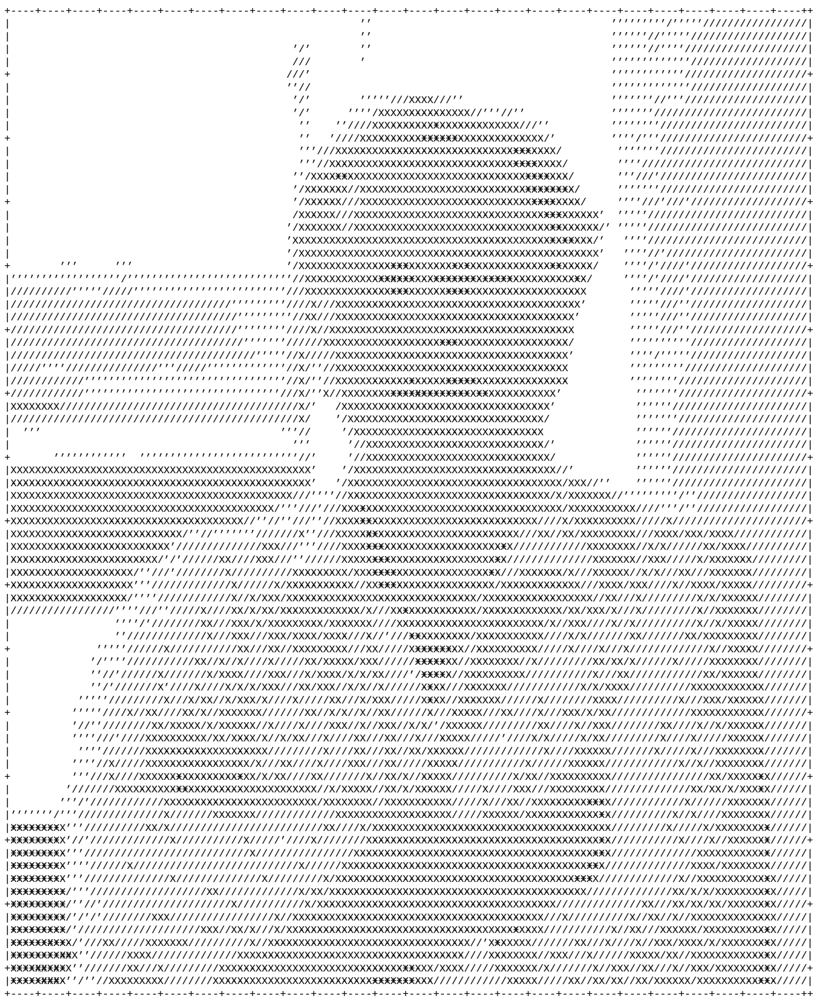
ADRIAN F. CLARK

COMPUTER VISION

COMPUTER SCIENCE AND ELECTRONIC ENGINEERING
UNIVERSITY OF ESSEX



Adrian Clark has a degree in Physics and a PhD in digital image processing. His post-doctoral research was some of the first into parallel image processing. He has worked in industry in the general area of real-time image analysis, where he gained experience of many types of image and a wide range of applications.

At Essex, his principal research interests are into the automatic construction of computer vision systems, and into virtual and augmented reality — the maths underlying them all is quite similar. He has collaborated with companies large (Airbus, Leonardo, MBDA, Vitec) and small (CG Eye, Filament, Railscape). In the dim and distant past, he led UK involvement in an international standard called *Image Processing and Interchange*. He recently finished a stint as Chairman of the Bristish Machine Vision Association, the learned society for academic and industrial vision researchers.

Why is his picture made up from over-printed characters? When he started his PhD, computers required air-conditioned rooms and this was *the only way* anyone could visualize images. Raster displays came along shortly afterwards but were hugely expensive and limited to about 485 lines and 512 pixels/line. It's good to remember how primitive things were when most of the ground-breaking research into computer vision was done!

Contents

1 An Introduction to Computer Vision . . . 11
- 1.1 Introduction . . . 11
- 1.2 Applications of image processing . . . 11
- 1.3 Computer vision in action . . . 12
- 1.4 Other types of sensor . . . 13
- 1.5 Image and video file formats . . . 14
- 1.6 Chris Greening's *Sudoku Grab* . . . 15
- 1.7 About the module . . . 19
- 1.8 Installing OpenCV on your own machine . . . 22
- 1.9 Further reading . . . 23

2 The Human Visual System . . . 24
- 2.1 The eye . . . 24
- 2.2 From the eye to the brain . . . 27
- 2.3 Inside the brain . . . 27

3 Vision Software . . . 29
- 3.1 Introduction . . . 29
- 3.2 Images and pixels . . . 30
- 3.3 Coding up algorithms . . . 33
- 3.4 Histograms and their manipulation . . . 35
- 3.5 Contrast-stretching and an improved histogram routine . . . 38
- 3.6 Histogram equalisation . . . 40
- 3.7 Content-based image retrieval using histograms . . . 40

4 Convolution . . . 46
- 4.1 Introduction . . . 46
- 4.2 Enhancing isolated points using convolution . . . 46
- 4.3 Blurring . . . 48
- 4.4 Using the median . . . 49
- 4.5 Other types of masks . . . 50
- 4.6 Implementing convolution . . . 50
- 4.7 Mathematical morphology . . . 52
- 4.8 Finding known patterns in images . . . 54

5 Low-Level Vision . . . 56
- 5.1 Introduction . . . 56
- 5.2 Isolating regions by thresholding . . . 56
- 5.3 Region labelling . . . 58

5.4 Describing regions . . . 60
5.5 A practical shape-based vision system . . . 62
5.6 Measuring texture . . . 64

6 Evaluating Vision Systems . . . 70
6.1 Introduction . . . 70
6.2 Evaluating a vision system . . . 71
6.3 Comparing vision systems . . . 73
6.4 Assessing `colrec1` . . . 74
6.5 Recognising Handwritten Digits . . . 75
6.6 Comparing Performance Figures on Different Datasets . . . 77

7 Intermediate-Level Vision . . . 80
7.1 Introduction . . . 80
7.2 The Canny edge detector . . . 80
7.3 Moravec's corner detector . . . 83
7.4 The corner detector of Harris and Stephens . . . 83
7.5 Other corner detectors . . . 85
7.6 The Hough transform for straight lines . . . 85
7.7 Describing corners . . . 89
7.8 SIFT and related techniques . . . 90
7.9 Local binary patterns . . . 92

8 Looking at Humans . . . 93
8.1 Introduction . . . 93
8.2 Locating Faces by Colour . . . 93
8.3 Viola-Jones: Haar Features and Adaptive Boosting . . . 94
8.4 Processing Face Images . . . 98
8.5 Recognising Faces: Eigenfaces . . . 99
8.6 HOG, the histogram of oriented gradients . . . 102
8.7 OpenPose and friends . . . 103

9 Vision in a 3D World . . . 105
9.1 Introduction . . . 105
9.2 The geometry of imaging . . . 106
9.3 Computational stereo . . . 107
9.4 Computational stereo in action . . . 109
9.5 Propagating uncertainty . . . 110
9.6 3D coordinate geometry . . . 111
9.7 Correcting perspective effects . . . 114
9.8 Visual structure from motion . . . 116
9.9 Visual SLAM . . . 118

10 High-Level Vision with Machine Learning . . . 121
10.1 Introduction . . . 121
10.2 Unsupervised learning using k-means . . . 122
10.3 Simple supervised learning with WISARD . . . 123
10.4 The MNIST test case . . . 124
10.5 Support vector machines . . . 125
10.6 The genetic algorithm . . . 128

10.7 Genetic programming 131

11 Deep Learning and Neural Networks 137
11.1 Introduction 137
11.2 The multi-layer perceptron 137
11.3 MLP in software 140
11.4 Convolutional neural networks 141
11.5 Transfer learning. 147
11.6 Discussion 147

List of Figures

1.1 Applications of computer vision 11
1.2 Autonomous cars (images taken from the web) 12
1.3 RoboCup (images from http://www.robocup.org/) 12
1.4 Remotely-sensed imagery of the Mediterranean region . . . 13
1.5 Thermal image of a house 13
1.6 An X-ray image 13
1.7 A synthetic aperture radar (SAR) image 14
1.8 A depth image captured from a Microsoft Kinect 14
1.9 Sudoku grid captured from a newspaper 16
1.10 After thresholding 17
1.11 Tidying up the thresholded image 17
1.12 Finding the grid 17
1.13 The Hough transform of the lines 18
1.14 Hough features superimposed on the grid 18
1.15 The image after rectification to a rectangle 18
1.16 The final grid, with recognised numbers superimposed . . . 19

2.1 A cross-section through the human eye 24
2.2 The relative spectral sensitivity of cones 25
2.3 The spectral sensitivity of rods 26
2.4 The distribution of rods and cones on the retina 26
2.5 The excitatory–inhibitory nature of ganglion cells 27

3.1 A digital image of an upward-pointing arrowhead 30
3.2 Pixels have a spatial extent 30
3.3 Magnified newsprint 32
3.4 Representing colour as hue, saturation and value 32
3.5 Representations of colour images in software 33
3.6 Histograms corresponding to differently-exposed images . . 36
3.7 Result of histogram-equalising the image in Figure 3.6(c) . 41

4.1 A photograph of the Essex campus and its edges 46
4.2 Illustration of the convolution process 47
4.3 Morphological expand, shrink and opening 53
4.4 Finding an object's boundary and skeleton 53
4.5 Locating proteins on cells using matched filtering 54
4.6 Template matching used to locate occurrences of 'a' in text 55

5.1 Histogram of a typical image 56

5.2 Example 10-level image and its histogram, with the Otsu threshold marked . . . 58
5.3 Distinct regions of an image and the result of region labelling 59
5.4 Iris species used by Fisher . . . 60
5.5 Visualization of Fisher's iris data . . . 61
5.6 Extremal points can form a shape descriptor . . . 62
5.7 Biscuits on a virtual conveyor belt . . . 63
5.8 Identification of circular and rectangular biscuits . . . 64
5.9 Broken and overlapping biscuits are rejected . . . 64
5.10 Rectangular biscuits at an angle and their bounding boxes . 64
5.11 Brickwork is a regular texture . . . 64
5.12 Mosaic of irregular textures . . . 65
5.13 Grey-level co-occurrence matrix of the brickwork of Figure 5.11 for shifts of $X = 5, Y = 3$ . . . 65
5.14 Textures and corresponding GLCMs for shifts of $X = 10, Y = 7$ . . . 67
5.15 Rotated textures and corresponding GLCMs for shifts of $X = 10, Y = 7$ (shown with a logarithmic scale of grey levels to bring out detail) . . . 68
5.16 GLCMs for various shifts . . . 69

6.1 Examples of ROC curves . . . 72
6.2 ROC curves that cross . . . 72
6.3 The tails of a binomial distribution . . . 74
6.4 The various classes of fruit used for assessing `colrec1` . . . 74
6.5 Examples from the MNIST database . . . 76

7.1 Profile across a Gaussian mask . . . 81
7.2 Obtaining M and θ from H and V . . . 81
7.3 Quantization of $\theta(x, y)$ into four directions . . . 81
7.4 Non-maximum suppression . . . 82
7.5 Linking together edge segments using hysteresis thresholding 82
7.6 The aperture problem . . . 82
7.7 The different cases considered in the Moravec corner detector 83
7.8 Comparing the results of different edge and feature detectors 84
7.9 The FAST corner detector . . . 85
7.10 Edge detectors usually deliver only segments of edge . . . 85
7.11 Illustration of computing the Hough transformation . . . 86
7.12 Conventional and parameteric representations of a line . . . 87
7.13 An image and its Hough transform . . . 87
7.14 The Hough transform can be used in a lane-following system 88
7.15 SIFT features marked on a natural image . . . 90
7.16 An example of feature matching using SIFT . . . 90
7.17 LBP neighbours . . . 92

8.1 Successful detection of human skin by colour . . . 93
8.2 Skin detection by colour is easily confused . . . 94
8.3 Haar features . . . 94
8.4 Haar feature that helps detect eyes . . . 95
8.5 An image and its integral image . . . 95

8.6 Principle of calculating an integral image 96
8.7 Viola-Jones face location in action 97
8.8 Face dimensions that accord to the golden ratio are supposed to be more attractive 98
8.9 Newspaper article from 2017 on beauty and the golden ratio 99
8.10 Model-based image coding 99
8.11 Illustration of principal component analysis 100
8.12 Some eigenfaces 101
8.13 The six ‘universal’ expressions 102
8.14 HOG cells 103
8.15 HOG gradients 103
8.16 Tracking human motion with OpenPose 104

9.1 Imaging geometry of a pin-hole camera 106
9.2 Imaging geometry of a real camera 107
9.3 The geometry of computational stereo 107
9.4 Stereo pair of the Candide 3D head model 109
9.5 Locations are accurate only to the nearest pixel 110
9.6 The fundamental 2D transformations 111
9.7 Rectification of a perspective-distorted image 114
9.8 The principle of 3D reconstruction by structure from motion 116
9.9 Reconstruction of Square 2 117
9.10 A reconstruction of Elmstead’s church 117
9.11 A coral reef reconstruction 118
9.12 Visual SLAM in operation 119

10.1 The k-means algorithm 123
10.2 Illustration of the operation of WISARD 123
10.3 WISARD on display in the Science Museum 124
10.4 Typical images from the MNIST database 124
10.5 Principle of the support vector machine 127
10.6 The ‘kernel trick’ 127
10.7 A population of N chromosomes, each having P values . . . 129
10.8 Real-valued crossover 130
10.9 Representation of program code as a parse tree 131
10.10 Crossover exchanges sub-trees between parse trees 132
10.11 Jasmine, a GP-based vision system construction tool 133
10.12 Some problems solved by Jasmine 134
10.13 Training data and ground truth for a simple vision task . . . 135
10.14 A cascade of binary classifiers 136

11.1 Illustration of a multi-layer perceptron 138
11.2 Illustration of a single neuron 138
11.3 Common choices for the activation function of a neuron . . 138
11.4 LeNet 144
11.5 AlexNet 144
11.6 GoogLeNet 145
11.7 VGGNet 145
11.8 ResNet 146
11.9 Problematic classification with deep networks 148

List of Tables

1.1 Popular image file formats 14
1.2 Image formats devised for specific application areas 14
1.3 Common video formats 15

3.1 Means and standard deviations of the images of different exposures in Figure 3.6 42

6.1 Possible outcomes for a single test from a pair of algorithms 73
6.2 Converting Z values onto confidence limits 74
6.3 Error rates from `colrec1.res` 75
6.4 Class confusion matrix calculated from `colrec1.res` 75
6.5 Distribution of the 60,000 training and 10,000 test images amongst the classes in the MNIST database 76
6.6 Table of error rates for T1.000 76
6.7 Class confusion matrix for T1.000 77
6.8 Table of error rates for T0.400 77
6.9 Confusion matrix for T0.400 78
6.10 Comparison of results for T1.000 and T0.400 using McNemar's test 78

10.1 Numbers of training and test examples in MNIST 124
10.2 MNIST WISARD error rates 125
10.3 Class confusion matrix for WISARD on MNIST 125
10.4 Table of error rates for WISARD on a subset of MNIST . . . 126
10.5 Class confusion matrix for WISARD on a subset of MNIST . 126
10.6 Table of error rates for SVM on MNIST 129
10.7 Class confusion matrix for SVM on MNIST 129

1
An Introduction to Computer Vision

This chapter gives a short introduction to the discipline of computer vision. It gives examples of computer vision applications and technology, then outlines a complete vision system which involves both pixel-level operations and a machine learning component, to give an idea of what is typically involved.

The chapter goes on to discuss the nature of the module. Firstly, it describes teaching material such as the notes you are now reading and ancillary material to help your understanding: video clips, multiple-choice quizzes, and so on. Computer vision is a practical subject, so hand-in-hand with the taught material is a programme of experiments that you need to carry out. Finally, assessment of the module is discussed.

1.1 Introduction

This module is concerned with *image processing* and *computer vision*. As you might expect, this involves the manipulation of image data, usually in the context of making a computer 'understand' what it is looking at. People who have used a tool such as *Photoshop* or *Paint Shop Pro* to fiddle with digital camera or scanned images, or with *Premiere* to edit digital video, have done image processing. However, the emphasis in this course is on the *automatic* processing and analysis of image data, extracting useful information from the images. In fact, perhaps the most useful way to start this course is to consider where image processing and computer vision are used. After that, some fundamental notions about images and their representation are discussed.

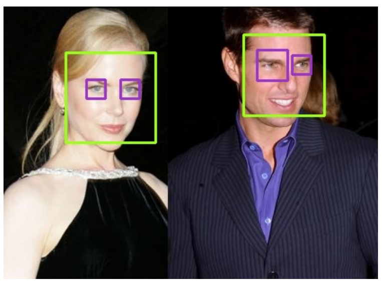
1.2 Applications of image processing

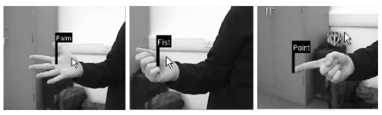
Image enhancement: contrast enhancement in Photoshop *etc.*, red-eye removal, correcting camera distortions, noise reduction, deblurring — all involve image processing.

OCR: by far the most successful image analysis application to date is optical character recognition, the conversion of printed text into machine-readable characters.

Surveillance: automatic analysis of town centre CCTV imagery is an obvious application, though one that cannot currently be achieved. However, automatic number plate recognition is entirely viable and formed the

(a) Reconstructing scenes from multiple views


(b) Face location and recognition



(c) Gesture recognition

(d) Reading number plates

(e) Gaming interfaces

(f) Augmented reality

Figure 1.1: Applications of computer vision

basis of London's pioneering congestion charging system; this functionality is now commonplace on road networks and in car parks.

Inspection: monitoring items on production lines, such as machined parts or foodstuffs.

Guidance: controlling the motion of robots based on imagery from a camera or other sensor; likewise other vehicles such as cars on roads — even guided missiles.

Coding: modern image coding techniques such as MPEG-4 ("DivX") use image analysis to segment the foreground from the background; and there are more impressive research techniques such as model-based coding that have the potential to reduce data rates a great deal further.

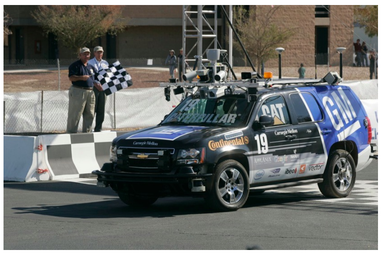
Biometrics: fingerprints, iris scanning, handwriting, face recognition — all biometric systems involve image analysis or its companion discipline of pattern recognition.

Scene reconstruction: virtual studios, compositing computer-generated and live action in movies, augmented reality.

Science: microscopy, remote sensing, astronomy, even trying to understand how the human visual system works.

Some of these applications, and some not listed above, are illustrated in Figure 1.1. The market for image processing applications is several *billions* of pounds per annum — and it is set to be worth a great deal more when techniques become more robust.

(a) Stanford, 2005


(b) Tartan Racing, 2007

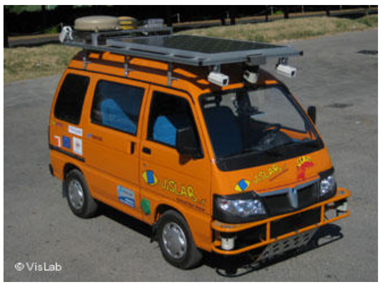

(c) Parma, 2010

Figure 1.2: Autonomous cars (images taken from the web)

1.3 Computer vision in action

Autonomous vehicles

The most exciting area of robotics is arguably autonomous vehicles. Since about 2000, a great deal of effort has been expended on research into driverless cars. The event that brought this to public attention was the DARPA "grand challenge," in which teams from different research groups competed to make their autonomous vehicles navigate a 132-mile course in the Mojave Desert in 2005. As has been well-documented, *Stanley* from Stanford University won the \$2 M prize, navigating the course in a little under 7 hours. There have been other, similar events since then. In 2007, the Tartan Racing team (Carnegie-Mellon and others) won an urban grand challenge for another \$2 M prize. In 2010, an autonomous vehicle travelled from Italy to China, an under-reported but ground-breaking event. These three vehicles are depicted in Figure 1.2. Subsequent to this, most research in the area has been led by industry, with Google and Tesla being major players. There are also driverless car initiatives here in the UK, with a major test under way in Milton Keynes. Here at Essex, we're not too interested in autonomous cars but we have active threads of research into robot submarines and robot aerial vehicles, both fixed- and rotary-wing.

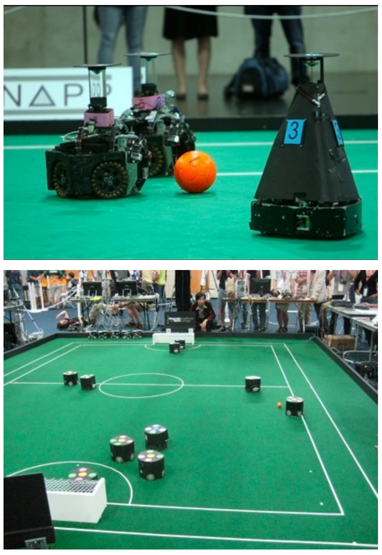


Figure 1.3: RoboCup (images from http://www.robocup.org/)

Another active area of mobile vehicle research is, of course, the exploration of other worlds. NASA's Mars rovers, *Spirit*, *Opportunity* and more recently *Curiosity*, have been exploring the planet. All of them are equipped with stereoscopic, colour cameras and are able to sense in visible and infra-red wavebands. Among other things, imagery from the cameras is used to guide the rovers' robotic arms.

RoboCup and related competitions

You can think of RoboCup as the World Cup for soccer-playing robots. Although seemingly facetious, the competition aims to foster intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined — and it has proven to be very effective at this. There are various classes of entry, ranging from simulation right up to humanoids. An Essex team finished third in the 1999 event; we haven't entered in recent years.

RoboCup spawned a number of related events, the best-known of which was first called the Robot Olympics but was re-named (mostly likely because of copyright problems) to RoboGames. It would be quite fun to enter these...

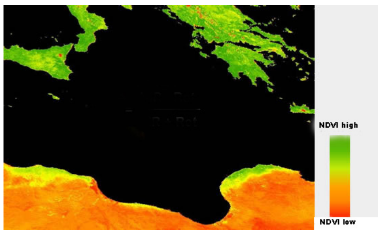


Figure 1.4: The *Normalised Difference Vegetation Index* (NDVI) of the Mediterranean region, using visible and infra-red images captured from satellites. The NDVI allows stressed vegetation to be identified.

1.4 Other types of sensor

There is no reason why an image must capture what we humans see. Satellites that observe the Earth from space (see Figure 1.4) regularly capture imagery not only in the visible wavebands but also in the infra-red. This is useful because the ratio of the intensity of some visible to near infra-red wavebands is a good way of determining whether vegetation is stressed, often through lack of water. This can be done using the *Normalised Difference Vegetation Index* (NDVI), defined for each pixel as

$$\frac{NIR - R}{NIR + R} \tag{1.1}$$

where NIR and R are the near infra-red and red values of the pixel.

There are also sensors that capture a spectrum for each pixel, known as *hyperspectral* sensors; these normally record well over 100 spectral bands simultaneously.

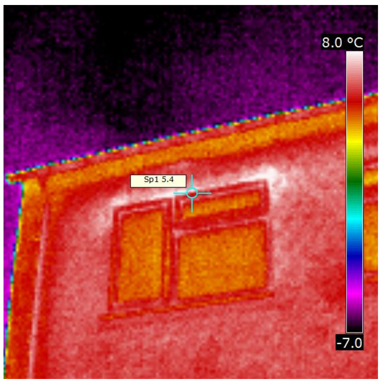


Figure 1.5: Thermal image of a house. The image is actually captured in greyscale, with black being cold and white hot, but the different values grey are mapped onto different colours to produce this "pseudo-colour" image.

Infra-red radiation in the 8–10 μm region corresponds to the temperatures that we experience on the Earth's surface and so allows us to form 'heat pictures' (Figure 1.5). Thermal imaging cameras are now available commercially and are used for things like identifying how well insulated houses are. Thermal cameras with a reasonable number of pixels on their sensor (say 640×480) are currently expensive, though new sensor technologies mean that the cost will probably fall if there is a market for them.

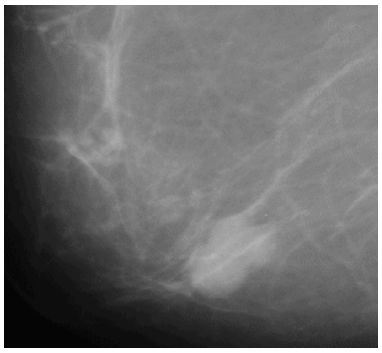


Figure 1.6: A mammogram (an X-ray image of a human breast) showing a lesion. Distinguishing cancerous lesions from benign ones automatically by computer vision is difficult.

Infra-red radiation uses wavelengths longer (less energetic) than visible light. If one employs radiation that is more energetic than visible, we pass through the ultra-violet wavebands and end up in the X-ray region. As we all know, X-rays are able to pass through soft tissue but are attenuated by bone, so they are a good way of imaging broken bones or lesions in soft tissue, or metal weapons in airline baggage. Figure 1.6 presents a mammogram (a breast X-ray) taken from the mini-MIAS database which shows lesions. Distinguishing lesions that are cancerous from benign ones is difficult for a radiologist to do, and substantial effort has been expended for a number of years in attempting to produce automatic ways of performing this task. It has still not been solved completely.

We are all used to seeing blips on radar screens in movies. There are now a number of imaging radar technologies which yield a complete image rather than an isolated blip. Unlike the other sensors we have considered to date, radar is an *active* sensor, meaning that it send out a radar 'beam' into the world and detects reflections. There are several unusual consequences of this; for example, tall features appear to be leaning towards the sensor. An image from a radar sensor is shown in Figure 1.7.

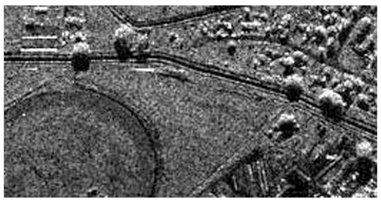


Figure 1.7: A synthetic aperture radar (SAR) image. Note how noisy it is compared to a conventional image, a combination of the need for the radar to illuminate the scene and the coherent nature of the imaging.

