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

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

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

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

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

9 Image Segmentation

9.1 Introduction

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

9.2 Thresholding

Single Thresholding

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

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

Thresholding is a vital part of image *segmentation*, where we wish to isolate objects from the background. It is also an important component of robot vision.

Thresholding can be done very simply in any of our systems. Suppose we have an 8-bit image, stored as the variable `X`. Then the command

```
X>T or X<T
```

will perform the thresholding. For example, consider an image of some birds flying across a sky; it can be thresholded to show the birds alone in MATLAB or Octave:

```
>> f = imread('flying.png');
>> imshow(f); figure,imshow(f<50)
```

MATLAB/Octave

and in Python:

```
In :  f = io.imread('flying.png')
In :  io.imshow(f)
In :  fig = plt.figure(); fig.show(io.imshow(f<50))
```

Python

These commands will produce the images shown in Figure 9.1. The resulting image can then be further processed to find the number, or average size of the birds.

Figure 9.1: Thresholded image of flying birds

To see how this works, recall that in each system, an operation on a single number, when applied to a matrix, is interpreted as being applied simultaneously to all elements of the matrix; this is vectorization, which we have seen earlier. In MATLAB or Octave the command `X>T` will thus return 1 (for true) for all those pixels for which the gray values are greater than `T`, and 0 (for false) for all those pixels for which the gray values are less than or equal to `T`. The result is a matrix of 0's and 1's, which can be viewed as a binary image. In Python, the result will be a Boolean array whose elements are `True` or `False`. Such an array can however be viewed without any further processing as a binary image. A Boolean array can be turned into a floating point array by re-casting the array to the required data type:

```
In :  (f<50).dtype
Out:  dtype('bool')

In :  f1 = (f<50).astype('float64')
In :  f1.dtype
Out:  dtype('float64')
```

Python

The flying birds image shown above has dark objects on a light background; an image with light objects over a dark background may be treated the same:

```
>> p = imread('paperclips.png')
>> imshow(p),figure,imshow(p>140)
```

MATLAB/Octave

will produce the images shown in Figure 9.2.

As well as the above method, MATLAB and Octave have the `im2bw` function, which thresholds an image *of any data type*, using the general syntax

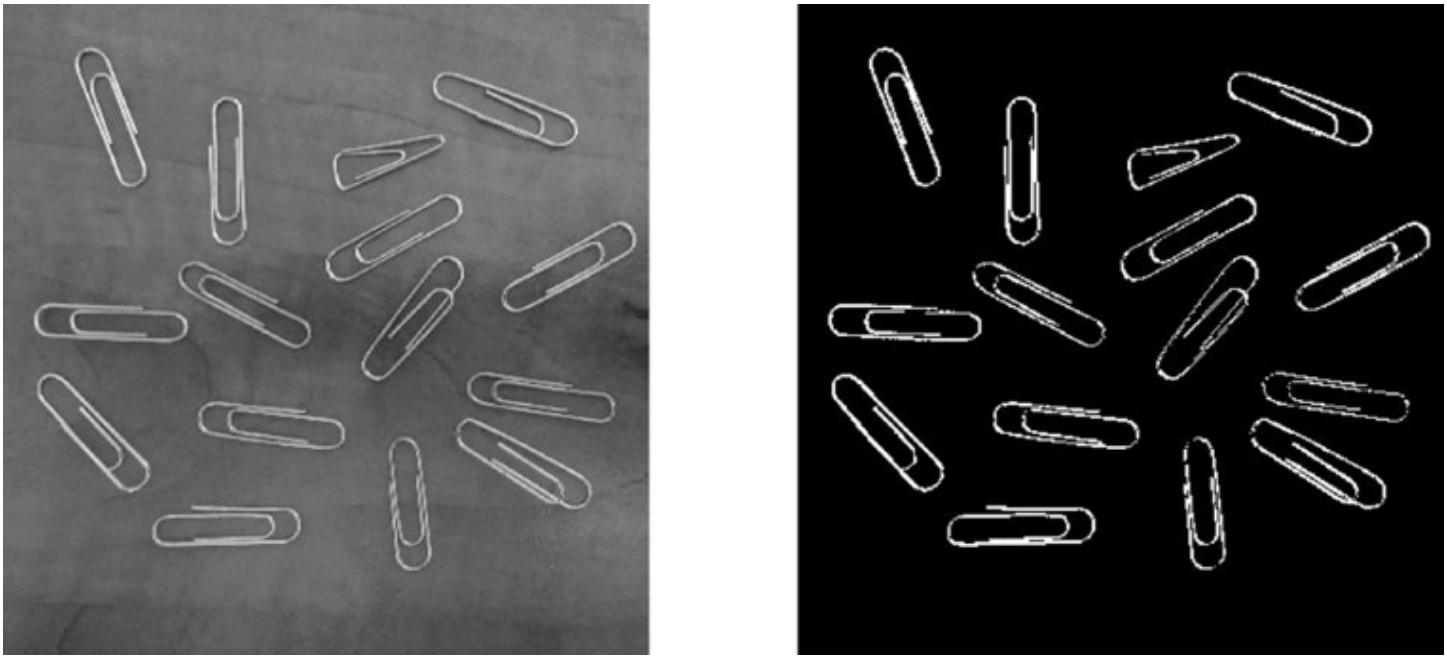
```
im2bw(image,level)
```

where `level` is a value between 0 and 1 (inclusive), indicating the fraction of gray values to be turned white. This command will work on grayscale, colored, and indexed images of data type `uint8`, uint16 or `double`. For example, the thresholded flying and paperclip images above could be obtained using

```
>> im2bw(f,0.3);
>> im2bw(p,0.55);
```

MATLAB/Octave

Figure 9.2: Thresholded image of paperclips



The `im2bw` function automatically scales the value `level` to a gray value appropriate to the image type, and then performs a thresholding by our first method.

As well as isolating objects from the background, thresholding provides a very simple way of showing hidden aspects of an image. For example, the image of some handmade paper `handmade.png` appears mostly white, as nearly all the gray values are very high. However, thresholding at a high level produces an image of far greater interest. We can use the commands

```
>> h = imread('handmade.png');
>> imshow(h),figure,imshow(h>242)
```

MATLAB/Octave

to provide the images shown in Figure 9.3.

Figure 9.3: The paper image and result after thresholding

Double Thresholding

Here we choose two values T_1 and T_2 and apply a thresholding operation as:

$$\text{A pixel becomes } \begin{cases} \text{white if its gray level is between } T_1 \text{ and } T_2, \\ \text{black if its gray level is otherwise.} \end{cases}$$

We can implement this by a simple variation on the above method:

```
X>T1 & X<T2
```

Since the ampersand acts as a logical "and," the result will only produce a one where both inequalities are satisfied. Consider the following sequence of commands:

```
>> x = imread('xray.png');
>> imshow(x),figure,imshow(x>50 & x<80)
```

MATLAB/Octave

The output is shown in Figure 9.4. Note how double thresholding isolates the boundaries of the lungs, which single thresholding would be unable to do. Similar results can be obtained using `im2bw`:

```
>> imshow(im2bw(x,0.2)&~im2bw(x,0.3))
```

MATLAB/Octave

In Python, the command is almost the same:

```
In :  io.imshow((x>40) & (x<80))
```

Python

Figure 9.4: The image `xray.png` and the result after double thresholding

9.3 Applications of Thresholding

We have seen that thresholding can be useful in the following situations:

1. When we want to remove unnecessary detail from an image, to concentrate on essentials. Examples of this were given in the flying birds and paperclip images: by removing all gray level information, the birds and paperclips were reduced to binary objects. But this information may be all we need to investigate sizes, shapes, or numbers of objects.

- 2. To bring out hidden detail. This was illustrated with paper and x-ray images. In both, the detail was obscured because of the similarity of the gray levels involved. But thresholding can be vital for other purposes:
- 3. When we want to remove a varying background from an image. An example of this was the paper clips image shown previously; the paper clips are all light, but the background in fact varies considerably. Thresholding at an appropriate value completely removes the background to show just the objects.

9.4 Choosing an Appropriate Threshold Value

We have seen that one of the important uses of thresholding is to isolate objects from their background. We can then measure the sizes of the objects, or count them. Clearly the success of these operations depends very much on choosing an appropriate threshold level. If we choose a value too low, we may decrease the size of some of the objects, or reduce their number. Conversely, if we choose a value too high, we may begin to include extraneous background material.

Consider, for example, the image `pinenuts.png`, and suppose we try to threshold using the `im2bw` function and various threshold values t for $0 < t < 1$.

```
>> n = imread('pinenuts.png');
>> imshow(n);
>> n1 = im2bw(n,0.35);
>> n2 =im2bw(n,0.55);
>> figure,imshow(n1),figure,imshow(n2)
```

MATLAB/Octave

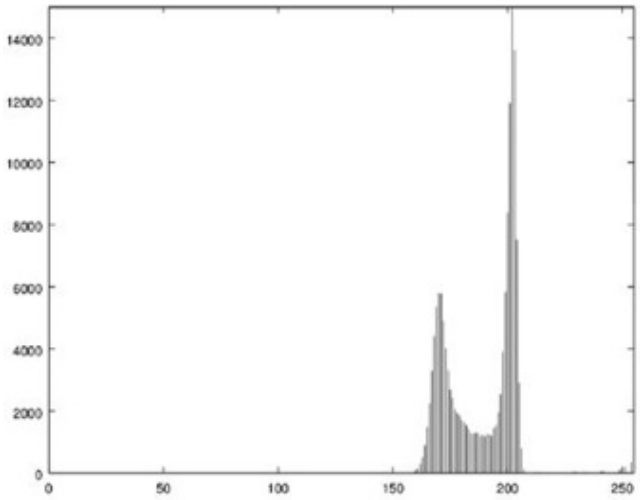
All the images are shown in Figure 9.5. One approach is to investigate the histogram of the image, and see if there is a clear spot to break it up. Sometimes this can work well, but not always.

Figure 9.5: Attempts at thresholding

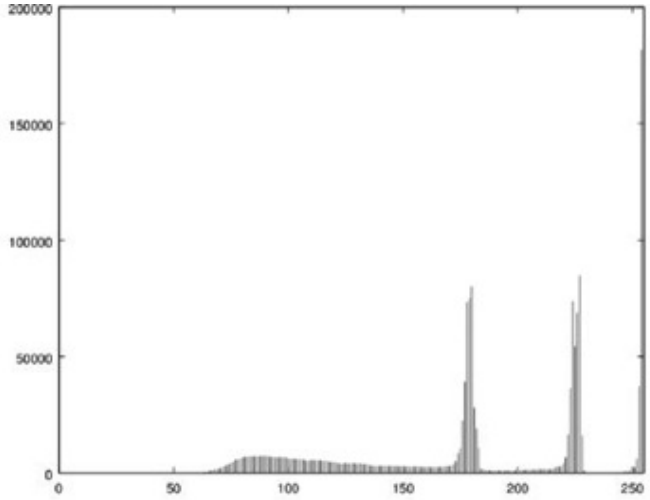
`n`: Original image `n1`: Threshold too low `n2`: Threshold too high

Figure 9.6 shows various histograms. In each case, the image consists of objects on a background. But only for some histograms is it easy to see where we can split it. In both the blood and daisies images, we could split it up about half way, or at the "valley" between the peaks, but for the paramecium and pinenut images, it is not so clear, as there would appear to be three peaks—in each case, one at the extreme right.

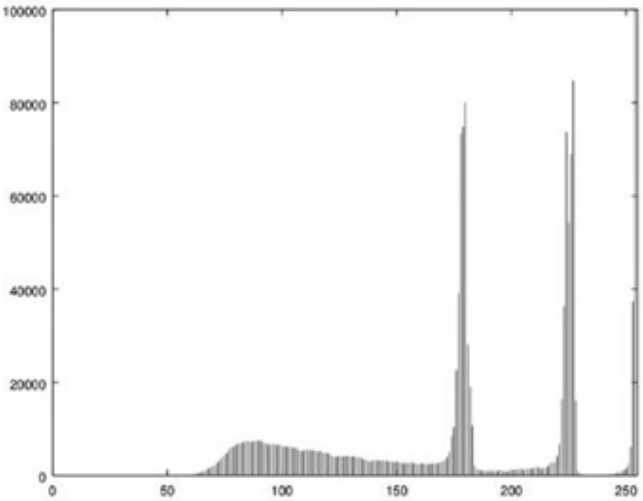
Figure 9.6: Histograms



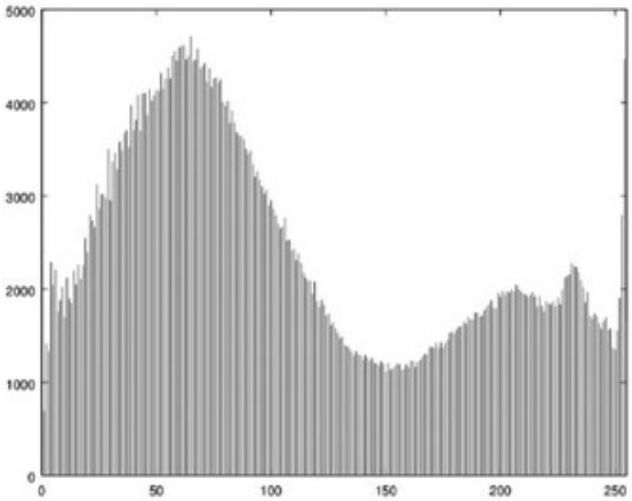
Blood smear image: `blood.png`



Paramecium image: `paramecium1.png`



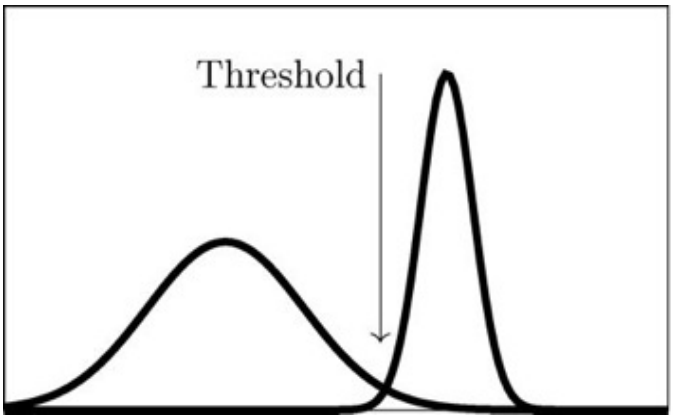
Pine nuts image: `pinenuts.png`

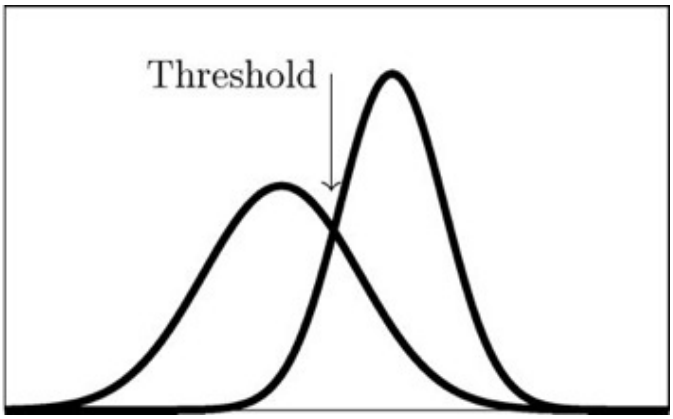


Daisies image: `daisies.png`

The trouble is that in general the individual histograms of the objects and background will overlap, and without prior knowledge of the individual histograms it may be difficult to find a splitting point. Figure 9.7 illustrates this, assuming in each case that the histograms for the objects and backgrounds are those of a normal distribution. Then we choose the threshold values to be the place where the two histograms cross over.

Figure 9.7: Splitting up a histogram for thresholding





In practice though, the histograms won't be as clearly defined as those in Figure 9.7, so we need some sort of automatic method for choosing a “best” threshold. There are in fact many different methods; here are two.

Otsu's Method

This was first described by Nobuyuki Otsu in 1979. It finds the best place to threshold the image into “foreground” and “background” components so that the *inter-class variance*—also known as the *between class variance*—is maximized. Suppose the image is divided into foreground and background pixels at a threshold t, and the fractions of each are w_f and w_b respectively. Suppose also that the means are μ_f and μ_b. Then the inter-class variance is defined as

$$w_f w_b (\mu_f - \mu_b)^2.$$

If an image has n_i pixels of gray value i, then define $p_i = n_i/N$ where N is the total number of pixels. Thus, p_i is the probability of a pixel having gray value i. Given a threshold value t then

$$w_b = \sum_{k=0}^{t-1} p_k,$$
$$w_f = \sum_{k=t}^{L-1} p_k.$$

where L is the number of gray values in the image. Since by definition we must have

$$\sum_{k=0}^{L-1} p_k = 1$$

it follows that $w_k + w_f = 1$. The background weights can be computed by a cumulative sum of all the p_k values.

The means can be defined as

$$\mu_b = \frac{1}{w_b} \sum_{k=0}^{t-1} k p_k,$$
$$\mu_f = \frac{1}{w_f} \sum_{k=t}^{L-1} k p_k.$$

Note that the sums

$$\sum_{k=0}^{t-1} k p_k$$

again can be computed by cumulative sums, and

$$\sum_{k=t}^{L-1} k p_k = \sum_{k=0}^{L-1} k p_k - \sum_{k=0}^{t-1} k p_k.$$

and the first term on the left is fixed.

