 & 64 & 224 & 96 \\ 48 & 176 & 16 & 144 \\ 240 & 112 & 208 & 80 \end{pmatrix}$$

We can apply the matrices to our thylacine image matrix `x` by using the following commands in MATLAB/Octave:

```
>> D = [0 128;192 64]
>> r = repmat(D,160,200);
>> x2 = x>r; imshow(x2)
>> D2 = [0 128 32 160;192 64 224 96;48 176 16 144;240 112 208 80];
>> r2 = repmat(D2,80,100);
>> x4 = x>r2; imshow(x4)
```

MATLAB/Octave

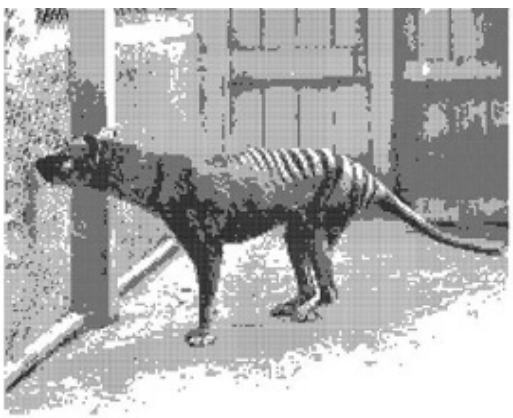
or in Python:

```
In :  D = array([[0,128],[192,64]])
In :  r = np.tile(D,(160,200));
In :  x2 = (x>r).astype(uint8)
In :  io.imshow(x2)
In :  D2 = array([[0, 128, 32, 160],[192, 64, 224, 96],[48, 176, 16, 144],\
...:  [240, 112, 208, 80]])
In :  r2 = np.tile(D2,80,100);
In :  x4 = (x>r2).astype(uint8)
In :  io.imshow(x2)
```

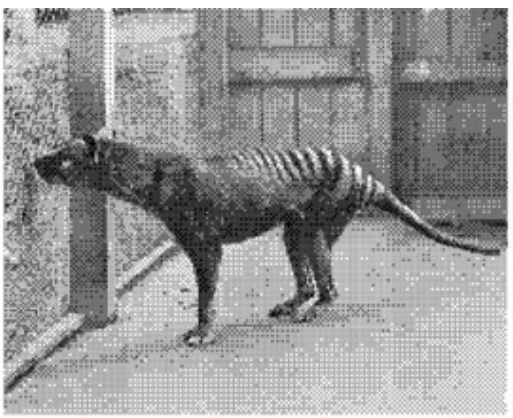
Python

The results are shown in Figure 3.14. The dithered images are an improvement on the uniformly quantized image shown in Figure 3.12. General dither matrices are provided by Hawley [15].

Figure 3.14: Examples of dithering



(a) The thylacine image dithered using D



(b) The thylacine image dithered using D_2

Dithering can be easily extended to more than two output gray values. Suppose, for example, we wish to quantize to four output levels 0, 1, 2, and 3. Since 255/3 = 85, we first quantize by dividing the gray value $x(i,j)$ by 85:

$$q(i,j) = [x(i,j)/85].$$

This will produce only the values 0, 1, and 2, except for when $x(i,j) = 255$. Suppose now that our replicated dither matrix $d(i,j)$ is scaled so that its values are in the range 0 ... 85. The final value $p(i,j)$ is then defined by

$$p(i,j) = q(i,j) + \begin{cases} 1 & \text{if } x(i,j) - 85q(i,j) > d(i,j) \\ 0 & \text{if } x(i,j) - 85q(i,j) \leq d(i,j) \end{cases}$$

This can be easily implemented in a few commands, modifying `D` slightly from above and using the `repmat` function which simply tiles an array as many times as given:

```
>> D = [0 56;84 28]
>> r = repmat(D,128,128);
>> x = double(x);
>> q = floor(x/85);
>> x4 = q+(x-85*q>r);
>> imshow(uint8(85*x4))
```