Our final example of an unusual sensor is the Microsoft Kinect. As well as a camera working in the visible, it features a camera that records in the near infra-red. There is also an arrangement that projects an infra-red pattern of dots into the environment, and the horizontal displacements of these dots let it determine how far away things are — this is similar in principle to the stereo arrangements considered in Chapter 9. The end result is an image in which each pixel represents the distance from the camera to an object: the higher the value of a pixel, the nearer it is to the camera — see Figure 1.8.

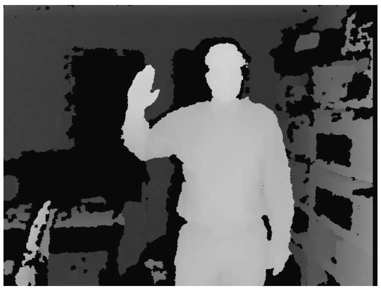


Figure 1.8: A depth image captured from a Microsoft Kinect, with brighter meaning closer to the camera.

The technologies touched on in the previous paragraphs are by no means the only ones that are able to yield images. People who have studied robotics will be familiar with sonar and lidar ("laser stripers"), mostly for obstacle detection; yet both of these can yield images. Radio astronomers are able to build up images of nearby stars and nebulae. Radiation with a frequency of around 1 THz is able to pass though walls and clothing and is being trialled in security monitors for detecting firearms hidden in clothing. Anything that can produce a regularly-sampled grid of numbers can be analysed using computer vision techniques.

1.5 Image and video file formats

There are well over a hundred image file formats used around the world. The most common are listed in Table 1.1 but there are also many others used in particular niches, some of which are listed in Table 1.2. They are all broadly similar, storing the values of the pixels along with ancillary information such as the dimensions of the image (the number of pixels per line and the number of lines).

GIF	index-colour graphics
PNG	Portable Network Graphics
JPEG	standard for digital cameras
PGM, PPM	used by PBMPLUS and NetPBM
HEIC	"high-efficiency" format used by Apple and others

Table 1.1: Popular image file formats. The first three are the most widely-used while the others are often encountered in computer vision.

BMP	Microsoft Windows
TIFF	printing and publishing industries
FITS	astronomy
DICOM	body scanners

Table 1.2: Image formats devised for specific application areas

However, there is an important distinction between some image file formats that you need to be aware of, as this affects their value for image analysis. Some image file formats are *lossy*, meaning that they discard some of the information contained in the pixels captured, though usually in a way that may not be visually obvious. The best-known lossy format is JPEG — which is unfortunate as it is by far the most commonly-used file format, not least because most digital cameras store photographs in JPEG format. The reason that JPEG is lossy goes back to the times when images were large relative to the capacity of disks *etc.*: removing visually imperceptible information results in smaller file sizes. However, *such image formats are poorly suited to image analysis* because the information that is 'visually imperceptible' tends to be around the edges of objects, and those are precisely what many vision algorithms work with.

If using JPEG is not a good idea, then what formats are good? For general work, PNG ("portable network graphics") has become popular and the author recommends you use that if you can. A reasonable proportion of the computer vision community uses the family of formats known as `PBMPLUS` or `NETPBM` as they are easy to read and write. Fortunately, there are good conversion utilities available: search for `NETPBM` or `ImageMagick` in Google. These are available in our Software Labs, under at least Linux.

Just as there are many image formats, there is a wide range of movie (video) formats. The most common are listed in Table 1.3. All of these involve lossy compression: the volume of data is too great for storage or transmission uncompressed or with loss-less schemes. The compression schemes typically use JPEG-like compression within a frame ("intra-frame" compression) and allow regions in one frame to be translated to appear in subsequent frames ("inter-frame" compression).

MPEG	the Motion Picture Expert Group's standard for broadcast video
MP4	MPEG version 4, a more sophisticated video encoder
AVI	Microsoft's "audio video interleave"
WMV	Microsoft's "windows media format"
MOV	Apple's QuickTime movie
FLV	Adobe's Flash video

Table 1.3: Common video formats

Only the first two in the list are internationally standardised. In fact, MPEG is a series of standards: MPEG-2 is essentially what is used for standard-definition broadcast digital television while the more recent MPEG-4 is used for high-definition terrestrial services, as well as "Freesat" and Sky. MPEG-4 is more effective at compressing video streams so that they require a lower data-rate but involve more complicated algorithms.

Most low-cost devices emit AVI-format video streams. Such video tends not to use inter-frame coding, only JPEG-compressing each frame. The resulting files are large but are fairly easy to unpack — in particular, OpenCV, which is used in the laboratory sessions associated with this module, is able to read the frames from (some types of) AVI and MP4 files. More sophisticated devices normally produce MPEG-4 files, at least until one is in the realm of professional video cameras.

1.6 Chris Greening's *Sudoku Grab*

The following is taken from the blog of a former CSEE research student, Chris Greening:

`http://sudokugrab.blogspot.com/2009/07/how-does-it-all-work.html`

In the entry, he talks about how he designed and implemented an iPhone App which captures images of Sudoku puzzles and converts them into text; many of the techniques will be familiar to you by the end of the module.

I get the odd email from people asking how *Sudoku Grab* works. I've replied to quite a few individually, but I thought it would be much easier to just explain it on the blog. Also when I demo the app people are often amazed (and equally, some people are also often not amazed). For the ones that are amazed they can get a bit enthusiastic and start coming up with all sorts of amazing ideas on what you could do with "the technology". I then have to explain to them that it's actually not doing anything really clever and that their idea might actually be quite a hard problem in comparison.

Sudoku Grab is a collection of some fairly basic image processing techniques that most engineering students at University could probably figure out how to put together. All of the algorithms used are either commonly available and can be found on the internet or can be written pretty easily. Obviously tweaking them and getting them all running together seamlessly is the real trick.

One of the things that makes recognising Sudoku puzzles an easier task than most image processing/recognition problem is that it is a highly constrained problem — a standard Sudoku puzzle is going to be a square grid and it will only contain the printed numbers 1–9. These two points are very important. The first point — it's a square grid — tells us what shape a puzzle is and what we should be looking for in an image. The second point — it will only contain the printed numbers 1–9 — tells us that we aren't going to need a sophisticated OCR system. When we look at the problem there's nothing that jumps out and says "nobody has solved this before — it's probably really hard." We can also add some additional assumptions:

- In a photograph of a sudoku puzzle, the puzzle is going to be the main/most important object on the page.
- A user is going to be photographing the puzzle — they aren't going to take a picture of a whole newspaper page, they won't be taking a photograph of a coffee shop and expecting us to find a sudoku puzzle that someone is playing four tables away. Also, the user is going to try and capture the whole puzzle, they won't miss a corner or chop off the top.
- The puzzle will be orientated reasonably correctly.

No-one (hopefully) is going to be taking a picture of an upside down puzzle, and typically they will be trying to align it nicely in the camera viewfinder so it is reasonably straight without too much distortion. So at the start of our Sudoku puzzle recognition we'd expect to be getting an image similar to the one in Figure 1.9.

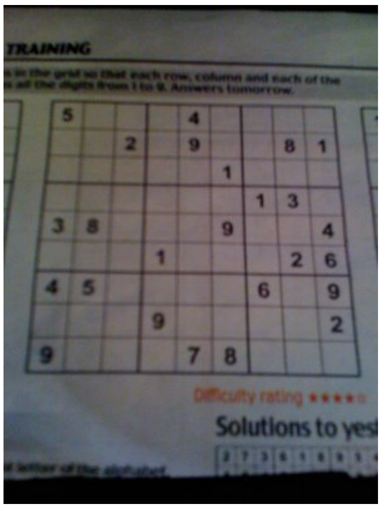


Figure 1.9: Sudoku grid captured from a newspaper

We can see that this meets all the assumptions above. We've got the whole puzzle — there are no bits missing, the puzzle is reasonably straight — it's not upside down or at some crazy angle, it's also the main thing in the picture — there's not a lot of distraction in the image, it's just a picture of a sudoku puzzle. So, how are we going to go about recognising this image? There are basically 4 main problems to tackle:

- Where is the puzzle?
- Once I've found the puzzle how do I turn it back into a square puzzle?
- I've got the puzzle — how do I find the numbers?
- How do I recognise the numbers?

Lets look at each of these in turn.

Where is the puzzle? The first thing to do in any image processing problem is to reduce the amount of data you are dealing with. We started from the full colour high resolution image. The first thing we can do is to throw away the colour information. Looking at our sample image, having colour does not add any information that is useful to us.

What else can we see? An obvious thing is that this is a printed page, it's basically a black and white image. So the first step in our image processing is to throw away even more information. We are going to threshold the image so that we have either background pixels (the paper) or foreground pixels (the printed elements).

There are a variety of thresholding techniques available to us:

http://en.wikipedia.org/wiki/Thresholding_(image_processing)

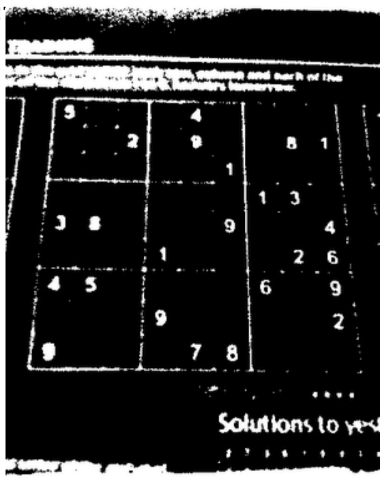


Figure 1.10: After thresholding

The initial naïve approach is the obvious one. Light pixels are the paper and dark pixels are the ink, so lets pick a number (say the average pixel value for the image) and anything less that that we'll set as foreground and anything higher than that is background. This would give us an image that looks like the one in Figure 1.10.

It's kind of OK — there's some paper showing up as foreground in the top left, but the puzzle is also showing up, but as we go down towards the bottom right the puzzle starts to break up. We can see that this simple approach doesn't really handle variable lighting on the page. And we can imagine that if there are any shadows on the page the results will be even worse.

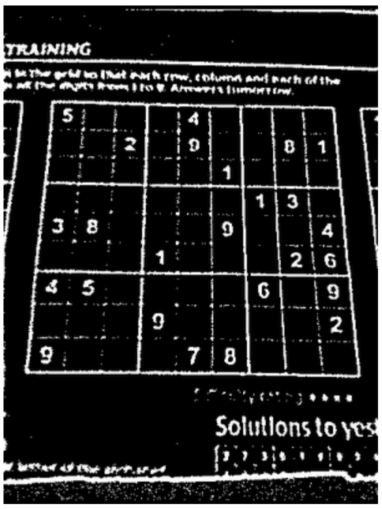


Figure 1.11: Tidying up the thresholded image

What we need is a thresholding method that can take account of this problem. My personal favourite is a simple adaptive threshold. For each pixel in the image take the average value of the surrounding area. If the pixel is less than 90% of this value then it is ink, if it is higher then it is paper. The reason for choosing 90% of the value is that this lets us filter out flat areas or areas that aren't changing very much (*i.e.,* blank bits of paper or solid black sections). This results in the image shown in Figure 1.11. As you can see it's a lot better than the previous attempt.

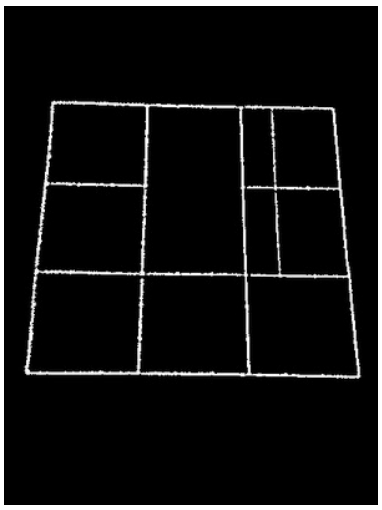


Figure 1.12: Finding the grid

We can now apply one of our assumptions: "In a photograph of a sudoku puzzle, the puzzle is going to be the main/most important object on the page." We can interpret this in the following way: the most important thing on the page probably has the most foreground pixels. So let's extract every blob of set pixels and see which blob has the most. That blob of pixels must be our sudoku puzzle. To do this we'll using a blob extraction algorithm. The simplest way of doing this is to scan through the image looking for a set pixel. Every time we find a set pixel we perform a "flood fill." The pixels that we fill in make up our blob. Running this process and then taking the blob with the largest number of pixels gives us the image in Figure 1.12.

So, that's great, we've managed to go from an picture of a puzzle in a newspaper to finding the pixels which are probably part of a Sudoku puzzle. Most importantly we have the pixels that make up the outside frame of the puzzle. Now what?

Ideally it would nice if we knew the coordinates of the corners of the puzzle frame — that would let use draw a box around it and know the exact location of it. There are quite a few ways that we could approach this. A simple approach might be to scan through the pixels looking for the top left, top right, bottom left, bottom right. That might work. I've chosen to use one of my favourite algorithms, the Hough transform (discussed later in the module). This can be used to detect straight lines (and many other shapes) in an image; rather handily, straight lines are what make up a sudoku grid. So lets feed our extracted grid pixels into a Hough transform. This gives us this Figure 1.13, which has the Hough transform mapped back into an image so we can see it.

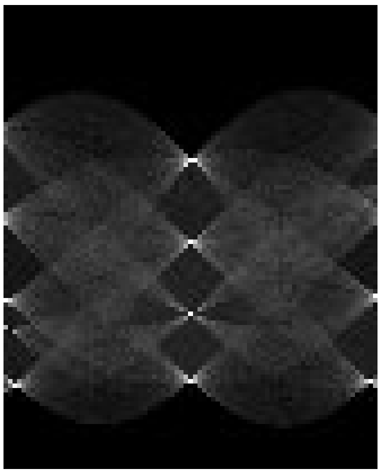


Figure 1.13: The Hough transform of the lines

This image represents all the possible lines that are in our image. The x coordinate is the angle of the line and the y coordinate is the distance of the line from the origin (I've trimmed the image to make it a bit smaller so our y coordinate looks a bit smaller than it would normally). We can see that we have a bunch of peaks around the 0 and 180 degrees positions (on the left and the right of the image) and a bunch of peaks in the middle of the image, around the 90 degrees mark. These correspond to horizontal and vertical lines in the image.

We really only care about the leftmost, rightmost, topmost and bottommost lines. These correspond to the peaks at the top and bottom of the two groups. If we take these peaks and turn them back into lines we have the lines along the top, bottom, left and right of the puzzle — find where they intersect and we have the corner coordinates of the puzzle. This lets us produce the image you can see in Figure 1.14. We now know exactly where in the original image — we have found the sudoku puzzle!

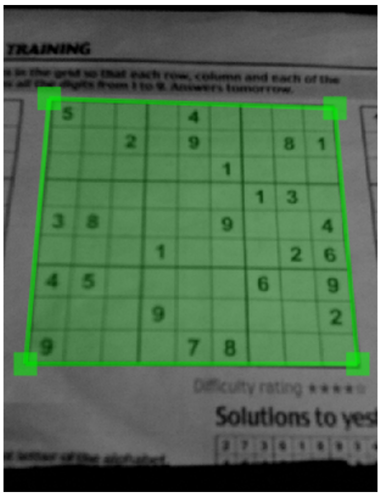


Figure 1.14: Hough features superimposed on the grid

Once I've found the puzzle how do I turn it back into a square puzzle? So, we've got the corner points of the puzzle — it's currently not really usable for much as it's a bit distorted. What we need is some way of mapping from the puzzle in the picture back into a square puzzle.

We need a transform that will maps one arbitrary 2D quadrilateral into another. For this we can use a perspective transform:

$$X = \frac{ax + by + c}{gx + hy + 1}$$
$$Y = \frac{dx + ey + f}{gx + hy + 1}$$

This will map a point given by x, y in one quadrilateral into a new point X, Y in another quadrilateral. As you can see there are eight unknowns in these two equations — but fortunately we have 8 values (the corner x and y coordinates of the puzzle and our arbitrary x and y corner points of our square image). Solving these equations gives us the a, b, c, d, e, f, g, h which provide us with a mapping to get our puzzle out nice and straight, as Figure 1.15 shows. A lot more information on this approach is on `http://alumni.media.mit.edu/~cwren/interpolator/` and the equations are discussed further in the notes on stereo vision. [*This link has disappeared since Chris wrote his notes.*]

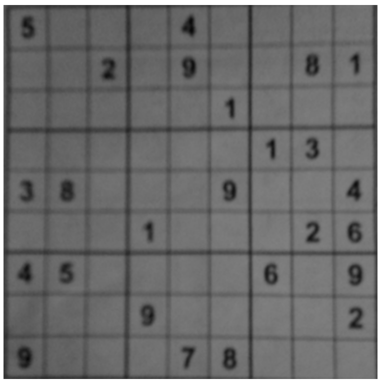


Figure 1.15: The image after rectification to a rectangle

I've got the puzzle — how do I find the numbers? So we've now got an undistorted square Sudoku puzzle. That's good, but it doesn't really help us that much — we could have got this far by making the user line up the puzzle with a square shape in the viewfinder when they took the picture!

Let's try and see which boxes in the puzzle actually have numbers in them. This is actually pretty straightforward, all we have to is divide the puzzle into a set of boxes, threshold each box and apply the blob extraction algorithm to the middle of the box. If we manage to extract a blob then it's more than likely that the box must contain a number. Throw away the empty boxes and you've got the numbers that you need to recognise.

How do I recognise the numbers? We can now take the blobs from the previous stage and try and work out what numbers they represent — this is where the world becomes your oyster. There are a huge number of techniques for performing OCR and a huge number of non-specific pattern recognition algorithms. For my implementation I chose to use a neural network. I collected a large number of extracted number images from some sample puzzles and hand classified them. I then used these to train a simple neural network to recognise the digits 1–9. This works remarkably well, but I suspect I am probably using a hammer to crack a nut in this instance... Anyway — this gives us our final result — a sudoku puzzle with recognised numbers (Figure 1.16)!

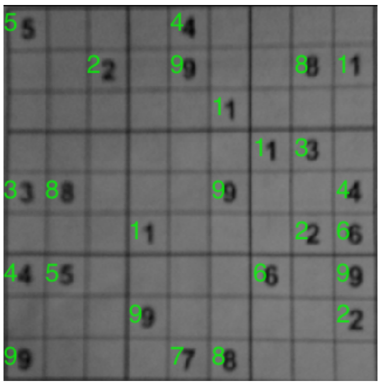


Figure 1.16: The final grid, with recognised numbers superimposed

There are a huge number of improvements that can be made to these basic steps. You can add intelligence at every step to improve your chances of recognising a puzzle. But basically, that's it!

1.7 About the module

The first thing to appreciate is that computer vision is far from being a solved problem. Research into it is extraordinarily active, with the UK contributing some of the most important results globally. The development of the discipline is so rapid that about half the content of this module couldn't have been taught as little a decade ago. Given what it is trying to do, you might think that vision is an area of artificial intelligence, placing it squarely in the realm of computer science. While this is true to some extent it is far from the whole story, with researchers also having backgrounds in disciplines such as electronic engineering, mathematics, physics, psychology and medicine. The contributions of the various disciplines will become more apparent as you learn about vision techniques.

The last decade or so has seen three major technological influences on computer vision. Firstly, *digital imaging* has transformed the capture of image and video data from something that pushed at the boundaries of real-time hardware and software into an everyday process. Secondly, *cheap data storage and processing* allows worthwhile quantities of image and video data to be stored and manipulated in reasonable timescales. The final influence, a consequence of having more CPU cycles to burn, is a huge expansion in the use of *machine learning* and an improvement in the learning algorithms available. Machine learning has actually been employed in real-world vision systems for a long time but it is now difficult to find vision applications the don't use machine learning. We shall explore some types of machine learning for vision tasks in the second half of the module. Real-world experience with machine learning is currently a highly-marketable skill, attracting good salaries from major international companies.

In fact, this is not one module but two, for it is followed by both final-year undergraduate students (CE316) and postgraduate students (CE866). Although most of the taught material applies to everyone, the assessments for the different groups is different and you will be reminded of this from time to time.

Learning outcomes. . . and afterwards

There are four learning outcomes for the module:

1. Describe the principles and main methods for computer vision.
2. Explain, on examples of visual data, how some methods facilitate aspects of two-dimensional vision.
3. Explain, on examples of visual data, how some methods facilitate aspects of three-dimensional vision.
4. Write computer programs to solve simple vision tasks.

The coursework and examination make sure you can do all of these.

You might think that a module such as this one is intended to give you knowledge about a subject. That's true of course but there is more to it. In order to award *accredited* degrees, the professional bodies that review them require us to move from spoon-feeding you content in year 1 to having you figure out how to solve problems yourself in year 3 — that is the underlying reason for all the team project modules you have been involved in as well as your individual project. MSc students are meant to be able to learn about new things with little external guidance.

Unlike many of my colleagues, the author has worked in industry and knows that a project often starts with your boss saying something like "Find out about this and tell me if it can be done, and if so how." Having found out how, you're then the poor sucker who has to implement it. The author is a great believer in preparing you for this kind of environment, and will do that through a combination of the lectures, the experiments and the examination. You have to play your part too though, trying to figure out what you know can help you solve problems you encounter.

Teaching materials

The notes you are now reading form the primary resource for the module. They present the underlying principles, show how things are calculated and discusses practicalities. Each topic, roughly a chapter in these notes, is backed up by a multiple-choice quiz which you take online and is automatically marked as you go through it. Each quiz has ten questions selected randomly from a pool, and the answers are juggled each time you take it. You can take these as many times as you like. People usually find these most useful for checking that they have grasped the ideas. They are thought to be a little easier than the multiple-choice questions used in the formal progress tests discussed below.

As indicated above, computer vision is an inherently practical subject so it essential that you learn how to use the principles and techniques described in lectures in practice. You will therefore carry out *a series of* *experiments*. These are examined in a pair of progress tests so it is essential you keep a record of what you did and what you found. If you have a long-held wish to learn LaTeX or to use `Jupyter` notebooks but never had a good reason to use them in anger, this is a good time to do so. However, a Word or OpenOffice document, a plain text file or a good old paper notebook are all equally good in terms of keeping a record.

Assessment

As mentioned above, the experiments are examined in two open-book progress tests, the first roughly half-way through the module and the second at its end; each test is worth 20% of the overall module mark. Note that the progress tests are different for CE316 and CE866 students. During the progress tests you are expected to refer to your records of the experiments, which focus on these specifically rather than on the lectured content.

The remainder of the assessment, 60% of the total module mark, is via a conventional examination which takes place in the summer term. Again, this is different for CE301 and CE866 students.

Software

In order to do the practical work discussed above, you need a computer and some vision software. The particular combination that is becoming most popular is OpenCV, which was written in C++ because it is intended for real-time processing but has now sprouted “wrappers” for Python. The particular image representation used is compatible with the numpy (“Numerical Python”) and scipy (“Scientific Python”) extensions, and with the machine learning capabilities of Scikit-learn, TensorFlow and other packages. OpenCV is not an application like Photoshop but a library of routines that you call from your own programs. OpenCV has wrappers for other languages too but using the Java wrappers is notoriously tricky, while programming things up in C++ *ab initio* is time-consuming; that is why Python is recommended. Pretty much all the code written during lectures is in Python. In any case, Python has become the *de facto* standard language for gluing together calls to standard libraries, widely used in high-tech industries.

You should become familiar with OpenCV as you do the experiments. It works fine under Windows, macOS and Linux, and recipes for installing it under your operating system of choice are given in Section 1.8. It is also available on the Horizon portal in CSEE. It normally works better in the Unix (Linux, macOS) world than in Windows, reflecting the operating systems on which it is most heavily used: a large proportion of vision researchers and developers — in fact, a large proportion of researchers in most of science and engineering — use Linux. This is partly because it tends to be better at squeezing the last ounce of performance from commodity hardware, important if your machine learning algorithm takes literally weeks to run or want your application to work on low-end systems. However, many of us find that the command line is a flexible and powerful way of controlling a computer and we look down on people wedded to pointy-clicky interfaces.

If your Linux skills need improvement, here are some places to look:

- The author's teaching material for CE222: *Operating Systems*. Drop him an email if you would like to be able to access it.
- A Linux tutorial, written by Michael Stonebank at Surrey.
- A a series of tutorials about the Unix shell. Incidentally, you can drive macOS with *exactly the same commands* as Linux.

1.8 Installing OpenCV on your own machine

You can install OpenCV on your own computer, either under Unix (macOS, Linux, *etc*) or Windows. As OpenCV is being developed so quickly, **there are significant differences between versions** so programs written for a recent version may well not work with older ones and *vice versa*. It makes sense for you to install the same version on your machine as is used in CSEE's teaching labs, and the easiest way to find out the version you are using is to print it out via the Python interpreter:

```
$ python3
>>> import cv2
>>> print cv2.__version__
4.6.0
>>>
```

You'll do this in the first experiment. Note that the version number printed out here is the one on the author's machine at the time of writing these notes; the version in the software labs may not be the same.

Under Linux (Ubuntu 20.04), I found the easiest route was to install Python 3 and `pip`, its package installer, then type the shell incantations:

```
   pip3 install numpy
   pip3 install opencv-python
   pip3 install opencv-contrib-python
```

If memory serves me right, this also installed `scipy` and `matplotlib`; but if it doesn't, it is easy enough to install them yourself with similar commands to the above. If using `pip` worries you, the one-liner

```
  sudo apt install python3-opencv
```

may suffice. The author is pretty sure this worked under Ubuntu 22.04 too.

Under macOS, I installed Python 3 using Homebrew then typed the same `pip3` commands as for Linux, except that I installed

```
   pip3 install opencv-contrib-python-headless
```

This "`headless`" version avoids installing old versions of some libraries used by OpenCV. If you install "`headless`" and find things are missing, de-install it and then install `opencv-contrib-python` as for Linux.

To be able to use `xv` and other Linux-like graphical utilities mentioned in the laboratory scripts, you need to download and install XQuartz, Apple's

X-server. You need to start XQuartz running before using `xv`; the author has it started automatically whenever he logs in. The source code of `xv` is on Github and is easy to build yourself if you install Homebrew and then `cmake`; speak you your friendly neighbourhood lecturer during a lab session if you run into problems installing it.

Under Windows, although I haven't done this myself, you should be able to install Python3 and `pip`, then follow the same route as for Linux.

If you try this and run into problems, do speak to a demonstrator during a lab session or again speak to your friendly neighbourhood lecturer. If you find one of the recipes above is no longer right and you have updated instructions, please let me know so that I can amend these notes. You will get an acknowledgement!

1.9 Further reading

In such a fast-moving subject area, printed textbooks are almost always out of date — and this is especially the case in the use of machine learning in computer vision. That is why these notes provide a comprehensive overview of the material. A couple of widely-recommended texts are:

- Richard Szeliski's book *Computer Vision: Algorithms and Applications* (Springer, 2010).
  This is intended for graduates with some vision experience; it is not really suitable for newbies.