For a simple example, consider an 8 × 8 3-bit image with the following values of n_k and $p_k = n_k/N = n_k/64$

i	n_i	p_i
0	4	0.062500
1	8	0.125000
2	10	0.156250
3	9	0.140625
4	4	0.062500
5	8	0.125000
6	12	0.187500
7	9	0.140625

Then the other values can be computed easily:

k	n_k	p_k	w_b	w_f	kp_k	$\sum kp_k$	μ_b	μ_f	vars
0	4	0.06250	0.06250	0.93750	0.00000	0.00000	0.00000	4.10000	0.98496
1	8	0.12500	0.18750	0.81250	0.12500	0.12500	0.66667	4.57692	2.32935
2	10	0.15625	0.34375	0.65625	0.31250	0.43750	1.27273	5.19048	3.46246
3	9	0.14062	0.48438	0.51562	0.42188	0.85938	1.77419	5.78788	4.02348
4	4	0.06250	0.54688	0.45312	0.25000	1.10938	2.02857	6.03448	3.97657
5	8	0.12500	0.67188	0.32812	0.62500	1.73438	2.58140	6.42857	3.26296
6	12	0.18750	0.85938	0.14062	1.12500	2.85938	3.32727	7.00000	1.63013
7	9	0.14062	1.00000	0.00000	0.98438	3.84375	3.84375	—	—

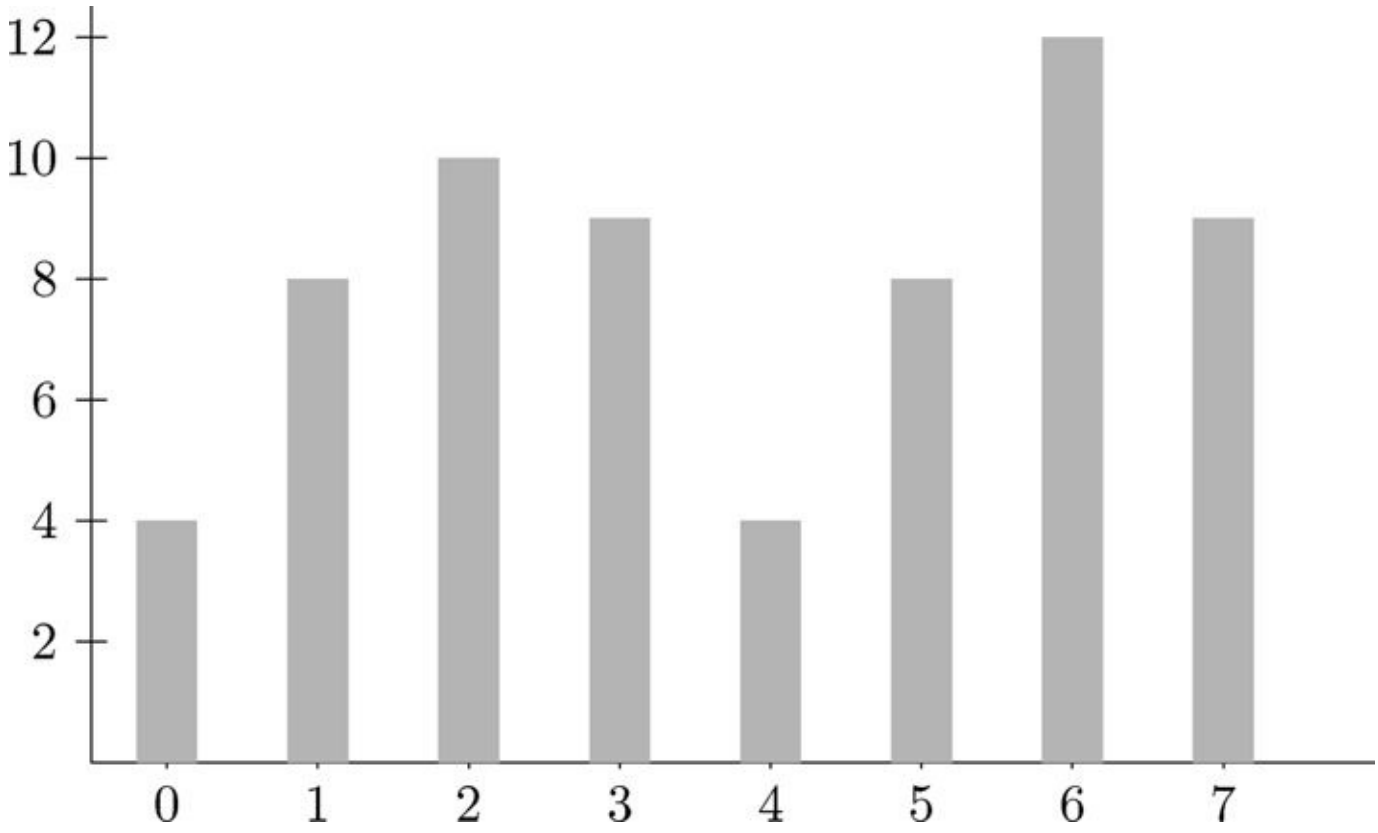
So in this case, we have

$$\mu_f = \frac{1}{w_f}\left(3.84375 - \sum_{k=0}^{t-1} kp_k\right).$$

The largest value of the inter-class variance is at $k = 3$, which means that $t - 1 = 3$ or $t = 4$ is the optimum threshold.

And in fact if a histogram is drawn as shown in Figure 9.8 it can be seen that this is a reasonable place for a threshold.

Figure 9.8: An example of a histogram illustrating Otsu's method



The ISODATA Method

ISODATA is an acronym for "Iterative Self-Organizing Data Analysis Technique A," where the final "A" was added, according to the original paper, "...to make ISODATA pronouncable." It is in fact a very simple method, and in practice converges very fast:

1. Pick a precision value ε and a starting threshold value t. Often $t = L/2$ is used.
2. Compute μ_f and μ_b for this threshold value.
3. Compute $t' = (\mu_b + \mu_f)/2$.
4. If $|t - t'| < \varepsilon$ the stop and return t, else put $t = t'$ and go back to step 2.

On the cameraman image `c`, for example, the means can be quickly set up in MATLAB and Octave using the `cumsum` function, which produces the cumulative sum of a one-dimensional array, or of the columns of a two-dimensional array:

```
>> k = (0:255)';
>> n = histc(c(:),k); p = n/prod(size(c));
>> wb = cumsum(p); wf = 1-wb;
>> kpc = cumsum(k.*p);
>> mu_b = kpc./wb; mu_f = (kpc(end)-kpc)./wf;
```

MATLAB/Octave

and in Python also using its `cumsum` function:

```
In :  k = np.arange(256)
In :  n = ndi.histogram(c,0,255,256); p = n/(c.size+0.0)
In :  wb = np.cumsum(p); wf = 1-wb
In :  kpc = np.cumsum(k*p)
In :  mu_b = kpc/wb; mu_f = (kpc[-1]-kpc)/wf;
```

Python

Now to compute 10 iterations of the algorithm (this saves having to set up a stopping criterion with a precision value):

```
>> t = 128
>> for i = 1:10,
>    t1 = floor((mu_b(t)+mu_f(t))/2),
>    t = t1;
>  end
t1 =  108
t1 =  95
t1 =  89
t1 =  88
t1 =  88
t1 =  88
t1 =  88
t1 =  88
t1 =  88
t1 =  88
```

MATLAB/Octave

```
In :  t = 128
In :  for i in range(10):
...:     t1 = int((mu_f[t]+mu_b[t])/2.0)
...:     print t1
...:     t = t1
...:
108
95
90
88
88
88
88
88
88
88
```

Python

The iteration quickly converges in only four steps.

MATLAB and Octave provide automatic thresholding with the `graythresh` function, with parameters `'otsu'` and `'intermeans'` in Octave for Otsu's method and the ISODATA method. In Python there are the methods `threshold_otsu` and `threshold_isodata` in the `filter` module of `skimage`.

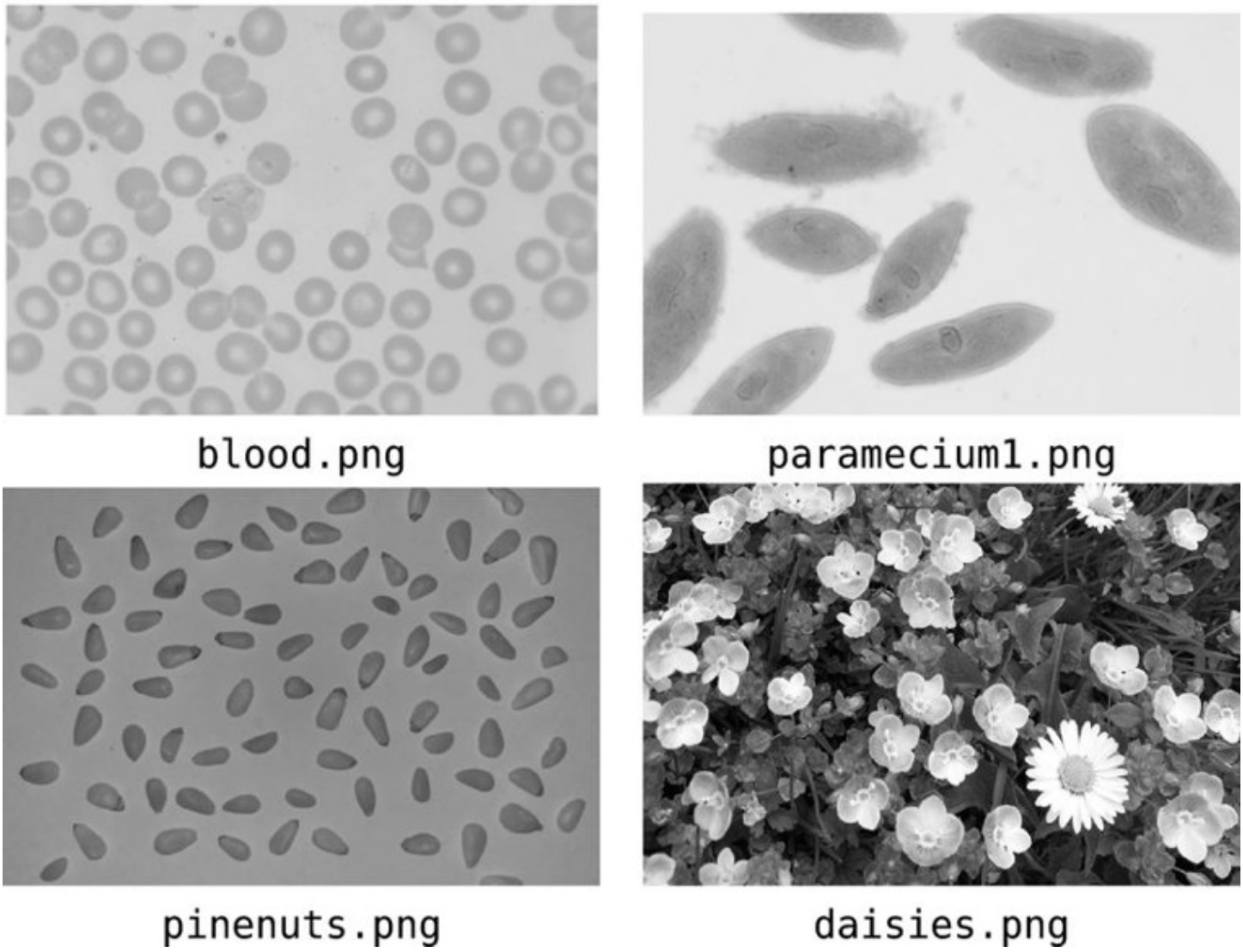
Consider four images, `blood.png, paramecium1.png, pinenuts.png`, and `daisies.png` which are shown in Figure 9.9.

With the images saved as arrays `b, p, n`, and `d`, the threshold values can be computed in Octave as follows, by setting up an anonymous function to perform thresholding:

```
>> thresh = @(x) [graythresh(x,'otsu'),graythresh(x,'intermeans')];
>> ts = cellfun(@thresh, {b,p,n,d},'UniformOutput',false);
>> for i = 1:4, disp(ts{i}), end
   0.74510   0.74510
   0.75294   0.75294
   0.47451   0.47451
   0.51373   0.51373
```

Octave

Figure 9.9: Images to be thresholded



This is an example of putting all the images into a cell array, and iterating over the array using `cellfun`, using a previously described function (in this case `thresh`). A cell array is useful here as it can hold arbitrary objects.

MATLAB only supports Otsu's method:

```
>> cellfun(@(x) graythresh(x),{b,p,n,d},'UniformOutput',false)
ans =

    [0.7451]    [0.7529]    [0.4745]    [0.5137]
```

MATLAB

In Python, we can iterate over the images by putting them in a list:

```
In :  for x in [b,p,n,d]:
...:     print [fl.threshold_otsu(x), fl.threshold_isodata(x)]
...:
[190, 190]
[192, 192]
[121, 122]
[131, 131]
```

Python

Note that Python returns an 8-bit integer if that is the initial image type, whereas both MATLAB and Octave return a value of type `double`.

Once the optimum threshold value has been obtained, it can be applied to the image using `im2bw`:

```
>> imshow(im2bw(b,graythresh(b)))
```

MATLAB/Octave

or

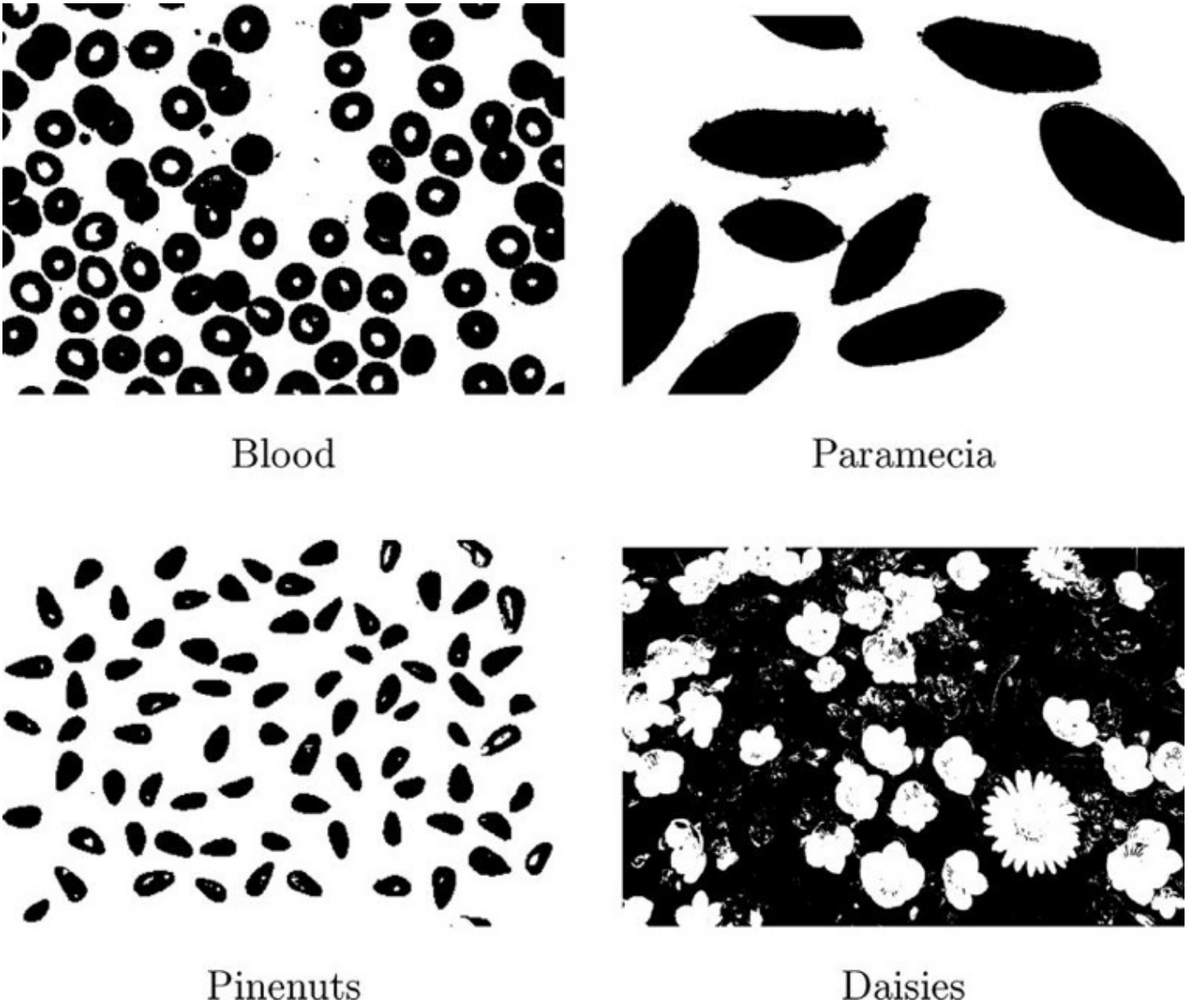
```
In :  io.imshow(b>fl.threshold_otsu(b))
```

Python

and similarly for all the others.

The results are shown in Figure 9.10. Note that for each image the result given is quite satisfactory.

Figure 9.10: Thresholding with values obtained with Otsu's method



9.5 Adaptive Thresholding

Sometimes it is not possible to obtain a single threshold value that will isolate an object completely. This may happen if both the object and its background vary. For example, suppose we take the paramecium image and adjust it so that both the objects and the background vary in brightness across the image. This can be done in MATLAB or Octave as:

```
>> p = im2double(imread('paramecium1.png'))
>> [r,c] = size(p);
>> [y,x] = meshgrid(linspace(0,1,c),linspace(0,1,r));
>> p2 = 1-p+y/2;
>> imshow(p2)
```

MATLAB/Octave

and in Python as

```
In :  p = io.imread('paramecium1.png').astype(float)
In :  x,y = np.mgrid[0:r,0:c].astype(float)
In :  p2 = 255.0-p+y/2
In :  io.imshow(p2)
```

Python

Figure 9.11 shows an attempt at thresholding, using `graythresh`, with MATLAB or Octave:

```
>> t = graythresh(p2)
ans
    t =  0.43529
>> figure,imshow(im2bw(p2,t))
```

MATLAB/Octave

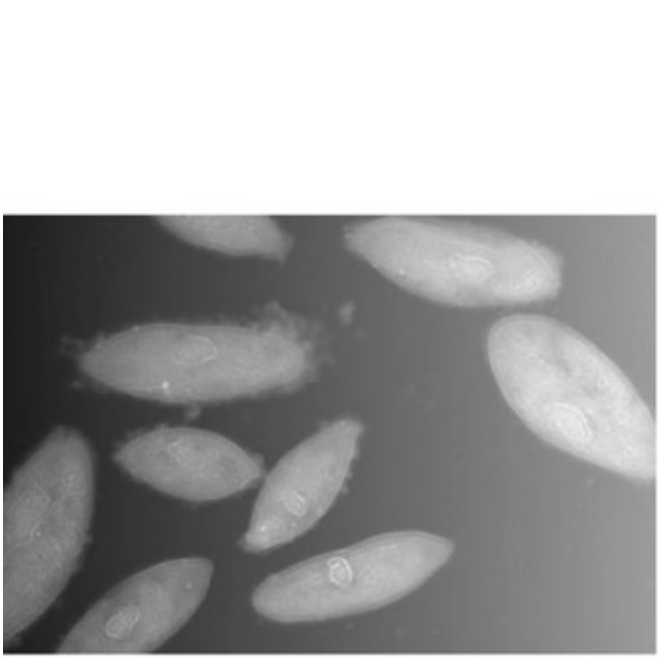
or with Python:

```
In :  t = fl.threshold_otsu(p2); t
Out:  290.7939453125
In :  io.imshow(p2>t)
```

Python

As you see, the result is not particularly good; not all of the objects have been isolated from their background. Even if different thresholds are used, the results are similar. To see

Figure 9.11: An attempt at thresholding



(a) Paramecium image: p2

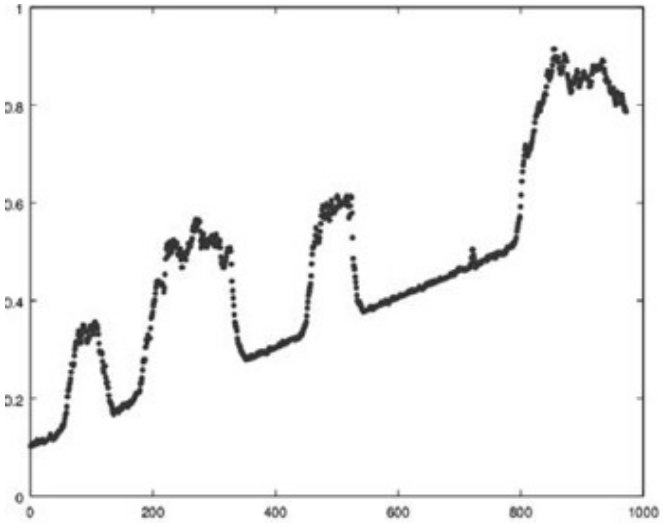
(b) Thresholding attempt

why a single threshold won't work, look at a plot of pixels across the image, as shown in Figure 9.12(a).