MATLAB/Octave

The commands in Python are similar:

```
In :  D = array([[0, 56],[84, 28]])
In :  r = np.tile(D,(128,128))
In :  x = x.astype(float64)
In :  q = floor(x/85)
In :  x4 = q+(x-85*q>r)
In :  io.imshow(uint8(x4*85))
```

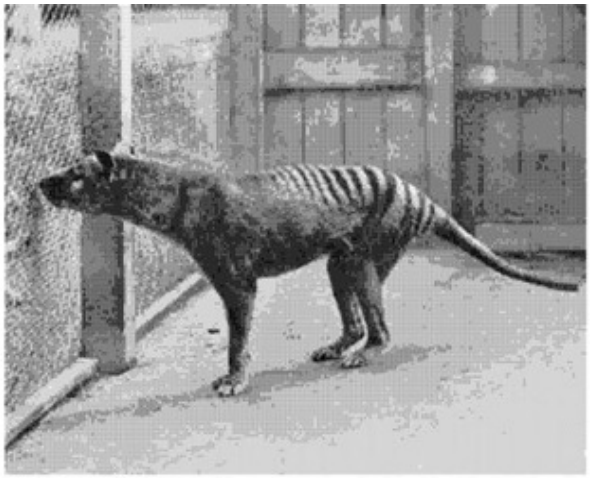
Python

and the result is shown in Figure 3.15(a). We can dither to eight gray levels by using 255/7 = 37 (we round the result up to ensure that our output values stay within range) instead of 85 above; our starting dither matrix will be

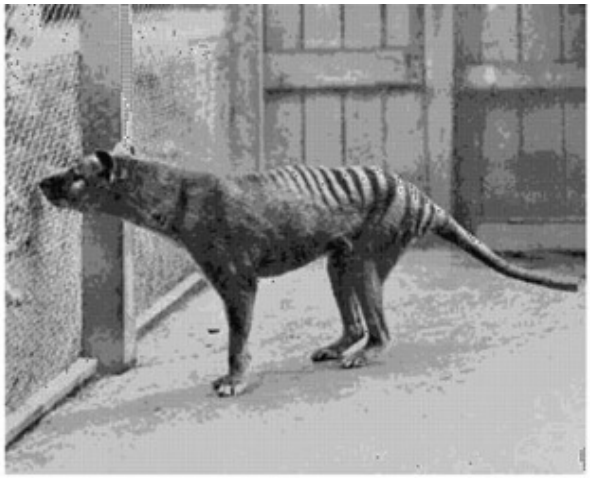
$$D = \begin{pmatrix} 0 & 24 \\ 36 & 12 \end{pmatrix}$$

and the result is shown in Figure 3.15(b). Note how much better these images look than the corresponding uniformly quantized images in Figures 3.11(b) and 3.11(a). The eight-level result, in particular, is almost indistinguishable from the original, in spite of the quantization.