- Roy Davies' book *Computer and Machine Vision: Theory, Algorithms, Applications, Learning*, (5th edition, Academic Press, 2017).
  This is a good book but unfortunately one you would have to buy. For people on campus, both it and the third edition are in the Library. It would be fine to use the older edition.

Regarding computer vision programming, the best places to look are:

- Google. Whatever you're trying to do and are struggling with, someone has probably ran into the same difficulty. You'll find useful questions and answers on sites such as Stack Overflow. Remember though that **you should not paste code directly from such places into your coursework without attribution** as that is plagiarism, an academic offence.

- The official OpenCV documentation. Be sure to look at the right version. A recipe for finding the version is shown in the previous section.

- Jan Erik Solem's book *Programming Computer Vision with Python* (O'Reilly, 2012). Rather than OpenCV, this book explains how to use Python, numpy and scipy to carry out computer vision tasks. I must admit to rather liking it.

You might also be interested to look at some useful online resources:

- CVonline, a collection of tutorials and papers which explain techniques;

- HIPR2, a compendium of interactive demonstrations of mostly low-level vision techniques.

2 The Human Visual System

Before considering computer vision in any detail, it is instructive to look at some of the capabilities of the human visual system. We shall start with the eye itself, which is fairly well understood, and then move on to what we know of the processing performed by the brain. ***Note that the content of this chapter is to aid your understanding; it is not examinable.***

2.1 The eye

Optically, there are close analogies between a camera and the eye. The front of the eye is covered by the *cornea*, which performs about 80% of the focusing that is needed. Behind that is the *lens*, whose shape and hence optical power can be varied a little by ciliary muscles, in much the same way that a camera can be focused. This focusing ability is known as *accommodation*. A young individual with no optical defects can focus easily down to a *near point* of about 10 cm, though as this individual ages, accommodation degrades and the near point moves further away — this is known as becoming *long-sighted* and its medical term is *presbyopia*. People who cannot focus on infinity — a common trait among people who have done a lot of close work when young — are *short-sighted* (the medical term is *myopia*) and generally wear spectacles or contact lenses all the time.

Around the lens lies the *iris*, which controls the area of the lens through which light is able to enter, commonly known as the *pupil*. Pupil sizes generally lie in the range 3–7 mm and correspond to about a five times change in pupil area and hence the amount of light allowed into the eye.

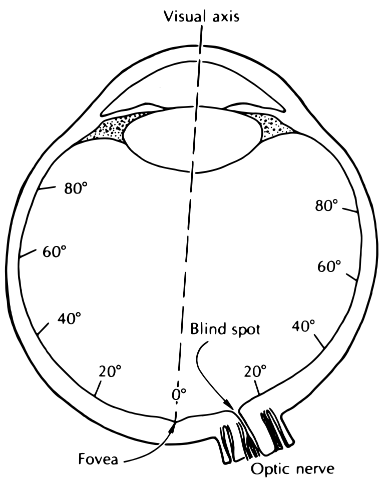


Figure 2.1: A cross-section through the human eye (from [Cornsweet, 1970] p137). The angles marked around the retina correspond to the angles in Figure 2.4.

The back of the eye, the *retina* (Figure 2.1), is covered with a layer of light-sensitive cells that is about the thickness of tissue paper. There are two main types of cells, known as *rods* and *cones*. The rods are responsible for vision in low light ($< 1\,\mathrm{cd/m^2}$), while the cones require more light to work. It is generally believed that there are three types of cones, having pigments that are more responsive in the red, green and blue parts of the spectrum (Figure 2.2), and this is what gives us colour vision. The red-, green- and blue-sensitive cones occur roughly in the ratio 12 : 6 : 1, so the eye is significantly less sensitive to blue light than to green or red, as Figure 2.2 shows. It is postulated that this is because humans evolved in a region where the staple diet was red berries growing amongst green foliage, though there is no way of verifying this. The presence of three different types of cone explains why some people are colour-blind: *e.g.,* there is a minor defect on the male chromosome that affects the pigmentation of the red- and green-sensitive cones in about 10% of men.

The rods are more sensitive in the green–blue part of spectrum than cones (Figure 2.3), presumably because of the need to retain some vision functionality in moonlight, which is bluish in colour. This also explains why ships normally illuminate their bridges with red light at night: it avoids destroying the dark-adaption of the rods.

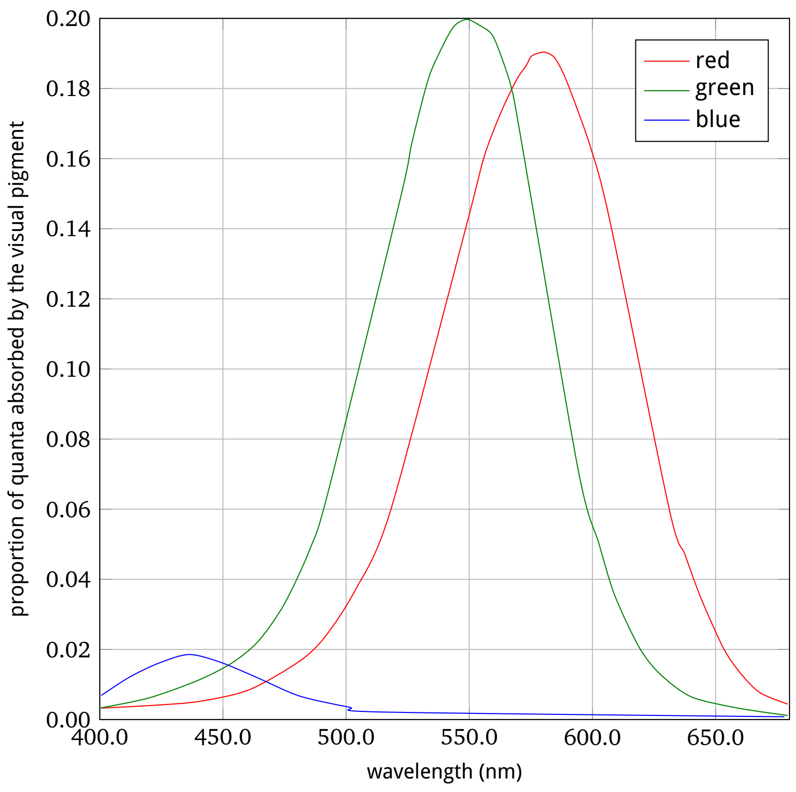


Figure 2.2: The relative spectral sensitivity of cones (adapted from [Cornsweet, 1970] p171, where it is reported that the subject from which these measurements were taken was about three times more blue-sensitive than an average subject).

The eye contains about 50,000 cones and about 120 million rods, about twenty times more — though groups of about 120 rods are 'wired together' into a single ganglion cell (see below) to amplify the faint responses that result from low illumination, while only about 6 cones converge into one ganglion. The distribution of rods and cones over the eye is far from uniform. Both rods and cones are roughly circular in cross-section and hence tend to pack together somewhat more like hexagons then rectangles — unlike, say, the sensors on most CCD cameras. In humans and most other predators, the cones are concentrated into a region known as the *fovea centralis* (Figure 2.1 and 2.4), which is almost rod-free. (Some birds, for example, have more than one fovea.) In humans, the fovea subtends about 2° of visual angle and contains about 1% of the cones. A higher concentration of sensors means that the resolving power is higher, *i.e.* that greater detail can be perceived, and that colour discrimination is better. Without realising it, people turn their heads and eyes so that incoming light is focused on the fovea and hence the perception of detail (and colour) is greatest (*fixation*). The better light-gathering efficiency of the rods around

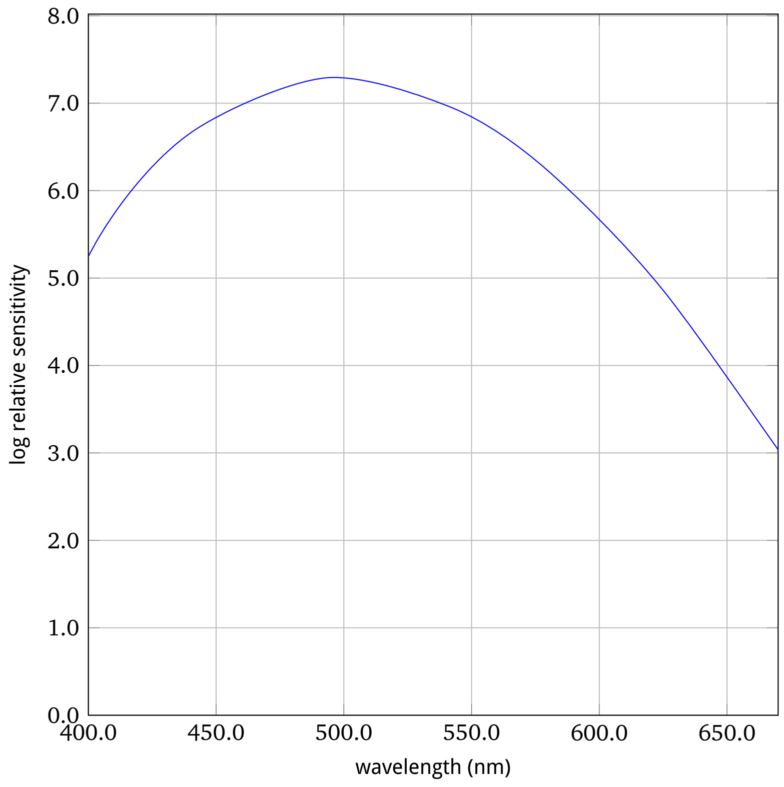


Figure 2.3: The spectral sensitivity of rods (adapted from [Cornsweet, 1970] p146)

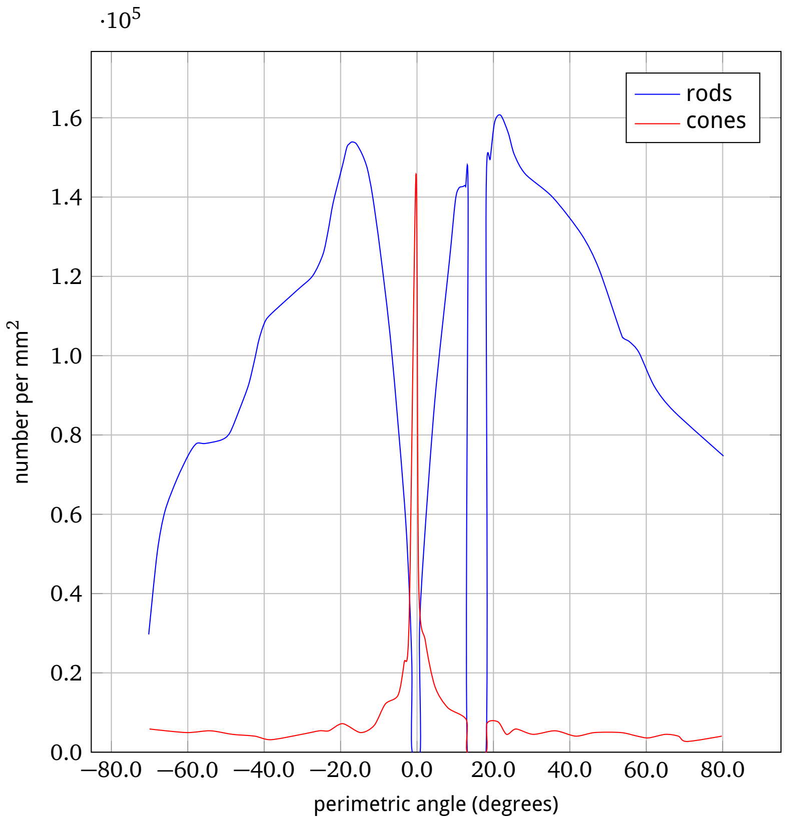


Figure 2.4: The distribution of rods and cones on the retina (adapted from [Cornsweet, 1970] p137). The angles correspond to those marked on Figure 2.1.

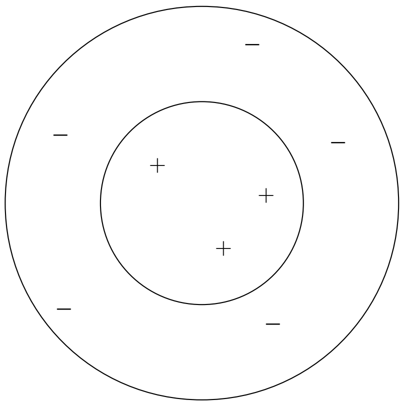
the periphery of the fovea is the reason that star-gazers are recommended to look 'above' stars rather than directly at them. Moreover, when one goes into the dark, it takes cones about six minutes to become maximally sensitive and rods about 30 minutes — and this is one reason that star-gazing is more effective in places without street lights and well away from roads.

2.2 From the eye to the brain

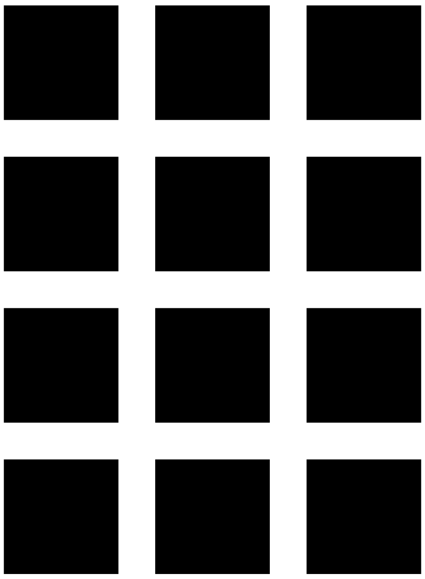
The electrical signals generated by photons hitting the rods and cones flow through four different types of cells (*amacrine*, *bipolar*, *horizontal* and *ganglion* cells), and the axons of the latter form the *optic nerve* which runs from the eye into the brain. The place where the optic nerve leaves the eye contains no rods or cones and hence leaves a blind spot (Figure 2.4). Fortunately, this is not apparent to us, partly because of fixation and partly because the brain appears to interpolate over the blind spot.

The way in which the signals from the retina converge into a ganglion cell provides an excitatory centre and inhibitory surround (Figure 2.5(a)) which, as we shall see when we consider convolution, is similar to some image operators. This phenomenon explains the *Hermann grid* (Figure 2.5(b)) in which grey 'ghosts' appear in the intersections of white 'corridors,' and Mach banding, an illusion that affects shaded objects in computer graphics near to illumination edges. The same effect causes *simultaneous contrast*, in which the apparent brightness of an area is affected by the brightness of its surroundings, and is thought to help *continuity*, where shapes that have incomplete boundaries are 'filled in' by the brain.

The optic nerves travel principally from the eyes to regions of the brain known as the *lateral geniculate nucleus* (LGN), though about 10% go to the *superior colliculus*, which controls eye movement. The LGN also receives signals from other parts of the brain, such as the *visual cortex* and the *thalamus*. There is one LGN in each half of the brain, and each of them comprises six distinct layers. Each eye sends half of its signals to the left LGN and the other half to the right LGN. The layers of the LGN retain the relationship between rods and cones on the retina, so we can think of it as working on images.



(a) The excitatory centre is surrounded by an inhibitory region



(b) One consequence of this is the Hermann grid, where grey spots are apparent at the junctions of the white lines

Figure 2.5: The excitatory–inhibitory nature of ganglion cells and its consequences

2.3 Inside the brain

About 1.5 million axons travel from the LGN to the region of the *visual striate cortex* known as *V1* (again, there is one in each half of the brain). The cortex contains about 250 million neurons and is responsible for most of the higher-level processing in the human visual system. Cells in the V1 region respond to specific aspects of images, such as orientation, edge and motion (Hubel and Wiesel won the Nobel prize for identifying this in 1950); for this reason, cells in the striate cortex are often known as *feature detectors*, and this is one of the reasons that vision researchers have expended a great deal of energy in developing edge, corner and feature detectors. Clearly, just like the LGN, cells in this region must relate to specific points in the retina. Interestingly, 8–10% of the cells are connected to the fovea even though it corresponds to about 0.01% of the number of cells on the retina. Each V1 transmits information to two pathways:

The dorsal stream goes through visual area V2, on to V5 and thence to the posterior parietal cortex. This stream is sometimes called the "where pathway" or "how pathway" as it is associated with motion, the representation of object locations, and control of the eyes and arms.

The ventral stream also goes through visual area V2 then through visual area V4 and on to the inferior temporal cortex. This stream is sometimes called the "what pathway" as it is associated with form recognition, object representation and the storage of long-term memory.

The four regions of V2 appear to produce a map of the visual world. As in V1, cells are tuned to properties of the visual field such as orientation, spatial frequency, and colour; however, the responses of many V2 neurons are also modulated by more complex properties, such as the orientation of illusory contours and whether the stimulus is part of foreground or background.

Beyond V2, detailed knowledge of the processing is less certain. It is thought that V3 plays a rôle in the determining global motion in the visual signal while V4, like V1, is tuned for orientation, spatial frequency, and colour. However, V4 is tuned for more complex object features than V1, for example simple geometric shapes (but not faces). V5 is a region of *extrastriate visual cortex* which is thought to play a major rôle in the perception of motion, the integration of local motion signals into global percepts and the guidance of some eye movements.

There is some knowledge of where the processing of particular types of features takes place in the brain, usually acquired from studies of people who have suffered oxygen starvation in strokes or road traffic accidents, resulting in the death of specific brain regions. For example, the ability to store and recognise faces appears to be centralised in one region of the brain, and similarly the conversion of 2D knowledge of shapes into 3D recognition appears to take place in a specific region. There are theories as to some of the mechanisms involved but it is fair to say that we are far from a complete understanding — certainly too far for us to develop computer vision systems simply by simulating what happens in the brain. There are a few Nobel prizes to be won before that becomes a realistic possibility.

3
Vision Software

This chapter first discusses the nature of pixels and images and describes the most common representations of colour. It then shows how images can be manipulated by software. A program that calculates some simple statistics and a histogram are presented and the result of using it interpreted. Finally, a complete system that uses colour histograms to recognize different types of produce is presented.

3.1 Introduction

In order to do any computer vision, you need some software. There is actually a lot of computer vision software on the Web, though much of it is quite difficult to get to grips with. One of the easiest to use is a GUI-driven package called ImageJ. Freely available and written in Java (so it runs on Windows, Linux and the Mac) and intended for use in the biomedical area, ImageJ is best thought of as a kind of 'photoshop for scientists.' If you want to play with some of the simpler operators to get a feel for what they do, ImageJ is ideal for the job. It can also be used for serious science, and it supports plug-ins that allow its functionality to be extended; with these, some fairly sophisticated things can be achieved.

Most people working in computer vision use one of two environments, Matlab or OpenCV. Matlab is commercial software; although it is used in some modules in CSEE, the author prefers to steer clear of it. As mentioned in Chapter 1, OpenCV is a library of routines which you call from your own code to perform computer vision tasks. It is written in C++ with 'wrappers' for other languages, and Python is probably the most widely-used one. OpenCV is intended for real-time applications (so Python is arguably not the best language to be using) but its ease of use and expressiveness make it easy to produce applications. You are meant to make use of OpenCV in your experiments to gain practical experience with it.

Although OpenCV is widely used in research and development, you need to realize that many of the algorithms it uses have been tweaked to run quickly. In order to check that its results are as expected (and to find where they are not, the author usually illustrates these lecture notes with code that doesn't make use of it, instead relying on the capabilities of only numpy and scipy. This non-OpenCV code is bundled into a Python module called EVE (for *Easy Vision Environemnt*) and you can pick up a copy of it from the web. Note that he fiddles with it on an almost daily basis.

		x increasing →							
← y increasing	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	255	0	0	0	0
2	0	0	0	255	255	255	0	0	0
3	0	0	255	255	255	255	255	0	0
4	0	0	0	0	255	0	0	0	0
5	0	0	0	0	255	0	0	0	0
6	0	0	0	0	255	0	0	0	0
7	0	0	0	0	255	0	0	0	0
8	0	0	0	0	255	0	0	0	0
9	0	0	0	0	0	0	0	0	0

(a) Values of pixels

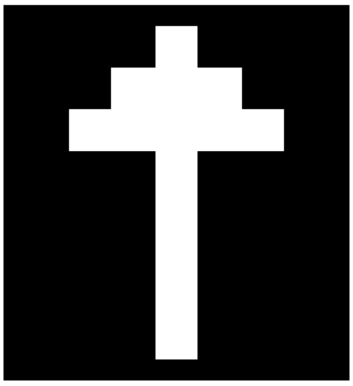
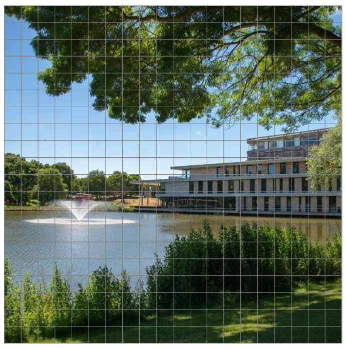

(b) Resulting image

Figure 3.1: A digital image of an upward-pointing arrowhead

3.2 Images and pixels

Before setting out on our exploration of computer vision software, it is important to have a grasp of some of the fundamentals. For that reason, let us start by considering what is meant by an image: for it to be processed by a computer, a good working definition is *a regularly-sampled, rectangular grid of pixels* — see Figure 3.1. (You'll see why the white values are all 255 shortly.) The "rectangular" part means that pixels are arranged as rows and columns in a grid as opposed to hexagons, even though the latter organisation is better at tessellating a plane: although a few people have explored hexagonally-sampled images, they are much difficult to work with and don't appear to yield any additional insights. The "regularly-sampled" part means that pixels are all the same size and shape, and are equally spaced along the lines that make up a digital image. We shall generally assume that pixels are square, the same dimensions in the x and y directions, as illustrated in Figure 3.2.


(a) Original image with pixels marked

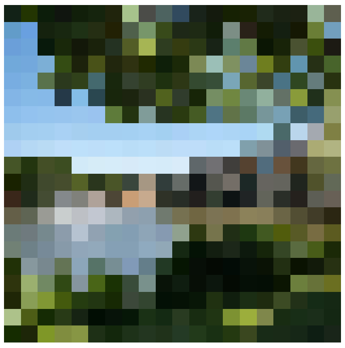

(b) Resulting pixels

Figure 3.2: Each pixel in an image is the average value of a region of the real world image

Figure 3.1 shows an image of an arrowhead that points vertically upwards; note how the origin is at the top left corner and that x increases along the lines while y increases down the columns. The x and y indices into the table start at zero in deference to C-like programming languages. The figure also shows how those values appear when displayed as an image.

This definition of an image means that we also need to be sure we know what is meant by 'pixel.' Most people think of a pixel as a sampled 8-bit value, *i.e.,* one with a value in the range 0–255. The '8 bits' results principally from being good enough to fool the human eye — it can identify about 64 different shades of grey — and representable in a single byte; good quality cameras and scientific sensors now almost always digitise pixels to 16 or more bits so this notion that pixels fit into 8 bits is slowly disappearing.

Representing pixels and images

Critical to any software is the set of data structures it uses — and for computer vision, this means the representation of pixels and how they are combined into an image. OpenCV works almost exclusively with integer pixels of various precisions, on the basis that integer operations are faster than floating-point ones. (They aren't always, it depends on your hardware.) Conversely, Python's numpy and Matlab favour floating-point numbers, even though images are naturally digitised into integer values. This is because many things one does to an image naturally yield floating-point numbers and, more critically, because it avoids having to worry about rounding except when writing out pixels. If one uses an integer representation for pixels, rounding has to be considered *every time* a pixel is manipulated. This is onerous if one has to do it consistently, and can lead to a loss of precision. The author contends that it is better to get an accurate result slowly than an inaccurate one quickly! In practice, OpenCV requires its images to be have particular representations for some routines and this will bite you from tine to time — this is why EVE uses a consistent representation throughout.

In Python, you might think that the natural way to represent an image is as a series of linked lists, with each line of the image being a separate linked list and all the lines combined into another linked list. You would then be able to write code like

```
arrowhead[1][2] = 255
```

Unfortunately, this is very slow to run. For images of the size captured by modern digital cameras, the time taken to walk along the linked lists to locate the right pixel quickly dominates processing. A more appropriate representation is needed, and in the Python world this is offered by the numpy extension. It provides arrays, chunks of memory which are indexed efficiently by sets of subscripts, so that

```
arrowhead[1,2] = 255
```

is as quick to execute as it can be (in Python). Note the difference in the way subscripts are written for a list of lists and a numpy array.

Compiled languages such as Fortran (much maligned by computer scientists of the past but now a really impressive language featuring classes, operator overloading and whole-matrix operations that work well on multi-processor hardware) have multi-dimensional arrays built into them. C (and hence C++) doesn't support them natively but, with a little care, one can create structured data types or classes that are essentially as efficient as those in Fortran. The same is true for Java: they are not quite supported natively but the implementation is pretty efficient.

The basic way of representing a monochrome (grey-scale) image is as shown in Figure 3.1: a matrix (a 2D array) of values. When we talk about the pixels in that arrowhead image, one might say that "the value of the pixel at $(4, 1)$ is white", using the x and y locations to identify the pixel in question. However, the indices of a matrix are *row and then column*, so that pixel will be indexed in Python as `[1,4]` rather than `[4,1]`:

```
>>> print (arrowhead[1,4])
255
>>> print (arrowhead[4,1])
0
>>>
```

It is very important that you get to grips with this. The code presented later in this chapter should give you further insight. However, before we start looking at code, it makes sense to understand how colour images are represented.

Representing colour

Colour images from cameras normally represent pixels as a combination of red, green and blue, so-called RGB images. These are consistent with the types of cone in the human eye (Section 2.1). Yet RGB is by no means the only colour representation. Images destined for printing are based around the 'subtractive primaries' of cyan, magenta and yellow inks (the complementary colours of red, green and blue respectively) with black regions being darkened by the addition of black ink — this is known as the CMYK colour model. If you look at a colour image in a newspaper (Figure 3.3), you'll see there are separate cyan, magenta, yellow and black dots.

