Computer Vision (CE316 and CE866): Low-level Vision

Adrian F. Clark
⟨*alien@essex.ac.uk*⟩

We consider principled ways of determining how to threshold imagery into foreground and background, then ways of segmented isolated objects from the background and allocating them unique labels. Ways of describing shape are considered. Finally, the Hough transform is presented and it is shown how that can be used to determine the equations of straight lines in images.

Contents

List of Figures

1 Introduction

The techniques we have explored to date are useful for looking at and manipulating the content of images but none of them really *extract* information from them. We shall take some steps in this direction here; however, you should be aware that this is an area where much research effort into computer vision is currently being expended as no combination of the (many) techniques available is able to come anywhere close to approaching the flexibility and accuracy of the human visual system.

Bearing in mind that we have discussed ways of emphasising particular types of features, a natural first step is to consider the use of thresholding to identify useful features; so that is where we shall start. This leads into a consideration of how one would best set the threshold, and then what one can do when one has managed to segment a feature successfully. We shall conclude with a discussion of a technique specifically for finding the parameters that describe straight lines in images, the Hough transform.

2 Isolating regions by thresholding

If we are lucky, our image processing gives a histogram that has two peaks, one due to the background and the other to the objects of interest. Setting a threshold that neatly divides objects from background is then fairly straightforward, and a few minutes' thought should tell you that the best place for the threshold is between the two peaks. However, look at the histogram in Figure 1: the best place to put the threshold is less obvious, so it would be useful if there was a principled way of determining where to put it.

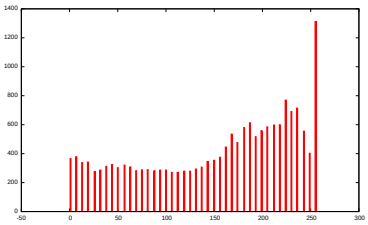


Figure 1: Typical histogram of an image. Where is the best place to set a threshold?

Perhaps surprisingly, there is a way to do this. The first thing to realise is that thresholding will be most effective if all the values of the pixels in an object have roughly the same value, and that is well-separated from the pixels in the background — that is essentially saying that we want two well-separated peaks in the histogram. We know a way of measuring the spread of a peak in a histogram: its variance. A peak formed from pixels having roughly the same value will have a small variance, so we want to choose a threshold that *minimizes* the variances of the object and background regions — these are so-called "within-class" variances.

Let us now consider this mathematically. We want to minimize the *within-class* variance

$$\sigma^2_{\text{within}} = \omega_B(t)\sigma^2_B(t) + \omega_F(t)\sigma^2_F(t) \tag{1}$$

where the weighting factor $\omega_B(t)$ is the probability of the background and $\omega_F(t)$ that of the foreground, and $\sigma^2_B(t)$ and $\sigma^2_F(t)$ the background and foreground variances respectively. All of these are functions of the threshold t. Nobuyuki Otsu was the first to show that (1) can be re-written in terms of the *between-class* variance:

$$\begin{aligned}\sigma^2_{\text{between}}(t) &= \sigma^2 - \sigma^2_{\text{within}}(t)\\ &= \omega_B(\mu_B-\mu)^2 + \omega_F(\mu_f-\mu)^2 \quad (\text{where } \mu = \omega_B\mu_b + \omega_F\mu_f)\\ &= \omega_B\omega_F(\mu_B-\mu_f)^2 \end{aligned} \tag{2}$$

where μ_B and μ_F are the background and foreground means respectively. So *minimising* the within-class variance is the equivalent to *maximising* the between-class variance, and the latter is much easier to do than the former.

