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 segmenting isolated objects from the background and allocating them unique labels. We then look at how edges and corners (intersections of edges) can be found; some of these techniques are useful in their own rights but their principles have also influenced the development of feature detectors described later in these notes.

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 do particularly useful things to them in terms of analysis. 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.

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.

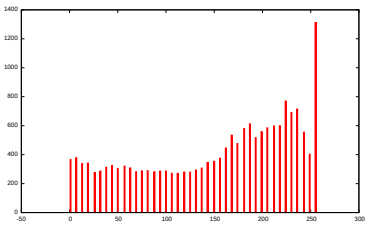


Figure 1: Typical histogram of an image. Where is the best place to set a threshold?

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:

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

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
```

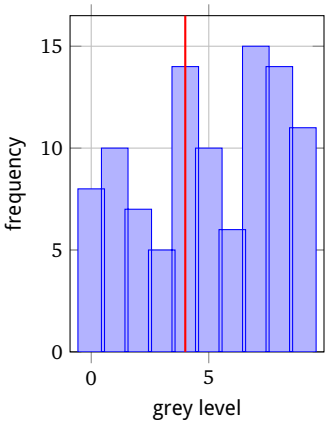
Note that this code yields the threshold correctly only if there is a single $\sigma^2_{\text{between}}$ that is maximum. Moreover, an implicit assumption is that a single, global threshold is enough to separate foreground from background, but the real world is not always that simple. On a positive note 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 The Canny edge detector

For a long time, edge detection was regarded as one of the main problems in computer vision, at least partly because humans find that line drawings of scenes are easy to interpret and processing lines rather than entire images is presumed to be easier for computers too — though the author doesn't follow this logic himself. Nevertheless, edge detection remains a topic of interest as the problem certainly has not been solved, especially for natural scenes.

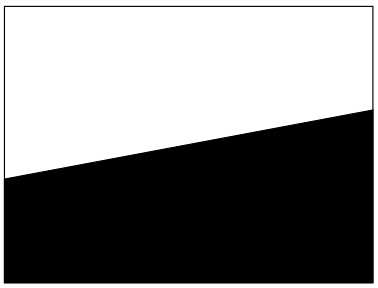


Figure 4: The aperture problem

Edge detection, no matter how well done, suffers from a significant drawback for image interpretation. As Figure 4 shows, when an edge between (say) a dark object and a white background fills the field of view, it is impossible to determine which part of an edge is associated with which part of the object; this is known as the *aperture problem*. Thus, for image analysis, edges are actually of limited value. In practice, it is much more useful to identify the locations of *corners* in an image, as they are related to characteristic features of the objects that created them.

Notwithstanding the above comments, we shall first consider the edge detector due to John Canny, which probably remains the most-used edge detector in image analysis and computer vision. We shall then move on to consider the problem of corner detection. The first scheme we shall look at, due to Moravec, is fairly easy to understand and represented the state of the art in around 1987. Subsequent work in the UK by Harris and Stephens circa 1989 produced a somewhat more effective corner detector which remains competitive with the best, though still vastly inferior to the human visual system.