Figure 3.15: Dithering to more than two grayscales



(a) Dithering to 4 output grayscales

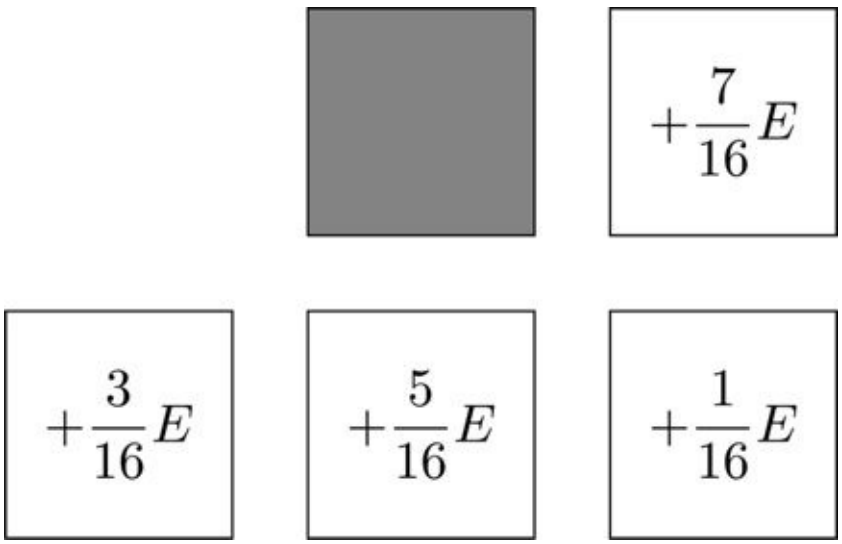


(b) Dithering to 8 output grayscales

Error Diffusion

A different approach to quantization from dithering is that of *error diffusion*. The image is quantized at two levels, but for each pixel we take into account the *error* between its gray value and its quantized value. Since we are quantizing to gray values 0 and 255, pixels close to these values will have little error. However, pixels close to the center of the range: 128, will have a large error. The idea is to spread this error over neighboring pixels. A popular method, developed by Floyd and Steinberg, works by moving through the image pixel by pixel, starting at the top left, and working across each row in turn. For each pixel $p(i, j)$ in the image we perform the following sequence of steps:

1. Perform the quantization.
2. Calculate the quantization error. This is defined as: $E = \begin{cases} p(i,j) & \text{if } p(i,j) < 128 \\ p(i,j) - 255 & \text{if } p(i,j) >= 128 \end{cases}$
3. Spread this error E over pixels to the right and below according to this table:



There are several points to note about this algorithm:

- The error is spread to pixels *before* quantization is performed on them. Thus, the error diffusion will affect the quantization level of those pixels.
- Once a pixel has been quantized, its value will never be affected. This is because the error diffusion only affects pixels to the right and below, and we are working from the left and above.
- To implement this algorithm, we need to embed the image in a larger array of zeros, so that the indices do not go outside the bounds of our array.

The `dither` function, when applied to a grayscale image, actually implements Floyd-Steinberg error diffusion. However, it is instructive to write a simple MATLAB function to implement it ourselves; one possibility is given at the end of the chapter.

The result of applying this function to the thylacine image is shown in Figure 3.16. Note that the result is very pleasing; so much so that it is hard to believe that every pixel in the image is either back or white, and so that we have a binary image.