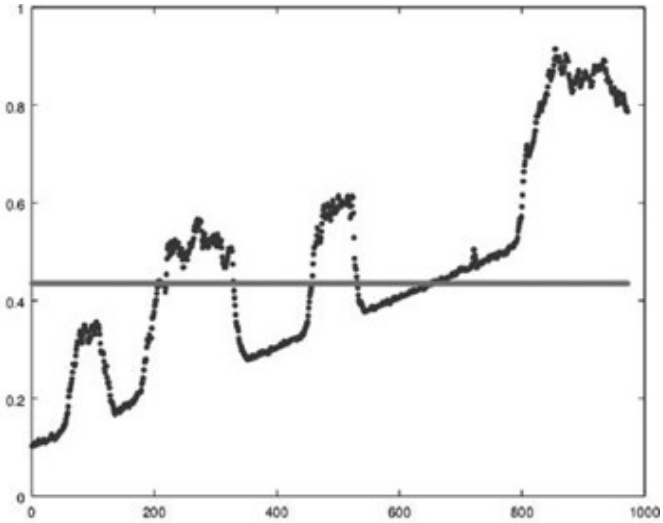
In this figure, the line of pixels is being shown as a function; the threshold is shown on the right as a horizontal line. It can be seen that no position of the plane can cut off the objects from the background.

What can be done in a situation like this is to cut the image into small pieces, and apply thresholding to each piece individually. Since in this particular example the brightness changes from left to right, we shall cut up the image into six pieces and apply a threshold to each piece individually. The vectors `starts` and `ends` contain the column indices of the start and end of each block.

Figure 9.12: An attempt at thresholding—functional version



(a) A horizontal line of pixels



(b) Thresholding attempt

```
>> out = zeros(r,c);
>> starts = 1:c/6:c
starts =

     1   163   325   487   649   811
>> ends = 0:c/6:c
ends =

   162   324   486   648   810   972
>> for i = 1:6,
>    temp = p2(:,starts(i):ends(i))
>    out((:,starts(i):ends(i)) = im2bw(temp,graythresh(temp));
>  end
```

MATLAB/Octave

In Python, the commands are very similar:

```
In :  starts = range(0,c-1,162)
In :  ends = range(162,c+1,162)
In :  z = np.zeros((r,c))
In :  for i in range(6):
...:      temp = p2[:,starts[i]:ends[i]]
...:      z[:,starts[i]:ends[i]]=(temp>fl.threshold_otsu(temp))*1.0
...:
```

Python

Figure 9.13(a) shows how the image is sliced up, and Figure 9.13(b) shows the results after thresholding each piece. The above commands can be done much more simply by using the command `blockproc`, which applies a particular function to each block of the image. We can define our function with

```
>> thresh = @(z) im2bw(z,graythresh(z));
```

MATLAB/Octave

Notice that this uses the same as the command used above to create each piece of the previous threshold, except that now `x` is used to represent a general input variable.

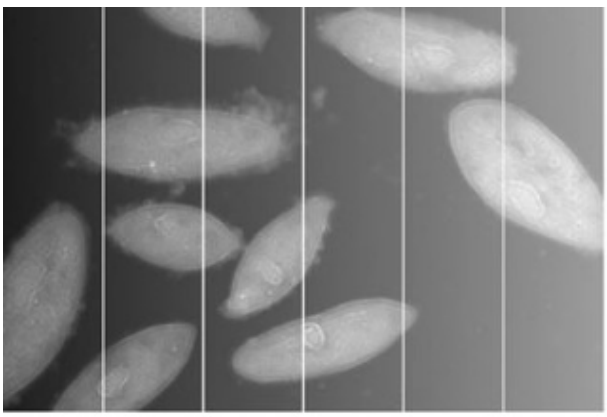
The function can then be applied to the image `p2` with

```
>> blockproc(p2,[r,c/6],thresh);
```

MATLAB/Octave

What this command means is that we apply our function `thresh` to each distinct 648 × 162 block of our image.

Figure 9.13: Adaptive thresholding



(a) Cutting up the image

(b) Thresholding each part separately

9.6 Edge Detection

Edges contain some of the most useful information in an image. We may use edges to measure the size of objects in an image; to isolate particular objects from their background; to recognize or classify objects.

There are a large number of edge-finding algorithms in existence, and we shall look at some of the more straightforward of them. The general command in MATLAB or Octave for finding edges is

```
edge(image,'method',parameters...)
```

where the parameters available depend on the method used. In Python, there are many edge detection methods in the `filter` module of `skimage`. In this chapter, we shall show how to create edge images using basic filtering methods, and discuss the uses of those edge functions.

An *edge* may be loosely defined as a local discontinuity in the pixel values which exceeds a given threshold. More informally, an edge is an observable difference in pixel values. For example, consider the two 4 × 4 blocks of pixels shown in Figure 9.14.

Figure 9.14: Blocks of pixels

51	52	53	59
54	52	53	62
50	52	53	68
55	52	53	55

50	53	155	160
51	53	160	170
52	53	167	190
51	53	162	155

In the right-hand block, there is a clear difference between the gray values in the second and third columns, and for these values the differences exceed 100. This would be easily discernible in an image—the human eye can pick out gray differences of this magnitude with relative ease. Our aim is to develop methods that will enable us to pick out the edges of an image.

9.7 Derivatives and Edges

Fundamental Definitions

Consider the image in Figure 9.15, and suppose we plot the gray values as we traverse the image from left to right. Two types of edges are illustrated here: a *ramp edge*, where the gray values change slowly, and a *step edge*, or an *ideal edge*, where the gray values change suddenly.

Figure 9.15: Edges and their profiles

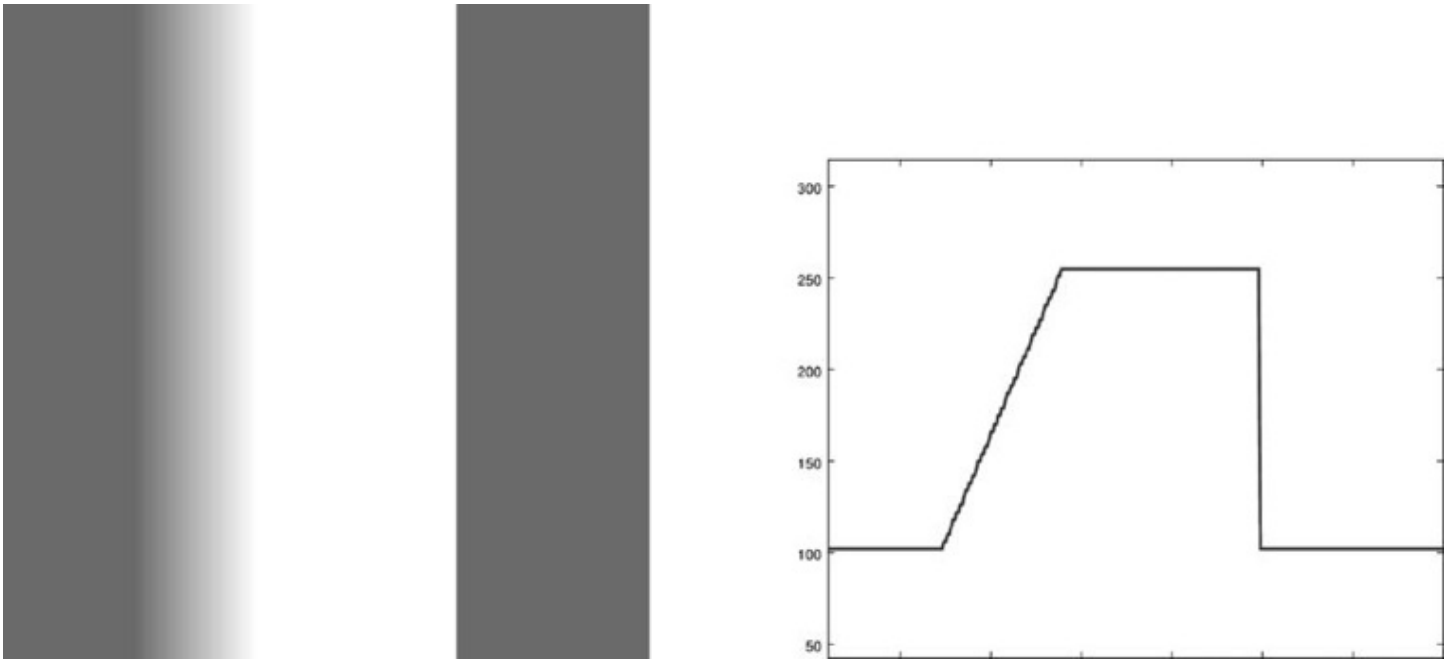
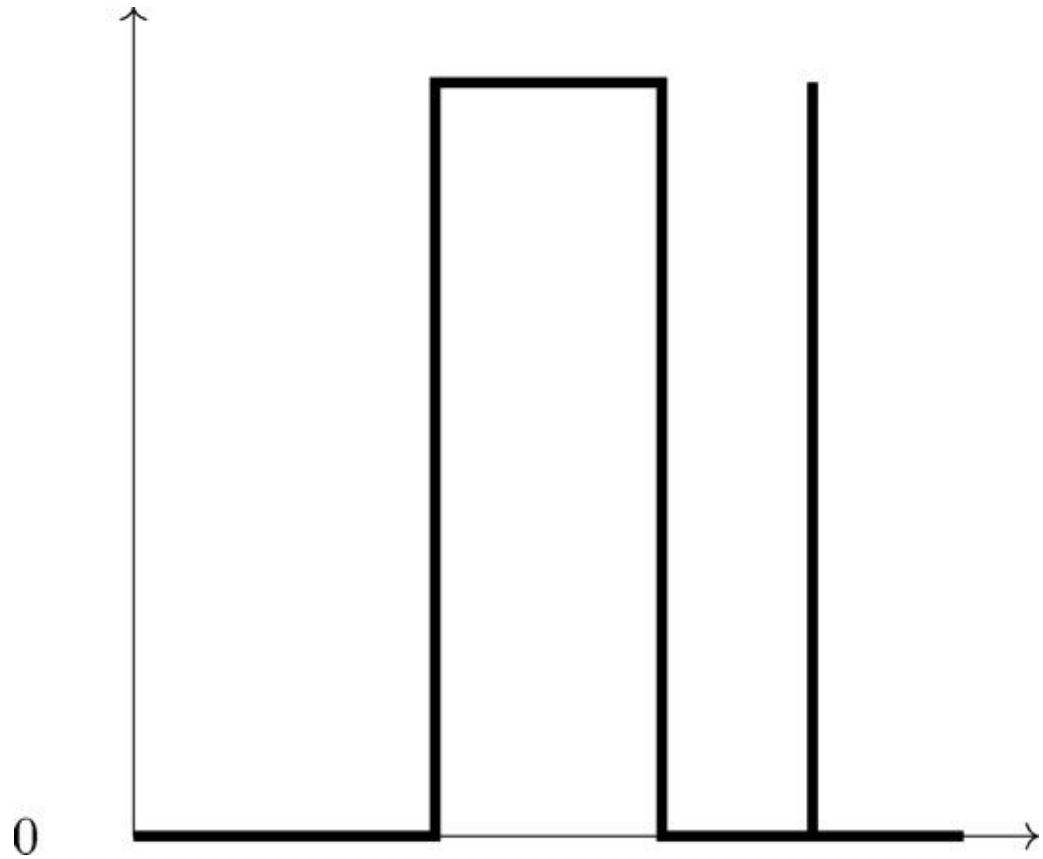


Figure 9.16: The derivative of the edge profile



Suppose the function that provides the profile in Figure 9.15 is $f(x)$; then its derivative $f'(x)$ can be plotted; this is shown in Figure 9.16. The derivative, as expected, returns zero for all constant sections of the profile, and is non-zero (in this example) only in those parts of the image in which differences occur.

Many edge finding operators are based on differentiation; to apply the continuous derivative to a discrete image, first recall the definition of the derivative:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Since in an image, the smallest possible value of h is 1, being the difference between the index values of two adjacent pixels, a discrete version of the derivative expression is

$$f(x+1) - f(x).$$

Other expressions for the derivative are

$$\lim_{h \to 0} \frac{f(x) - f(x-h)}{h}, \qquad \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

with discrete counterparts

$$f(x) - f(x-1), \qquad (f(x+1) - f(x-1))/2.$$

For an image, with two dimensions, we use partial derivatives; an important expression is the *gradient*, which is the vector defined by

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}$$

which for a function $f(x, y)$ points in the direction of its greatest increase. The direction of that increase is given by

$$\tan^{-1}\left(\frac{\partial f / \partial y}{\partial f / \partial x}\right)$$

and its magnitude by

$$\sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}.$$

Most edge detection methods are concerned with finding the magnitude of the gradient, and then applying a threshold to the result.

Some Edge Detection Filters

Using the expression $f(x + 1) - f(x - 1)$ for the derivative, leaving the scaling factor out, produces horizontal and vertical filters:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

These filters will find vertical and horizontal edges in an image and produce a reasonably bright result. However, the edges in the result can be a bit “jerky”; this can be overcome by smoothing the result in the opposite direction; by using the filters

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Both filters can be applied at once, using the combined filter:

$$P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

This filter, and its companion for finding horizontal edges:

$$P_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

are the *Prewitt* filters for edge detection.

If p_x and p_y are the gray values produced by applying P_x and P_y to an image, then the magnitude of the gradient is obtained with

$$\sqrt{p_x^2 + p_y^2}.$$

In practice, however, it is more convenient to use either of

$$\max\{|p_x|, |p_y|\}$$

or

$$|p_x| + |p_y|.$$

Figure 9.17: A set of steps: A test image for edge detection

This (and other) edge detection methods will be tested on the image `stairs.png`, which we suppose has been read into our system as the array `s`. It is shown in Figure 9.17. Applying each of P_x and P_y

individually provides the results shown in Figure 9.18 The images in Figure 9.18 can be produced with the following MATLAB commands:

```
>> px = [-1 0 1;-1 0 1;-1 0 1]; py = px'
>> sx = imfilter(s,px);
>> sy = imfilter(s,py);
>> imshow(sx),figure,imshow(sy)
```

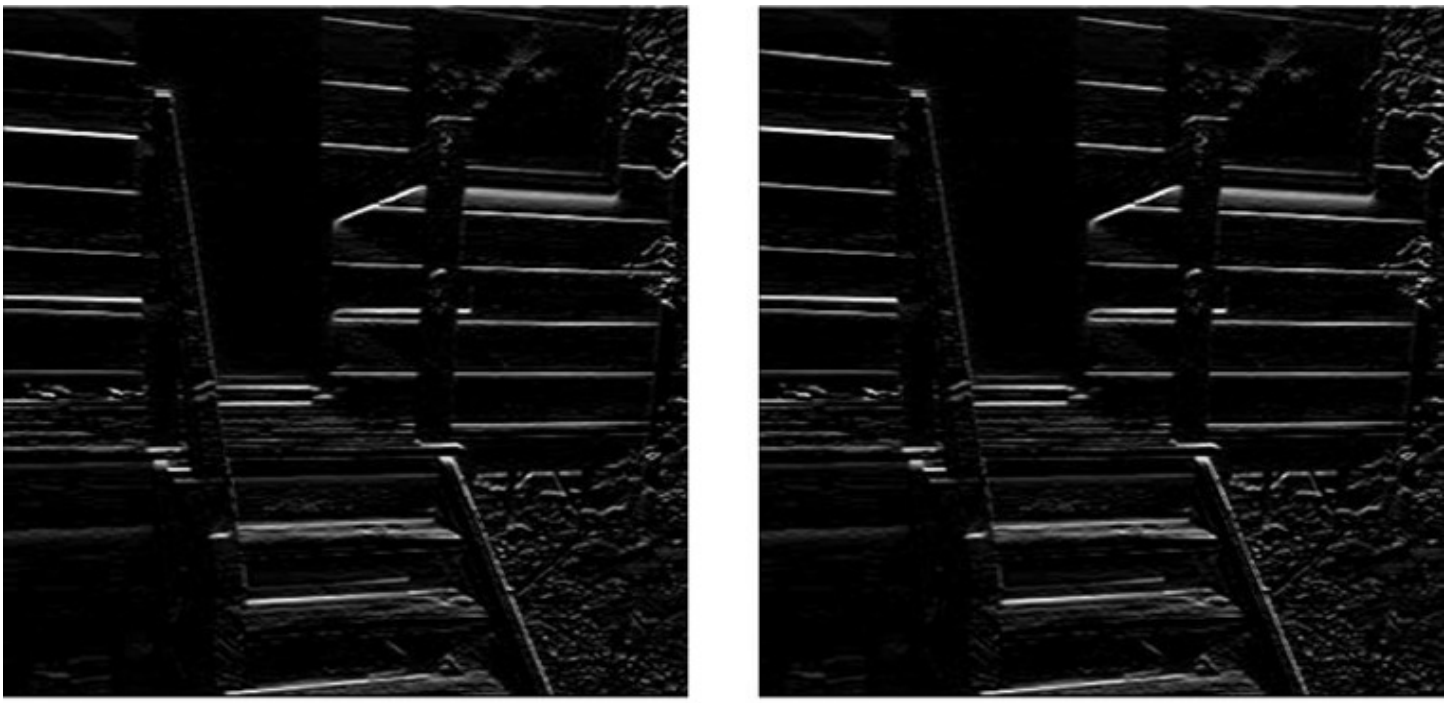
MATLAB/Octave

or these Python commands:

```
In :  sx = fl.hprewitt(s)
In :  sy = fl.vprewitt(s)
In :  io.imshow(sx)
In :  f = plt.figure(); f.show(io.imshow(sy))
```