The development of the Canny edge detector has an Essex connection. A researcher called Mike Brady was a lecturer in Essex's Department of Computer Science and he had a PhD student called Libor Spacek who was working on the problem of edge detection. Part-way through Libor's research programme, Mike took up an appointment at MIT where he got a Masters student, John Canny, to take a somewhat cut-down approach to the careful maths underlying Libor's edge detector. This was published in an MIT report around 1984, then in a journal paper [Canny, 1986] — and was taken up by most of the companies and institutions researching computer vision in the USA. Incidentally, Mike returned to the UK a little later in the 1980s as a professor in the Department of Engineering Science at Oxford and has become one of the prime movers of vision research in the UK. He moved from robot vision into medical imaging research in the 1990s and was knighted a few years ago. Libor was a member of academic staff in CSEE until his retirement a few years ago.

The Canny edge detector is based around three principles:

1. it should respond only to edges, and all edges should be found;
2. edges should be found in the correct places; and
3. multiple edges should not be found where only a single edge exists.

Canny considered how these three criteria could be obtained at a step edge in an image. Without going into the mathematics, he ended up with the following five-step algorithm.

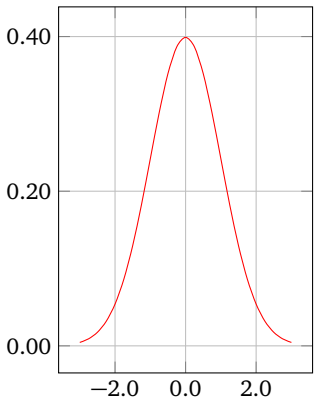


Figure 5: Profile across a Gaussian mask

Step 1. Convolve the image with a Gaussian-shaped mask (see Figure 5) to smooth the image and reduce the effects of noise. The s.d. of the Gaussian controls the amount of smoothing obtained, so this part of the algorithm can be tailored to the characteristics of the data.

Step 2. Find differences in the horizontal and vertical directions, averaging over 2×2 squares of the image S:

$$H(x,y) = \frac{(S(x,y+1)-S(x,y))+(S(x+1,y+1)-S(x+1,y))}{2} \tag{3}$$

$$V(x,y) = \frac{(S(x+1,y)-S(x,y))+(S(x+1,y+1)-S(x,y+1))}{2} \tag{4}$$

Step 3. Find the magnitudes and directions of these gradients:

$$M(x,y) = \sqrt{V(x,y)^2 + H(x,y)^2} \tag{5}$$

$$\theta(x,y) = \tan^{-1}\frac{V(x,y)}{H(x,y)} \tag{6}$$

Note that, when calculating $\theta(x,y)$ on a computer, you need to use the `atan2` function, which returns a value in the range $-\pi$ to $+\pi$, rather than `atan`, which returns a value in the range $-\pi/2$ to $+\pi/2$: `atan2` takes into account the signs of both of its arguments in determining the angle.

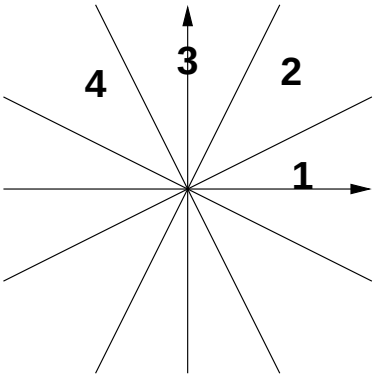


Figure 6: Quantization of $\theta(x,y)$ into four directions

Step 4. A broad edge in the image will tend to produce two edge responses, one at either side. This violates our third criterion, so something must be done to reduce any broad lines in $M(x,y)$. The approach taken is to perform *non-maximum suppression*, *i.e.* to remove all those parts of the edge except where there is the greatest local change. This is carried out in two stages:

- Firstly, $\theta(x,y)$ is quantised into four values that indicate roughly the direction of the edge gradient (Figure 6).
- Then, for each 3×3 neighbourhood in $M(x,y)$, the value at x,y is compared with its neighbours along the four possible gradient directions and, if $M(x,y)$ is less than any neighbour, it is set to zero.

The result is that any broad edges in $M(x,y)$ are thinned, usually to be a single pixel in width.