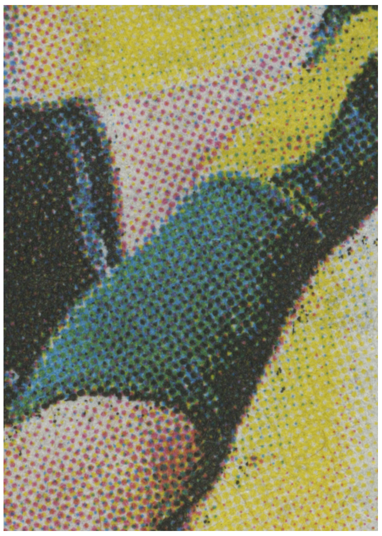


Figure 3.3: Magnified newsprint, showing that an image is made up of dots of coloured ink

Another common image representation is YIQ, used in NTSC analogue colour television, where the Y channel is grey-scale (which is what black-and-white televisions display) and I and Q are two colour difference signals which are processed by the television receiver to reconstruct RGB:

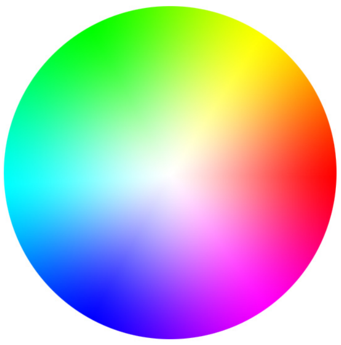
$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.600 & -0.274 & -0.321 \\ 0.211 & -0.523 & 0.311 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

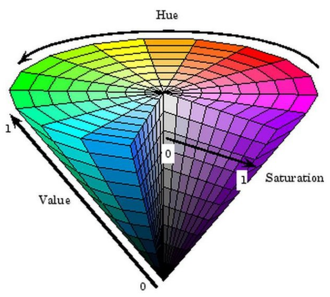
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.956 & 0.619 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.703 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

The colour difference signals are actually sampled at half the rate of the Y signal because the eye is less sensitive to changes in colour than to changes in intensity (see Chapter 2). This same phenomenon is also exploited in a similar way in the JPEG image format, another reason for it being a poor choice for computer vision.

A colour model that is better-suited to analysis is HSV ('hue–saturation–value'), which introduces terms akin to those that an artist might use when painting [Smith, 1978; Foley et al., 1990]. The best way to think of HSV is to take a cube with the R, G and B values running along three sides and then to cut the top off the cube, giving a hexagonal shape filled with the colours in the colour wheel (Figure 3.4). If R, G and B are the red, green and blue values of a single pixel, they can be converted to HSV using the following series of transformations. Firstly, calculate the maximum and minimum of R, G and B:

$$V = \max(R, G, B)$$
$$m = \min(R, G, B)$$


(a) A colour wheel, showing the additive primaries, their combinations and complementary colours



(b) HSV colour representation

Figure 3.4: Representing colour as hue, saturation and value (taken from a paper)

From this, calculate the chroma, $C = V - m$. Then calculate the hue as

$$H = 60^\circ \times \begin{cases} \text{undefined} & C = 0 \\ \frac{G-B}{C} \mod 6 & V = R \\ \frac{B-R}{C} + 2 & V = G \\ \frac{R-G}{C} + 4 & V = B \end{cases}$$

to give H in degrees. We calculate S as

$$S = \begin{cases} 0 & C = 0 \\ \frac{C}{V} & \text{otherwise} \end{cases}$$

There is a single OpenCV routine that converts between different colour models.

All of the above colour representations are known as ‘uncalibrated’ because they are based purely on the data coming from the camera. There are also ‘calibrated’ colour spaces (with names like XYZ, Lab, and Luv), some of which make a fair job of matching the perceptual differences detected by the human visual system — but calibrating a capture system is far from straightforward. There is not time to cover all of these, and they are somewhat specialized, but you should be aware that they exist.

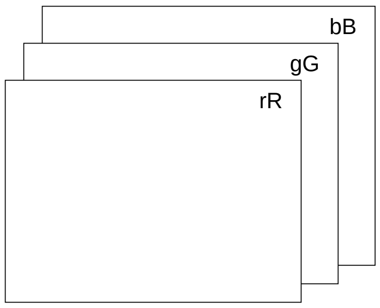
It is often easiest to think of a colour image as a series of colour “planes,” as shown in Figure 3.5(a). Here, `r` and `R` represent the last two pixel values of the first line of the red channel, `g` and `G` those of the green channel and `b` and `B` those of the blue. If we represent the colour channel by `c` then, bearing in mind the discussion above, it would be indexed as

```
image[c,y,x]
```

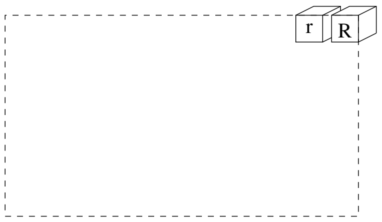
However, this is not the representation we have ended up using for images in Python/numpy, OpenCV or Java; instead, it is Figure 3.5(b) in which the `y` and `x` subscripts index the row and column in the image and `c` then indexes the channel:

```
image[y,x,c]
```

Perhaps the best way to fix this representation in our minds is to see it used in real code, so let us now do that.



(a) A colour image can be thought of as comprising red, green and blue channels



(b) The representation of a colour image in OpenCV and `numpy`, a 2D array of 1D pixels

Figure 3.5: Representations of colour images in software

3.3 Coding up algorithms

Calculating the mean of an OpenCV image

The computer vision equivalent of a “hello world” program is arguably one that calculates mean of an image. Getting it working involves not only getting the hang of editor and compiler (or IDE) but also reading in images, so you have to get to grips with quite a lot of ‘infrastructure’ to get it working.

The mean (“average”) of a set of numbers is found by adding them up and dividing them by the size of the set. If `im` is a colour OpenCV image represented in Python as a numpy array, then suitable code is:

```
def mean (im):
    "Calculate the mean of the image im."
    ny, nx, nc = im.shape
    total = 0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                total += im[y,x,c]
    return total / ny / nx / nc
```

The `def` line declares a procedure. The remainder of the line lists its parameters and the trailing colon introduces the procedure body. The quoted string provides minimal procedure documentation, and the remainder of the code is the procedure body. Indentation is used to indicate loop structure. The `range` function provides a list of integers in the range zero to (but excluding) its second argument, and the other lines of code accumulate the sum of all the values in the image in `total` and then divide it by the number of pixels in the image. The mean is returned as the value of the function.

The equivalent C++ code is quite similar:

```
// ave -- return the mean of all pixels of an image
double ave (Mat im)
{
  int ny, nx, nc, y, x, c, v;
  double total = 0.0;
  Vec3b pixel;

  ny = im.rows;
  nx = im.cols;
  nc = im.channels();
  for (y = 0; y < ny; y++) {
    for (x = 0; x < nx; x++) {
      pixel = im.at<Vec3b>(y, x);
      for (c = 0; c < nc; c++) {
        v = int (pixel[c]);
        total += v;
      }
    }
  }
  return total / ny / nx / nc;
}
```

This C++ code is interesting. My natural way of working with images is to use `float` representations because I need the dynamic range of floating-point representations but not the extended precision of a `double`. My first attempt at this code declared `total` as `float` *but the resulting code yielded the wrong value for the mean!* I had to remind myself what was wrong with it. . . if you think you know why, do speak to me in a lab session about it. There are no marks for understanding why but you will get a significant number of brownie points if you can figure it out correctly.

with roughly a one-for-one mapping of lines of Python code to C++ code. To cope with the fact that the image may be of any data type (`unsigned char`, `int`, *etc.*), templated code can be used to return the values of all channels of a pixel, then the innermost loop steps along that; this is hidden behind the scenes in Python.

As with all programming languages, the order of the loops is important when accessing image data. The order used here is the most efficient in Python and languages derived from C (C itself, C++, C#, Java), where the *last* subscript needs to vary fastest. For languages with a different heritage (*e.g.,* Fortran, Matlab), the *first* subscript should vary fastest. Why should this be so? It is because of the order in which the array elements are stored in memory: accessing sequential memory locations maximises cache hits, reduces page faults, and so on.

3.4 Histograms and their manipulation

Having seen how to work through the pixels of an image, let us move on to what is probably the most fundamental way of characterizing an image, its *histogram*. This is a graph, usually drawn as a bar-chart, which shows how many pixels have each possible grey level value. With just a little experience, a glance at a histogram will tell a person whether an image is under- or over-exposed (which is why good digital cameras are able to show them), how best to threshold an image, and so on. For example, three similar photographs that demonstrate under-, over- and good exposure are shown in Figure 3.6 along with their histograms. It is clear from the figure that under-exposed images have a preponderance of pixels with low grey-level values while over-exposed images have many pixels with high grey-level values. Images that are exposed correctly tend to have more pixels corresponding to mid grey-level values and fewer pixels near the extrema of the histogram.

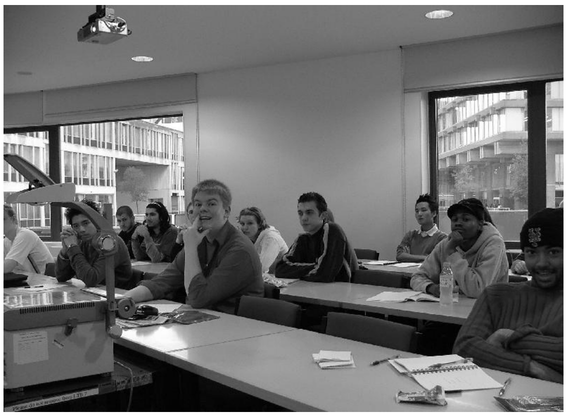
How does one compute a histogram? Clearly, when a pixel has the value v, some corresponding counter needs to be incremented by unity. The easiest way to do this is to have an array of the same size as the number of possible grey levels, then to use v as an index into it. Beyond that, the algorithm is similar to the one for calculating the mean in the way it accesses pixels.

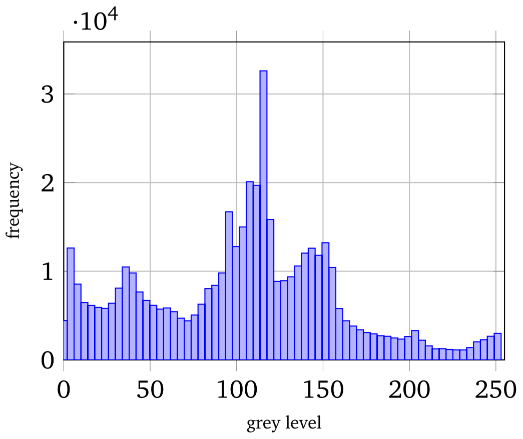
A suitable Python routine is presented in the following complete program. You might be interested to learn that it is presented using a "literate programming" idiom [Knuth, 1984]. The author has written software which pulls the program source code out of the (LaTeX) source of this document, so that the code you read here is *guaranteed* to be identical to the program one executes.

It is suggested that you study this program carefully: as well as having a routine to calculate the histogram, it also has code to plot it *and* display the image. The code is clever in that it re-sizes large images down to a reasonable size before display; you may need to do similar re-sizing yourself at some point. Finally, the code has no knowledge of the filename built into it, instead taking one or more file filenames from the command line — you will see this approach used a lot in the laboratory sessions.

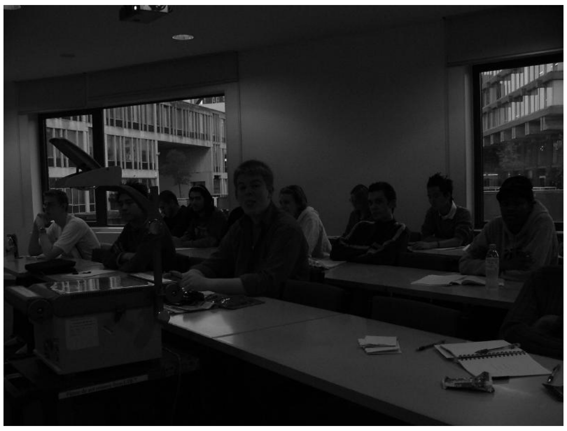
⟨*summarize.py*⟩ ≡

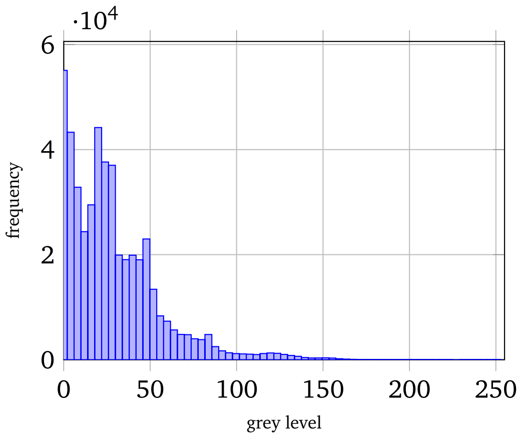
```python
#!/usr/bin/env python3
"Summarize the content of images by showing their statistics and histogram"
import sys, cv2, numpy, pylab

#-------------------------------------------------------------------------------
# Routines.
#-------------------------------------------------------------------------------
def iround (x):
    "Convert x to the nearest integer."
    return int (round (x))

```


(a) Well-exposed

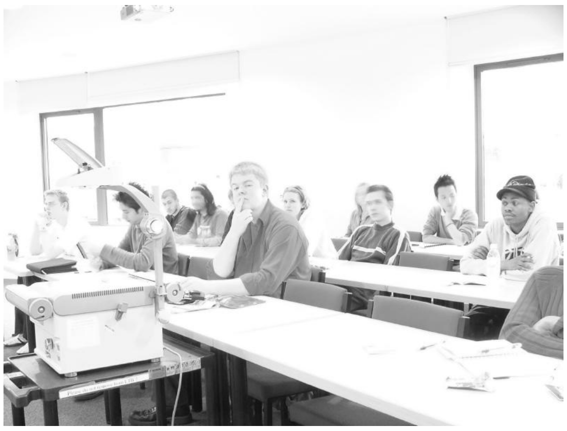


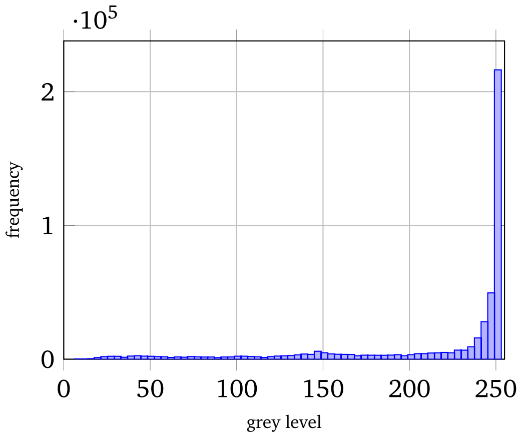
(b) Histogram of (a)


(c) Under-exposed



(d) Histogram of (c)


(e) Over-exposed



(f) Histogram of (e)

Figure 3.6: Histograms corresponding to differently-exposed images

```
def mean (im):
    "Calculate␣the␣mean␣of␣the␣image␣im."
    ny, nx, nc = im.shape
    total = 0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                total += im[y,x,c]
    return total / ny / nx / nc

def hist (im):
    "Return␣the␣grey-level␣histogram␣of␣an␣image,␣ready␣for␣plotting."
    # The maximum image value that we support.
    maxgrey = 256

    # Create the values for the abscissa (x-axis).
    ab = numpy.ndarray (maxgrey)
    for i in range (0, maxgrey):
        ab[i] = i

    # Create the histogram array and set it from the image.
    h = numpy.zeros (maxgrey)
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = im[y,x,c]
                h[v] += 1

    # Return the x and y values.
    return ab, h

def plot_hist (x, y, fn):
    "Plot␣the␣histogram␣of␣image␣fn."
     # Set up pylab.
    pylab.figure ()
    pylab.xlim (0, 255)
    pylab.grid ()
    pylab.title ("Histogram␣of␣" + fn)
    pylab.xlabel ("grey␣level")
    pylab.ylabel ("number␣of␣occurrences")
    pylab.bar (x, y, align="center")
    pylab.show ()

#-------------------------------------------------------------------------------
# Main program.
#-------------------------------------------------------------------------------
# Set-up.
maxdisp = 800

# Ensure the command line is sensible.
if len (sys.argv) < 2:
    print ("Usage:", sys.argv[0], "<image>...", file=sys.stderr)
    sys.exit (1)

# Process the files given on the command line.
for fn in sys.argv[1:]:
    # Read in the image and print out its dimensions.
    im = cv2.imread (fn)
    ny, nx, nc = im.shape
    print (fn + ":")
    print ("␣␣Dimensions:", nx, "pixels,", ny, "lines,", nc, "channels.")

    # Calculate and output some important statistics.
    print ("␣␣Range:␣%d␣to␣%d" % (im.min (), im.max ()))
    print ("␣␣Mean:␣%.2f␣(using␣mean)" % mean (im))
    print ("␣␣Mean:␣%.2f␣(using␣numpy␣method)" % im.mean ())
    print ("␣␣Standard␣deviation:␣%.2f" % im.std ())

    # Work out and display the histogram.
    x, h = hist (im)
    plot_hist (x, h, fn)

    # For display, ensure the image is no more than maxdisp pixels in x or y.
    if ny > maxdisp or nx > maxdisp:
        nmax = max (ny, nx)
        fac = maxdisp / nmax
        nny = iround (ny * fac)
        nnx = iround (nx * fac)
        print ("␣␣[re-sizing␣to␣%d␣x␣%d␣pixels␣for␣display]" % (nnx, nny))
        im = cv2.resize (im, (nnx, nny))

    # Display the image.
    cv2.imshow (fn, im)
    cv2.waitKey (0)
    cv2.destroyWindow (fn)
    print ()

#-------------------------------------------------------------------------------
# End of summarize.
#-------------------------------------------------------------------------------
```

The example code presented above works fine for colour images. However, when OpenCV encounters a monochrome (single-channel) image, the representation changes: rather than being a data structure indexed by the three values discussed above, y, x and c, the c part disappears and the image needs to be indexed by only two values, y and x. This means that one cannot easily use a consistent way of indexing, and some coding callisthenics are required, a situation the author finds profoundly irritating. There is a work-around this in EVE.

3.5 Contrast-stretching and an improved histogram routine

Our first attempt at a histogram routine doesn't work well if an image has 16-bit pixels, both because 65,536 points are difficult to draw on a graph and because most of the grey level bins would have a count of zero. More sophisticated software will scale the available range of grey levels into a more sensible number of bins. We shall do this in two stages.

If an image's histogram is 'bunched up' into a narrow range of grey levels or there are more grey levels than a display can cope with, its appearance can be improved by *contrast stretching*. This involves finding the lowest and highest grey-level values in the image and then linearly scaling them to have the values zero and 255 (typically, because that is what computer displays support). If the lowest value in an image is $P_{\min}$ and the highest $P_{\max}$ and we want to scale them to $V_{\min}$ and $V_{\max}$ respectively, then a few moments' reflection will show that each pixel $P(x,y)$ should be changed using the formula

$$(V_{\max} - V_{\min}) \frac{P(x,y) - P_{\min}}{P_{\max} - P_{\min}} + V_{\min} \tag{3.1}$$

The contrast-stretching algorithm tells us how to re-scale the grey-levels linearly between any limits. Armed with this, we can easily write a histogram routine that works for any number of bins; the following code is adapted from EVE and should work with colour OpenCV images, albeit quite slowly.

⟨*Calculate the histogram*⟩ ≡

```
def histogram (im, bins=64, limits=None):
    """Work out the histogram of the image 'im'.  The histogram is
    accumulated in 'bins' bins.  By default, these lie between the
    minimum and maximum values of 'im' but other extrema can be
    provided in 'limits', a list comprising the low and high limits
    to be used.  Values outside these extrema are ignored."""

    # Find the extreme values in the image.
    if limits is None: limits = [im.min(), im.max()]
    lo, hi = limits

    # Create the arrays to hold the values to be plotted.
    h = numpy.zeros (bins)
    a = numpy.zeros (bins)

    # Fill the x array with the centres of the bins.
    inc = (hi - lo) / (bins - 1)
    for i in range (0, bins):
        a[i] = lo + i * inc

    # Accumulate the histogram.
    ny, nx, nc = im.shape
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = int ((im[y,x,c] - lo) / (hi - lo) * (bins-1) + 0.5)
                if v >= 0 and v < bins:
                    h[v] += 1.0

    # Return the abscissa and ordinate arrays.
    return a, h
```

The three lines in the innermost loop do all the interesting work, scaling the pixel value into the number of available bins using (3.1) and then incrementing the appropriate element of the histogram.

3.6 Histogram equalisation

The best-known image processing technique is arguably histogram equalisation. This is a non-linear mapping of grey levels intended to improve the appearance of low-contrast regions of an image: it works by stretching out regions of similar grey value and compressing regions where few pixels have distinct grey levels. This might seem difficult to perform but turns out to be quite straightforward.

Despite being so well-known, serious vision work *avoids* histogram equalisation because it is *ad hoc* and dangerous: the non-linear mapping involved wreaks havoc with the pixel distribution (the 'shape' of the histogram), making impossible any subsequent processing based around an understanding of the image formation processes involved. You have been warned!

If we think of the code that calculated the histogram, we end up with an array h where, for a subscript g, it holds how many pixels have the grey value g. To perform histogram equalisation, we first convert the histogram into the *cumulative* histogram; after this stage, each element of h holds how many pixels hold a value of g *or lower*.

⟨*Calculate a cumulative histogram*⟩ ≡

```
def cumulative_histogram (im, bins=256, limits=None):
    "Find␣the␣cumulative␣histogram␣of␣an␣image"
    a, h = histogram (im, bins=bins, limits=limits)
    for i in range (1, len(h)):
      h[i] = h[i] + h[i-1]
    return a, h
```

The second and final stage of histogram equalisation is to use the cumulative histogram as a look-up table. We scan over the image, pixel by pixel, and for a pixel whose value is g, we simply replace it by h[g]; this performs the necessary non-linear mapping alluded to above.

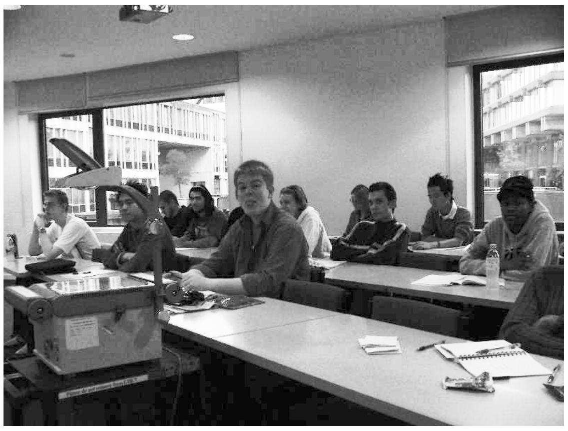
⟨*Look up each pixel in a table*⟩ ≡

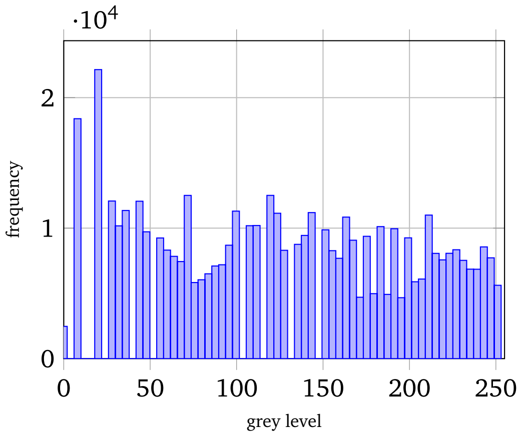
```
def lut (im, table, limits=None):
    "Look␣up␣each␣pixel␣of␣an␣image␣in␣a␣table"
    if limits is None: limits = extrema (im)
    lo, hi = limits
    ny, nx, nc = sizes (im)
    bins = len (table)
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = (im[y,x,c] - lo) / (hi - lo) * (bins-1)
                im[y,x,c] = table[int(v)]
```

The effect of histogram-equalising the image in Figure 3.6(b) is shown in Figure 3.7.

3.7 Content-based image retrieval using histograms

Anyone who has tried to find an image on the Web will have realised that the major search engines index images by the text that surrounds them


(a) Histogram-equalised image



(b) Histogram of (a)

Figure 3.7: Result of histogram-equalising the image in Figure 3.6(c)

or links to them. It would be much better if an image could be found by presenting an example of something and saying effectively "find me images that look like this." This is an active area of research known as *content-based image retrieval* (CBIR) or sometimes *query by image content* (QBIC). Let us develop a simple CBIR program, one that similar to the state of the art in the early 1990s.

A reasonable way of characterising an image, so the argument goes, is through the colours it contains, *i.e.* its histogram. If we find images with similar histograms, they should look similar.[1] But how can we determine the similarity of histograms? The most obvious way is simply to find the difference between them — although as these differences can be positive or negative at different places in an image, it is wise to take the absolute value or the square of the difference. In fact, it is normal in most of science and engineering to use the sum-squared difference

$$\sum_i (H_1(i) - H_2(i))^2 \tag{3.2}$$

so this is one way to determine the similarity of histograms H_1 and H_2.

The problem with the sum-squared difference is that a change in the illumination of the scene (say, by the sun going behind clouds) makes the difference between histograms large. We really need something that works on the *shape* of the histograms rather than the actual values. There is something that does this but to get to how it works, we'll need remind ourselves of a little more statistics first.

We are all familiar with the mean of a set of data — in fact, we considered how to calculate the mean of an image earlier in these notes. The spread of values in a set of data is measured by its *standard deviation*, closely related to the width of any peak in a histogram.

Because we're working in the image realm, let us consider how to calculate the standard deviation of an image P; the basic approach is the same for any dataset though. However, we cannot just calculate

$$\frac{1}{MN} \sum_{x,y} (P(x,y) - \langle P \rangle)$$

[1] You might like to reflect on these statements before proceeding.

where $\langle\cdot\rangle$ denotes averaging: this *must* come out to be zero because of the definition of $\langle P\rangle$. However if we calculate the mean *squared* deviation from the mean, we end up with the variance, the square of the standard deviation:

$$\sigma^2 = \frac{1}{MN}\sum_{x,y}\left(P(x,y) - \langle P\rangle\right)^2 \tag{3.3}$$

image	mean	s.d.
well-exposed	105.6	56.5
under-exposed	30.2	26.4
over-exposed	217.8	60.4

Table 3.1: Means and standard deviations of the images of different exposures in Figure 3.6

This makes all the variations from the mean non-negative, and so σ^2 will be greater than zero unless every pixel has the same value. The means and standard deviations of the images in the differently-exposed images shown in Figure 3.6 are presented in Table 3.1.