The class probabilities *etc*. can conveniently be calculated from the histogram:

$$\omega_B(t) = \sum_{i=0}^{t} p(i) \qquad \mu_B(t) = \sum_{i=0}^{t} p(i)x(i)$$
$$\omega_F(t) = \sum_{i=t+1}^{G} p(i) \qquad \mu_F(t) = \sum_{i=t+1}^{G} p(i)x(i)$$

where G is the number of grey-level bins in the histogram, $x(i)$ is the middle of the i^{th} histogram bin and $p(i)$ the corresponding probability. This leads to a fairly straightforward algorithm

```
def otsu (im):
    "Determine the threshold by Otsu's method."
    # Initialization.
    nr, nc, nb = im.shape
    npixels = nr * nc * nb
    # Work out the histogram.
    ngreys = int (im.max () + 1.5)  # round the value
    hist = numpy.zeros (ngreys)
    for r in range (0, nr):
        for c in range (0, nc):
            for b in range (0, nb):
                v = int (im[r,c,b] + 0.5)  # round the value
                hist[v] += 1

    # Step over all the possible thresholds, calculating the between-class
    # variance at each step and working out its maximum as we go.
    sum = eve.sum (im)
    sumB = totB = threshold = max_var = 0
    for t in range (0, ngreys):
        sumB += hist[t]
        if sumB == 0: continue
        sumF = npixels - sumB
        if sumF == 0: break
        totB += t * hist[t]
        mB = totB / sumB
        mF = (sum - sumB) / sumF
        var = (sumB / sum) * (sumF / sum) * (mB - mF)**2
        if var > max_var:
            max_var = var
            threshold = t
    return threshold
```

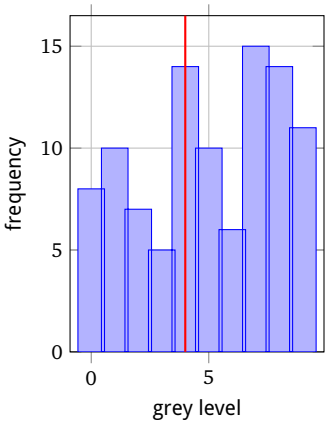
Note that this code yields the threshold correctly only if there is a single $\sigma^2_{\text{between}}$ that is maximum. Moreover, an implicit assumption is that a single, global threshold is enough to separate foreground from background, but the real world is not always that simple. On a positive note however, the approach can be extended to cope with multiple thresholds.

In practice, Otsu's method is found to work well when the foreground and background have roughly similar numbers of pixels; it is less effective when, for example, there are a few foreground pixels superimposed on a large, dark background.

To illustrate Otsu's method, consider the 10×10 image shown in Figure 2(a). The histogram of this image is plotted in Figure 2(b) and the threshold determined by the `otsu` routine above is shown with a vertical line at the value 4 — pixel values below this threshold represent background and those above foreground.

0	3	4	2	1	7	7	8	8	8
1	2	5	4	3	7	7	9	9	8
0	1	4	5	4	8	9	9	8	8
0	0	3	4	4	7	7	9	8	7
0	1	2	4	5	6	6	7	7	5
1	1	2	5	5	5	5	6	7	4
1	2	3	4	4	6	7	8	8	7
2	3	4	4	5	8	8	9	9	9
0	1	2	4	4	6	7	8	9	8
0	0	1	1	5	7	7	9	9	6

(a) 10×10-pixel image



(b) Corresponding histogram

Figure 2: Example 10-level image and its histogram, with the Otsu threshold marked

3 Region labelling

Otsu's method is able to help us partition or *segment* objects of interest from their surrounding background. The next step is typically to identify which pixels belong to which object — in other words, we want to assign a number to each region of the image as shown in Figure 3. With regions numbered in this way, all pixels that have the value (say) 3 belong to the same object, and that is a different object to pixels whose value is (say) 4. So how do we move from an image in which all regions have been segmented from the background to one in which the regions are numbered? This task is known as *region labelling*, *connected-component labelling* or *blob labelling*.

Studying Figure 3 for a little while should give you an idea of an algorithm: each pixel in a numbered region has a neighbouring pixel in the same region above, below or to the left or right of it, or the neighbouring pixel is background. Hence, we can scan across the image in the usual way, top to bottom and left to right, and at each pixel look at the neighbours immediately above it and immediately to its left; if either of these pixels is set, the centre pixel is also part of the same region.