Python

Figure 9.18: The result after filtering with the Prewitt filters



(a) sx (b) sy

(There are in fact slight differences in the outputs owing to the way in which each system manages negative values in the result of a filter.) Note that the filter P_x highlights vertical edges, and P_y horizontal edges. We can create a figure containing all the edges with:

```
>> edge_p = uint8(sqrt(double(sx).^2+double(sy).^2));
>> figure,imshow(edge_p))
```

MATLAB/Octave

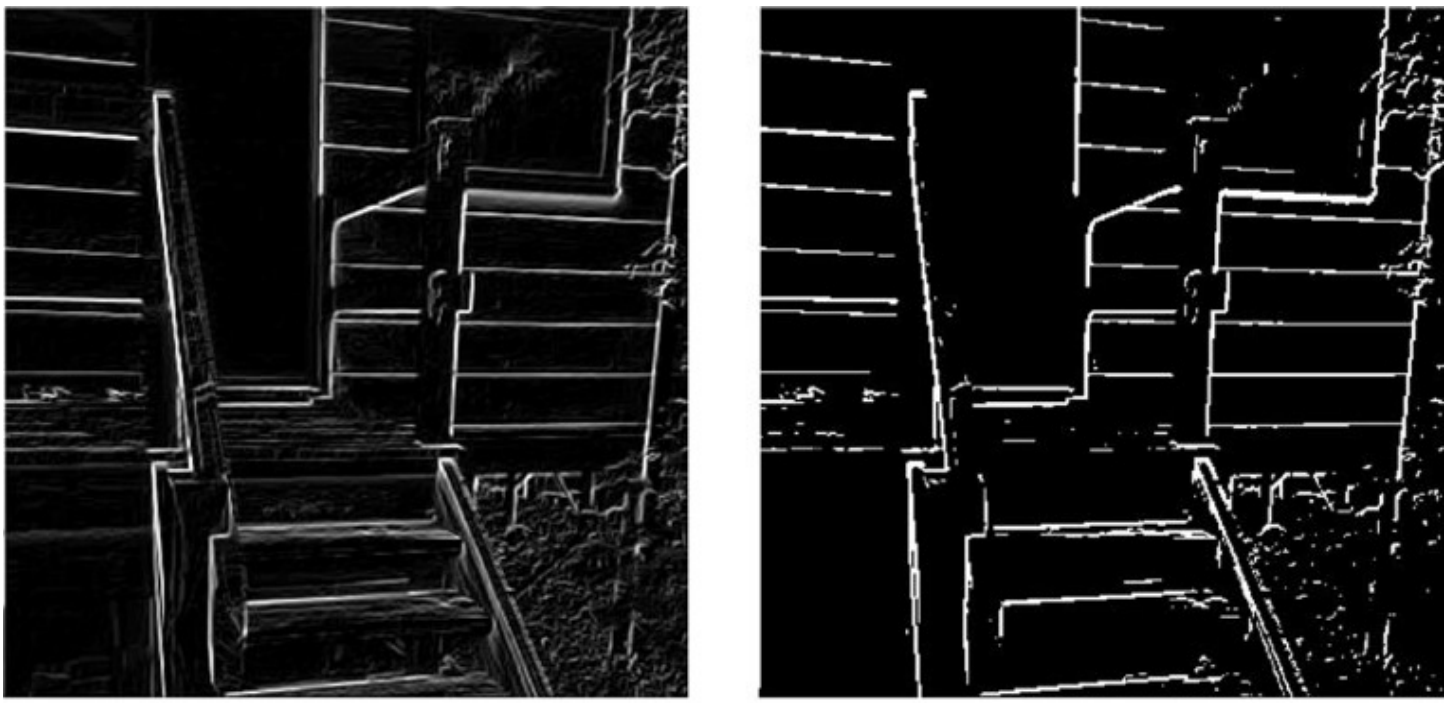
or

```
In :  edge_p = sqrt(sx**2+sy**2)
```

Python

and the result is shown in Figure 9.19(a). This is a grayscale image; a binary image containing edges only can be produced by thresholding. Figure 9.19(b) shows the result after thresholding with a value found by Otsu's method; this optimum threshold value is 0.3333.

Figure 9.19: All the edges of the image



(a) (b)

We can obtain edges by the Prewitt filters directly in MATLAB or Octave by using the command

```
>> edge_p = edge(s,'prewitt');
```

MATLAB/Octave

and the `edge` function takes care of all the filtering, and of choosing a suitable threshold level; see its help text for more information. The result is shown in Figure 9.20. Note that Figures 9.19(b) and 9.20 seem different from each other. This is because the `edge` function does some extra processing over and above taking the square root of the sum of the squares of the filters.

Figure 9.20: The `prewitt` option of `edge`

Python does not have a function that automatically computes threshold values and cleans up the output. However, in Chapter 10, we will see how to clean up binary images.

Slightly different edge finding filters are the *Roberts cross-gradient filters*:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and the *Sobel filters*:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The *Sobel filters* are similar to the Prewitt filters in that they apply a smoothing filter in the opposite direction to the central difference filter. In the Sobel filters, the smoothing takes the form

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

which gives slightly more prominence to the central pixel. Figure 9.21 shows the respective results of the MATLAB/Octave commands

```
>> edge_r = edge(s,'roberts');
>> figure,imshow(edge_r)
```

MATLAB/Octave

```
>> edge_s = edge(s,'sobel');
>> figure,imshow(edge_s)
```

MATLAB/Octave

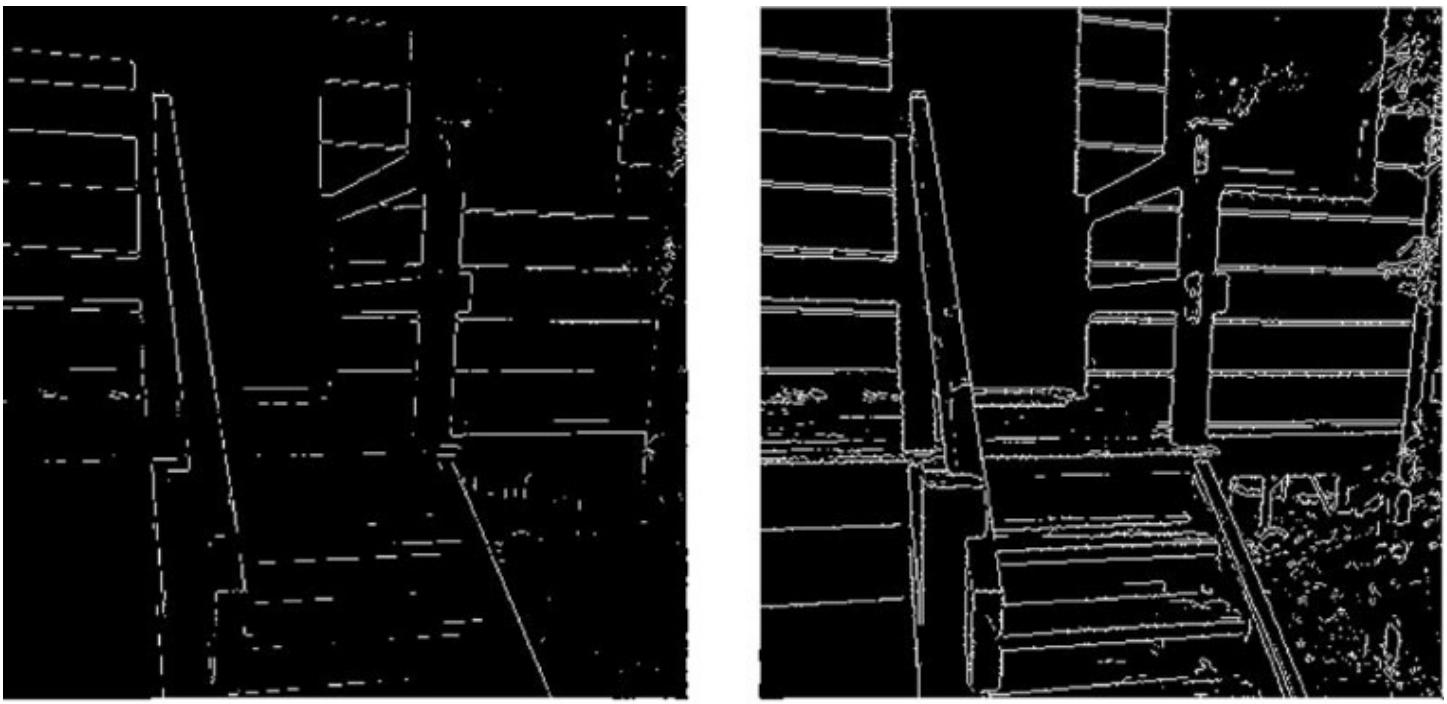
and

```
>> edge_s = edge(s,'sobel');
>> figure,imshow(edge_s)
```

MATLAB/Octave

The appearance of each of these can be changed by specifying a threshold level.

Figure 9.21: Results of the Roberts and Sobel filters



(a) Roberts edge detection (b) Sobel edge detection

The Python outputs, with

```
In :  edge_r = fl.roberts(s)
In :  edge_s = fl.sobel(s)
```

Python

are shown in Figure 9.22. Of the three filters, the Sobel filters are probably the best; they provide good edges, and they perform reasonably well in the presence of noise.

Figure 9.22: Results of the Roberts and Sobel filters in Python

(a) Roberts edge detection

(b) Sobel edge detection

9.8 Second Derivatives

The Laplacian

Another class of edge-detection method is obtained by considering the second derivatives.

The sum of second derivatives in both directions is called the *Laplacian*; it is written as

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

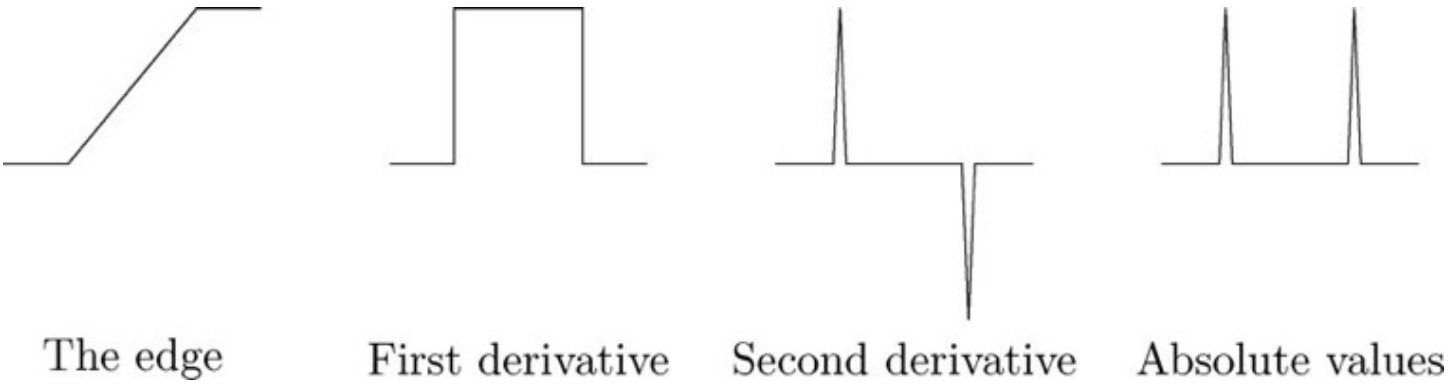
and it can be implemented by the filter

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

This is known as a *discrete Laplacian*. The Laplacian has the advantage over first derivative methods in that it is an *isotropic filter* [40]; this means it is invariant under rotation. That is, if the Laplacian is applied to an image, and the image is then rotated, the same result would be obtained if the image were rotated first, and the Laplacian applied second. This would appear to make this class of filters ideal for edge detection. However, a major problem with all second derivative filters is that they are very sensitive to noise.

To see how the second derivative affects an edge, take the derivative of the pixel values as plotted in Figure 9.15; the results are shown schematically in Figure 9.23.

Figure 9.23: Second derivatives of an edge function



The Laplacian (after taking an absolute value, or squaring) gives double edges. To see an example, suppose we enter the MATLAB/Octave commands:

```
>> lap = fspecial('laplacian',0);
>> s_lap = imfilter(s,lap);
```

MATLAB/Octave

or the Python commands

```
In :  s2 = ut.img_as_float(s)
In :  s_lap = abs(fl.laplace(s2))
```

Python

the result of which is shown in Figure 9.24.

Figure 9.24: Result after filtering with a discrete Laplacian

Although the result is adequate, it is very messy when compared to the results of the Prewitt and Sobel methods discussed earlier. Other Laplacian masks can be used; some are:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{bmatrix}.$$

In MATLAB and Octave, Laplacians of all sorts can be generated using the `fspecial` function, in the form

```
fspecial('laplacian',ALPHA)
```

which produces the Laplacian

$$\frac{1}{\alpha+1}\begin{bmatrix} \alpha & 1-\alpha & \alpha \\ 1-\alpha & -4 & 1-\alpha \\ \alpha & 1-\alpha & \alpha \end{bmatrix}.$$

If the parameter `ALPHA` (which is optional) is omitted, it is assumed to be 0.2. The value 0 gives the Laplacian developed earlier.

Zero Crossings

A more appropriate use for the Laplacian is to find the *position* of edges by locating *zero crossings*. From Figure 9.23, the position of the edge is given by the place where the value of the filter takes on a zero value. In general, these are places where the result of the filter changes sign. For example, consider the the simple image given in Figure 9.25(a), and the result after filtering with a Laplacian mask in Figure 9.25(b).

We define the *zero crossings* in such a filtered image to be pixels that satisfy either of the following:

1. They have a negative gray value and are next to (by four-adjacency) a pixel whose gray value is positive
2. They have a value of zero, and are between negative and positive valued pixels

To give an indication of the way zero-crossings work, look at the edge plots and their second differences in Figure 9.26.

Figure 9.25: Locating zero crossings in an image

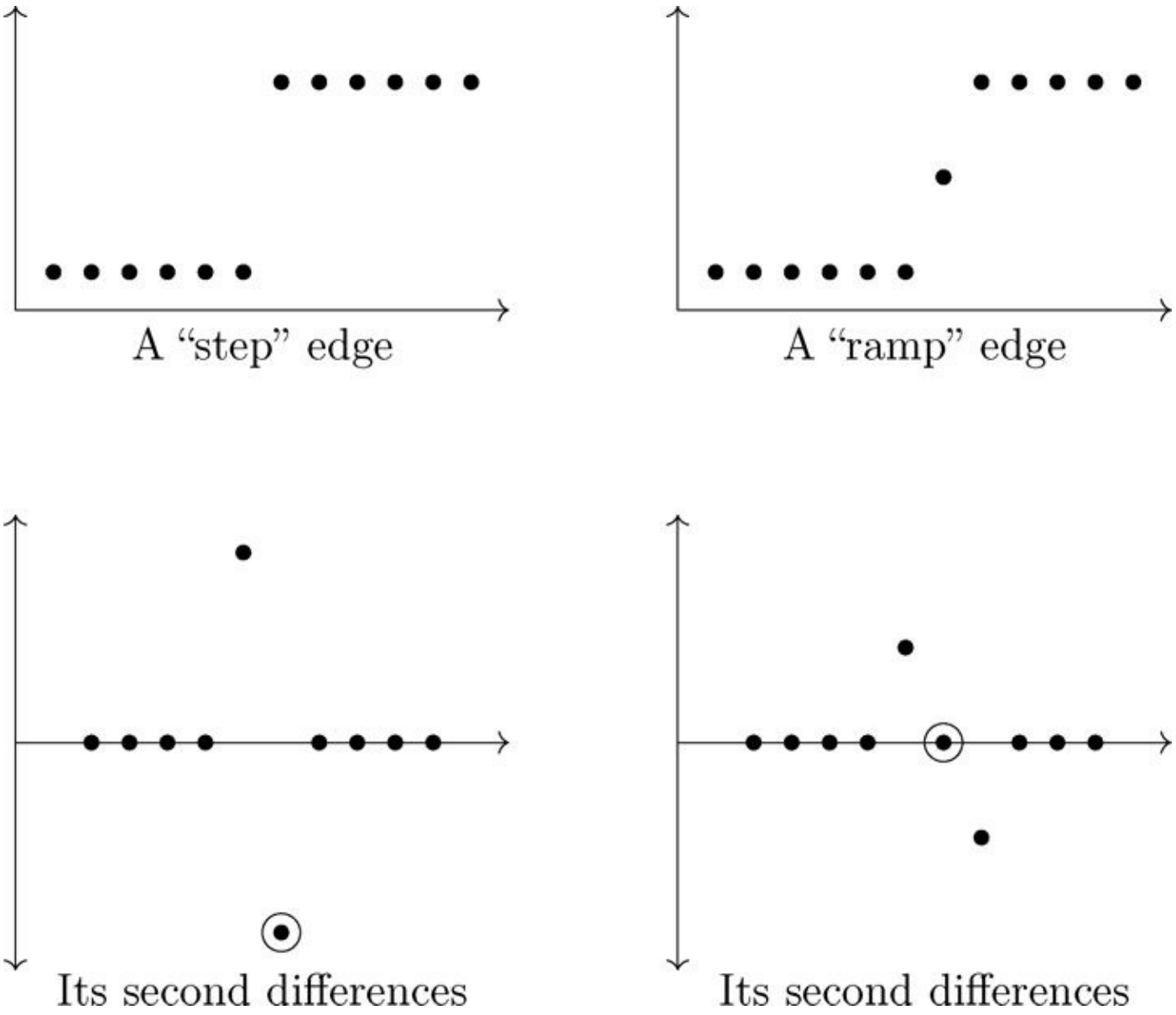
50	50	50	50	50	50	50	50	50	50
50	50	50	50	50	50	50	50	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	200	200	200	200	200	200	50	50
50	50	50	50	200	200	200	200	50	50
50	50	50	50	200	200	200	200	50	50
50	50	50	50	50	50	50	50	50	50
50	50	50	50	50	50	50	50	50	50

(a) A simple image

-100	-50	-50	-50	-50	-50	-50	-50	-50	-100
-50	0	150	150	150	150	150	150	0	-50
-50	150	-300	-150	-150	-150	-150	-300	150	-50
-50	150	-150	0	0	0	0	-150	150	-50
-50	150	-150	0	0	0	0	-150	150	-50
-50	150	-300	-150	0	0	0	-150	150	-50
-50	0	150	0	-150	0	0	-150	150	-50
-50	0	0	150	-300	-150	-150	-300	150	-50
-50	0	0	0	150	150	150	150	0	-50
-100	-50	-50	-50	-50	-50	-50	-50	-50	-100

(b) After Laplacian filtering

Figure 9.26: Edges and second differences



In each case, the zero-crossing is circled. The important point is to note that across any edge there can be only *one* zero-crossing. Thus, an image formed from zero-crossings has the potential to be very neat.