A straightforward implementation of (3.3) involves two passes through all the pixels of an image, the first to calculate the mean and the second to sum the squared deviations from it — rather pesky:

⟨*Calculate the standard deviation in two passes*⟩ ≡

```
def sd_slow (im):
    "Return␣the␣standard␣deviation␣of␣an␣image␣(two-pass)"
    ny, nx, nc = im.shape
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                total += im[y,x,c]
    n = ny * nx * nc
    mean = total / n

    total = 0.0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = im[y,x,c] - mean
                total += v * v
    v = total / n
    return math.sqrt (v)
```

However, a little algebra helps here. Expanding the square gives

$$\sigma^2 = \frac{1}{MN}\sum_{x,y}\left(P(x,y)^2 - 2\langle P\rangle P(x,y) + \langle P\rangle^2\right) \tag{3.4}$$

$$= \frac{1}{MN}\sum_{x,y}P(x,y)^2 - \frac{2\langle P\rangle}{MN}\sum_{x,y}P(x,y) + \langle P\rangle^2 \tag{3.5}$$

$$= \frac{1}{MN}\left(\sum_{x,y}P(x,y)^2 - \frac{1}{MN}\left(\sum_{x,y}P(x,y)\right)^2\right) \tag{3.6}$$

which might look messier but *can* be calculated in a single pass through the image as it involves just the sum of the pixels and the pixels' squares. We end up with the following routine.

⟨*Calculate the standard deviation*⟩ ≡

```
def sd (im):
    "Return␣the␣standard␣deviation␣of␣an␣image"
    ny, nx, nc = im.shape
    total = total2 = 0.0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = im[y,x,c]
                total += v
                total2 += v * v
    n = ny * nx * nc
    v = (total2 - total**2/n) / n
    return math.sqrt (v)
```

This conversion of the straightforward way of calculating the standard deviation into a seemingly more complicated but computationally more efficient algorithm is an approach that crops up fairly often in computer vision. We shall see some examples later where algorithms that initially look to be complicated can be made significantly simpler with a little careful thought.

With the standard deviation under our belt, we can turn our attention to measuring the similarity of a pair of images (or, more generally, sets of data). Of course, statisticians have had to do this for many years, and they have come up with a statistic to do it known as the *sample cross-correlation coefficient*, conventionally written as r. We can use this directly on the pixels of our images, P and Q, and r is given by:

$$r = \frac{\langle PQ \rangle}{\sqrt{\langle P^2 \rangle \langle Q^2 \rangle}} \tag{3.7}$$

where $\langle \cdots \rangle$ denotes averaging over the N pixels of the image. Now $\langle P^2 \rangle$ is essentially the square of the standard deviation of P (and likewise for Q); and $\langle PQ \rangle$ is called their *covariance*; we shall see this again when we explore face recognition.

If we expand the $\langle \cdots \rangle$ notation and simplify, we can actually calculate r from sums accumulated in a single pass through the pixels of an image in much the same way as for the standard deviation itself:

$$r = \frac{\sum PQ - \frac{\sum P \sum Q}{N}}{\sqrt{\left(\sum P^2 - \frac{(\sum P)^2}{N}\right)\left(\sum Q^2 - \frac{(\sum Q)^2}{N}\right)}}$$

where each of the sums is performed over the N pixels of the image. Note that $\sum P^2 \neq (\sum P)^2$ for precisely the same reason that the s.d. is not the same as the mean. Translating this into code, we end up with the following.

⟨*Calculate the correlation coefficient*⟩ ≡

```
def correlation_coefficient (im1, im2):
    "Calculate␣the␣correlation␣coefficient␣between␣two␣images"
    ny, nx, nc = im1.shape
    sumx = sumy = sumxx = sumyy = sumxy = 0.0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v1 = im1.im[y,x,c]
                v2 = im2.im[y,x,c]
                sumx += v1
                sumy += v2
                sumxx += v1 * v1
                sumxy += v1 * v2
                sumyy += v2 * v2
    n = ny * nx * nc
    v1 = sumxy - sumx * sumy / n
    v2 = math.sqrt((sumxx-sumx*sumx/n) * (sumyy-sumy*sumy/n))
    return v1 / v2
```

r has some interesting properties. It lies in the range -1 to $+1$. The value $+1$ occurs when P and Q vary *identically*. That is not to say that the images are identical: they may differ in scale and offset but their variations have to be 'in step,' as in, for example, an image and a contrast-stretched version of it. The opposite extreme, $r = -1$, occurs when one image is like a photographic negative of the other — where one image increases from pixel to pixel, the other decreases — and again scale and offset have no effect. Finally $r \approx 0$ means that the two images are, on average, not at all like each other.

It has taken us some time to get there, but we have ended up with two different ways of assessing the similarity of histograms, sum squared difference and correlation. Armed with these, we can measure the similarity of histograms and hence try to do some content-based image retrieval.

Let us now consider a program, `colrec1`, that implements this idea. `colrec1` is invoked with a number of image files on the command line. The first of these is the *probe* image, the one for which matches are to be found, while the others are the *test set* of images in which a match is sought.

⟨*colrec1.py*⟩ ≡

```
#!/usr/bin/env python3
"Demonstrate␣content-based␣image␣retrieval␣using␣histograms"
import sys, math, cv2
#------------------------------------------------------------------------------

<<Calculate the correlation coefficient>>
<<Calculate the histogram>>

#------------------------------------------------------------------------------
# Say hello and initialize things.

if len(sys.argv) < 3:
    print ("Usage:", sys.argv[0], "<probe>␣<test-images>", file=sys.stderr)
    sys.exit (1)
probe_file = sys.argv[1]
v_best = 0
f_best = "?"

# Read in the probe image and find its histogram.

im = cv2.imread (probe_file)
a, probe = histogram (im, limits=[0,255])

# We now enter the main loop.  The basic idea is to load an image,
# find its histogram, then compare that with the histogram of the
# probe image.  We are careful to skip the case when the test
# image is the same as the probe.

for file in sys.argv[2:]:
```

```
30      if file != probe_file:
31          print ("Processing", file)
32          im = cv2.imread (file)
33          a, h = hist (im, limits=[0,255])
34          v = correlation_coefficient (probe, h)
35          if v > v_best:
36              v_best = v
37              f_best = file
38
39  # We've finished our work so say which of the test set best matches the
40  # probe and exit.
41  print ("Best␣match␣is", f_best, "with␣correlation", v_best)
```

where `histogram` is the routine presented above. You might like to reflect on why line 30 of this code is present.

As it stands, this program calculates a single histogram that spans all the colour channels of an image — so an image having many pixels of blue sky could be mistaken with one containing many pixels of green grass. Hence, we should not expect it be very effective. A better approach would be to calculate a separate histogram for each channel and then combine them in some way, perhaps by calculating a separate correlation coefficient for each channel and then combining them.

4 Convolution

Convolution is arguably the most important way of processing an image, processing the region around each pixel in turn. We look at how it is performed, the effects of different masks and ways of combining values. We conclude the chapter with considerations of mathematical morphology and matched filtering, designing masks to bring out particular image features.

4.1 Introduction

Now that we have some idea of how an image is represented and manipulated within a computer, we can start thinking more seriously about processing images. The techniques we looked at in Chapter 3 all processed each pixel independently of all others. However when we look at an image, we do not look at each pixel in isolation; instead, we tend to focus on the parts of the image where things change — edges, corners and so on. This will be clear from Figure 4.1, in which the edges (obtained using the Canny edge detector of Chapter 7) alone give a good impression of the scene. Our first step away from considering each pixel in isolation is therefore to consider small regions of images.

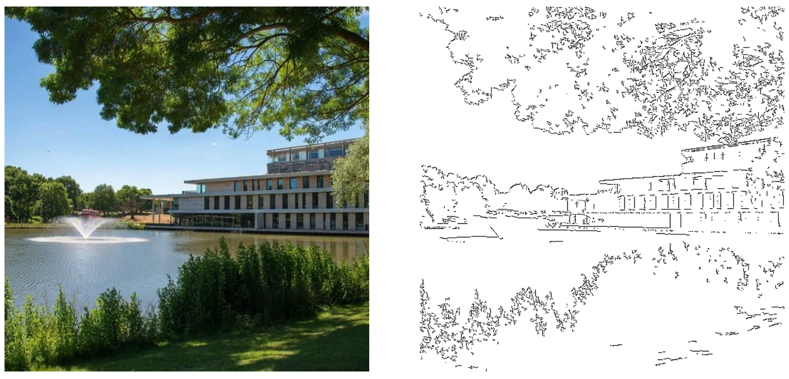

(a) The Essex campus (b) Edges obtained using Canny's detector

Figure 4.1: A photograph of the Essex campus and its edges

4.2 Enhancing isolated points using convolution

Let us start by trying to identify pixels that differ widely from their surroundings. These tend to be due to defects in the sensor but also help us understand what is happening. Fairly obviously, the only way to determine whether a pixel is lighter or darker than its surroundings is to examine the values around it; and we want to perform exactly the same series of operations to each pixel of the image in turn, scanning over the lines and pixels within each line in the way we saw in Chapter 3. Hence, given a pixel at location (x, y) in the image P, we need to examine it and its eight nearest neighbours:

	$x-1$	x	$x+1$
$y-1$	$P(x-1, y-1)$	$P(x, y-1)$	$P(x+1, y-1)$
y	$P(x-1, y)$	$P(x, y)$	$P(x+1, y)$
$y+1$	$P(x-1, y+1)$	$P(x, y+1)$	$P(x+1, y+1)$

If $P(x, y)$ is light, we want its value to become much larger (and hence even lighter) than the other pixels shown above, and the easiest way to do that is to multiply it by a large number. But that is not enough; for if (say) $P(x-1, y)$ were also quite large, it would also be large after multiplication by the same number. So we need to multiply $P(x, y)$ by a large number *and simultaneously* suppress the values of those pixels around it by multiplying them by smaller numbers. We can do this by placing a 3×3 'mask' of coefficients on the region of the image:

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}

and multiply each pixel by the corresponding coefficient in this mask. Writing this out explicitly, we calculate

$$\begin{aligned} v \; = \; & P(x-1, y-1)m_{11} + P(x, y-1)m_{12} + P(x+1, y-1)m_{13} \\ + \; & P(x-1, y)m_{21} + P(x, y)m_{22} + P(x+1, y)m_{23} \\ + \; & P(x-1, y+1)m_{31} + P(x, y+1)m_{32} + P(x+1, y+1)m_{33} \end{aligned}$$

and use v to replace $P(x, y)$. The overall process is summarized in Figure 4.2.

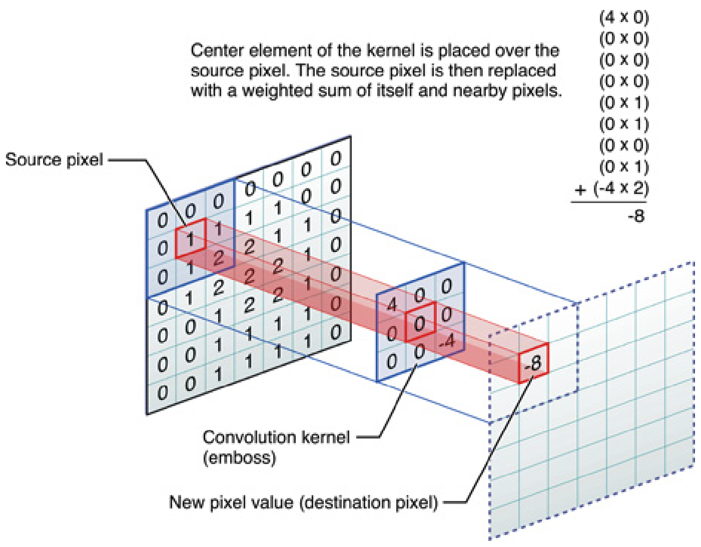

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 4.2: Illustration of the convolution process (image from https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html)

We must be careful about the way we store results when performing a convolution. In particular, we cannot perform store them 'in place,' writing the results into the image we're reading values from as we would end up using modified values for $P(x-1,y-1)$, $P(x,y-1)$, $P(x+1,y-1)$ and $P(x-1,y)$ in calculating the result for $P(x,y)$ — this is known as "recursive filtering" for obvious reasons. Hence, convolution is normally done from one image structure into another.

We have now decided what we are going to do but it remains to choose a set of values for the mask coefficients. Let us choose

−1	−1	−1
−1	8	−1
−1	−1	−1

which is known as the Laplacian (after the French mathematician Laplace). This mask fulfils our necessary criteria: the pixel in the centre of the mask is multiplied by a large number, while those around are multiplied by small ones. But why those particular values? The reason is that processing homogeneous region of the image (*i.e.,* one whose pixels all have identical values) will produce a response of zero. A few minutes' consideration should then tell you what processing with this mask will do to an isolated dark pixel: it will result in a large *negative* value.

4.3 Blurring

If the Laplacian enhances isolated pixels, we might ask ourselves what effects other choices of coefficients have. The obvious other extreme is to use

1	1	1
1	1	1
1	1	1

When we convolve this with an image, we are summing up the pixels in 3×3 regions around each pixel, effectively averaging them. (In practice, we would probably use $\frac{1}{9}$ instead of unity in each of the 9 positions in the mask, thereby calculating a true mean.) On a homogeneous region, this would have no effect: the average of 9 identical values is the same as each of them. But if one pixel was widely different from its neighbours, convolution with this mask would tend to reduce its effect; in other words, this can be used to perform noise reduction — though there are better ways, as we shall see shortly. On the other hand, a sharp boundary between uniformly light and dark regions would be smeared out by the region-averaging process — in other words, the image would be *blurred* in much the same way as focusing a camera poorly. Indeed, the above mask is usually called 'a 3×3 blur.'

On images captured by a modern digital camera, a 3×3 blur has a small but noticeable effect; the amount of blurring can be increased by using 5×5, 7×7 *etc*. masks, or by applying a 3×3 on several times. Mask sizes are almost always odd so that the region considered is symmetrical around the pixel to be replaced.

People who have used Photoshop or similar tools may well have come across blurring and convolution. One popular technique they employ for enhancing images is *unsharp masking*. What this effectively does is subtract the blurred image from the original one, enhancing the edges, and add the result to the image.

4.4 Using the median

Our consideration of blurring masks tells us that we are effectively calculating the arithmetic mean of the $9, 25, 49\ldots$ values for 3×3, 5×5, $7 \times 7\ldots$ masks. But the mean is only one measure of the behaviour of random variables; others are the median and the mode. (For a symmetric distribution with a single peak, all three coincide; but this is never achieved on real data — and, as has been said before, computer vision is an experimental discipline.)

The *mode* is the most commonly-occurring value, *i.e.* the peak of the histogram. Modal filtering has never become popular in image processing, perhaps because the numbers of values involved are so small, though extensive work has been done on it [Davies, 2017]. On the other hand, the *median*, the value in the middle position when all values are sorted into order, has proved popular. The main reason for this is what happens in the vicinity of a pixel whose value is widely different from those surrounding it — just the kind of image feature the Laplacian finds. Although such features can arise naturally in images, they tend to arise more commonly from timing problems in image or video capture circuitry — so-called "salt and pepper" noise — and we normally want to remove them. Convolving using the mean will smear out the effect of a single light or dark pixel over a region of the size of the mask; on the other hand, when the median is used, the value sorts to one extreme of the pixel values and hence has a much smaller effect on the median. So median filtering tends to be better at noise removal, one of the main reasons for contemplating averaging over a region, and introduces less blur too.

In terms of computation, the median takes more effort to calculate than the mean as it involves sorting numbers into order. As people who have studied algorithmics will know, there are many sort algorithms out there. The easiest to code is the 'bubble' sort but it is pretty inefficient. The algorithm with the best theoretical performance is the 'quicksort' and is often taught in courses on algorithmics, though more because it is a way of teaching recursion in programming. The fact is that quicksort also can be a very poor sorting algorithm if the data happen to be ordered unfortunately. A better choice for this kind of task is Shell's sort, which is almost as good as the quicksort in the best case and a lot better in the worst case.

These mask-based operators are absolutely central to image processing. In later chapters, we shall look at how they are used in feature detection, with emphasis on finding first lines and then corners in images.

4.5 Other types of masks

Having considered masks that enhance isolated spots or blur images, it is useful to look at a couple more masks to motivate the discussion in later chapters.

Consider what effect on an image the mask

1	1	1
0	0	0
−1	−1	−1

will have. This takes the pixels in a line of the image and subtracts off it the pixels two lines below that. As usual, there is a zero response in uniform regions, where all the pixels have the same value. Now consider a *vertical* edge in the image directly below the centre of the mask: again, the response will be zero. But what if the edge is *horizontal*? If the image contains a lighter region (*i.e.,* with higher-valued pixels) above a darker one, this mask will produce a large positive response when it spans the edge. Conversely, if the upper region is darker than the lower one, the mask produces a large negative response. So this mask tends to detect horizontal edges. By analogy, it is fairly easy to design a mask to detect vertical edges:

−1	0	1
−1	0	1
−1	0	1

What happens if we wish to detect edges that are neither vertical nor horizontal? We could design masks for the 45° angles. However, an edge running diagonally across the image will produce a response from both these masks, though less in both cases than would be obtained from its optimally-oriented edge — so a way ahead is to combine the responses from these two masks.

This approach is what is used in Sobel's edge detector. It uses two masks which differ only a little from those above:

$$H = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \qquad V = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

and combines their responses in quadrature:

$$\sqrt{H^2 + V^2} \tag{4.1}$$

to calculate the overall response at a pixel. The Sobel detector is able to run at video rate on quite modest hardware so is widely used in practice for edge detection. The edge detector due to Canny, presented in Chapter 7, is a little better than Sobel's though somewhat more complicated. It is able to run at video rate on modern PC-class hardware too.

4.6 Implementing convolution

It is not tricky to code up a routine that implements straightforward convolution: it is only a matter of multiplying pixels by their coefficients and adding up the result. The biggest problem is around the edges of the image, for it is not clear what should be done. There are several possible approaches, ranging from padding out the edges of the image with zero values (or, equivalently, ignoring mask pixels that 'fall off' the image), mirroring the image, and so on.

However, only one way of handling the edges agrees with the underlying Fourier theory (which takes too long to go into in this lecture course) and that is to wrap those mask coefficients that 'fall off' one edge of the image around to the other edge. Fortunately, all programming languages have a way of calculating the remainder when one number is divided by another and we can use this to work out subscripts; in C and Python, this is the '%' operator. With this, the routine that implements convolution in EVE is shown below. You could use it on images read in Using OpenCV but it runs pretty slowly because all the loops are coded in Python — OpenCV routines are much faster because their loops are written in C++ and compiled down to machine instructions. You will see that this supports different types of convolution (mean, median *etc.*) and the similar morphological operators (see the next section) via minimum and maximum.

⟨*Convolve an image with a mask*⟩ ≡

```
def convolve (im, mask, statistic='sum'):
    """
    Perform a convolution of im with mask, returning the result.

    Arguments:
           im  the image to be convolved with mask (modified)
         mask  the convolution mask to be used
    statistic  one of:
                  sum  conventional convolution
                 mean  conventional convolution
               median  median filtering
                  min  grey-scale shrink (reduces light areas)
                  max  grey-scale expand (enlarges light areas)
    """
    im = reshape3 (im)
    ny, nx, nc = sizes (im)
    mask = reshape3 (mask)
    my, mx, mc = sizes (mask)
    yo = my // 2
    xo = mx // 2

    # Create an output image of the same size as the input.
    result = image (im)

    # We need a special case for 'min' statistic to erase the mask elements
    # that are zero.
    nzeros = len ([x for x in mask.ravel() if x == 0])

    # Loop over the pixels in the image.  For each pixel position, multiply
    # the region around it with the mask, summing the elements and storing
    # that in the equivalent pixel of the output image.
    v = numpy.zeros ((my*mx*mc))
    vi = 0
    for yi in range (0, ny):
        for xi in range (0, nx):
            for ym in range (0, my):
                yy = (ym + yi - yo) % ny
                for xm in range (0, mx):
                    xx = (xm + xi - xo) % nx
                    v[vi] = im[yy,xx,0] * mask[ym,xm,0]
                    vi += 1
            if statistic == 'sum':
                ave = numpy.sum (v)
            elif statistic == 'mean':
                ave = numpy.mean (v)
            elif statistic == 'max':
                ave = numpy.max (v)
            elif statistic == 'min':
                v = sorted (v)
                ave = numpy.min (v[nzeros:])
            elif statistic == 'median':
                ave = numpy.median (v)
            result[yi,xi,0] = ave
            vi = 0
    return result
```

4.7 Mathematical morphology

You might be asking yourself at this point whether there is anything else that can be used in convolution as well as the mean, mode and median. Two other interesting possibilities are the minimum and maximum values, and these give rise to grey-scale versions of what are known as *morphological* operators. Taking the minimum of a region will naturally tend to make light regions smaller; a 3×3 minimum will usually remove the outermost light pixels from around a region, so this operation is called an *erode* or *shrink*. Conversely, taking the maximum will tend to grow light regions, by one pixel for a 3×3 region, two for a 5×5 region, and so on; this is known as a *dilate* or *expand*.[1]

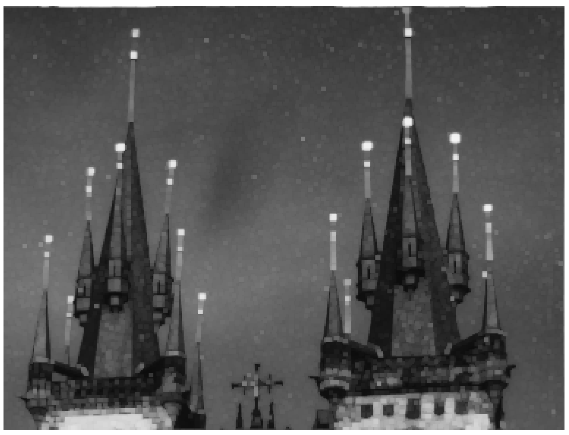
[1] A little care is necessary here as some texts consider the shrinking and expanding of a dark region on a white background, so you may encounter the opposite terminology to what I've used.

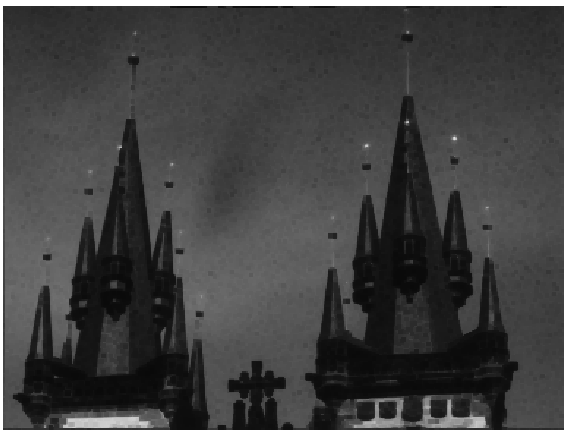
If we perform an erode followed by a dilate, a few moments' thought should tell you that we will delete any isolated light pixels inside dark regions; this is known as an *opening*. The converse, dilate followed by erode, is a *closing*. Some of these operations are illustrated in Figure 4.3. It is clear that the specular reflections in Figure 4.3(b) are much more apparent than those in Figure 4.3(a), the original image, due to the expansion of light-coloured regions. Similarly, the reflections in Figure 4.3(c) are much less apparent than the original image. Finally, the result of the opening in Figure 4.3(d) essentially finds the outlines of bright regions. These morphological versions of convolution are especially useful when one has identified regions of images that correspond to objects of interest, and those regions need to be 'tidied up.'

If a morphological shrink reduces the size of light feature on a dark background by one pixel around its boundary, then subtracting the result of the shrink from the original image will leave only its boundary — see Figure 4.4.

Let us consider one final morphological algorithm, one that finds the

(a) Original image


(b) Expand


(c) Shrink

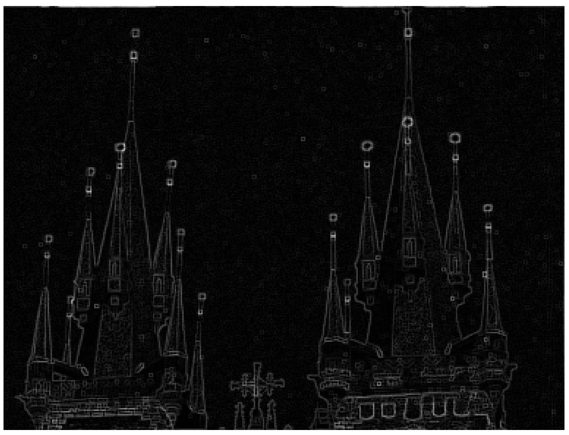
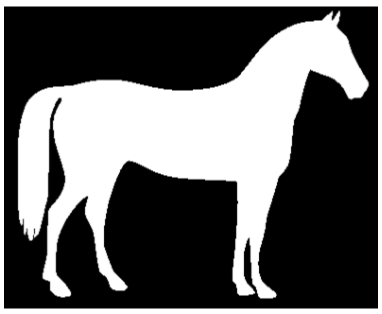
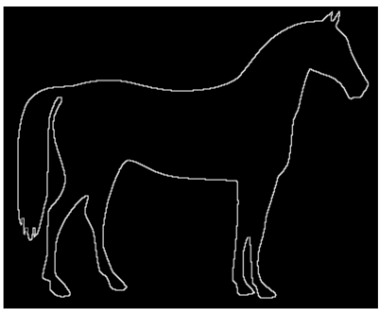

(d) Opening

Figure 4.3: Morphological expand, shrink and opening


(a) Original image


(b) Difference between original and result of shrink

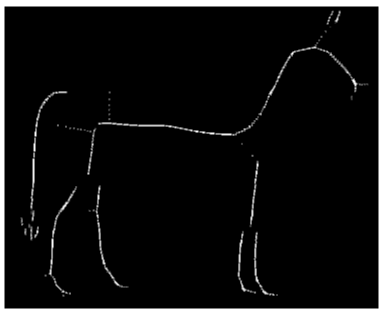

(c) Morphological skeleton

Figure 4.4: Finding an object's boundary and skeleton

'skeleton' of a region — by which we mean something that retains its connectivity but discards most of the pixels. Starting with the original image, each iteration of the loop shown below makes `skel` into a version of itself with small regions removed or the image after a shrink — so it contracts the shape of white features until just before all white pixels are removed.