0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	1	0	0	1	1
1	1	1	0	0	1	0	0	1	1
0	0	0	0	0	1	1	0	0	0
1	1	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	0	0	0
0	1	1	1	0	0	1	1	1	1
0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0	0	0

(a) Thresholded image

0	0	0	0	0	0	0	0	0	0
0	1	1	0	2	2	0	0	3	3
1	1	1	0	0	2	0	0	3	3
0	0	0	0	0	2	2	0	0	0
4	4	0	0	0	0	2	2	0	0
0	0	0	0	0	0	0	0	0	0
0	5	0	5	0	6	0	0	0	0
0	5	5	5	0	0	7	7	7	7
0	0	0	0	0	0	0	7	7	0
0	0	0	0	0	0	0	0	0	0

(b) Corresponding labelled regions

Figure 3: Distinct regions of an image and the result of region labelling. Regions 6 and 7 with four-connected processing will all be in region 6 if eight-connected processing is used.

This approach works well until a shape like region 5 in Figure 3 is encountered. The first pixel encountered is the top-left one and it will be correctly numbered as "5" but the next pixel of the object on the same line is not adjacent to it and would then be considered as part of a new region and numbered "6." It is not until the following line of the image is processed that these two pixels are known to be connected.

When situations such as these are encountered, the solution is to put an entry into an "equivalence table," recording that labels 5 and 6 are the same region. Then, when the entire image has been labelled, a second pass is made through it changing all pixels labelled with a 6 to a 5. (A little refinement of this approach is also normally made to renumber the regions so there are no 'missing' numbers.)

It is worth noting that the need for the label equivalence table can be dispensed with if one programs the algorithm recursively. While this works on small images, as soon as one starts to label images taken from a video camera, the program's stack (which is normally limited in size, to catch errant recursive programs) tends to overflow, leading to a run-time error even on a computer with a vast amount of memory. The recursive approach is also normally much slower than the equivalence table one.

The observant reader will be wondering, as the algorithm considers pixels to above and to the left, what happens to the first line and column of the image. The answer is that they are treated specially: the first pixel of the first line is set to zero, then the remainder of the first line is processed by considering only the neighbour to the left. When that line has been processed, the first column is processed by considering only the neighbour above. These special cases and the equivalence table processing together make the code to implement region labelling rather long — and the algorithm is inherently sequential. My EVE package provides two implementations of this algorithm: `eve.label_regions` uses an implementation in a compiled language that forms part of the `scipy` package, while `eve.label_regions_slow` is a pure Python alternative. The latter is *vastly* slower to execute than the former. A complete program that performs thresholding and region labelling using EVE looks like:

```
import sys, eve

# The first argument should be the threshold, and all others filenames.
if len (sys.argv) < 3:
    print >>sys.stderr, "Usage:", sys.argv[0], "<threshold> <file>..."
    sys.exit (1)
threshold = float (sys.argv[1])

# Loop over the images, processing each in turn.
for fn in sys.argv[2:]:
    im = eve.image (fn)
    mim = eve.mono (im)
    mask = eve.binarize (mim, threshold)
    mask, nregs = eve.label_regions (mask)
    print nregs, "regions found."
    eve.display (mask, stretch=True)
```

There is a roughly one-to-one mapping from EVE calls to OpenCV ones, though the latter are more tricky to invoke.

Careful examination of Figure 3 will show that there is a single pixel labelled "6" and that it is diagonally-connected to region 7. Some implementations of region labelling will look at the neighbours above, left and above-left of the pixel under consideration to find other pixels in the same region; this is known as 8-connected region labelling (the basic approach described above is 4-connected). If the image of Figure 3 was processed

with an 8-connected algorithms, all those pixels labelled "7" in the figure would instead be labelled "6."

4 Describing regions

Having isolated regions of interest and labelled them, the next problem is to think about what to do with them. For many vision problems, the underlying method is to identify these regions and classify them — to identify, say, whether a lesion on a person's skin is a mole or a melanoma, a dangerous form of skin cancer. When this is the case, the normal approach is to make a set of 'measurements' of each region and use that as the basis of distinguishing the different classes. It is rare that a single such measurement will discriminate the different classes, so the normal approach is to make a number of measurements of a feature which are concatenated into a *feature vector*.