In Figure 9.25(b) the zero crossings are shaded. We now have a further method of edge detection: take the zero-crossings after a Laplacian filtering. This is implemented in MATLAB with the `zerocross` option of `edge`, which takes the zero crossings after filtering with a given filter:

```
>> lap = fspecial('laplace',0);
>> sz = edge(s,'zerocross',[],lap);
>> imshow(sz)
```

MATLAB/Octave

The result is shown in Figure 9.27(a). This is not in fact a very good result—far too many gray level changes have been interpreted as edges by this method. To eliminate them, we may first smooth the image with a Gaussian filter. This leads to the following sequence of steps for edge detection; the *Marr-Hildreth* method:

1. Smooth the image with a Gaussian filter
2. Convolve the result with a Laplacian
3. Find the zero crossings

This method was designed to provide an edge detection method to be as close as possible to biological vision. The first two steps can be combined into one, to produce a "Laplacian of Gaussian" or "LoG" filter. These filters can be created with the `fspecial` function. If no extra parameters are provided to the `zerocross` edge option, then the filter is chosen to be the LoG filter found by

```
>> fspecial('log',13,2)
```

MATLAB/Octave

This means that the following command:

```
>> edge(s,'log');
```

MATLAB/Octave

produces exactly the same result as the commands:

```
>> log = fspecial('log',13,2);
>> edge(s,'zerocross',[],log);
```

MATLAB/Octave

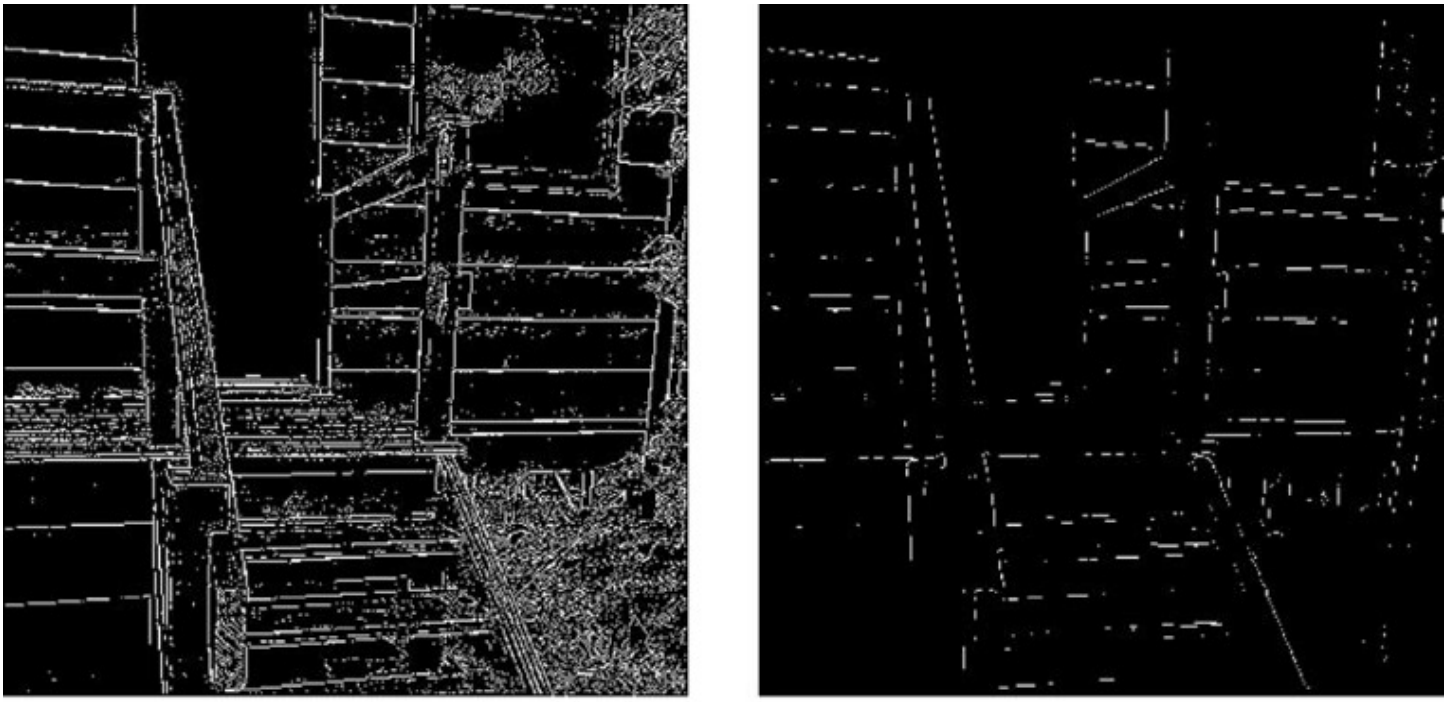
In fact, the `LoG` and `zerocross` options implement the same edge finding method; the difference being that the `zerocross` option allows you to specify your own filter. The result after applying an LoG filter and finding its zero crossings is given in Figure 9.27(b).

Python does not have a zero crossing detector, but it is easy to write one, and a simple one is given at the end of the chapter. Using this function, the edges can be found with

```
In :  s2 = ndi.gaussian_laplace(float64(s),3)
In :  s_edge = zerocross(s2)
```

Python

Figure 9.27: Edge detection using zero crossings



(a) Zero crossings (b) Using an LoG filter first

Note that the image must be converted to type `float64` first. The filtering function applied to an image of type `uint8` will return an output of the same type, with modular "wrap around" of values outside the range 0–255. This will introduce unnecessary artifacts in the output.

9.9 The Canny Edge Detector

All the edge finding methods so far have required a straightforward application of linear filters, with the addition of finding zero crossings. All of our systems support a more complex edge detection technique, first described by John Canny in 1986 [6] and so named for him.

Canny designed his method to meet three criteria for edge detection:

1. **Low error rate of detection.** It should find *all* edges, and nothing but edges.
2. **Localization of edges.** The distance between actual edges in the image and edges found by this algorithm should be minimized.
3. **Single response.** The algorithm should not return multiple edge pixels when only a single edge exists.

Canny showed that the best filter to use for beginning his algorithm was a Gaussian (for smoothing), followed by the derivative of the Gaussian, which in one dimension is

$$\left(-\frac{x}{\sigma^2}\right) e^{-\frac{x^2}{2\sigma^2}}. \tag{9.1}$$

These have the effect of both smoothing noise and finding possible candidate pixels for edges. Since this filter is separable, it can be applied first as a column filter to the columns, next as a row filter to the rows.

We can then put the two results together to form an edge image. Recall from Chapter 5 that this is more efficient computationally than applying a two-dimensional filter.

Thus at this stage we have the following sequence of steps:

1. Take our image `x`.
2. Create a one-dimensional Gaussian filter `g`.
3. Create a one-dimensional filter `dg` corresponding to the expression given in Equation 9.1.
4. Convolve `g` with `dg` to obtain `gdg`.
5. Apply `gdg` to `x` producing `x1`.
6. Apply `gdg'` to `x` producing `x2`.

We can now form an edge image with

$$\mathtt{xe} = \sqrt{\mathtt{x1}^2 + \mathtt{x2}^2}.$$

So far we have not achieved much more than we would achieve with a standard edge detection by using a filter.

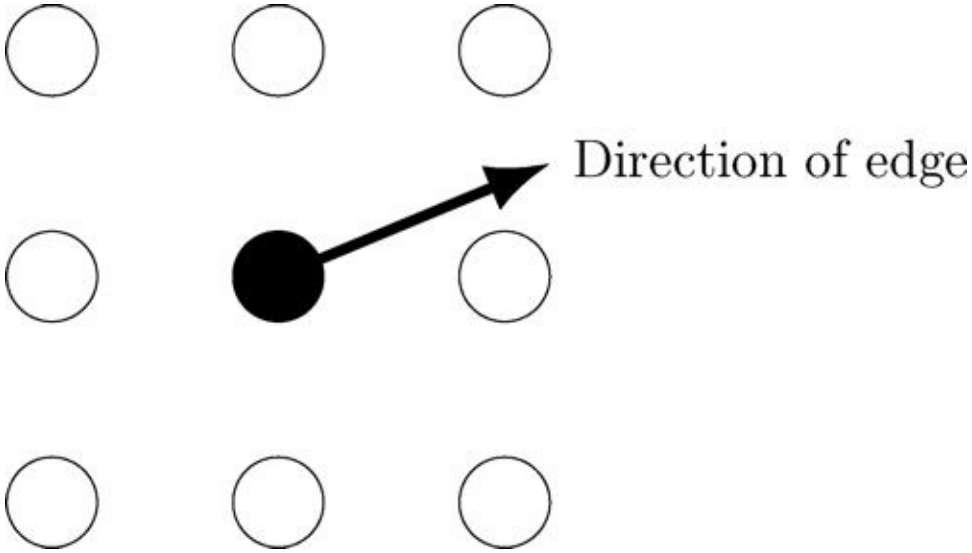
The next step is that of *non-maximum suppression*. What we want to do is to threshold the edge image `xe` from above to keep only edge pixels and remove the others. However, thresholding alone will not produce acceptable results. The idea is that every pixel p has a direction φ_p (an edge "gradient") associated with it, and to be considered as an edge pixel, a p must have a greater magnitude than its neighbors in direction φ_p.

Just as we computed the magnitude `xe` above we can compute the edge gradient using the inverse tangent function:

$$\mathtt{xg} = \tan^{-1}\left(\frac{\mathtt{x2}}{\mathtt{x1}}\right).$$

In general that direction will point between pixels in its 3 × 3 neighborhood, as shown in Figure 9.28. There are two approaches here: we can compare the gradient of the current (center) pixel with the value obtained from linear interpolation (see Chapter 6); basically we just take the weighted average of the gradients of the two pixels. So in Figure 9.28 we take the weighted average of the upper two pixels on the right.

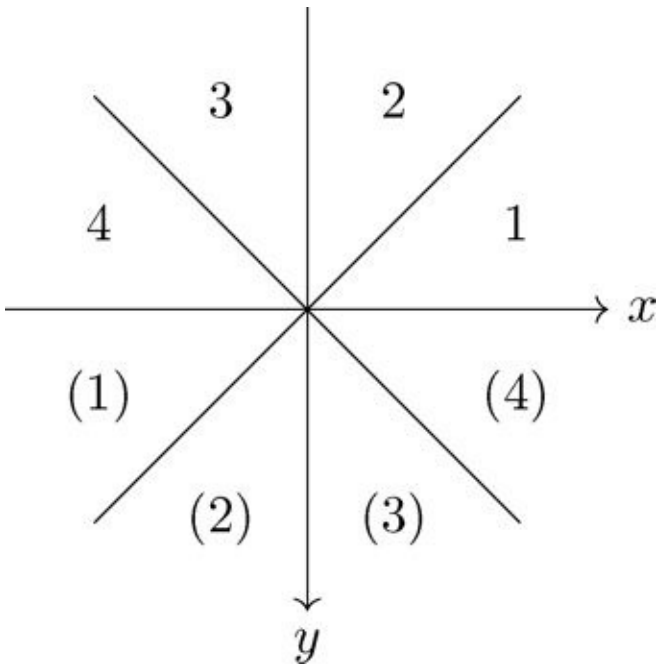
Figure 9.28: Non-maximum suppression in the Canny edge detector



The second approach is to quantize the gradient to one of the values 0°, 45°, 90°, or 135°, and compare the original gradient to the gradient of the pixel to which the quantized gradient points. That is, suppose the gradient at position (x, y) was $\varphi(x, y)$. We quantize this to one of the four angles given to obtain $\varphi'(x, y)$. Consider the two pixels in direction $\varphi'(x, y)$ and $\varphi'(x, y) + 180°$ from (x, y). If the edge magnitude of either of those is greater than the magnitude of the current pixel, we mark the current pixel for deletion. After we have passed over the entire image, we delete all marked pixels.

We can in fact compute the quantized gradients without using the inverse tangent: we simply compare the values in the two filter results `x1` and `x2`. Depending on the relative values of `x1`(x, y) and `x2`(x, y) we can place the gradient at (x, y) into one of the four gradient classes. Figure 9.29 shows how this is done. We divide the image plane into eight regions separated by lines at 45°, as shown, with the x axis positive to the right, and the y axis positive down. Then we can assign regions, and degrees, to pixel values according to

Figure 9.29: Using pixel locations to quantize the gradient

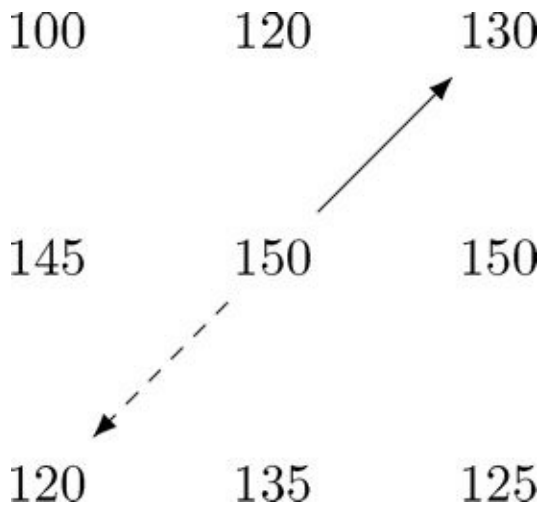


this table:

Region	Degree	Pixel Location
1	0°	$y \leq 0$ and $x > -y$
(1)	0°	$y \geq 0$ and $x < -y$
2	45°	$x > 0$ and $x \leq -y$
(2)	45°	$x < 0$ and $x \geq -y$
3	90°	$x \leq 0$ and $x > y$
(3)	90°	$x \geq 0$ and $x < y$
4	135°	$y < 0$ and $x \leq y$
(4)	135°	$y > 0$ and $x \geq y$

Suppose we had a neighborhood for a pixel whose gradient had been quantized to 45°, as shown in Figure 9.30. The dashed arrow indicates the opposite direction to the current gradient. In this figure, both magnitudes at the ends of the arrows are smaller than the center magnitude, so we keep the center pixel as an edge pixel. If, however, one of those two values was *greater* than 150, we would mark the central pixel for deletion.

Figure 9.30: Quantizing in non-maximum suppression



After performing non-maximum suppression, we can threshold to obtain a binary edge image. Canny proposed, rather than using a single threshold, a technique called *hysteresis thresholding*, which uses two threshold values: a low value t_L, and a high value t_H. Any

pixel with a value greater than t_H is assumed to be an edge pixel. Also, any pixel with a value p where $t_L \leq p \leq t_H$ and which is adjacent to an edge pixel is also considered to be an edge pixel.

The Canny edge detector is implemented by the `canny` option of the `edge` function in MATLAB and Octave; and in Python by the `canny` method in the `filter` module of `skimage`. by the `canny` option of the `edge` function.

MATLAB and Octave differ slightly in their parameters, but the results are very similar. With no choosing of parameters, and just

```
>> edge(s,'canny')
```

MATLAB/Octave

the result is shown in Figure 9.31(a). Python's syntax is

```
In :  fl.canny(s)
```

Python

and the result is shown in Figure 9.31(b) Two other results with different thresholds and

Gaussian spreads are given in Figure 9.32. The higher we make the upper threshold, the less edges will be shown. We can also vary the standard deviation of the original Gaussian filter.

Figure 9.31: Canny edge detection

(a) MATLAB/Octave canny

(b) Python canny

The Canny edge detector is the most complex of the edge detectors we have discussed; however, it is not the last word in edge detectors. A good account of edge detectors is given by Parker [32], and an interesting account of some advanced edge detection techniques can be found in Heath et al. [16].

9.10 Corner Detection

Second only to edges as means for identifying and measuring objects in images are corners, which may be loosely defined as a place where there are two edges in significantly different directions. An edge may have a very slight bend in it—a matter of only a few degrees—but that won't qualify as a corner.

Figure 9.32: Canny edge detection with different thresholds

$t_L = 2, t_H = 10, \sigma = 0.1$

$t_L = 0, t_H = 4, \sigma = 2$

As with edges, there are many different corner detectors; we will look at just two.

Moravec Corner Detection

This is one of the earliest and simplest detectors: fundamentally, a corner is identified as a pixel whose neighborhood is significantly different from each other local neighborhood. It can be described as a sequence of steps:

1. We suppose we are dealing with a square "window" (like a mask), of odd dimensions, placed over the current pixel p; suppose that W is the array of neighboring pixels surrounding p.
2. Move this mask one pixel in each of the eight directions from p.
3. For each shift $s = (i, j)$ compute the sum of squared differences: $I_s = \sum (W(x,y) - W_s(x,y))^2$ This sum is called the *Intensity Variation*.
4. Compute the minimum M of all the I_s values.

If p is in a region with not much difference in any direction, the values I_s would be expected to be small. A corner is a place where M is maximized.