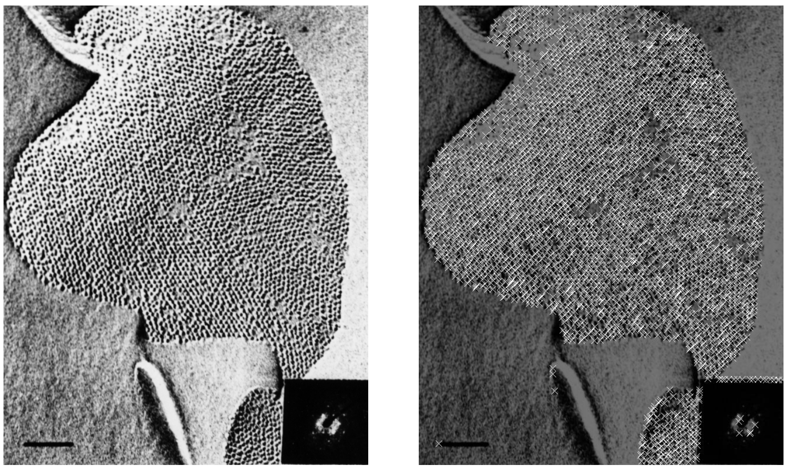
⟨*Skeletonize a binary image*⟩ ≡

```
skel = im
while True:
    eroded = shrink (im, mask)
    opened = expand (eroded, mask)
    temp = im - opened
    skel = bitwise_or (skel, temp)
    if eroded.sum () == 0:
        break
    im = copy (eroded)
```

For the horse image of Figure 4.4(a), some 55 iterations of this loop yield the skeleton of Figure 4.4(c).

4.8 Finding known patterns in images

Matched filtering. There are many cases where we might wish to find or emphasize arbitrary-shaped regions within an image. To give a concrete example of this, Figure 4.5 shows a micrograph of a cell where proteins on the cell membrane (outer surface) give it a characteristic 'lumpy' appearance. In modelling the biological processes of cells, one needs to know how much the proteins clump together into regions; and to do that, one needs to determine where each protein lies in the image.


(a) Cell membrane image showing lumps of proteins

(b) Protein locations found by matched filtering

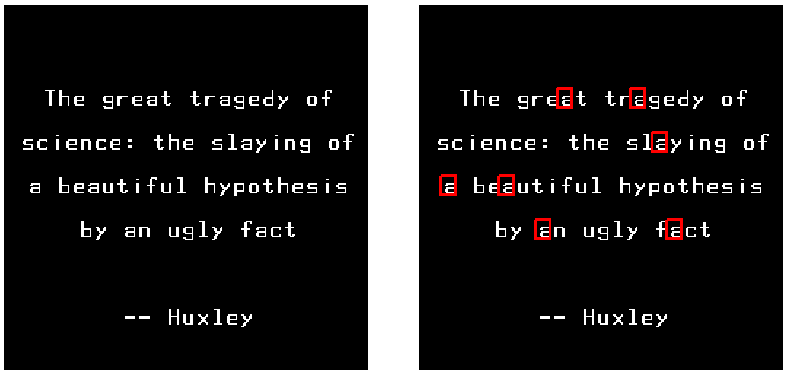
Figure 4.5: Locating proteins on cells using matched filtering

If, as in this case, we are able to identify the general appearance of the feature that we are looking for in the image, we can design a so-called *matched filter* which will produce a large response when that feature is encountered. In this case, we observe that the proteins appear to be illuminated from the upper left corner of the image, so a mask that has a positive response to its upper left and a negative response in its lower right will tend to enhance these features. Hence, a suitable convolution mask will be something like:

$$\begin{array}{|c|c|c|}\hline 2 & 1 & 0 \\ \hline 1 & 0 & -1 \\ \hline 0 & -1 & -2 \\ \hline \end{array} \qquad (4.2)$$

As usual, the coefficients in the mask sum to zero so that it will produce a response of zero for a uniform region of the input image. Figure 4.5(b) shows all the significant peaks found with this matched filter. There are some false positives which can be removed by tuning the size of the peak responses that we accept; but for literally five minutes' work, this is pretty good.

Template matching. When a suitable matched filter is not obvious or the processing needs to be made automatic, we need an alternative strategy. One suitable technique is to identify and extract from the image a typical feature. This extracted region forms a *template* that we want to look for in the image. We can then centre the template over each pixel in the image in turn and work out the similarity between the template and image region — and in Chapter 3 we considered two ways of determining similarity when we looked at content-based image retrieval, namely simple differencing and correlation.



(a) Original image (b) Best matches with the letter 'a'

Figure 4.6: Template matching used to locate occurrences of 'a' in text

To illustrate this process, Figure 4.6(a) contains the author's favourite quotation. This was correlated with the letter 'a' at every position in the image and the best matches are marked in Figure 4.6(b). Incidentally, readers who have a knowledge of video coding will recognize that template matching forms part of the motion estimation techniques of the H.261 and MPEG standards.

5
Low-Level Vision

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 these labelled regions can be described and hence how one can build working vision systems based around the shapes of features.

5.1 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. In this chapter, we consider how low-level processing of image data is able to extract useful features, and how those features can be used to build complete, working vision systems. This is illustrated by an industrial inspection example.

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.

5.2 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 5.1: 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.

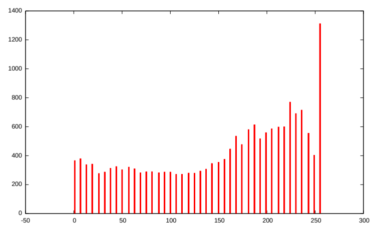


Figure 5.1: Histogram of a typical image. Where is the best place to set a threshold?

Perhaps surprisingly, there is a way to do this. The first thing to realise is that thresholding will be most effective if all the pixels in an object have roughly the same value, and that is well-separated from the values of 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 data in a histogram: its standard deviation (or its square, the 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

$$\sigma^2_{\text{within}} = \omega_B(t)\sigma^2_B(t) + \omega_F(t)\sigma^2_F(t) \tag{5.1}$$

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 (5.1) can be re-written in terms of the *between-class* variance:

$$\begin{aligned}\sigma^2_{\text{between}}(t) &= \sigma^2 - \sigma^2_{\text{within}}(t)\\ &= \omega_B(\mu_B - \mu)^2 + \omega_F(\mu_f - \mu)^2 \quad \text{(where } \mu = \omega_B\mu_b + \omega_F\mu_f\text{)}\\ &= \omega_B\omega_F(\mu_B - \mu_f)^2 \qquad (5.2)\end{aligned}$$

where μ_B and μ_F are the background and foreground means respectively [Otsu, 1979]. 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

⟨*Otsu's method for choosing a threshold*⟩ ≡

```
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
```

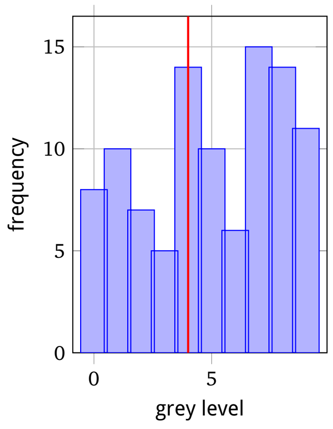
Note that this code yields the threshold correctly only if there is a single $\sigma^2_{\text{between}}$ that is maximum. Moreover, an implicit assumption is that a single, global threshold is enough to separate foreground from background, and the real world is not always that simple. On a positive note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

To illustrate Otsu's method, consider the 10×10 image shown in Figure 5.2(a). The histogram of this image is plotted in Figure 5.2(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) $10x10$-pixel image



(b) Corresponding histogram

Figure 5.2: Example 10-level image and its histogram, with the Otsu threshold marked

5.3 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 shown in Figure 5.3(a). With regions numbered as in Figure 5.3(b), 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 5.3 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.

This approach works well until a shape like region 5 in Figure 5.3(b) is encountered. The first pixel encountered is the top-left one and it will

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 5.3: 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.

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 actually 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, renumbering 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. EVE 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:

⟨*Threshold and label images*⟩ ≡

```
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. The associated laboratory sessions will see you build OpenCV-compatible routines that do similar tasks to these EVE ones.

Careful examination of Figure 5.3 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 whereas the approach described above is 4-connected. If the image of Figure 5.3 was processed with an 8-connected algorithms, all those pixels labelled "7" in the figure would instead be labelled "6."

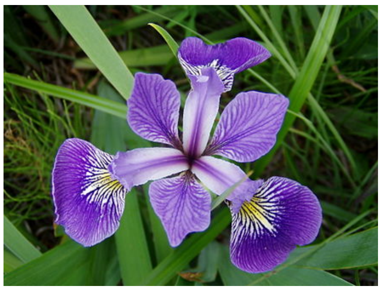
Although not advertised as being region labelling, OpenCV does provide this functionality: regions are what are returned by its `findRegions` routine. In that case, subsequent OpenCV calls on individual regions are able to provide a further information; let us look at the kind of information that can be returned and how it can be used.

5.4 Describing regions

Having isolated regions of interest and labelled them, we need 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 which are concatenated into a *feature vector* and use that to distinguish classes.

Many people find the idea of feature vectors difficult to grasp so let us step away from computer vision for a moment to consider a classic piece of work that involves a feature vector. As long ago as 1936, biologist Ronald Fisher tried to distinguish three species of iris (Figure 5.4) from four measurements of them: sepal length, sepal width, petal length, petal width [Fisher, 1936]. He measured 150 individual flowers and the first

(a) *Iris setosa*


(b) *Iris versicolor*

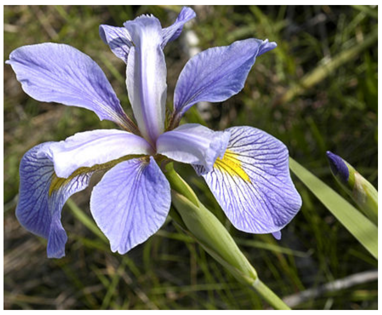

(c) *Iris virginica*

Figure 5.4: Iris species used by Fisher (images from the Wikipedia)

few lines of the dataset are:

```
#SL  SW  PL  PW CLASS
5.1 3.5 1.4 0.2 Iris-setosa
4.9 3.0 1.4 0.2 Iris-setosa
4.7 3.2 1.3 0.2 Iris-setosa
...
```

I have software for visualizing datasets such as these in my research lab on a full-wall stereoscopic display, meaning that you gain a good impression of depth. Figure 5.5 plots three of the four features along the axes, with the species being shown by colour. An interactive demonstration of this allows the user to spin around the axes[1] but from the image, you should be able to see that one species, coloured red, has all its points well away from the others. You should be able to imagine a plane in the 3D space which separates the cloud of red points from the others. However, that is not the case for the other two sets of points, which are inter-mixed or *confused*: they cannot be separated by a plane no matter which way it is oriented.

[1] In my lab, you can do this using gestures.

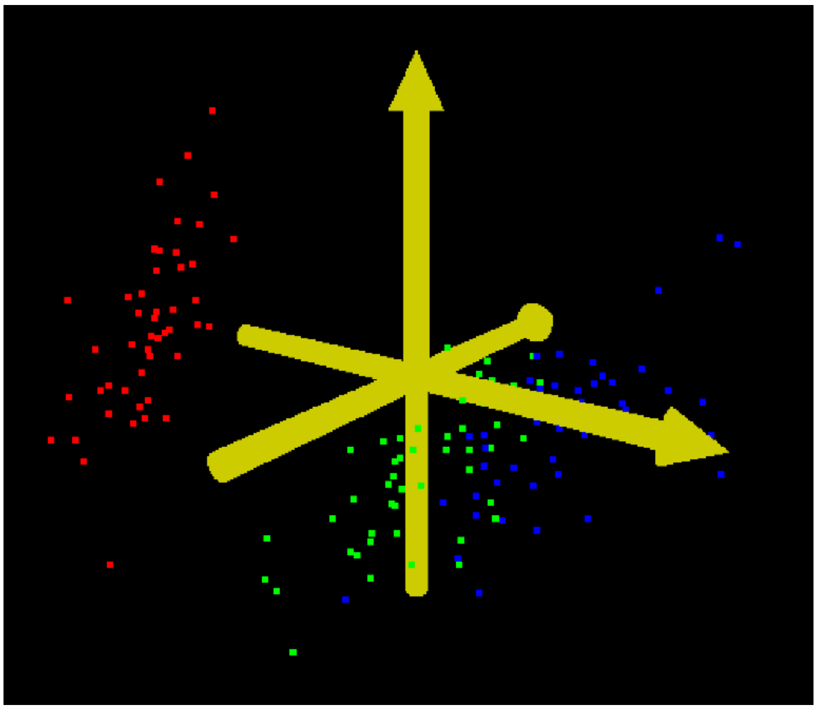


Figure 5.5: Visualization of Fisher's iris data showing how one species (in red) is easily distinguished but the other two are confused

This iris example is a very useful one because it shows how a feature vector can make it fairly easy to distinguish classes: the technique known as a *support vector machine* introduced in Chapter 10 computes the separating plane described in the previous paragraph. In fact, the most effective current way of identifying 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 will almost certainly have heard of neural networks) and we shall look at some of them in Chapter 10 and 11. You should also note that it is sometimes impossible to distinguish confused classes.

Returning to the vision domain, 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; you will recall from Chapter 4 how it can be found. 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, its area is $A = \pi r^2$ and the perimeter $C = 2\pi r$. Hence, if we calculate

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

to make it independent of the radius, then the closer the ratio of C^2/A is to 4π the closer the shape is to being circular. The circle is 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. The closer this is to unity, the more rectangular is the shape.

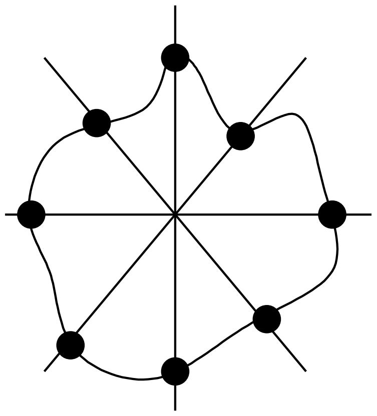


Figure 5.6: Extremal points can form a shape descriptor

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 5.6). 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.

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 described in Chapter 10, 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 were able to use that to make a vision system learn how to read car number plates, the first purely-learnt system to do so — see Chapter 10.

5.5 A practical shape-based vision system

The research community has been fixated on the use of high-level vision for solving all types of vision problem for about a decade now. That's a pity because the old adage of "using a sledgehammer to crack a nut" applies: simpler systems using only the low-level techniques described in this chapter are perfectly adequate — and run faster using less resource. Let us explore such a system here.

Industry uses computer vision extensively. If you watch the BBC's series *Inside the Factory*, you'll find that most of the huge factories visited have vision-based quality assurance as an intrinsic part of their production pipelines. Although the author doesn't have a production line handy, it is fairly easy to mimic the type of imagery they produce.

Imagine a biscuit factory. After the biscuits have been baked they are loaded onto a conveyor belt, where they pass below a vision system *en route* to where they are packaged. As vision engineers, our job is to build a system that identifies misshapen or broken biscuits, or ones that have been under- or over-cooked. Let us concentrate on the first problem here.

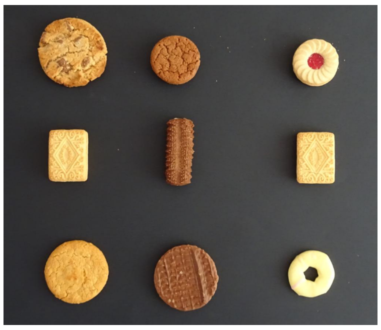


Figure 5.7: Biscuits on a virtual conveyor belt

The first thing we need to do is consider the environment. Biscuits that have a chocolate coating tend to be shiny so the first thing we need to do is ensure that the lighting reduces or avoids specular reflections from them. The easiest way to do that is to make the lighting diffuse. It will also make our task much easier if the conveyor belt is a good contrast from the biscuits. Typical images from the conveyor belt might look like those in Figure 5.7.

Based on the pipeline we have examined above, a suitable series of stages is:

1. convert the image to monochrome
2. threshold the image
3. delete small regions and fill in gaps with morphologial processing
4. label the regions
5. compute descriptors for each region
6. determine whether regions are circular, rectangular or reject

The result of this processing is shown in Figure 5.8. You will see that it has done well for all the biscuits that contrast well with the background but not for the chocolate ones, which tend to be darker. Two small regions have been detected on one biscuit and two others have been missed completely! However, lowering the threshold would cause the lighter region at the right of the image to grow and so is probably not the best strategy. There are *adaptive* thresholding schemes and one of those might work better, or the system developer needs to improve the lighting.

What we have found here is not uncommon: something that looks obvious to a human and has a seemingly well-designed series of steps may not work in practice because some feature of the real world has been overlooked.

A real production line would not carry many different types of biscuit at the same time; a more common requirement is to distinguish well-formed biscuits and reject those that are broken or badly formed. Despite the difficulty with segmenting chocolate biscuits, the shape processing stage works well, as Figure 5.9 shows.

There is one difficulty with rectangular biscuits that does not occur with circular ones: if the sides of a biscuit are not aligned with the edges

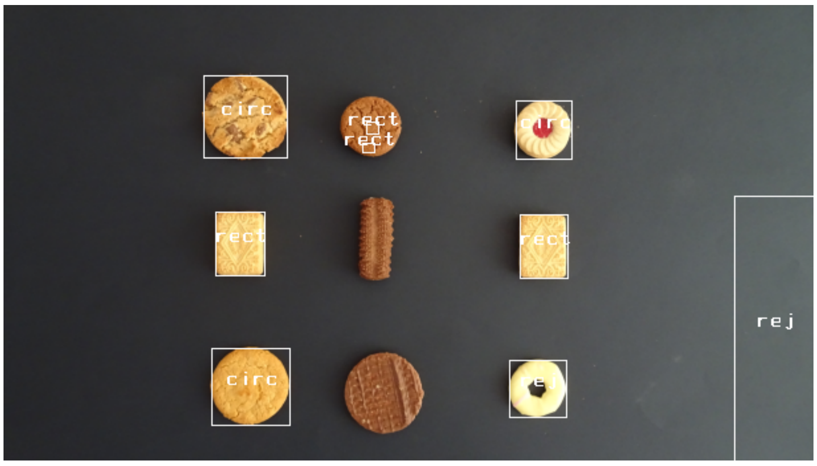
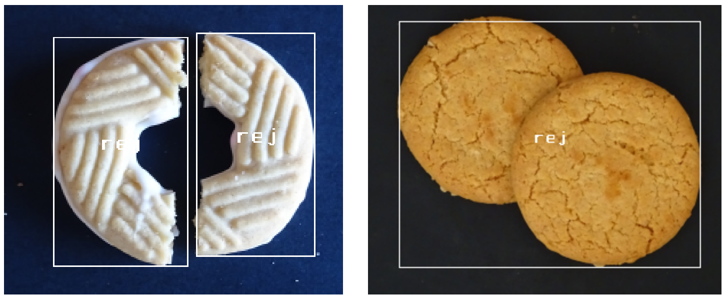


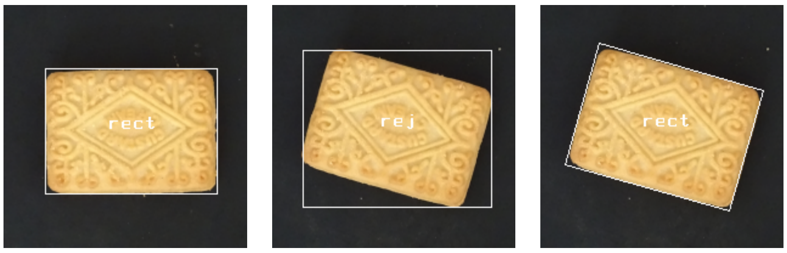
Figure 5.8: Identification of circular and rectangular biscuits. Note that there is a spurious region due to uneven illumination and how difficult the brown biscuits have been to distinguish from the background



(a) Broken (b) Overlapping

Figure 5.9: Broken and overlapping biscuits are rejected

of the image, they can be rejected even if undamaged; see Figure 5.10. The solution here is to compute the *oriented* bounding box. This is done using *principal component analysis*, a technique outlined in Chapter 8 and illustrated in Figure 8.11 — but all you need know at the moment is that it works, as Figure 5.10(c) demonstrates.



(a) Biscuit aligned with axes (b) Axis-aligned bounding box fails when the biscuit is oriented (c) The solution is to compute an *oriented* bounding box

Figure 5.10: Rectangular biscuits at an angle and their bounding boxes

5.6 Measuring texture

We considered colour in Chapter 3 and have just looked at shape. Does the human vision system have other capabilities which help recognize things? The answer is clearly that it does, and perhaps the most important one that we have not yet considered is *texture*.

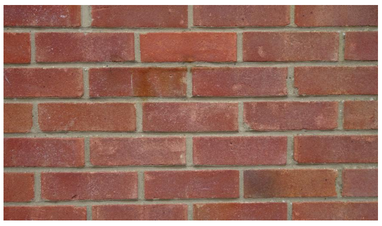


Figure 5.11: Brickwork is a regular texture

Clearly a brick wall (Figure 5.11) has a texture, one that is regular in appearance. Identifying a regular texture such as this is not too difficult: if we know the 'unit cell' from which the texture is built, we can use template matching (Chapter 4) to identify where it appears. If we don't know the unit cell, we can use an approach called *autocorrelation* (which will be familiar to readers whose background encompasses signal processing) to establish it *and* where the repeats occur.

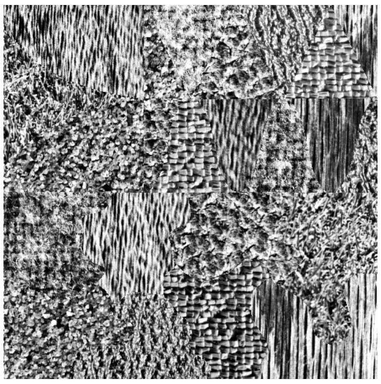


Figure 5.12: Mosaic of irregular textures (from USC-SIPI's Brodatz texture dataset)

However, many textures do not have a regular pattern; Figure 5.12 is a mosaic of several types. For these random textures, one can produce only statistical descriptions of them. It is probably fair to say that identifying textures such as these, and hence segmenting regions by the texture they contain, in as effective a way as colour and shape is an unsolved problem in computer vision.

We shall now examine two classic ways of describing random textures. We shall return to the topic in our consideration of intermediate-level vision in Chapter 7 and machine learning in Chapter 11 but the techniques covered are not necessarily more useful than those we shall consider here, which remain in widespread use several decades after they were devised.

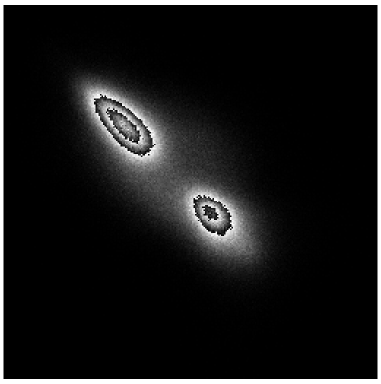


Figure 5.13: Grey-level co-occurrence matrix of the brickwork of Figure 5.11 for shifts of $X = 5, Y = 3$

Grey-level co-occurrence matrices Clearly, it is not possible to describe a texture from the values of individual pixels taken in isolation. A natural approach is to take *pairs of pixels* separated by a particular distance and direction, determining how similar they are. Let's think of looking a pair of regions of an image separated by X and Y along the two directions. Something like the mean square error or correlation won't work well because the relationship between the regions is only statistical, so there won't necessarily be a definite minimum or maximum. Instead we plot a *scattergram* with the grey-level value of one region along one axis and the grey-level value of the other region along the other axis — this is known as a *grey-level co-occurrence matrix* or GLCM. The GLCM of the brick texture of Figure 5.11 is shown in Figure 5.13. As you would expect, $(0,0)$ lies at the top-left corner of the image and $(255,255)$ at the bottom right. Non-zero values lie mostly along the leading diagonal of the GLCM, showing that — for the shift considered — most pixels are similar in grey level to each other. Some other examples of textures from the Brodatz texture dataset and their corresponding GLCMs are shown in Figure 5.14. You might reasonably think that, for textures with a definite directionality, the GLCMs would exhibit this too. Figure 5.15 shows different rotations of a Brodatz texture with a definite directionality and the corresponding GLCMs: there is little visible difference between GLCMs, though the two differ in detail. You might also wonder what effect the shifts have on the GLCM, so some examples are shown in Figure 5.16.

The GLCMs themselves only extract information that may be useful, they do not actually describe textures. To do that, [Haralick et al., 1973] devised 14 measures and used them successfully to describe textures such as corn, grass, water and wood. However, [Connors and Harlow, 1980] found that only five of them were really useful; if $N(i,j)$ is the GLCM, the useful measures are:

$$\begin{aligned}
\text{energy} &= \sum_i \sum_j N_d^2(i,j) \\
\text{entropy} &= -\sum_i \sum_j N_d(i,j) \log_2 N_d(i,j) \\
\text{contrast} &= \sum_i \sum_j (i-j)^2 N_d(i,j) \\
\text{homogeneity} &= \sum_i \sum_j \frac{N_d(i,j)}{1+|i-j|} \\
\text{correlation} &= \frac{\sum_i \sum_j (i-\mu_i)(j-\mu_j) N_d(i,j)}{\sigma_i \sigma_j}
\end{aligned}$$

where μ_i, μ_j are the means and σ_i, σ_j the standard deviations of the row and column sums $N_d(i)$ and $N_d(j)$ defined by

$$\begin{aligned}
N_d(i) &= \sum_j N_d(i,j) \\
N_d(j) &= \sum_i N_d(i,j)
\end{aligned}$$

One problem with GLCMs is choosing the displacements $d = d_x, d_y$ along the axes. A solution is to select the values that yield the most structure, which can be done by maximising

$$\chi_d^2 = \sum_i \sum_j \frac{n_d^2(i,j)}{N_d(i) N_D(j)} - 1$$

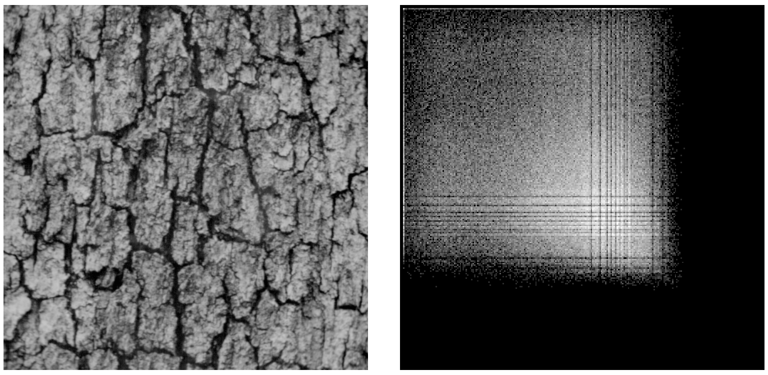
The result of this is a series of numbers that (hopefully) characterize a texture uniquely — but you are still left with the problem of distinguishing which range of numbers identifies which texture, arguably best done using some kind of machine learning (Chapter 10). In fact, there has been a recent trend towards computing all 14 texture measures and using them to train a neural network to distinguish texture classes; while this often works to some extent, you should always bear in mind the "first law of machine learning": *garbage in, garbage out* — in other words, you have to train on data that stand a chance of distinguishing classes.

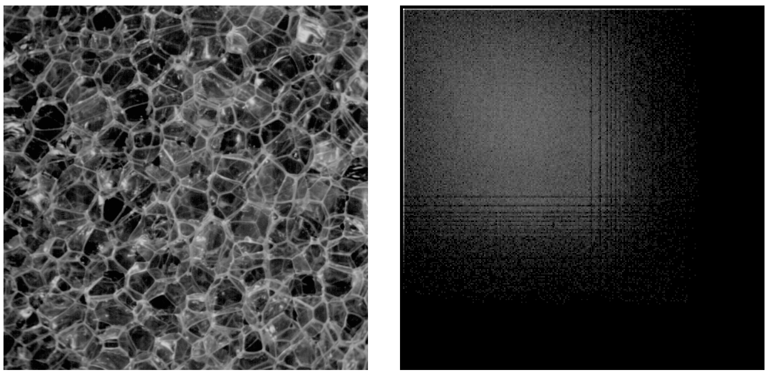
Laws' masks Laws' approach [Laws, 1979, 1980a,b] was somewhat different from that in GLCMs. He used a small number of convolution filters to identify points of high "texture energy" in images and from them characterized different textures. His masks are constructed from fairly simple 1-D ones; the set of five-element masks is

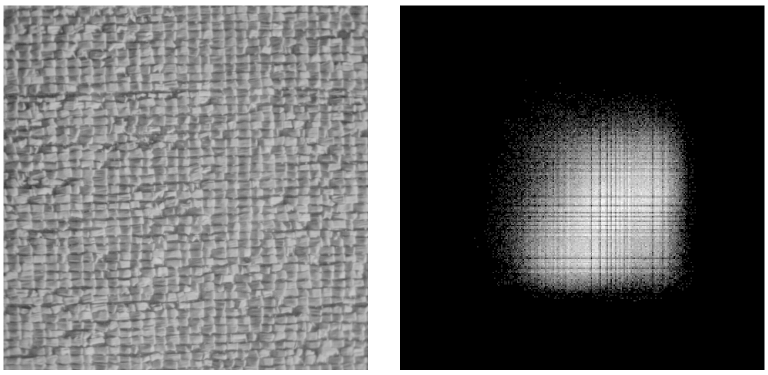
$$\begin{aligned}
\text{local average}, L_5 &= [1 \quad 4 \quad 6 \quad 4 \quad 1] \\
\text{edge detection}, E_5 &= [-1 \quad -2 \quad 0 \quad 2 \quad 1] \\
\text{spot detection}, S_5 &= [-1 \quad 0 \quad 2 \quad 0 \quad -1] \\
\text{ripple detection}, R_5 &= [1 \quad -4 \quad 6 \quad -4 \quad 1] \\
\text{wave detection}, W_5 &= [-1 \quad 2 \quad 0 \quad -2 \quad 1]
\end{aligned}$$

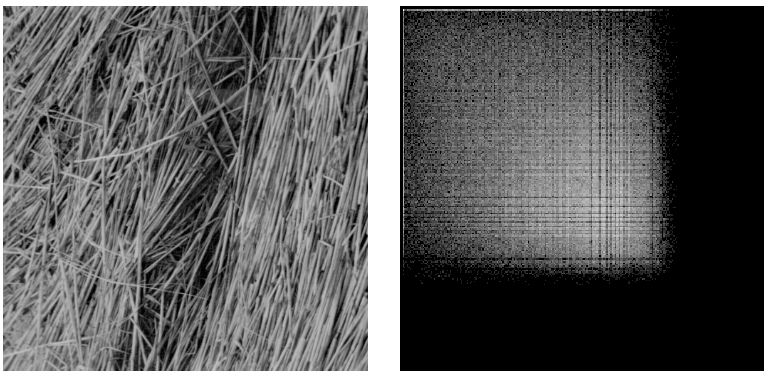
Note that the sum of elements of many, though not all, of these masks is zero for the reason discussed in Chapter 4. They are sufficiently simple that they can be used in real-time applications.

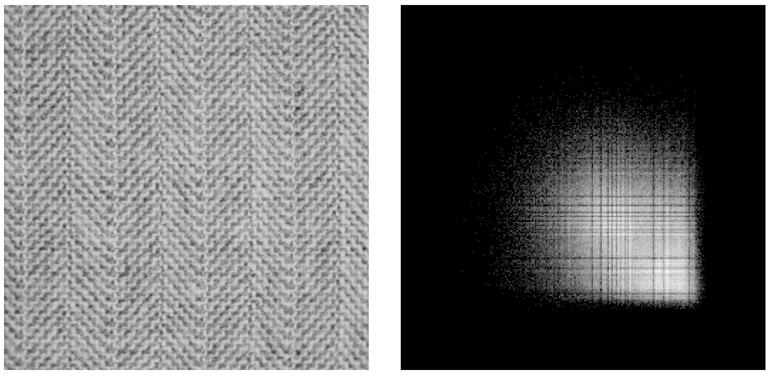
Figure 5.14: Textures and corresponding GLCMs for shifts of $X = 10, Y = 7$ (shown with a logarithmic scale of grey levels to bring out detail)

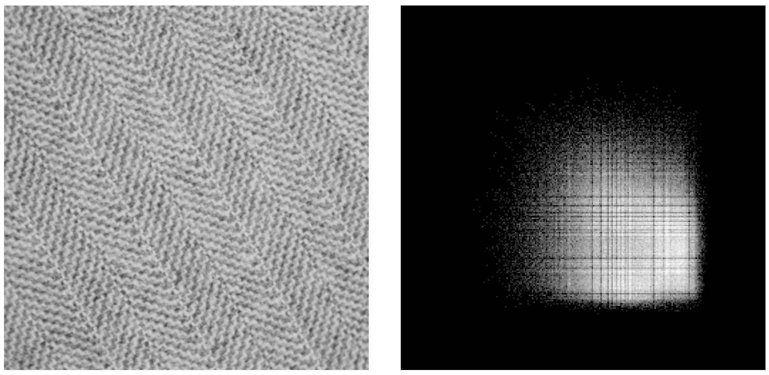

(a) Bark

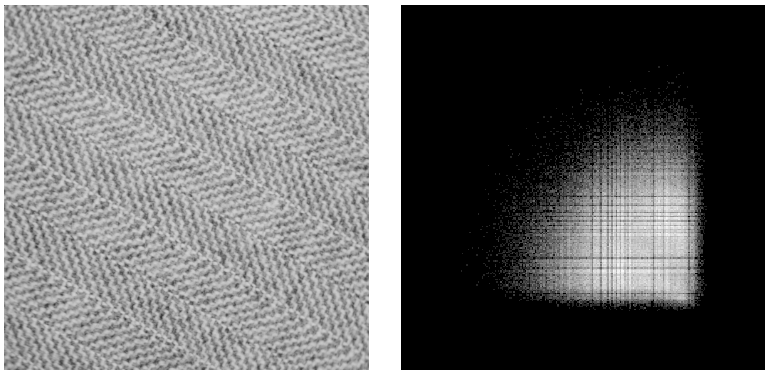

(b) Bubbles


(c) Raffia


(d) Straw


(a) 0°


(b) 30°


(c) 60°

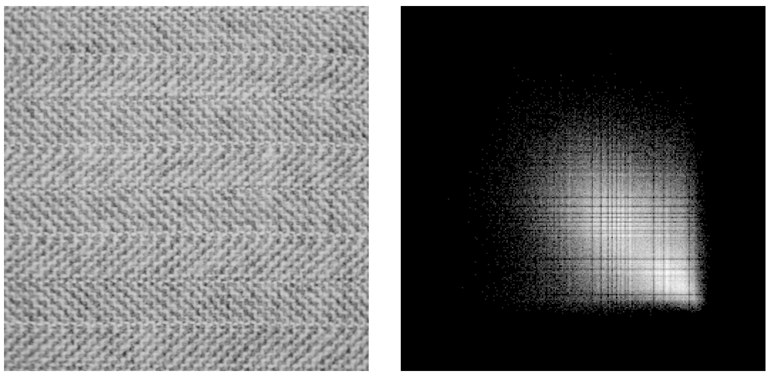
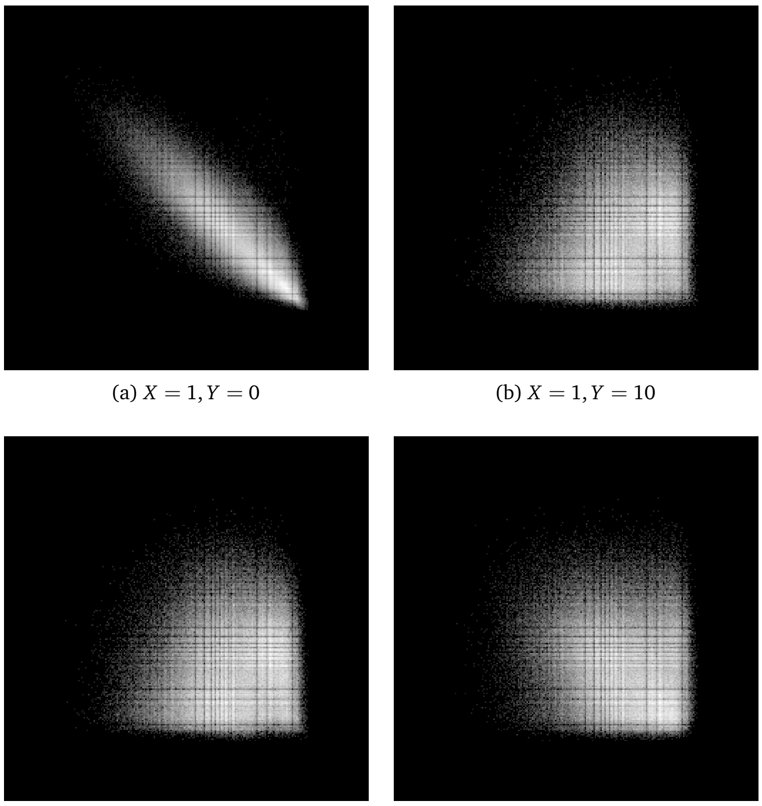

(d) 90°

Figure 5.15: Rotated textures and corresponding GLCMs for shifts of $X = 10, Y = 7$ (shown with a logarithmic scale of grey levels to bring out detail)



(a) $X = 1, Y = 0$

(b) $X = 1, Y = 10$

(c) $X = 10, Y = 0$

(d) $X = 20, Y = 0$

Figure 5.16: GLCMs for various shifts (shown with a logarithmic scale of grey levels to bring out detail)

Matrix multiplication of these 1-D masks yield the 5 × 5 ones that are actually applied to images. For example

$$E_5S_5 = \begin{pmatrix} -1 & 0 & 2 & 0 & -1 \\ -2 & 0 & 4 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & -4 & 0 & 2 \\ 1 & 0 & -2 & 0 & 1 \end{pmatrix}.$$

Masks obtained in this way are convolved with the original image to produce output images which are passed through a second stage, a moving window estimation of the energy within the images — hence the name “texture energy” above. You end up with a series of energy values from the various masks — but, just as with GLCMs, you still have to find a way of determining which ranges of energy values correspond to which texture.

6
Evaluating Vision Systems

This chapter considers the way in which vision systems are assessed. The underlying principles are expounded and techniques such as ROC curves explained. The statistical basis for comparing systems is presented, and two examples of assessment presented: the first assesses the histogram-based approach to content-based image retrieval expounded in Chapter 3 while the second explores the use of a support vector machine on a dataset of handwritten digits.

6.1 Introduction

It has already been stated that computer vision is a practical discipline, and some working — though not necessarily good — vision systems have been explored in previous chapters. We have now reached the point where a deeper understanding of how vision systems can and should be assessed is needed. Assessment is an important part of both research and development: for research, if you cannot measure how well a system works, you cannot be doing science; and in development, you want your system to be more effective than those of your competitors.

The first thing to understand is that an assessment of the effectiveness of a vision system can currently be made only by running it on imagery for which the correct answer is known — what is normally termed *ground truth*. In cases where the imagery are synthetic, such as the use of computer-generated images when we look at stereo vision in Chapter 9, the correct answer is known precisely. However, vision systems are more commonly assessed using real imagery and the 'ground truth' might actually contain inaccuracies too. Let us take, for example, the application of detecting cancerous tumours in mammograms (X-ray images of human breasts): this is an important application because breast cancer used to be the largest killer of women in the UK, so much so that the NHS introduced a regular screening programme. Whether a particular feature in an X-ray is a tumour or not is assessed by a radiologist and it is natural to use their identifications as the 'ground truth' for training a vision system. However, if the radiologist has made an incorrect diagnosis of a particular feature, the assessment of the effectiveness of the vision system is measuring how well it agrees that the radiologist's opinion rather than the underlying truth of whether or not features are cancerous. For that reason, ground truth imagery is often assessed by several experts and their consensus taken.

There are cases in which ground truth capture need not involve experts. The Galaxy Zoo project and its successors invite 'citizen scientists' to classify images for which automatic classification approaches work poorly. As well as yielding results of direct interest to scientists, these kinds of project deliver large quantities of valuable training data to computer vision researchers: for example, the author has about 800,000 images with ground truth classification from the first Galaxy Zoo project, two orders of magnitude more than would normally be available. Crowd-sourcing has much value in computer vision.

Most modern vision systems involve an element of adaption. This may be a case of the developer tailoring the values of convolution masks *etc*. to work well with the imagery or, more commonly, it may involve using a machine learning technique. Either way, it is essential that the data used for developing or training the vision system are disjoint (*i.e.,* are separate from) the data used for testing.

6.2 Evaluating a vision system

Almost all vision systems take in images and yield labelled results. In the mammogram example discussed above, a labelled result might be 'cancerous' or 'benign' for each feature identified in an image. For a face recognition system, an image would be labelled with the identity of a person. An assessment procedure must present images from the test set to the vision system in turn and see whether it identifies features and assigns them the correct labels. The vision community usually works on the basis that there are four possible outcomes:

true positive (TP): the feature has been found and allocated the correct label;

false positive (FP): the feature has been found and allocated an incorrect label;

false negative (FN): a feature that should have been found was not;

true negative (TN): a feature was not present and no feature was found.

This is, in this author's view, an over-simplification as it does not, for example, distinguish between a feature that was missed and a program crash, or for totally spurious results.

Two of these cases may seem strange at first sight and so merit some explanation. The *true negative* would occur if we were trying to identify cancers in mammograms and we presented an image of, say, a giraffe: the vision system definitely shouldn't identify any lesions. A *false positive* arises if, say, a lesion was correctly found in a mammogram but identified as benign when it is actually cancerous. In this context this is a dangerous mistake to make, much more so than the identifying a benign tumour as cancerous — you should think about why this is so.

It should be appreciated that there is always a trade-off between true positive and false positive detection. If an algorithm is tuned to detect all the true positive cases then it will also tend to give a larger number of false positives. Conversely, if it is set to minimize false positive detection then the number of true positives it detects will likely be reduced too. This is an important factor to bear in mind when tailoring a vision system to a particular task.

Tables of true positives *etc*. are difficult to analyze and compare, so results are frequently shown graphically using ROC or precision–recall curves. A ROC ("receiver operating characteristic") curve is a plot of false positive rate against true positive rate as some tuning parameter in the algorithm is varied. ROC curves were developed to assess the performance of radar operators during the second World War: operators had to make the distinction between friend or foe targets, and also between targets and noise, from the blips they saw on their screens. Their ability to make these vital distinctions was called the receiver operating characteristic. These curves were taken up by the medical profession in the 1970s, who found them useful in bringing out the *sensitivity* (true positive rate) and *specifity* ($1 -$ false positive rate). ROC curves are as interpreted as follows (see Figure 6.1):

- the closer the curve approaches the top left-hand corner of the plot, the more effective the technique is;
- the closer the curve is to a 45° diagonal, the worse it performs;
- the area under the curve is a measure of the accuracy of the technique;
- the plot highlights the trade-off between the true positive rate and the false positive rate: an increase in true positive rate is usually accompanied by an increase in false positive rate.

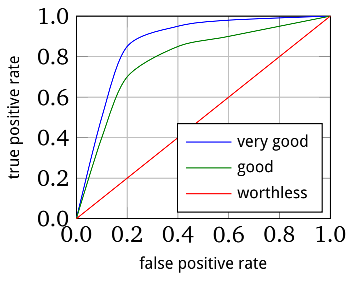


Figure 6.1: Examples of ROC curves. Curves closer to the upper left corner indicate better performance, as that true positive rate is being achieved with a lower false positive rate. Note that ROC curves are often drawn with logarithmic axes to separate performances close to the upper left corner.

Figure 6.1 shows ROC curves for very good, good and poor (worthless) tests. In practice, one usually finds that ROC curves cross, as in Figure 6.2; which system one would then use depends on whether a high false positive rate is too large a price to pay for a high true positive one:

- For cancer detection say, we want the TP rate to be as high as possible so we choose alg1 of Figure 6.2 and operate it close to the right edge of the plot, accepting that it will generate more FPs.
- Conversely, for a task such as secure access by face recognition, we want the FP rate to be as low as possible so we choose alg2 and operate it near the left edge of Figure 6.2, accepting that people may have to try several times before being authenticated.

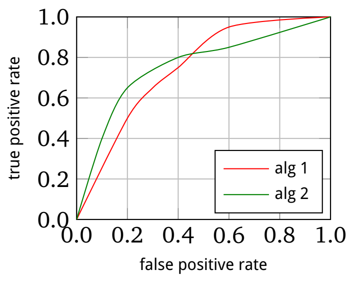


Figure 6.2: ROC curves that cross means that one has to be careful about the settings of tuning parameters in order to achieve best performance.

Precision and recall are calculated from TP *etc*. as follows:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6.1}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6.2}$$

We also sometimes see these combined into a single quantity, the *F-measure*:

$$F = 2\,\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{6.3}$$

A precision–recall curve obviously has some relationship to a ROC curve as both encode similar information. Whereas a ROC curve is regarded as better if it is closer to the top left corner of the graph, a precision–recall curve is better if it closer to the top right corner. The major advantage of precision–recall curves over ROC curves is that the former take account of true negatives, *i.e.* the system rejecting incorrect data. For this reason, precision–recall curves are slowly replacing ROC curves in assessments of vision systems.

The author's experience is that measures such as F are of little practical value in assessing the performance of a vision system. Although less well known, we have found that Matthews correlation coefficient (MCC or ϕ) is more meaningful, especially because its value is bounded into ± 1. Using the notation introduced above, we can define the MCC as

$$\phi = \frac{\mathrm{TN} \times \mathrm{TP} - \mathrm{FN} \times \mathrm{FP}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}} \tag{6.4}$$

6.3 Comparing vision systems