The most effective current approach to identify classes from feature vectors is to use *machine learning* — algorithms that mimic a person's ability to learn. There are many machine learning algorithms (you might have heard of neural networks, for example) and we shall look at some of them in the last two or three lectures of the module.

What sort of measurements of a region can go into a feature vector? The most obvious is the number of pixels in the region — in other words, its *area*. Closely related to that is the region's *perimeter*, the number of pixels in its boundary. From those, we can obtain further a useful quantity, the *circularity*: how close the shape of the region is to a circle. For a circle of radius r, the area is $A = \pi r^2$ and the perimeter $C = 2\pi r$. Hence, if we calculate

$$\frac{C^2}{A} = \frac{4\pi^2 r^2}{\pi r^2} = 4\pi$$

then the closer the ratio of C^2/A is to 4π, the closer the shape is to being circular. The circle is known as being the most compact 2D shape, so all other shapes will have a value larger than 4π.

In a similar vein, if one determines the lowest and highest x and y positions of pixels in the object, then one can determine its *bounding box*, the smallest rectange that encloses it. It is then easy to calculate the area of the bounding box; and if we divide that by the number of pixels in the object, we end up with the *rectangularity*, how similar the shape is to a rectangle.

Extending the idea of the bounding box, we can determine not only the limits of the object above, below and to the left and right, but also in the diagonal directions (Figure 4). By connecting opposing pairs of these extremal points (top left to bottom right, *etc.*), we create four axes that describe the shape. These can be used directly in the feature vector or in combination, *e.g.* the ratio of the longest to the shortest can be used to define an *aspect ratio* or *eccentricity* of the shape. The direction of the longest axis can also be used as a *direction* of the shape.

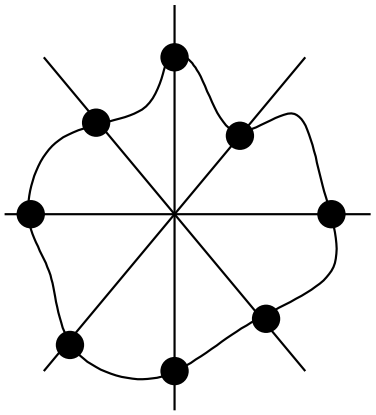


Figure 4: Extremal points can form a shape descriptor

An object which contains within it other labelled regions can be thought of as having holes, and the number of these can be helpful in describing shape — think of distinguishing a D-shaped region from a B-shaped one:

the D has one hole but the B two.

A somewhat different approach is to count the number of pixels lying in each column of a shape, to give a *vertical profile*; and similarly for a *horizontal profile* — even a *diagonal profile*. These can be thought of as 'shape signatures' and used for matching similar shapes on their own, or added to a feature vector.

The descriptors described above only touch on the ways in which shape can be described. In some of the author's research, there are > 120 shape and texture descriptors; a machine learning system identifies which of them are the most effective for a particular problem by trying them out on segmented shapes and then goes on to learn how best to identify the classes of object correctly. We have been able to use that to make a vision system learn how to read car number plates, the first purely-learnt system to do so.

5 The Hough transform for straight lines

Although operators like Canny's edge detector are fairly good at identifying edges in images, they rarely deliver accurate results. For an edge in an image which is perfectly straight, an edge detector's output will often contain breaks or even wiggles. Hence, it would be useful to have a technique that lets us improve the output. Moreover, an edge detector only does half of what we typically want: it identifies which pixels belong to an edge but not what the parameters of that edge are (*i.e.,* what the equation of the line following the edge is).

This is where the Hough (pronounced "huff") transform comes in. It is a general technique, applicable anywhere one needs parameters to be estimated. We shall restrict our discussion to detecting straight lines in images but it is also widely used for identifying circles, for example. The method wasn't actually developed for image analysis: Paul Hough patented his technique in the USA in the early 1960s for identifying the tracks left by particles in bubble chamber images; it was adapted for use in image analysis in the 1970s and became popular with vision researchers in the 1980s. It is now one of the most important tools in the vision toolbox.