To implement this detector in MATLAB or Octave is fairly straightforward. Pad the image by zeros, and on this larger image plane move it in each direction, compute the squared distances, and sum them with a

linear filter. For example:

```
>> s = im2double(imread('stairs.png'));
>> [r,c] = size(s);
>> dirs = [1 1;1 0;-1 0;0 1;0 -1;-1 -1;1 -1;-1 1];
>> sp = padarray(s,[1,1]);
>> z = zeros(r+2,c+2,8);
>> w = zeros(r+2,c+2,8);
>> for i = 1:8
>    z(2+dirs(i,1):1+dirs(i,1)+r,2+dirs(i,2):1+dirs(i,2)+c,:) = s;
>    w(:,:,i) = imfilter((z(:,:,i)-sp).^2,ones(3,3));
>  end
>> corners = min(w,3);
```

MATLAB/Octave

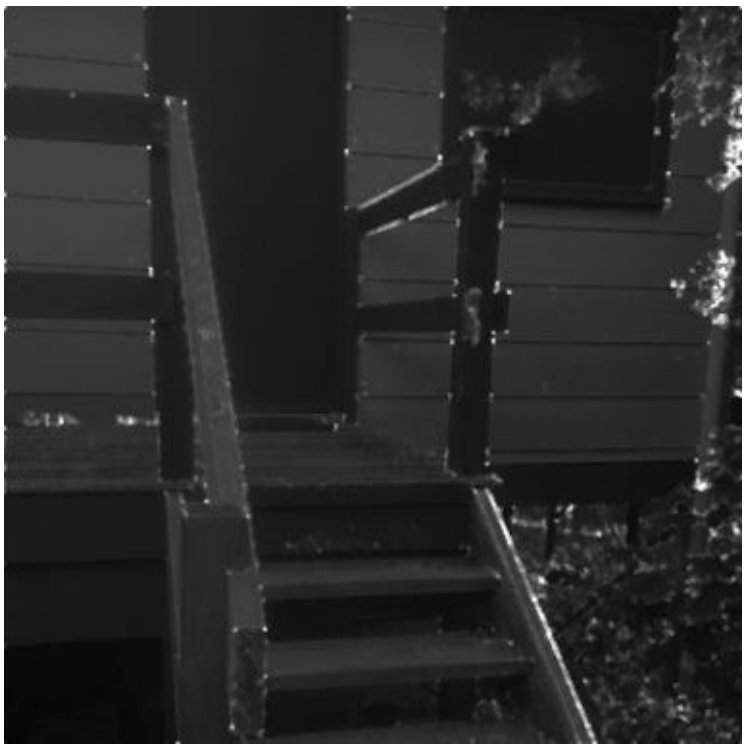
To show the corners, create an image that includes a darkened version of the original added to a brightened version of the corners array:

```
>> imshow(s/4+4*corners(2:r+1,2:c+1))
```

MATLAB/Octave

and the result is shown in Figure 9.33.

Figure 9.33: Moravec corner detection



Python has a `corner_moravec` method in `skimage.feature`; however, this implementation produces different results to the method given above. For example, suppose we take a simple image with just one corner, as shown in Figure 9.10. The MATLAB/Octave result and the Python results are shown in Figure 9.35. It can be seen that the Python results would require further processing so as not to incorrectly

classify internal pixels of a region as corner points. Our very naive implementation above also incorrectly classifies points at the edge of the image as corner points, but these can be easily removed.

The Moravec algorithm is simple and fast, but its main drawback is that it cannot cope with edges not in one of the eight directions. If there is an edge at an angle, for example, of 22.5°, then one of the local intensity variations will be large, and will incorrectly identify a corner. The Moravec detector is thus *non-isotropic* as it will produce different results for different angles.

Figure 9.34: A simple image with one corner

```
50  50  50  50  50  50  50  50  50  50
50  50  50  50  50  50  50  50  50  50
50  50  50  50  50  50  50  50  50  50
50  50  50  50  50  50  50  50  50  50
50  50  50  50 150 150 150 150 150 150
50  50  50  50 150 150 150 150 150 150
50  50  50  50 150 150 150 150 150 150
50  50  50  50 150 150 150 150 150 150
50  50  50  50 150 150 150 150 150 150
50  50  50  50 150 150 150 150 150 150
```

Figure 9.35: Moravec detection in MATLAB, Octave, and Python compared

```
5  0  0  0  0  0  0  0  0  5
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0 10 10  0  0  0  0 20
0  0  0 10 20  0  0  0  0 20
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
5  0  0 20 20  0  0  0  0 45
```

Corners in MATLAB/Octave

```
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0 10 20 30 30 30  0  0
0  0  0 20 30 30 30 30  0  0
0  0  0 30 30  0  0  0  0  0
0  0  0 30 30  0  0  0  0  0
0  0  0 30 30  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
```

Corners in Python

The Harris-Stephens Detector

This corner detector is often called simply the Harris corner detector, and it was designed to alleviate some of the problems with Moravec's method. It is based in part on using the first order Taylor approximation for a function of two variables:

$$f(x+h, y+k) \approx f(x,y) + h\frac{\partial f}{\partial x}(x,y) + k\frac{\partial f}{\partial y}(x,y).$$

Suppose the derivatives in the *x* and *y* directions are computed using a linear filter.

As with the Moravec detector, the Harris detector computes the squared differences with a neighborhood and its shift. For a shift (*s, t*) then the sum of squares, over a mask *K* is:

$$\sum_{(u,v)\in K} (I(u+s, v+t) - I(u,v))^2.$$

Using the Taylor approximation above, this can be written as

$$\begin{aligned}
&\sum_{(u,v)\in K} (I(u,v) + sI_x(u,v) + tI_y(u,v) - I(u,v))^2 \\
&= \sum_{(u,v)\in K} (sI_x(u,v) + tI_y(u,v))^2 \\
&= \sum_{(u,v)\in K} (s^2 I_x^2 + 2stI_xI_y + t^2 I_y) \\
&= \sum_{(u,v)\in K} \begin{bmatrix} s & t \end{bmatrix} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \begin{bmatrix} s//t \end{bmatrix}
\end{aligned}$$

The Harris method looks at the matrix

$$H = \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

For a given (*s, t*), the expression

$$s^2 I_x^2 + 2stI_xI_y + t^2 I_y = c$$

for a constant *c* has the shape of an ellipse. The length of its axes *a* and *b* and their direction are given by the *eigenvalues* λ_1 and λ_2 and *eigenvectors* $\mathbf{v}_1$ and $\mathbf{v}_2$ of the matrix *H* as shown in Figure 9.36. These satisfy

$$H\mathbf{v}_i = \lambda_i \mathbf{v}_i.$$

In particular, it can be shown that

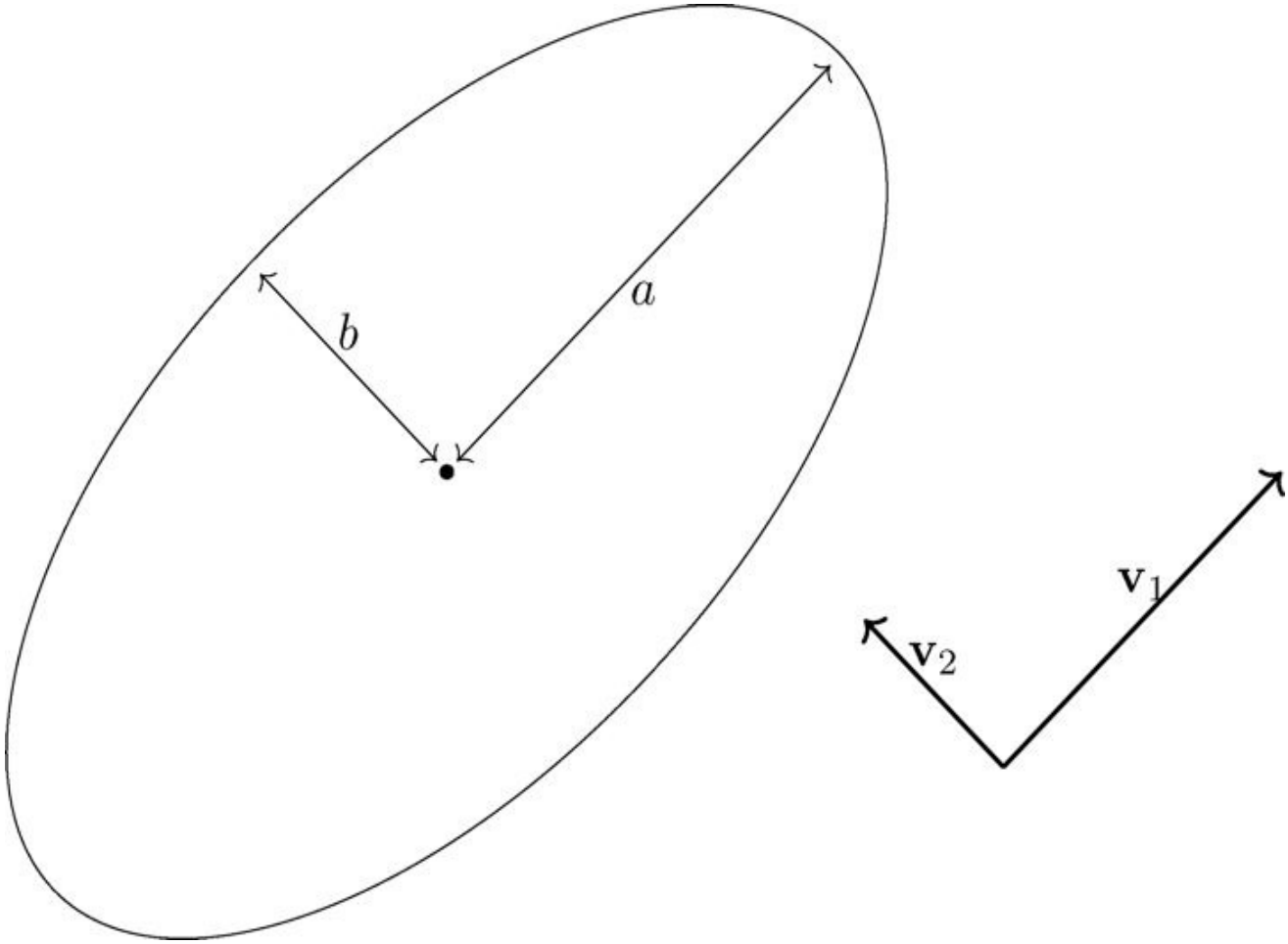
$$a = (\lambda_1)^{-1/2}, b = (\lambda_2)^{-1/2}.$$

Since the squares a^2 and b^2 of the axes lengths are inversely proportional to the eigenvalues, it follows that the *slowest* change of intensity is in the direction of the *largest* eigenvalue, and the fastest change in the direction of the smallest.

Given the eigenvalues then, we distinguish three cases:

1. Both are large. This corresponds to not much change in any direction, or a low frequency component of the image.

Figure 9.36: The ellipse associated with the Harris matrix *H*



2. One is large and the other is small. This corresponds to an edge in that direction.
3. Both are small. This corresponds to change in both directions, so it may be considered a corner.

For a general symmetric matrix

$$M = \begin{bmatrix} x & y \\ y & z \end{bmatrix}$$

the eigenvalues can be found to be the solutions of the equation

$$\lambda^2 - (x + z)\lambda + (xz - y^2) = 0$$

where $x + z = \mathrm{Tr}(M)$, the *trace* of M, and $xz - y^2 = \det(M)$, the *determinant* of M. However, the solution of this equation will involve square roots, which are computationally expensive operations. Harris and Stephens suggested a simpler alternative: for a previously chosen value of k, compute

$$R = \det(M) - k(\mathrm{Tr}(M))^2.$$

Corners will correspond to large values of R.

The mathematics may seem complicated, but in fact the implementation is very simple. It consists of only a few steps:

1. For the given image I, compute the edge images I_x and I_y in the x and y directions. This can be done using standard linear filters.
2. Over a given mask, compute the sums $S = \sum I_x^2, \quad T = \sum I_x I_y, \quad U = \sum I_y^2.$
3. Compute $R = (SU - T^2) - k(S + U)^2$.

To ensure isotropy, and to make the corner detector more robust in the presence of noise, Harris and Stephens recommended that in Step 2, instead of merely adding values over the mask, that a Gaussian filter be applied instead. Thus, the following variant would be used:

2'. Over a given mask with a Gaussian filter G, compute the convolutions

$$S = (I_x^2) * G, \quad T = (I_x I_y) * G, \quad U = (I_y^2) * G.$$

MATLAB contains a Harris-Stephens implementation in the `corner` function, and Python with the `corner_harris` method in the `skimage.feature` module.

So for Octave, here is how we might perform corner detection on the stairs image `s`, which we assume to be of data type `double`. First compute the derivatives (edges) and their products:

```
>> Ix = imfilter(s,[1 0 -1]);
>> Iy = imfilter(s,[1; 0; -1]);
>> Ixx = sx.*sx;
>> Ixy = sx.*sy;
>> Iyy = sy.*sy;
```

Octave

The next step is to convolve these either with a 3 × 3 sum filter, or its variant, a Gaussian filter. Take the first, simple option:

```
>> S = imfilter(Ixx,ones(3))
>> T = imfilter(Ixy,ones(3))
>> U = imfilter(Iyy,ones(3))
```

Octave

Now to compute the corner image, with a standard value $k = 0.05$.

```
>> k = 0.05
>> R = (S.*U-T.^2)-k*(S+U).^2;
```

Octave

This image R can be used directly by simple thresholding:

```
>> imshow(s/4+(R>k)*1)
```

Octave

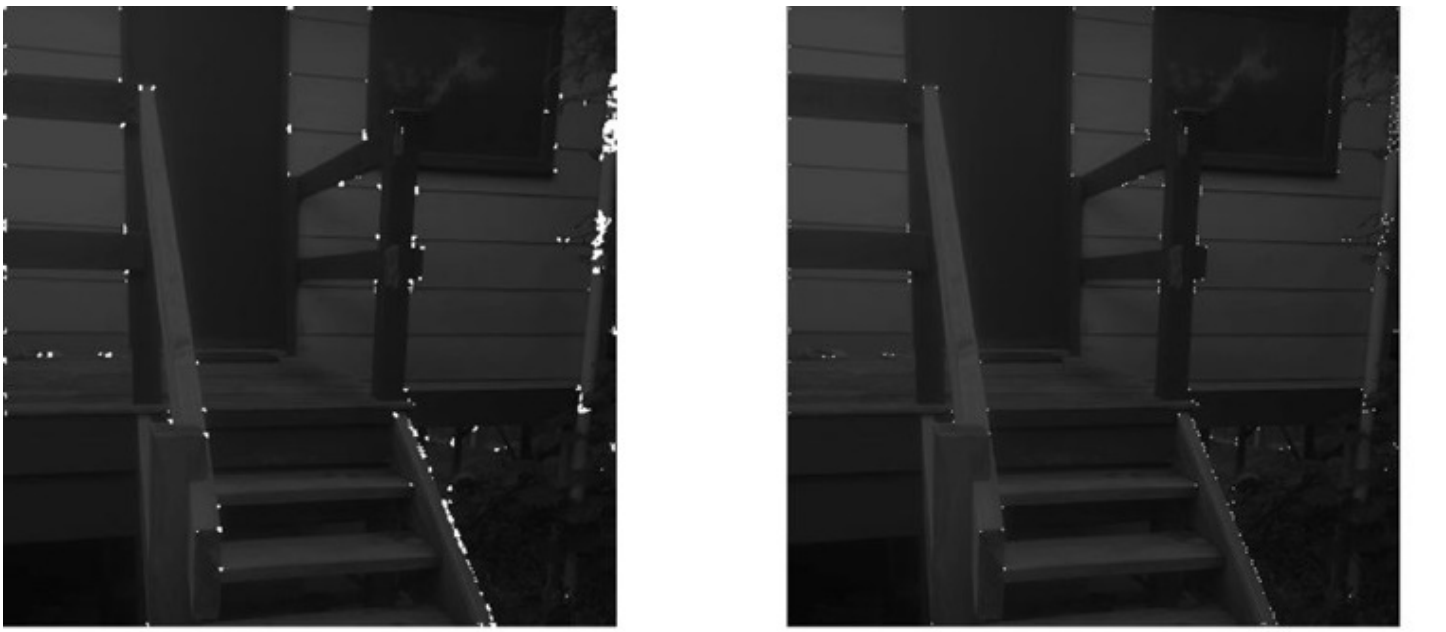
and this is shown in Figure 9.37(a). However, non-maximum suppression, as used in the Canny edge detector, can be used to ensure that any single corner is represented by only one pixel. In this case, simply remove pixels which are not the largest in their neighborhood:

```
>> Rmax = ordfilt2(R,9,ones(3));
>> Rs = (Rmax==R).*(R>k)*1;
>> imshow(s/4+Rs)
```

Octave

and this is shown in Figure 9.37(b).

Figure 9.37: Harris-Stephens corner detection



(a) With thresholding (b) With non-maximum suppression

9.11 The Hough and Radon Transforms

These two transforms can be used to find lines in an image. Although they seem very different on initial explanation, they provide very similar information. The Hough transform is possibly more efficient and faster; whereas the Radon transform is mathematically better behaved—in its continuous form the Radon transform is fully invertible, and in its discrete form (as in images) there are many different approximations to the inverse which can nearly completely recover the initial image.