Figure 3.16: The thylacine image after Floyd-Steinberg error diffusion

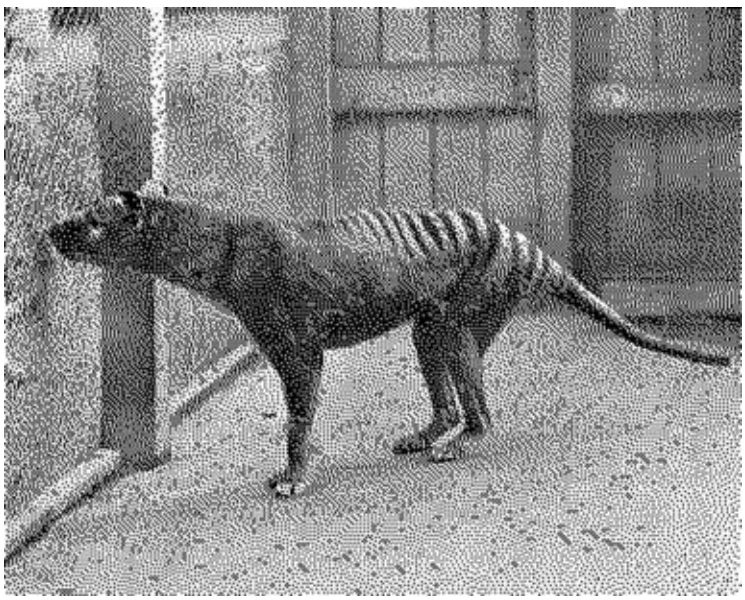
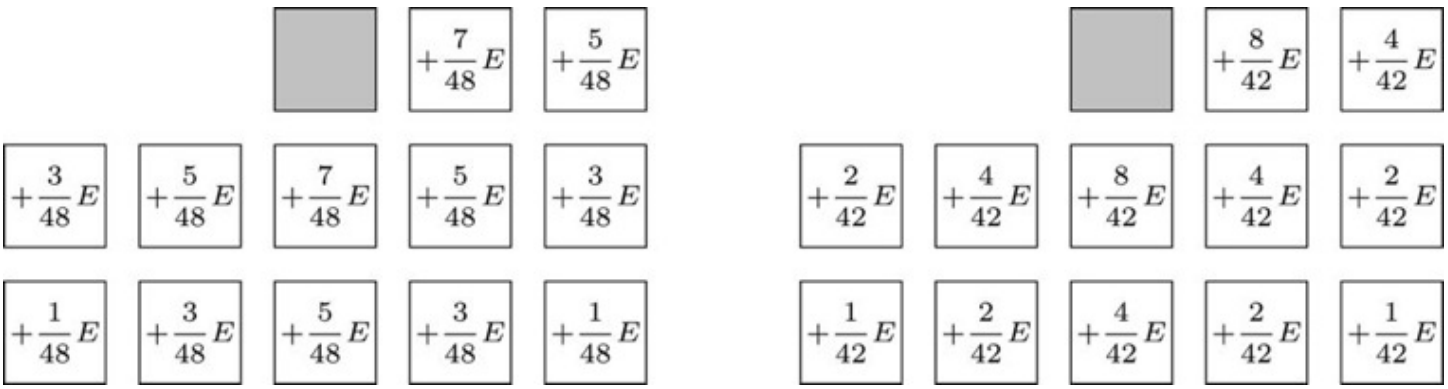


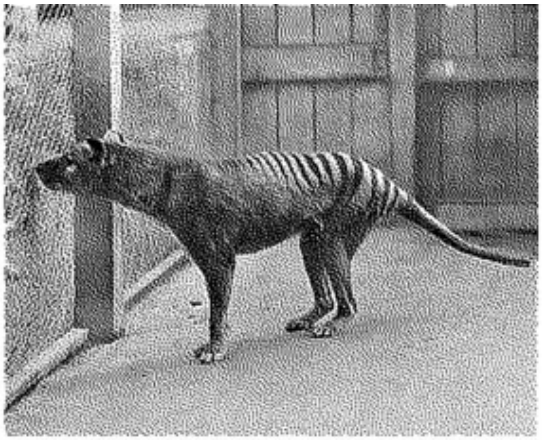
Figure 3.17: Different error diffusion schemes



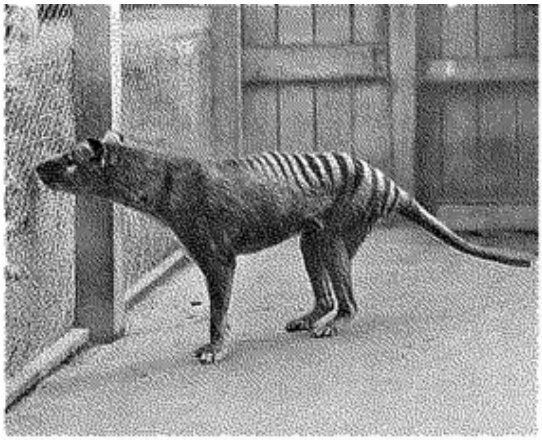
Other error diffusion schemes are possible; two such schemes are Jarvis-Judice-Ninke and Stucki, which have error diffusion schemes shown in Figure 3.17.

They can be applied by modifying the Floyd-Steinberg function given at the end of the chapter. The results of applying these two error diffusion methods are shown in Figure 3.18.