A line in 2D can be described using the equation

$$y = ax + b \tag{3}$$

where a and b are parameters that describe the line, the *gradient* and *intercept* respectively. The goal of the Hough transform is to find values for them such that as many edge points as possible lie on the line they describe.

If we had to program this ourselves, what approaches could we take? We could search for an edge point and then look in its 3×3 neighbourhood for other edge points, and so on, trying to trace along the line — but this runs into problems if there are breaks or if the line is not straight. Alternatively, for each pixel in the image, we could draw a straight line at every possible angle and count the number of image pixels that lie exactly on them — but this would takes a long time to run.

The Hough transform takes a different approach. If we re-write (3) in the form

$$b = -xa + y \tag{4}$$

then we can now think of x and y as being the parameters of a straight line and a and b as variables — in other words, this is a line in (a, b) space parameterized by x and y. A single point in (x, y) space describes a line in (a, b) space, and another point in (x, y) space will give rise to a line in (a, b) space with a different gradient and intercept, as illustrated in Figure 5. Although points that lie on the same line in (x, y) space yield different lines in (a, b) space, *those lines all cross in the same place*, and the values of a and b at that point are the gradient and intercept of the line in (x, y) space.

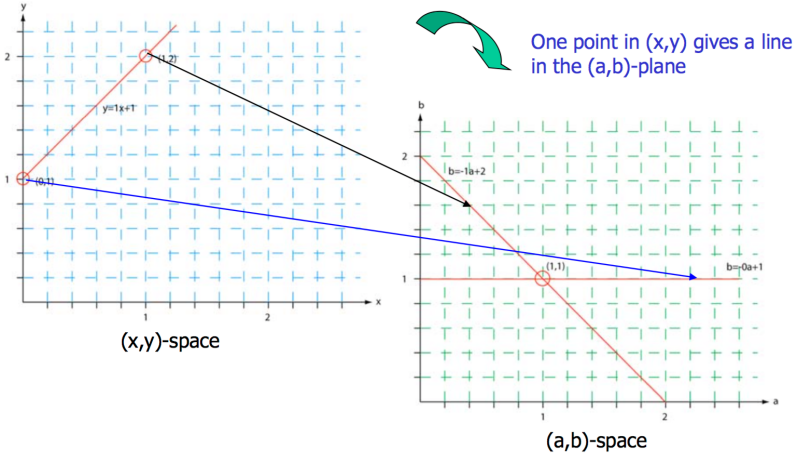


Figure 5: Illustration of computing the Hough transformation (from Anne Solberg's lecture notes)

What we do in practice is form an 'accumulator,' a 2D array indexed by a and b. Then we consider each pixel in the image in turn. For each pixel that appears to be part of a line, we increment those values in the accumulator that correspond to all possible values of a and b. Then, when we have processed all pixels in the image, we look for peaks in the accumulator array.

In practice, the procedure described in the previous paragraph does not work well because a vertical line has $a = \infty$. However, we can represent a straight line in *parametric* or *Hessian normal* form as

$$x \cos\theta + y \sin\theta = r \tag{5}$$

using parameters r and θ — see Figure 6 [Duda and Hart, 1973]. Each point in the (x, y)-plane gives a sinusoid in the (r, θ)-plane, so M colinear point lying on the line of (5) gives rise to M curves that intersect at (r, θ) in the parameter plane.

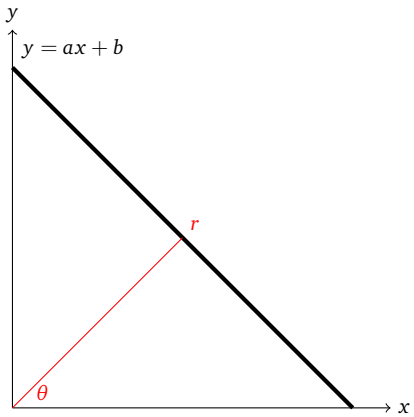


Figure 6: Conventional and parameteric representations of a line

We use the same approach of filling an accumulator array, but drawing in sinusoids rather than straight lines. When the accumulator has been filled, one looks through it for peaks; these yield the (r, θ) values corresponding to the lines that have been found. It is normal to sort them into descending order of peak height, as the highest peaks have the most number of image pixels contributing to them and hence are the longest