Step 5. Even after non-maximum suppression, there will usually be many false edge fragments, due to texture and noise in the image. These typically have much poorer contrast than edges obtained from the boundaries of objects, which is what computer vision is more interested in. To reduce the effects of these false edge segments, we attempt to link them together.

In the Canny detector, we use a thresholding strategy which employs two thresholds, τ_L and τ_H, where typically $\tau_H \approx 2\tau_L$. This scheme is sometimes called *hysteresis thresholding*. Simply thresholding $M(x,y)$ using a sensibly-chosen value for τ_H will reduce the number of false edge segments but the true edge segments will inevitably contain gaps. The idea is to try to join up edge segments.

When the contour formed by following successive pixels with values $> \tau_H$ (shown in red in Figure 7) ends, the hysteresis thresholding algorithm looks in the 3×3 region around that pixel for any neighbours whose value is $> \tau_L$ (shown in black in Figure 7). If so, it continues the edge to that pixel. This process continues until there are no suitable neighbours or the another edge segment with value $> \tau_H$ has been found.

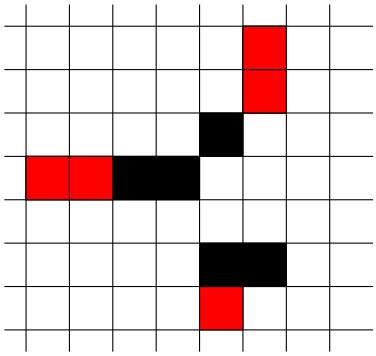


Figure 7: Linking together edge segments

As we can see from Figure 9, Canny's edge detector gives better results than the Laplacian or Sobel detectors — which it should, of course, since it is somewhat more sophisticated. However, we can also see that the Canny result is far from perfect on images of natural scenes. For robot vision in research laboratories, where illumination tends to be strong and the boundaries of objects straight and well-defined, it is fairly effective.

5 Moravec's corner detector

Like so many other image processing operators, the Moravec corner detector is based around the processing of a region with a mask of coefficients. In this case, the strategy is to devise a mask with the property that convolution with the mask should produce a maximum in its output when it is centred on a corner, and a large decrease when the mask moves away from the corner.

If we think about this, we'll see that there are three main cases, shown in Figure 8:

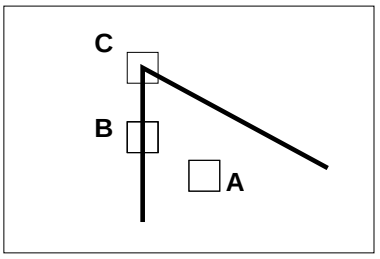


Figure 8: The difference cases considered in the Moravec corner detector

A: if the region of the image is fairly uniform, all shifts result in a small change in response;

B: if the region straddles an edge, a shift along the edge produces a small change but a shift perpendicular to the edge produces a large change;

C: at a corner or isolated point, all shifts produce a large change.

If we write $I(x,y)$ as the value of the pixel at x,y and $W(u,v)$ as the coefficient in the mask for some values of (u,v), then we can write this as

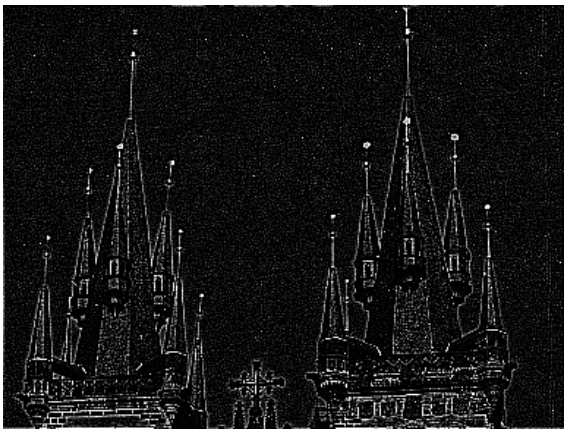
$$E(x,y) = \sum_{u,v} W(u,v)\,(I(x+u,y+v) - I(u,v))^2. \qquad (7)$$

If we consider shifts in x,y of $(1,0)$, $(1,1)$, $(0,1)$ and $(-1,1)$, then we explore the same four directions that we used when quantising $\theta(x,y)$ in the Canny edge detector (Figure 6). Hence, if we look for local *maxima* in $\min(E)$ above some threshold value, we should be able to identify corners.

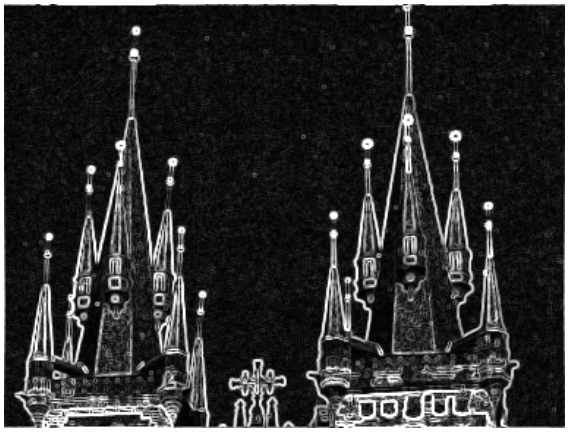
6 The corner detector of Harris and Stephens

As mentioned above, the Moravec detector represents the state of the art in around 1987. Subsequent researchers have improved on this, in partic-