Hough transform. First note that on the Cartesian plane a general line can be represented in the form

$$x\cos\theta + y\sin\theta = r$$

where r is the perpendicular distance to the origin, and θ is the angle of that perpendicular to the positive x axis, as shown in Figure 9.38. Note that the vector $\overrightarrow{OA}$ has direction ⟨cos θ, sin θ⟩ and the vector $\overrightarrow{AX} = \langle x - r\cos\theta, y - r\sin\theta\rangle$. Since these vectors are perpendicular, their dot product is zero, which means that

$$\cos\theta(x - r\cos\theta) + \sin\theta(y - r\sin\theta) = 0.$$

This can be rewritten as

$$x\cos\theta - r\cos^2\theta + y\sin\theta - r\sin^2\theta = 0$$

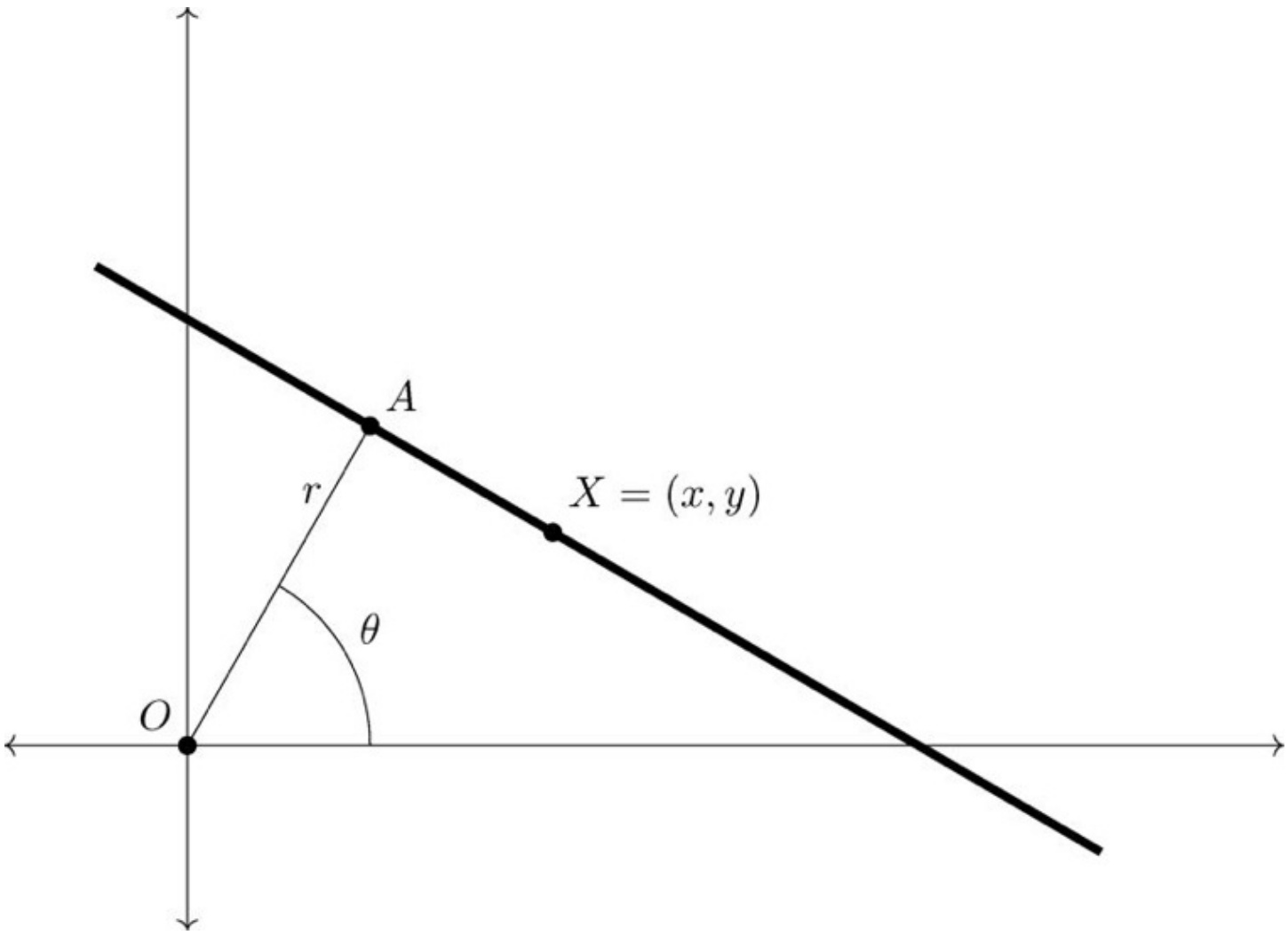
or

$$x\cos\theta + y\sin\theta = r.$$

This parameterization includes perpendicular lines, which cannot be represented in the more usual $y = ax + b$ form.

On a binary image, for every foreground point (x, y) consider the values of $x \cos \theta + y \sin \theta$ for a set of values of θ. One standard is to choose $0 \leq \theta \leq 179°$ in steps of 1°. It will be found that many different values of (x, y) will give rise to the same $(r\theta)$ pairs. The pair (r, θ) corresponding to the greatest number of (x, y) points corresponds to the "strongest line"; that is, the line with the greatest number of points in the image.

Figure 9.38: A line and its parameters



For example, consider the small image with just six foreground points as shown in Figure 9.39, and with the angle set $\theta = 0,45,90.135$.

The values of $x \cos \theta + y \sin \theta$ are:

x	y	0	45	90	135
1	2	1	2.12132	2	0.70711
3	1	3	2.82843	1	−1.41421
2	2	2	2.82843	2	0
2	3	2	3.53553	3	0.70711
3	4	3	4.94975	4	0.70711
4	4	4	5.65685	4	0

Notice that the value $r = 0.7071$ (that is,

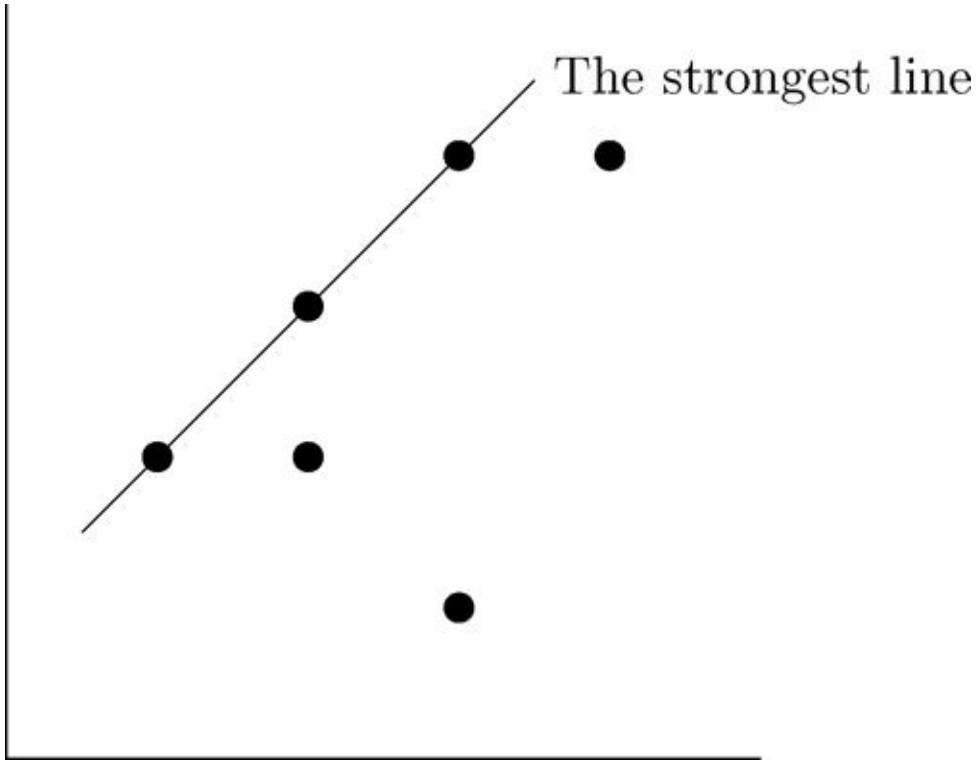
	$r =$	−1.41	0	0.71	1	2	2.12	2.83	3	3.54	4	4.95	5.66
$\theta =$	0	0	0	0	1	2	0	0	2	0	1	0	0
	45	0	0	0	0	0	1	2	0	1	0	1	1
	90	0	0	0	1	2	0	0	1	0	2	0	0
	135	1	2	3	0	0	0	0	0	0	0	0	0

) occurs most often, in the last column. This means that the strongest line has values $(r, \theta) = (1/\sqrt{2}, 135)$ which has standard Cartesian form $y = x + 1$. Value (x, y) are transformed by this procedure into (r, θ) values. We can thus set up an (r, θ) array where each value corresponds to the number of times it appears in the initial computation. For the previous table, the corresponding array is

	$r =$	−1.41	0	0.71	1	2	2.12	2.83	3	3.54	4	4.95	5.66
$\theta =$	0	0	0	0	1	2	0	0	2	0	1	0	0
	45	0	0	0	0	0	1	2	0	1	0	1	1
	90	0	0	0	1	2	0	0	1	0	2	0	0
	135	1	2	3	0	0	0	0	0	0	0	0	0

This array is known as the *accumulator array*, and its largest values correspond to the strongest line. The *Hough transform*1 thus transforms a Cartesian array to an (r, θ) array, from which the lines in the image can be determined.

Figure 9.39: A simple Hough example



A quick-and-dirty, if not particularly efficient, Hough transform can be implemented by following the description above, starting with a binary image:

```
>> e = edge(c,'canny');
>> [x,y] = find(e==1);
>> th = 0:179; cos_th=cosd(th);sin_th = sind(th);
>> rtable = floor(x*cos_th+y*sin_th);
```

MATLAB/Octave

At this stage we have a table of value of r. Note that in the last command matrix products are being computed: if there are N foreground pixels in the binary image, then each product is formed from an N x 1 matrix and an 1 × 180 matrix, resulting in an N x 180 matrix. To form the accumulator array, find all the different values in the table:

```
>> u = unique(rtable);
>> N = length(u);
```

MATLAB/Octave

and then count them:

```
>> for j = 1:180,
>>    for i = 1:N,
>>      hough(i,j) = length(find(rtable(:,j)==u(i)));
>>    end;
>>  end
```

MATLAB/Octave

In Python, we need a little more care to ensure that the correct variable type is being used–list or array:

```
In :  e = fl.canny(c,2,5)
In :  x,y = np.nonzero(e)
In :  xt = np.array([x]).T
In :  yt = np.array([y]).T
In :  th = linspace(0,np.pi,180)
In :  cos_th = np.array([np.cos(th)])
In :  sin_th = np.array([np.sin(th)])
In :  rtable = np.floor(xt.dot(cos_th)+yt.dot(sin_th))
```

Python

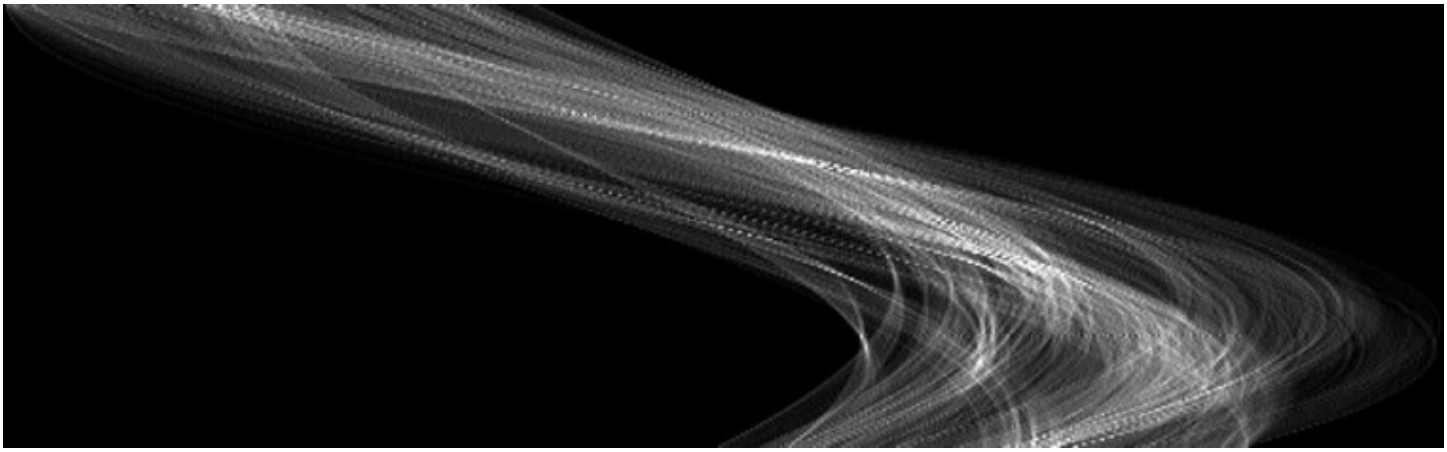
Recall that in Python a single asterisk performs element-by-element multiplication; standard matrix products are performed with the `dot` method applied to a matrix. The `T` method is matrix transposition. Now for the accumulator array, using the `unique` function, which lists the unique elements in an array:

```
In :  u = np.unique(rtable)
In :  N = len(u)
In :  hough = np.zeros((N,180))
In :  for j in range(180):
...:     for i in range(N):
...:         hough[i,j] = np.where(rtable[:,j]==u[i])[0].shape[0]
...:
```

Python

This final array is the Hough transform of the initial binary image; it is shown (rotated 90° to fit) in Figure 9.40. The maximum value in this transform will correspond to the strongest line in the image. Obtaining this line will be discussed at the end of the next section.

Figure 9.40: A Hough transform

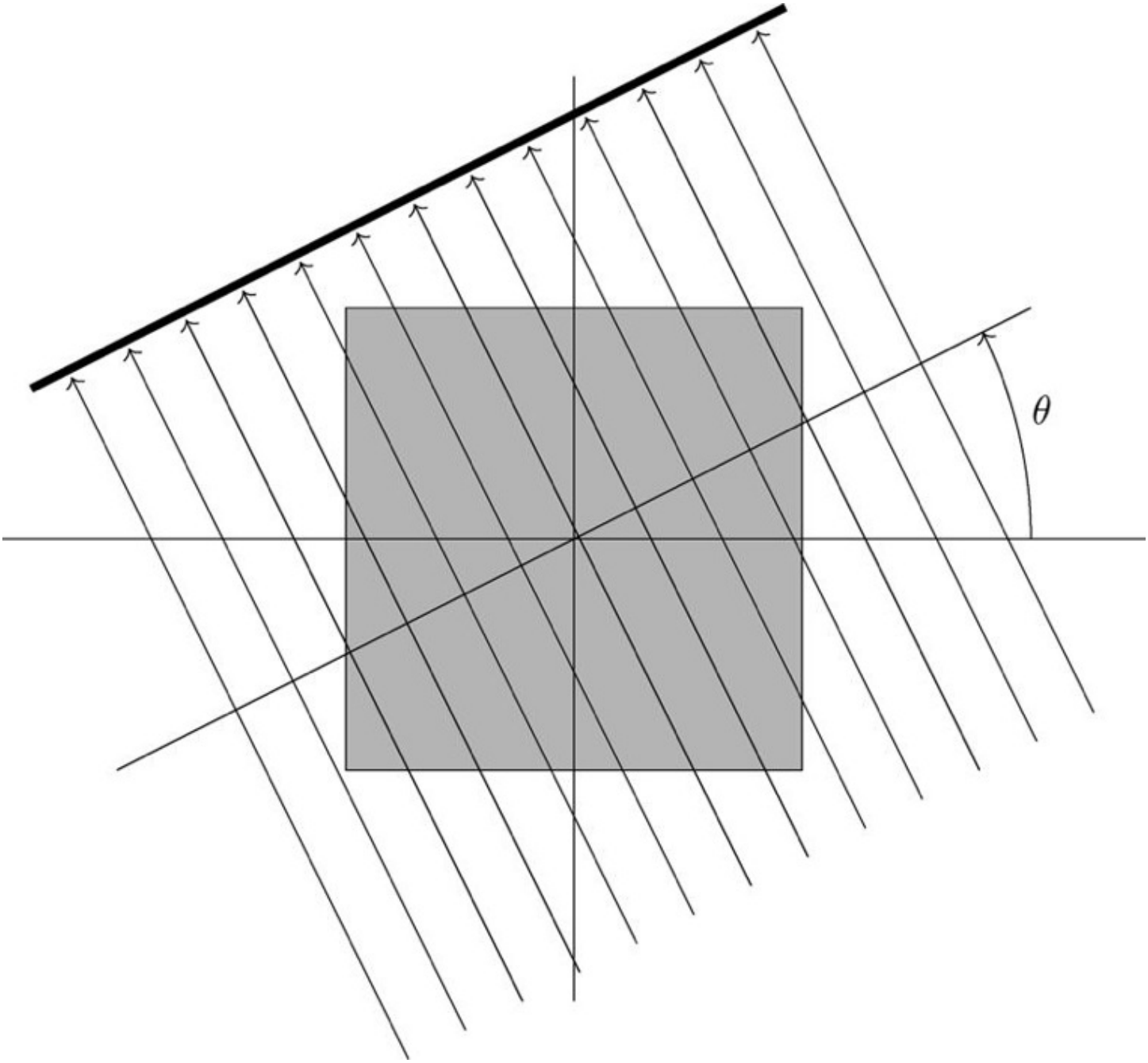


The Radon transform. Whereas the Hough transform works point by point, the Radon transform works across the entire image. Given an angle θ, consider all the lines at that angle which pass across the image, as shown in Figure 9.41.

As the lines pass through the image, add up the pixel values along each line, to accumulate in a new line. The *Radon transform* is the array whose rows are the values along these new lines.

The mathematical elegance of the Radon transform is partly due to a result called the *Fourier slice theorem*, which claims that the Fourier transform of a line for an angle θ is equal to the line (a "slice") through the Fourier transform at that angle. To see this, take an image and its Fourier and Radon transforms. In MATLAB/Octave:

Figure 9.41: The Radon transform




```
>> c = imread('cameraman.png');
>> cf = fftshift(fft2(c));
>> cr = radon(c);
```

MATLAB/Octave

and in Python we can use the `radon` function from the `transform` module of `skimage`:

```
In :  from numpy.fft import fft, fft2, fftshift
In :  c = io.imread('cameraman.png')
In :  cf = fftshift(fft2(c))
In :  cr = tr.radon(float64(c),range(180))
```

Python

Pick an angle, say θ = 30. Given that the first row of the Radon transform corresponds to θ = 0, the thirty-first row will correspond to θ = 30. Then the Fourier transform of that row can be obtained, scaled, and plotted with

```
>> c30 = fftshift(fft(cr(:,31)));
>> c30l = log(1+abs(c30));
>> plot(c30l)
```

MATLAB/Octave

or with

```
In :  c30 = fftshift(fft(cr[:,30]))
In :  c30l = np.log(1+np.abs(c30))
In :  plt.plot(c30l)
```

Python

To obtain the Fourier slice, the easiest way is to rotate the transform of the image and then read off the center row:

```
>> cl = log(1+abs(cf));
>> cl30 = imrotate(cl,30);
>> [rs,cs] = size(cl30);
>> plot(cl30(floor(rs/2),:))
```

MATLAB/Octave

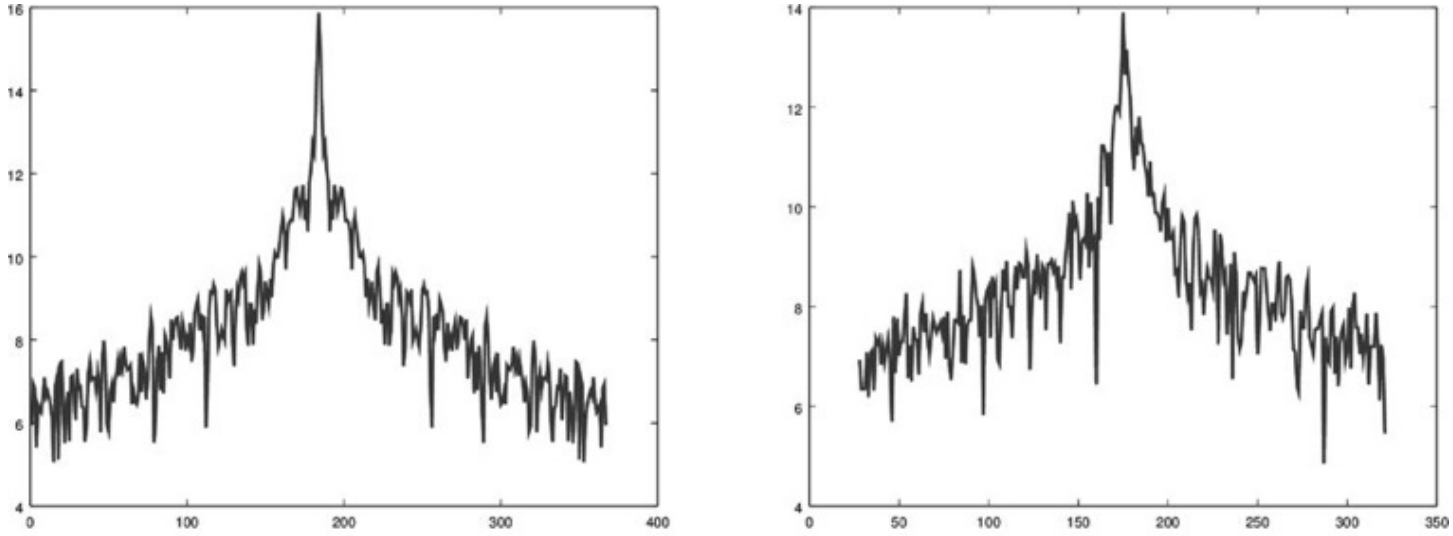
or with

```
In :  cl = np.log(1+np.abs(cf))
In :  cl30 = tr.rotate(cl,30,resize="True",order=3)
In :  rs = cl30.shape[0]
In :  plt.plot(cl30[rs/2,''])
```

Python

The results are shown in Figure 9.42. The plots are remarkably similar, with differences accounted for by rounding errors and interpolation when rotating. This means, at least in theory, a Radon transform can be reversed. For each projection in the transform corresponding to an angle θ, take its Fourier transform and add it into an array as a line at angle θ. The final array may be consider as the Fourier transform of the initial image, and so inverting it will provide the image.