lines. Python code that implements all of these stages is shown below, and an image and its Hough transform are shown in Figure 7.

```
def find_peaks (im, threshold=10):
    ny, nx, nc =  im.shape
    peaks = list ()
    for y in range (1, ny-1):
        for x in range (1, nx-1):
            if      im[y,x,0] > im[y-1,x-1,0] \
                and im[y,x,0] > im[y-1,x  ,0] \
                and im[y,x,0] > im[y-1,x+1,0] \
                and im[y,x,0] > im[y  ,x-1,0] \
                and im[y,x,0] > im[y  ,x+1,0] \
                and im[y,x,0] > im[y+1,x-1,0] \
                and im[y,x,0] > im[y+1,x  ,0] \
                and im[y,x,0] > im[y+1,x+1,0] \
                and im[y,x,0] > threshold:
                peaks.append ([im[y,x,0], y, x])
    # Return the peaks sorted into descending order.
    peaks.sort (reverse=True)
    return peaks

def hough_line (im, nr=512, na=512, yc=None, xc=None, threshold=10,\
                    disp=False, dispacc=False):
    ny, nx, nc = im.shape
    if yc is None:  yc = ny / 2
    if xc is None:  xc = nx / 2
    acc = image ((na, nr, 1))
    ainc = math.pi / na
    rinc = nr / math.sqrt (ny**2 + nx**2)
    # Find edge points and update the Hough array.
    for y in xrange (0, ny):
        for x in xrange (0, nx):
            v = im[y,x,0]
            if v > 0:
                for a in xrange (0, na):
                    ang = a * ainc
                    r = ((x - xc) * math.cos(ang) + (y - yc) * math.sin (ang))
                    r += ny
                    if r >= 0 and r < nr:
                        acc[a,r,0] += 1
    # Now find peaks in the accumulator.
    peaks = find_peaks (acc, threshold=threshold)
    return peaks, acc
```

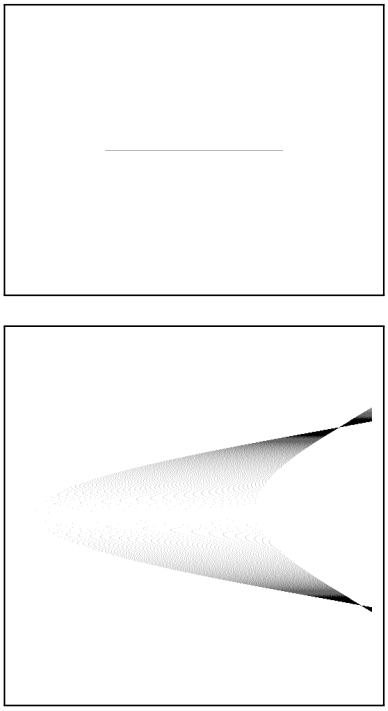


Figure 7: An image and its Hough transform (presented so that black is larger)

A couple of footnotes are in order here. A line running diagonally across the middle of the image must contain more pixels than one running near a corner, so the Hough transform has an inherent bias; there are ways of correcting for this bias, though we shan't go into them here. Similarly, rather than incrementing the value in the accumulator array by unity irrespective of the quality of an edge pixel, one can use its edge strength.

You will see from the above that the Hough transform procedure can be applied to any case where one needs to determine the values of parameters in an equation. It is commonly used to find circles in images, for example the irises and pupils of people's eyes. If you're interested in finding out more about the Hough transform, there is an excellent discussion

in [Burger and Burge, 2008], or look on the web.

References

Burger, Wilhelm and Mark J. Burge (2008). *Digital Image Processing: An Algorithmic Introduction Using Java*. Springer.

Duda, Richard O. and Peter E. Hart (1973). *Pattern Classification and Scene Analysis*. Wiley.