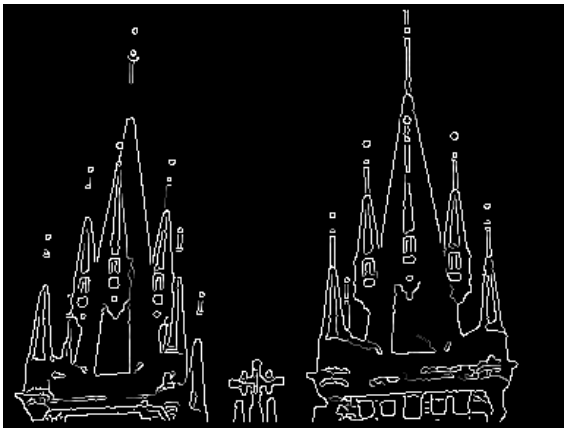
(a) Original image



(b) Result of Laplacian



(c) Result of Sobel



(d) Result of Canny

(e) Result of Harris & Stephens

(f) Result of SUSAN

Figure 9: Comparing the results of different edge and feature detectors

ular in the detector due to [Harris and Stephens, 1988], which combines the ideas of Moravec and Canny has been the most widely-used corner detector since the early 1990s. The advance due to Harris and Stephens seems fairly straightforward, given our knowledge of Canny's edge detector: rather than considering shifted patches, Harris and Stephens considered the direction of the derivatives at the corner directly.

Consider a patch of a grey-scale image $I(u,v)$ shifted by (x,y). The weighted sum-squared difference is given by (7). If one expands $I(u+x,v+y)$ as a Taylor series then, if I_x and I_y are the partial derivatives of I in the x- and y-directions respectively,

$$I(u+x,v+y) \approx I(u,v) + xI_x(u,v) + yI_y(u,v)$$

which means that

$$E(x,y) \approx \sum_{u,v} w(u,v)\left(xI(u,v) + yI(u,v)\right)^2.$$

If we calculate the matrix of partial derivatives at point (x,y)

$$\mathbf{A} = \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix}$$

then

$$S(x,y) \approx \begin{pmatrix} x & y \end{pmatrix} \mathbf{A} \begin{pmatrix} x \\ y \end{pmatrix}$$

will have a large variation irrespective of direction x and y. Rather than calculate $S(x,y)$, Harris and Stephens calculate $\det(\mathbf{A}) - \kappa \mathrm{trace}^2(\mathbf{A})$, κ being a tuning parameter usually in the range 0.04–0.15, to determine whether a corner is present. The output from Harris and Stephens is shown in Figure 9. These days, it is able to run at video rate on off-the-shelf hardware.

7 Other corner detectors

Work subsequent to that of Harris and Stephens has produced more sophisticated detectors, two of which are worthy of mention here. The first is the *smallest univalue segment assimilating nucleus* ("SUSAN" — a strained acronym if there ever was) operator of Steve Smith and Mike Brady (again) [Smith and Brady, 1997]. An example of its output is also shown in Figure 9.

Secondly, Edward Rosten and colleagues at Cambridge developed the FAST corner detector [Rosten, Porter and Drummond, 2010]. This is based on a particularly simple observation: at a corner, more than half the pixels will be dark or light, as illustrated in Figure 10. Although some sophistication is required to make FAST work well, a C implementation is easily able to operate at video rates.

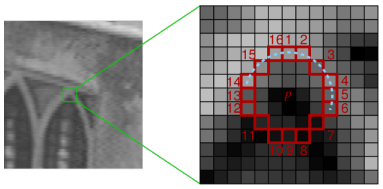


Figure 10: The FAST corner detector (from http://mi.eng.cam.ac.uk/~er258/work/fast.html)

References

Canny, John F. (1986). "A computational approach to edge detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.6, pp. 679–698.

Harris, C. and M. Stephens (1988). "A combined corner and edge detector". In: *Proceedings of the 4th Alvey Vision Conference*. http://www.bmva.org/bmvc/1988/avc-88-023.pdf, pp. 147–151.

Rosten, Edward, Reid Porter and Tom Drummond (2010). "Faster and better: A machine learning approach to corner detection". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 32, pp. 105–119.

Smith, Stephen M. and J. Michael Brady (1997). "SUSAN—A New Approach to Low Level Image Processing". In: *Int. J. Comput. Vision* 23 (1), pp. 45–78.