Figure 9.42: The Fourier slice theorem



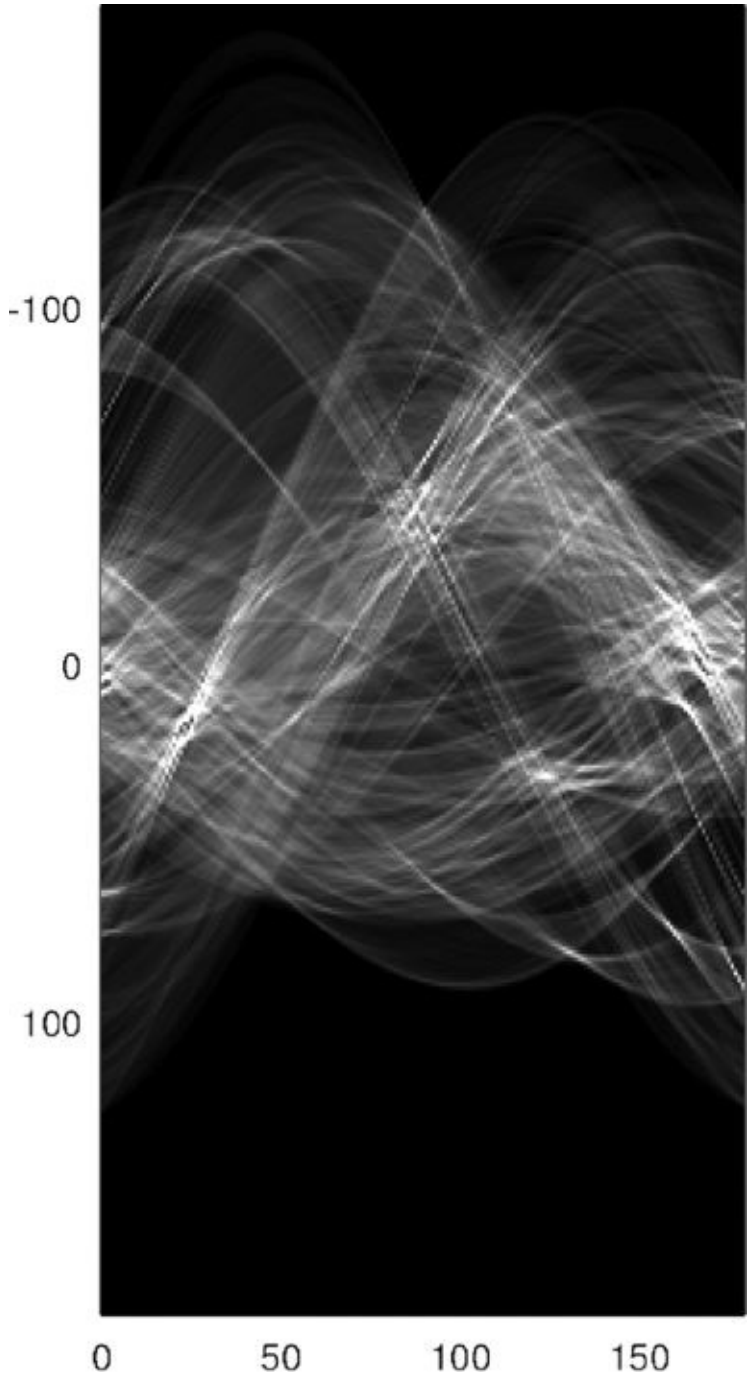
(a) The FFT of a single projection

(b) A slice of the 2D FFT

With continuous functions the Radon transform—considered an integral transform—is indeed invertible. For images, being discrete objects, the transform is not completely invertible. However, it is possible to obtain a very good approximation of the image by using a similar method. An important consideration is preventing the DC coefficient from overpowering the rest of the result; this is usually managed by some filtering.

Finding lines using the Radon transform. To find the strongest lines, first create a binary edge image, and compute its Radon transform. This can be done in MATLAB/Octave as

Figure 9.43: The Radon transform of the edges of an image




```
>> e  = edge(c,'canny');
>> angles = 0:179;
>> [rad,x] = radon(e,angles);
```

MATLAB/Octave

or in Python as

```
In :  e = fl.canny(c)
In :  rad = tr.radon(e)
```

Python

The extra output parameter `x` in the MATLAB/Octave implementation gives the x-axis of the rotated coordinate. The result can be displayed using `imagesc`, which displays the image as a plot:

```
>> imagesc(angles,x,rad),colormap(gray(256))
```

MATLAB/Octave

and is shown in Figure 9.43. The strongest line will be the point in the transform of greatest value:

```
>> [r,theta] = find(rad==max(rad(:)))
r =  200
theta =  24
```

MATLAB/Octave

The corresponding x value can be found using the `x` array:

```
>> x(r)
ans =
     16
```

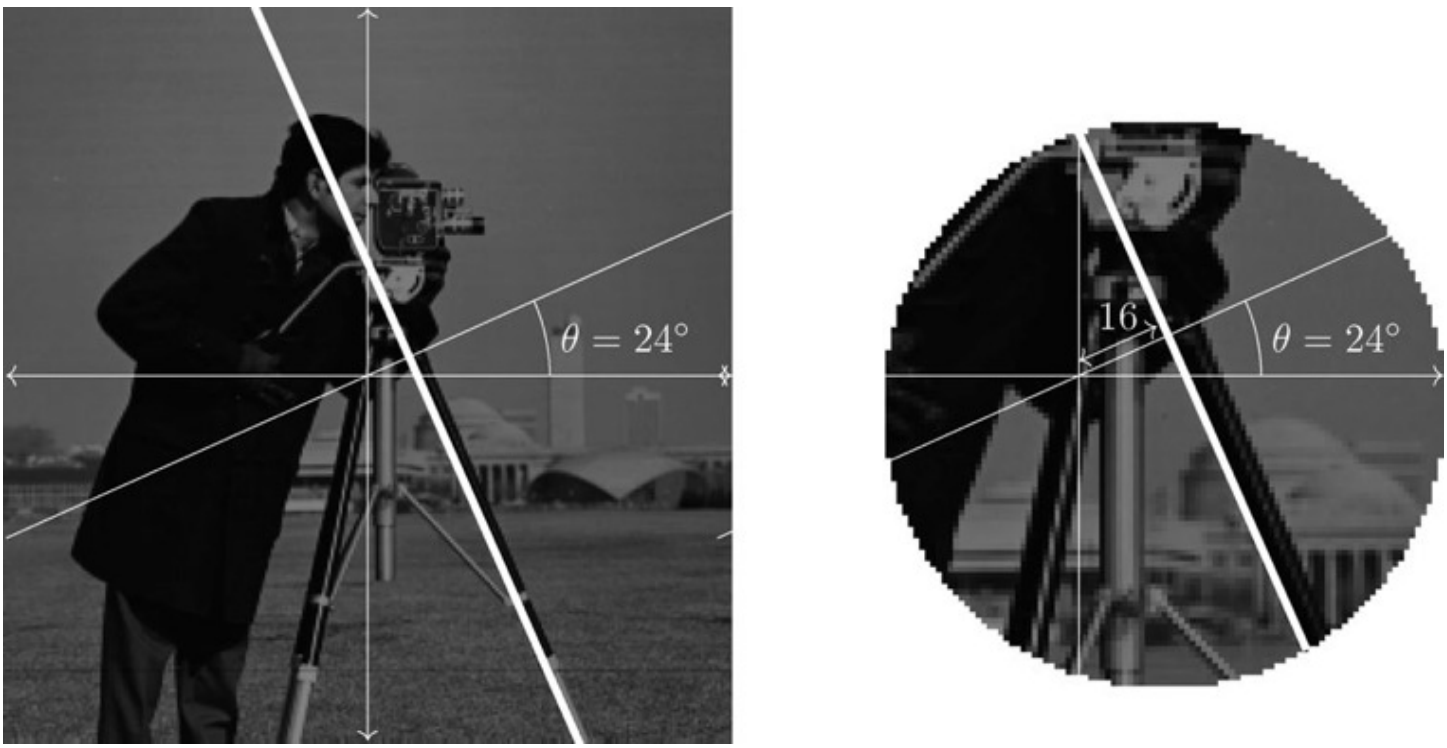
MATLAB/Octave

If the image is centered about the origin, then the strongest line will be

$$x\cos(24) + y\sin(24) = 16$$

and this is shown superimposed on the original image in Figure 9.44, along with an enlarged view of the center.

Figure 9.44: Finding a line in an image



Exercises

Thresholding

1. Suppose you thresholded an image at value t_1, and thresholded the result at value t_2. Describe the result if

   (a) $t_1 > t_2$
   (b) $t_1 < t_2$

2. Create a simple image with

```
>> [x,y] = meshgrid(1:256,1:256);
>> z = sqrt((x-128).^2+(y-128).^2);
>> z2 = 1-mat2gray(z);
```

MATLAB/Octave

or with

```
In :  x,y = np.mgrid[0:256,0:256].astype(float)
In :  z = np.sqrt((x-128)**2+(y-128)**2)
In :  z2 = z.max()-z
```

Python

Threshold `z2` at different values, and comment on the results. What happens to the amount of white as the threshold value increases? Can you state and prove a general result?

3. Repeat the above question, but with the image `cameraman.png`.
4. Can you create a small image that produces an "X" shape when thresholded at one level, and a cross shape "+" when thresholded at another level? If not, why not?
5. Superimpose the image `nicework.png` onto the image `cameraman.png`. You can do this with:

```
>> n = im2uint8(imread('nicework.png'));
>> c = imread('cameraman.png');
>> m = imlincomb(0.5,c,0.5,n);
```

MATLAB/Octave

or with

```
In :  c = io.imread('cameraman.png')
In :  n = io.imread('nicework.png')
In :  m = c//2+n//2
```

Python

(Look up `imlincomb` to see what it does.) Can you threshold this new image `m` to isolate the text?

6. Try the same problem as above, but define `m` as:

```
>> m  = c.*(n==0);
```

7. Create a version of the circles image with

```
>> t = imread('circles.png');
>> [x,y] = meshgrid(1:256,1:256);
>> t2 = double(t).*(x+y)/512;
>> t3 = im2uint8(t2);
```

MATLAB/Octave

or

201	195	203	203	199	200	204	190	198	203
201	204	209	197	210	202	205	195	202	199
205	198	46	60	53	37	50	51	194	205
208	203	54	50	51	50	55	48	193	194
200	193	50	56	42	53	55	49	196	211
200	198	203	49	51	60	51	205	207	198
205	196	202	53	52	34	46	202	199	193
199	202	194	47	51	55	48	191	190	197
194	206	198	212	195	196	204	204	199	200
201	189	203	200	191	196	207	203	193	204

Attempt to threshold the image `t3` to obtain the circles alone, using adaptive thresholding (and if you are using MATLAB or Octave, the `blkproc` function). What sized blocks produce the best result? **Edge Detection**

- 8. Enter the following matrix using either

```
>> im = 200*ones(10,10);im(3:5,3:8)=50;im(6:8,4:7)=50;
>> im = im + round(9*randn(10,10))
```

MATLAB/Octave

or

```
In :  im = 200*np.ones((10,10))
In :  im[2:5,2:8]=50;im[5:8,3:7]=50
In :  im = (im + np.round(8*sc.randn(10,10))).astype('uint8')
```

Python

This will create something like this:

201	195	203	203	199	200	204	190	198	203
201	204	209	197	210	202	205	195	202	199
205	198	46	60	53	37	50	51	194	205
208	203	54	50	51	50	55	48	193	194
200	193	50	56	42	53	55	49	196	211
200	198	203	49	51	60	51	205	207	198
205	196	202	53	52	34	46	202	199	193
199	202	194	47	51	55	48	191	190	197
194	206	198	212	195	196	204	204	199	200
201	189	203	200	191	196	207	203	193	204

and use the appropriate filter to apply each of the Roberts, Prewitt, Sobel, Laplacian, and zero-crossing edge-finding methods to the image. In the case of applying two filters (such as with Roberts, Prewitt, or Sobel) apply each filter separately, and join the results. Apply thresholding if necessary to obtain a binary image showing only the edges. Which method seems to produce the best results?

- 9. If you are using MATLAB or Octave, apply the `edge` function with each of its possible parameters in turns to the array above. Which method seems to produce the best results?
- 10. Open up the image `cameraman.png` in MATLAB, and apply each of the following edge-finding techniques in turn:

    (a) Roberts
    (b) Prewitt
    (c) Sobel
    (d) Laplacian
    (e) Zero-crossings of a Laplacian
    (f) The Marr-Hildreth method
    (g) Canny

Which seems to you to provide the best looking result?

- 11. Repeat the above exercise, but use the image `arch.png`.
- 12. Obtain a grayscale flower image with:

```
ir = imread('iris.png');
i = rgb2gray(ir);
```

MATLAB/Octave

or

```
In :  ir = io.imread('iris.png')
In :  i = co.rgb2gray(ir)
```

Python

Now repeat Question 10.

- 13. Pick a grayscale image, and add some noise to it using the commands introduced in Section 8.2. Create two images: `c1` corrupted with salt and pepper noise, and `c2` corrupted with Gaussian noise. Now apply the edge finding techniques to each of the "noisy" images `c1` and `c2`. Which technique seems to give

    (a) The best results in the presence of noise?
    (b) The worst results in the presence of noise?

The Hough Transform

- 14. Write the lines $y = x - 2$, $y = 1 - x/2$ in (r, θ) form.
- 15. Use the Hough transform to detect the strongest line in the binary image shown below. Use the form $x \cos\theta + y \sin\theta = r$ with θ in steps of 45° from −45° to 90° and place the results in an accumulator array.

y \ x	−3	−2	−1	0	1	2	3
−3	0	0	0	0	0	1	0
−2	0	0	0	0	0	0	0
−1	0	1	0	1	0	1	0
0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0
2	1	0	0	0	0	1	0
3	0	0	0	0	0	0	0

- 16. Repeat the previous question with the images:

y \ x	−3	−2	−1	0	1	2	3
−3	0	0	0	0	1	0	0
−2	0	0	0	0	0	0	0
−1	0	0	1	0	0	0	1
0	0	1	0	0	1	0	0
1	1	0	0	0	0	0	1
2	0	0	0	0	1	0	0
3	0	0	0	1	1	0	0

y \ x	−3	−2	−1	0	1	2	3
−3	0	0	0	1	0	0	0
−2	1	0	0	0	1	0	0
−1	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
1	0	1	0	1	0	0	0
2	1	0	0	0	0	0	1
3	0	0	1	0	0	1	0

- 17. Find some more lines on the cameraman image, using the Radon transform method of implementing the Hough transform.
- 18. Read and display the image `stairs.png`.

    (a) Where does it appear that the "strongest" lines will be?
    (b) Plot the five strongest lines.

“Hough” is pronounced “Huff.”

apply the distance transform to approximate distances from the region containing 1's to all other pixels in the image, using the masks:

$$
\begin{array}{ll}
 & \begin{array}{ccc} & 1 & \\ 1 & 0 & 1 \end{array} \\
\text{(i)} & \begin{array}{ccc} & 1 & \\ 1 & 1 & 1 \end{array} \\
 & \begin{array}{ccc} 1 & 0 & 1 \\ 1 & 1 & 1 \end{array} \\
\text{(ii)} & \begin{array}{ccc} 4 & 3 & 4 \\ 3 & 0 & 3 \end{array} \\
 & \begin{array}{ccc} 4 & 3 & 4 \end{array} \\
\text{(iii)} & \begin{array}{ccccc} & 11 & & 11 & \\ 11 & 7 & 5 & 7 & 11 \\ & 5 & 0 & 5 & \\ 11 & 7 & 5 & 7 & 11 \end{array} \\
\text{(iv)} & \begin{array}{ccc} & 11 & \\ & 11 & \end{array}
\end{array}
$$

and applying any necessary scaling at the end.

- 28. Apply the distance transform and use it to find the skeleton in the images `circles2.png` and `nicework.png`.
- 29. Compare the result of the previous question with the results given by the Zhang-Suen and Guo-Hall methods. Which method produces the most visually appealing results? Which method seems fastest?

10 Mathematical Morphology

10.1 Introduction

Mathematical morphology, or *morphology* for short, is a branch of image processing that is particularly useful for analyzing shapes in images. We shall develop basic morphological tools for investigation of binary images, and then show how to extend these tools to grayscale images. MATLAB has many tools for binary morphology in the image processing toolbox; most of which can be used for grayscale morphology as well.

10.2 Basic Ideas

The theory of mathematical morphology can be developed in many different ways. We shall adopt one standard method which uses operations on sets of points. A very solid and detailed account can be found in Haralick and Shapiro [14].

Translation

Suppose that A is a set of pixels in a binary image, and $w = (x, y)$ is a particular coordinate point. Then A_w is the set A “translated” in direction (x, y). That is

$$A_x = \{(a, b) + (x, y) : (a, b) \in A\}.$$

For example, in Figure 10.1, A is the cross-shaped set and $w = (2, 2)$. The set A has been