Figure 3.18: Using other error-diffusion schemes



(a) Result of Jarvis-Judice-Ninke error diffusion



(b) Result of Stucki error diffusion

A good introduction to error diffusion (and dithering) is given by Schumacher [44].

3.6 Programs

Here are programs for error diffusion, first Floyd-Steinberg (with arbitrary levels), in MATLAB/Octave:

```
function y = fs(x,k)
height = size(x,1);
width = size(x,2);
ed = [0 0 0 7 0;0 3 5 1 0;0 0 0 0 0]/16;
y = uint8(zeros(height,width));
z = zeros(height+4,width+4);
z(3:height+2,3:width+2) = x;
for i = 3:height+2,
  for j = 3:width+2,
    quant = floor(255/(k-1))*floor(z(i,j)*k/256);
    y(i-2,j-2) = quant;
    e = z(i,j)-quant;
    z(i:i+2,j-2:j+2) = z(i:i+2,j-2:j+2)+e*ed;
  endfor
endfor
endfunction
```

MATLAB/Octave

and now Jarvis-Judice-Ninke with two levels:

```
function out = jjn(im)
height = size(im,1);
width = size(im,2);
out = zeros(size(im));
ed = [0 0 0 7 5;3 5 7 5 3;1 3 5 3 1]/48;
z = zeros(size(im)+4);
z(3:height+2,3:width+2) = double(im);
for i = 3:height+2,
  for j = 3:width+2,
    quant = 255*(z(i,j)>=128);
    out(i-2,j-2) = quant;
    e = z(i,j)-quant;
    z(i:i+2,j-2:j+2) = z(i:i+2,j-2:j+2)+e*ed;
  endfor
endfor
out = im2uint8(out);
endfunction
```

MATLAB/Octave

Here is Floyd-Steinberg in Python:

```
def fs(im,k):  #FS error diffusion at k levels
    rs,cs = im.shape
    ed = array([[0,0,7],[3,5,1]])/16.0
    z = zeros((rs+2,cs+2))
    z[1:rs+1,1:cs+1] = float64(im)
    for i in range(1,rs+1):
        for j in range(1,cs+1):
            old = z[i,j]
            new = (old//(255//k))*(255//(k-1))
            z[i,j] = new
            E = old - new
            z[i:i+2,j-1:j+2] = z[i:i+2,j-1:j+2]+E*ed;
    return uint8(z[1:rs+1,1:cs+1])
```

Python

and Jarvis-Judice-Ninke:

```python
def jjn(im): #JJN error diffusion at two levels
    rs,cs = im.shape
    ed = array([[0,0,0,7,5],[3,5,7,5,3],[1,3,5,3,1]])/48.0
    z = zeros((rs+4,cs+4))
    z[2:rs+2,2:cs+2] = float64(im)
    for i in range(2,rs+2):
        for j in range(2,cs+2):
            old = z[i,j]
            new = (old//128)*255
            z[i,j] = new
            E = old - new
            z[i:i+3,j-2:j+3] = z[i:i+3,j-2:j+3]+E*ed;
    return uint8(z[2:rs+2,2:cs+2]>0)
```

Python

Exercises

1. Open the grayscale image `cameraman.png` and view it. What data type is it?
2. Enter the following commands (if you are using MATLAB or Octave):

```
>> [em,map] = imread('emu.png');
>> e = ind2gray(em,map);
```

MATLAB/Octave

and if you are using Python:

```
In :  import skimage.color as co
In :  em = io.imread('emu.png')
In :  e = co.rgb2gray(em)
```

Python

These will produce a grayscale image of type `double`. View this image.

3. Enter the command

```
>> e2 = im2uint8(e);
```

MATLAB/Octave

or

```
In :  import skimage.util as ut
In :  e2 = ut.img_as_ubyte(e)
```

Python

and view the output. What does the function `im2uint8/img_as_ubyte` do? What affect does it have on

(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.