The most common way that algorithms are compared in the literature is by means of their ROC or precision–recall curves. Curves from algorithms being compared often cross each other, and then it is up to the user to decide which represents the best method for their application. As a result, comparisons of algorithms tend to be performed with a specific set of tuning parameter values, depending on the target application.

It is becoming common to compare the performance of vision systems using the *area under the curve* on a ROC plot, an approximation to the integral of the curve. The idea is that the larger this value, the closer it is to the top left-hand corner and so the better it is. However, this is *a particularly poor way of performing such a comparison*, and for two reasons. Firstly, this approach makes no sense in any operational context: a vision system is operated with a single set of tuning values and hence at a single point on the ROC curve as discused above. The critical test is which algorithm fulfils the requirements for true and false classifications, not how well it performs with different tuning values. The second reason is that it takes no account of the amount of test data involved: a 2% difference in performance in likely to be significant if 500,000 images have been processed but not if the number of images was 50.

Thankfully, there is a better way. If one is able to run algorithms on the same data and record the outcome of each individual test — perhaps under the control of a test harness such as FACT from the laboratory sessions for this module — an appropriate statistical test can be employed that takes into account not only the number of false positives *etc.* but also the number of tests. Although not yet in widespread use in the computer vision research community, the appropriate test to employ for this type of comparison is McNemar's test. This is a form of the sign test for matched paired data. Consider the 2×2 table of results for two algorithms in Table 6.1.

	algorithm A failed	algorithm A succeeded
algorithm B failed	N_{ff}	N_{sf}
algorithm B succeeded	N_{fs}	N_{ss}

Table 6.1: Possible outcomes for a single test from a pair of algorithms

The form of McNemar's test that we shall use is:

$$Z^2 = \frac{(|N_{sf} - N_{fs}| - 1)^2}{(N_{sf} + N_{fs})} \tag{6.5}$$

where the -1 is a continuity correction. We see that the test employs both false positives and false negatives, rather than just one of them. If $N_{sf} + N_{fs}$ (*i.e.,* the number of tests where the algorithms differ) is greater than about 20, then the value of the "Z-score" Z will be meaningful.

If algorithm A and algorithm B give similar results then Z will be near zero. As their results diverge, Z will increase. Confidence limits can be associated with the Z value as shown in Table 6.2; it is normal to look for $Z > 1.96$, which means that the results from the algorithms would be expected to differ by chance about one time in 20. Values for two-tailed and one-tailed predictions are shown in the table as either may be needed, depending on the hypothesis used: if we assessing whether the performances of two algorithms differ, a two-tailed test should be used; but if we are determining whether one algorithm performs better than the other, a one-tailed test is needed.

Z value	two-tailed confidence	one-tailed confidence
1.646	90%	95.0%
1.960	95%	97.5%
2.326	98%	99.0%
2.576	99%	99.5%

Table 6.2: Converting Z values onto confidence limits

Figure 6.3 shows the tails of a binomial distribution: if we want to ascertain whether the performances differ, we are interested in either of the two shaded regions; but if we are asking whether one algorithm out-performs the other, we use only one tail.

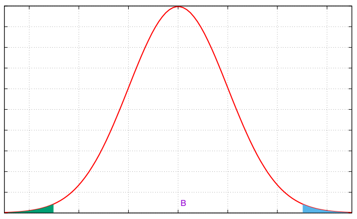


Figure 6.3: The tails of a binomial distribution

Further information can also be gleaned from N_{sf} and N_{fs}: if these values are both large, then we have found places where algorithm A succeeded while algorithm B failed and *vice versa*. This is valuable to know, as we can devise a new algorithm that uses both in parallel and takes the value of algorithm B where algorithm A fails, and *vice versa* — this should yield an overall improvement in accuracy. This is actually a significant statement with regard to the design of vision systems: rather than combining the results from algorithms in the rather *ad hoc* manner that usually takes place, McNemar's test provides a principled approach that tells us not only *how to do it* but also *when it is appropriate to do so* on the basis of evaluation — in other words, evaluation needs to be an inherent part of the algorithm design process.

As a footnote to this discussion of McNemar's test, the reason that the FACT program you use in laboratories works on *files of results* rather than just providing a user-callable routine in a module or class is to promote the sharing of results without also having to give away the software that created them.

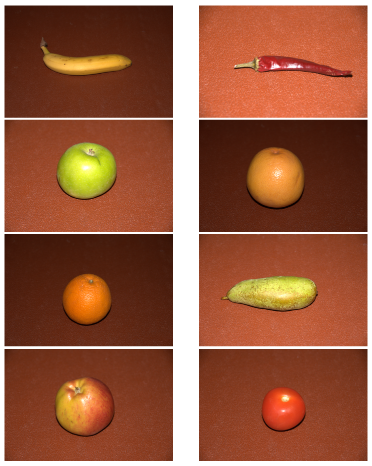


Figure 6.4: The various classes of fruit used for assessing `colrec1`: banana, chili, green apple, grapefruit, orange, pear, red apple, tomato. (Yes, I know some people would argue that a tomato is not a fruit.)

6.4 Assessing `colrec1`

It is fairly straightforward to assess `colrec1` from Chapter 3 using the FACT test harness alluded to above. A series of test images of fruit were captured against a plain background, as shown in Figure 6.4, and used as tests. Tables 6.3 and 6.4 are the result — try to interpret them before reading on. Note that some classes use abbreviated names to stop the columns becoming too spread out.

Firstly, it is notable that there are only ten tests for each class, clearly not enough for any resulting statistics to be reliable — at least 20 are needed if the dataset is well-behaved, and about one hundred would give more confidence in any conclusions drawn. Having said that, it is clear that `colrec1` is not particularly effective. We see that all ten test images

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
10	10	0	0	0	1.00	1.00	1.00	0.00	banana
10	7	0	3	0	0.70	1.00	0.70	0.00	chili
10	6	0	4	0	0.60	1.00	0.60	0.00	gapple
10	10	0	0	0	1.00	1.00	1.00	0.00	gfruit
10	8	0	2	0	0.80	1.00	0.80	0.00	orange
10	9	0	1	0	0.90	1.00	0.90	0.00	pear
10	5	0	5	0	0.50	1.00	0.50	0.00	rapple
10	9	0	1	0	0.90	1.00	0.90	0.00	tomato
80	64	0	16	0	0.80	1.00	0.80	0.00	overall

Table 6.3: Error rates from `colrec1.res`

	expected							
actual	banana	chili	gapple	gfruit	orange	pear	rapple	tomato
banana	10	0	0	0	0	0	0	0
chili	0	7	0	0	0	0	0	1
gapple	0	0	6	0	0	1	4	0
gfruit	0	0	0	10	2	0	0	0
orange	0	0	0	0	8	0	0	0
pear	0	0	0	0	0	9	0	0
rapple	0	0	3	0	0	0	5	0
tomato	0	3	1	0	0	0	1	9

Table 6.4: Class confusion matrix calculated from `colrec1.res`

were recognised correctly only for banana and grapefruit, while red apples were recognised poorly. In this case, the 'recall' and 'specificity' columns of Table 6.3 are of little value.

Table 6.4 summarizes what the test images were classified as. Here, the element at row r and column c indicates that class r was obtained when class c should have been found; for example, there were three cases in which a chili was incorrectly identified as a tomato and one case where a tomato was incorrectly identified as a chili. Intuitively, we might have expected these to be confused more often because they are of similar colours. Interestingly, the red apple was often confused with a green one, indicating that the way `colrec1` bundles its colours together when determining the histograms to compare is a poor one.

6.5 Recognising Handwritten Digits

As a second example of evaluation, we turn our attention to one of the standard test cases for machine learning, the MNIST database of handwritten digits [LeCun et al., 1998]. This is because the database has been carefully pre-processed to remove variations other than the shapes of the characters written. The training set is large yet the individual images are small, only 28×28 pixels, so that the overall computation time remains manageable. Some typical examples are shown in Figure 6.5 and the sizes of the training and test sets are presented in Table 6.5.

Rather than use `colrec1`, which we now know is pretty poor at doing this kind of thing, we shall use MNIST with a fairly state-of-the-art machine learning algorithm, a *support vector machine* (SVM). A SVM was trained using the 60,000 training images (which takes about 10 minutes on the author's computer) and then tested on the 10,000 test images; tables 6.6 and 6.7 present the statistics and class confusion matrix respectively on the test images. (These were calculated using FACT's big brother, which explains why they look a little different.) Accuracy on all the digit classes is good, with 96% accuracy being obtained in each case. The class confusion matrix shows that there are few places where one digit is consistently mistaken for another.

One thing that these tables show that were missing in those from FACT concerning fruit classification is the mention of classes called `reject` and `fail`. `reject` occurs when the program being tested believes the image presented to it is invalid (in this case, rather than a digit, the image might contain a coffee cup or giraffe). Conversely, `fail` occurs when the program crashes. A good program will have no `fail` cases!

In most evaluation studies in the computer vision domain, there are few datasets that explicitly test the `reject` case — there is no testing for true negatives, using the terminology introduced earlier in this chapter. The author believes that this is a mistake, being one of the major reasons that computer vision systems have a reputation for being *fragile*, seemingly working well in the research laboratory but failing dismally when used in the field.

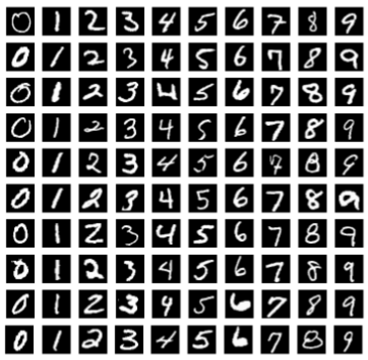


Figure 6.5: Examples from the MNIST database

class	training	test
0	5923	980
1	6742	1135
2	5958	1032
3	6131	1010
4	5842	982
5	5421	892
6	5918	958
7	6265	1028
8	5851	974
9	5949	1009

Table 6.5: Distribution of the 60,000 training and 10,000 test images amongst the classes in the MNIST database

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
980	972	0	8	0	0.9918	1.0000	0.9918	0.0000	0
1135	1126	0	9	0	0.9921	1.0000	0.9921	0.0000	1
1032	1013	0	19	0	0.9816	1.0000	0.9816	0.0000	2
1010	993	0	17	0	0.9832	1.0000	0.9832	0.0000	3
982	963	0	19	0	0.9807	1.0000	0.9807	0.0000	4
892	867	0	25	0	0.9720	1.0000	0.9720	0.0000	5
958	944	0	14	0	0.9854	1.0000	0.9854	0.0000	6
1028	997	0	31	0	0.9698	1.0000	0.9698	0.0000	7
974	949	0	25	0	0.9743	1.0000	0.9743	0.0000	8
1009	973	0	36	0	0.9643	1.0000	0.9643	0.0000	9
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	reject
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	fail
10000	9797	0	203	0	0.9797	1.0000	0.9797	0.0000	overall

Table 6.6: Table of error rates for T1.000

A similar set of results, this time training the SVM with significantly different settings, is shown in tables 6.8 and 6.9. It can be seen, though with some difficulty, that the accuracy is worse and the classes significantly more confused. However, it is not clear whether the performance differences are *significantly* different in the statistical sense.

When a comparison is made using McNemar's test (Table 6.10), the difference in performance is much more apparent. The one case where

true	class returned by algorithm											
class	0	1	2	3	4	5	6	7	8	9	reject	fail
0	972	0	1	0	0	3	1	1	2	0	0	0
1	0	1126	3	1	0	1	1	1	2	0	0	0
2	5	1	1013	0	1	0	1	7	3	1	0	0
3	0	0	2	993	0	2	0	6	5	2	0	0
4	0	0	5	0	963	0	3	0	1	10	0	0
5	3	0	0	10	1	867	4	1	4	2	0	0
6	5	2	1	0	2	3	944	0	1	0	0	0
7	1	7	10	2	2	0	0	997	1	8	0	0
8	3	0	2	6	5	2	2	2	949	3	0	0
9	3	3	1	7	10	1	1	7	3	973	0	0
reject	0	0	0	0	0	0	0	0	0	0	0	0
fail	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.7: Class confusion matrix for T1.000

T0.400 out-performs T1.000 is for digit '1', and the number of times this happens is not enough to make the value of Z significant, yet the T1.000 case is the better for every other digit and overall, with $Z^2 = 5553$, three orders of magnitude larger than the critical value of 1.96!

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
980	189	0	791	0	0.1929	1.0000	0.1929	0.0000	0
1135	1127	0	8	0	0.9930	1.0000	0.9930	0.0000	1
1032	416	0	616	0	0.4031	1.0000	0.4031	0.0000	2
1010	77	0	933	0	0.0762	1.0000	0.0762	0.0000	3
982	843	0	139	0	0.8585	1.0000	0.8585	0.0000	4
892	546	0	346	0	0.6121	1.0000	0.6121	0.0000	5
958	136	0	822	0	0.1420	1.0000	0.1420	0.0000	6
1028	801	0	227	0	0.7792	1.0000	0.7792	0.0000	7
974	28	0	946	0	0.0287	1.0000	0.0287	0.0000	8
1009	27	0	982	0	0.0268	1.0000	0.0268	0.0000	9
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	reject
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	fail
10000	4190	0	5810	0	0.4190	1.0000	0.4190	0.0000	overall

Table 6.8: Table of error rates for T0.400

6.6 Comparing Performance Figures on Different Datasets

The comparison process described above depends critically on having access to the same dataset. This is reasonably common these days but authors rarely make available either their software or the outcomes for each of the test inputs, the kind of information needed for McNemar's test. However, it still is possible to carry out some kind of statistical examination of the performance figures, as long as there is some indication of accuracy.

true	class returned by algorithm											
class	0	1	2	3	4	5	6	7	8	9	reject	fail
0	189	3	508	0	86	112	0	82	0	0	0	0
1	0	1127	0	0	0	6	0	0	2	0	0	0
2	0	452	416	0	125	17	0	22	0	0	0	0
3	0	688	101	77	16	95	0	33	0	0	0	0
4	0	68	21	0	843	2	0	47	0	1	0	0
5	0	136	58	1	107	546	0	44	0	0	0	0
6	0	96	205	0	396	104	136	21	0	0	0	0
7	0	115	68	0	36	7	0	801	0	1	0	0
8	0	498	120	6	207	101	0	11	28	3	0	0
9	0	130	28	1	526	22	0	275	0	27	0	0
reject	0	0	0	0	0	0	0	0	0	0	0	0
fail	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.9: Confusion matrix for T0.400

N_{ss}	N_{sf}	N_{fs}	N_{ff}	Z	better	class
189	783	0	8	781.0013	T1.000	0
1121	5	6	3	0.0000	T0.400	1
414	599	2	17	591.0416	T1.000	2
77	916	0	17	914.0011	T1.000	3
838	125	5	14	108.9308	T1.000	4
536	331	10	15	300.2933	T1.000	5
136	808	0	14	806.0012	T1.000	6
798	199	3	28	188.2426	T1.000	7
28	921	0	25	919.0011	T1.000	8
27	946	0	36	944.0011	T1.000	9
0	0	0	0	0.0000	neither	reject
0	0	0	0	0.0000	neither	fail
4164	5633	26	177	5553.4964	T1.000	overall

Table 6.10: Comparison of results for T1.000 and T0.400 using McNemar's test

Let us assume we want to compare two accuracy figures, S_1 successes from N_1 tests and S_2 successes from N_2 tests. These mean we have probabilities of success given by

$$p_1 = \frac{S_1}{N_1} \text{ and } p_1 = \frac{S_2}{N_2}.$$

We first compute the factor

$$\hat{P} = \frac{S_1 + S_2}{N_1 + N_2} \tag{6.6}$$

and then the test statistic

$$Z = \frac{|p_1 - p_2|}{\sqrt{\hat{P}(1-\hat{P})\left(\frac{1}{N_1} + \frac{1}{N_2}\right)}} \tag{6.7}$$

Exactly as with McNemar's test, you can use the critical value of $Z = 1.96$ to decide whether or not the two performance figures are different (at the one in twenty or 5% level). For the interpretation to be valid, you need to be confident that the two datasets involved are equally difficult for the vision algorithms you are examining.

7
Intermediate-Level Vision

The techniques examined in Chapter 5 work on regions of images without any knowledge of what those regions are. In this chapter, we explore finding features such as edges and corners and how they may be described. We then take a look at SIFT and similar techniques, which are able to match features in one image with the same features in other images. We close the chapter by examining a technique that use a similar approach for identifying texture.

7.1 Introduction

Detecting the various types of feature mentioned in Chapter 5 is not the end of the story. For example, in developing a "co-pilot" aid for a person driving a car discussed later in this chapter, we might want to identify the white lines painted along the edges of many UK roads so that we can alert the driver if they stray out of their lane. We shall start with the detection of straight lines and corners, as they turn out to be useful in many contexts, then move on to consider describing such features. We first look at a technique known as the Hough transform for pulling the equation of lines from images. We then look at techniques matching features found in images as this capability underlies some of the most important modern vision applications.

7.2 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.

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. The development of the Canny edge detector actually 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 became 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. He has now retired. 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.

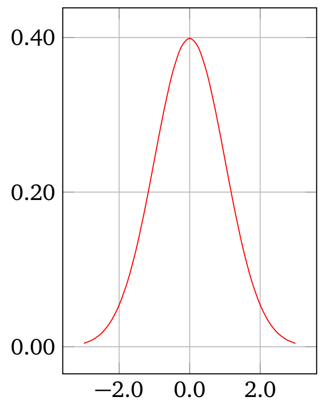


Figure 7.1: Profile across a Gaussian mask

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.

Step 1. Convolve the image with a Gaussian-shaped mask (see Figure 7.1) 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.

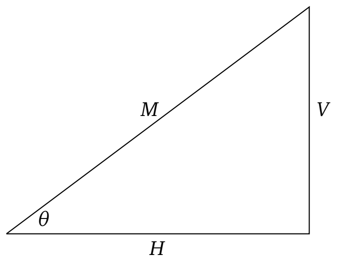


Figure 7.2: Obtaining M and θ from H and V

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

$$H(x,y) = \frac{(S(x,y+1) - S(x,y)) + (S(x+1,y+1) - S(x+1,y))}{2} \tag{7.1}$$

$$V(x,y) = \frac{(S(x+1,y) - S(x,y)) + (S(x+1,y+1) - S(x,y+1))}{2} \tag{7.2}$$

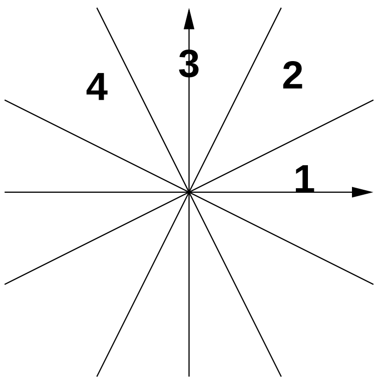


Figure 7.3: Quantization of $\theta(x,y)$ into four directions

Step 3. Find the magnitudes and directions of these gradients (Figure 7.2):

$$M(x,y) = \sqrt{V(x,y)^2 + H(x,y)^2} \tag{7.3}$$

$$\theta(x,y) = \tan^{-1} \frac{V(x,y)}{H(x,y)} \tag{7.4}$$

Note that, when calculating $\theta(x,y)$ on a computer, you need to use the `atan2` function, which returns correctly 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.

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 value. This is carried out in two stages (Figure 7.4):

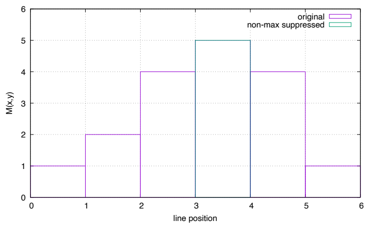


Figure 7.4: Non-maximum suppression

- Firstly, $\theta(x, y)$ is quantised into four values that indicate roughly the direction of the edge gradient (Figure 7.3).
- 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.

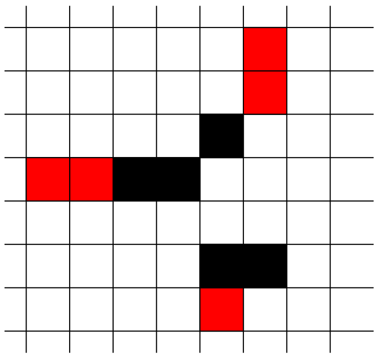
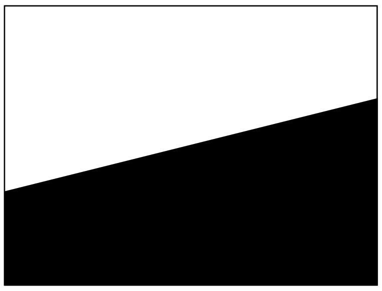


Figure 7.5: Linking together edge segments using hysteresis thresholding

When the contour formed by following successive pixels with values $> \tau_H$ (shown in red in Figure 7.5) 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 7.5). If so, it continues the edge to that pixel. This process continues until there are no suitable neighbours or another edge segment with value $> \tau_H$ has been found.

As we can see from Figure 7.8, 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.


(a) You cannot tell which part of an edge you see

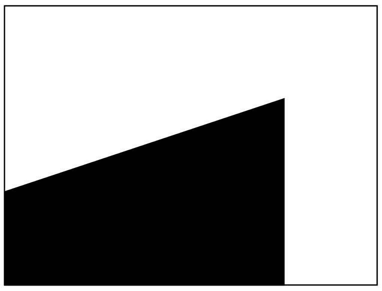

(b) It is easy when a corner is visible

Figure 7.6: The aperture problem

Edge detection, no matter how well done, suffers from a significant drawback for image interpretation. As Figure 7.6 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. So let us look at how to find corners.

7.3 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 7.7:

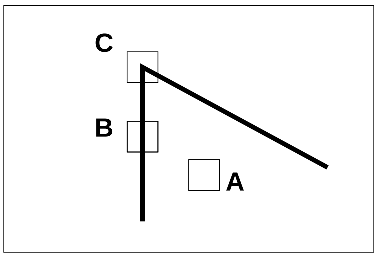


Figure 7.7: The different cases considered in the Moravec corner detector

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

$$E(x,y) = \sum_{u,v} W(u,v)\left(I(x+u,y+v) - I(u,v)\right)^2. \qquad (7.5)$$

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 7.3), obtaining four values corresponding to the four shifts. We retain the *minimum* of the four at each and store the result in an array the same size as the image. We then look for local *maxima* in this array above some threshold value to identify corners.

7.4 The corner detector of Harris and Stephens

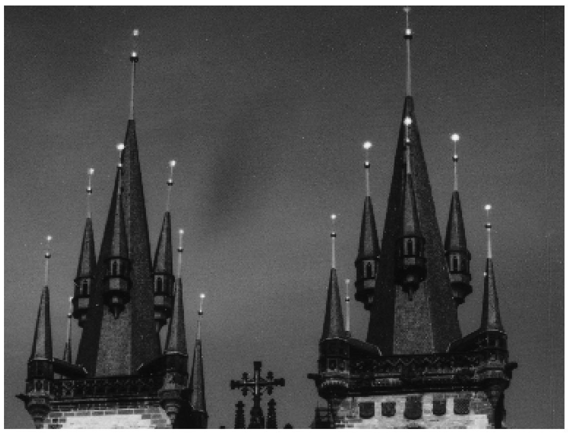
As mentioned above, the Moravec detector represents the state of the art in around 1987. Subsequent researchers have improved on this, in particular in work of [Harris and Stephens, 1988], which combines the ideas of Moravec and Canny and has been the most widely-used corner detector since the early 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.

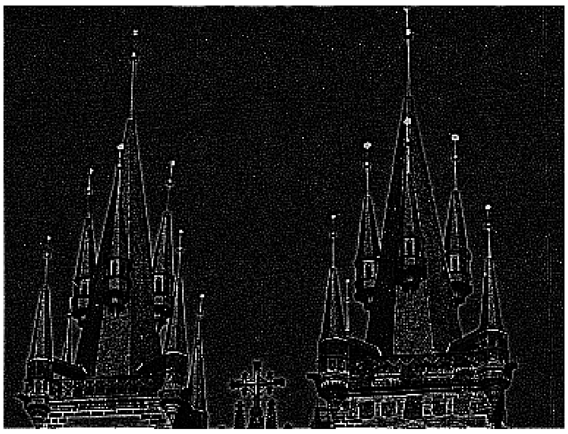
Consider a patch of a grey-scale image $I(u,v)$ shifted by (x,y). The weighted sum-squared difference is given by (7.5). If one expands $I(u+x,v+y)$ as a Taylor series then, if I_x and I_y are the partial derivatives of I in the x- and y-directions respectively,

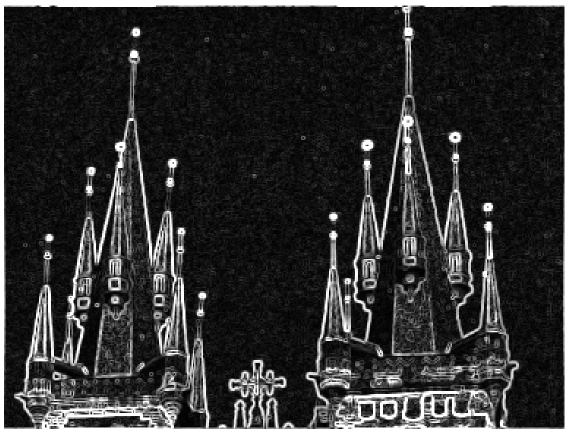
$$I(u+x,v+y) \approx I(u,v) + xI_x(u,v) + yI_y(u,v)$$

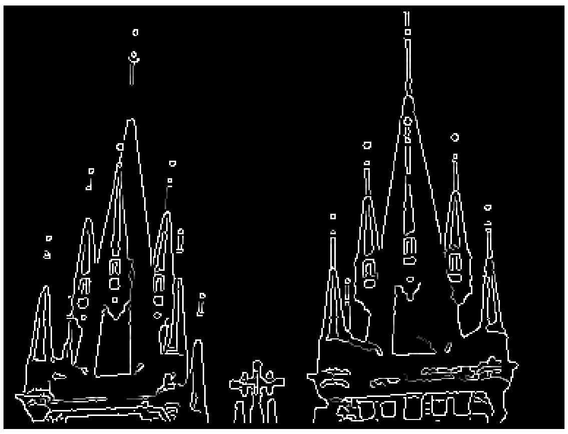
which means that

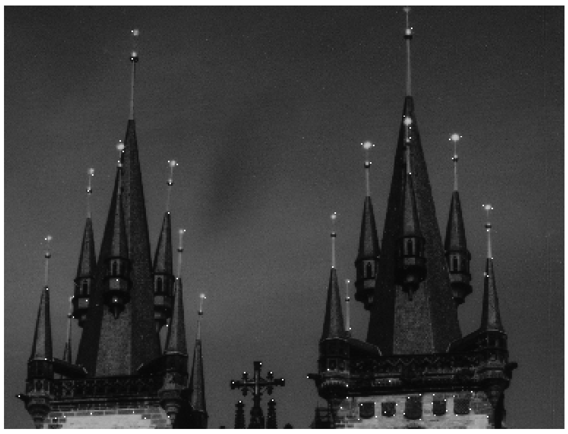
$$E(x,y) \approx \sum_{u,v} w(u,v)\left(xI(u,v) + yI(u,v)\right)^2.$$


(a) Original image


(b) Result of Laplacian


(c) Result of Sobel


(d) Result of Canny


(e) Result of Harris & Stephens

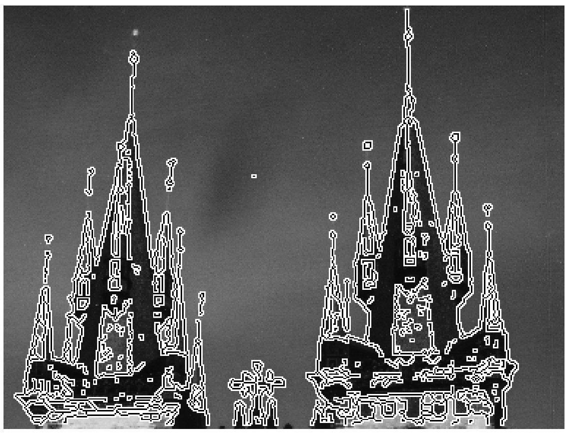

(f) Result of SUSAN

Figure 7.8: Comparing the results of different edge and feature detectors

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 7.8. These days, it is able to run at video rate on off-the-shelf hardware.

7.5 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 7.8.

Secondly, Edward Rosten and colleagues at Cambridge developed the FAST corner detector [Rosten et al., 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 7.9. Although some sophistication is required to make FAST work well, a C implementation is easily able to operate at video rates. The author's experience of FAST is that it yields huge numbers of spurious corners.

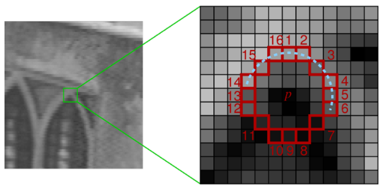


Figure 7.9: The FAST corner detector (from `http://mi.eng.cam.ac.uk/~er258/work/fast.html`)

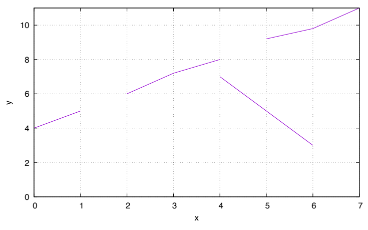


Figure 7.10: Edge detectors usually deliver only segments of edge

7.6 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 (Figure 7.10). 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

$$y = ax + b \tag{7.6}$$

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, though rather in the spirit of the second alternative discussed above. If we re-write (7.6) in the form

$$b = -xa + y \tag{7.7}$$

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 7.11. 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.

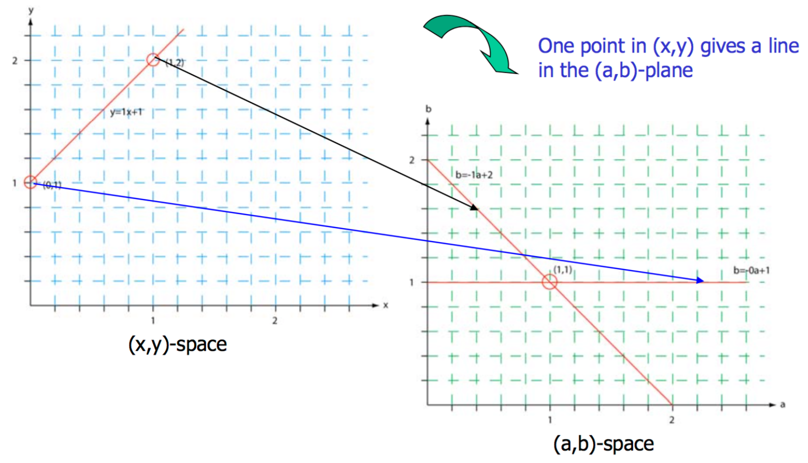


Figure 7.11: Illustration of computing the Hough transformation (from Anne Solberg's lecture notes)

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

$$x \cos\theta + y \sin\theta = r \tag{7.8}$$

using parameters r and θ — see Figure 7.12 [Duda and Hart, 1973]. Each point in the (x, y)-plane gives a sinusoid in the (r, θ)-plane, so M co-linear point lying on the line of (7.8) gives rise to M curves that intersect at (r, θ) in the parameter plane.

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 7.13.

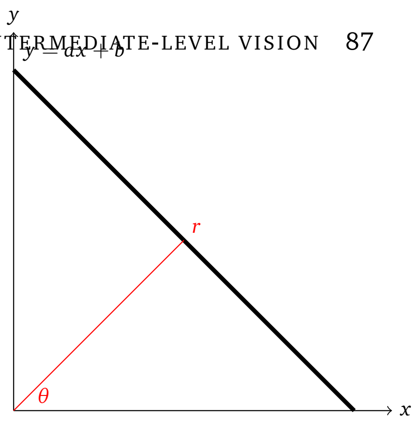


Figure 7.12: Conventional and parametric representations of a line

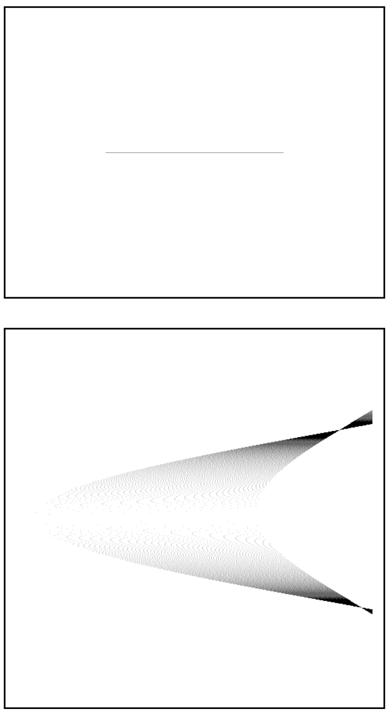


Figure 7.13: An image and its Hough transform (presented so that black is larger)

⟨*Hough transform*⟩ ≡

```
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=300, na=200, threshold=10, max_peaks=None):
    im = reshape3 (im)
    ny, nx, nc = sizes (im)
    acc = image ((nr, na, 1))
    ainc = math.pi / na
    rinc = math.sqrt (ny**2 + nx**2) / nr

    # Fill arrays with the radius and angle values, to be returned.
    rvals = []
    for i in range (0, nr):
        rvals += [i * rinc]

    avals = []
    for i in range (0, na):
        avals += [i * ainc]

    # Find non-zero points and update the Hough accumulator.
    for y in range (0, ny):
        for x in range (0, nx):
            val = im[y,x,0]
            if val > 0:
                for ia in range (0, na):
                    ang = ia * ainc
                    r = x * math.cos(ang) + y * math.sin (ang)
                    ir = int (r / rinc)
                    acc[ir,ia,0] += 1

    # Find peaks in the accumulator.
    peaks = find_peaks (acc, threshold=threshold)
    if max_peaks is None:
        max_peaks = len (peaks)

    return peaks, acc, rvals, avals
```

What kinds of things are the Hough transform used for? The answer is really anywhere you need to find lines from line segments. For example, Figure 7.14 shows it being used as part of a driver's aid, alerting the driver whenever they are found to be straying outside their lane.

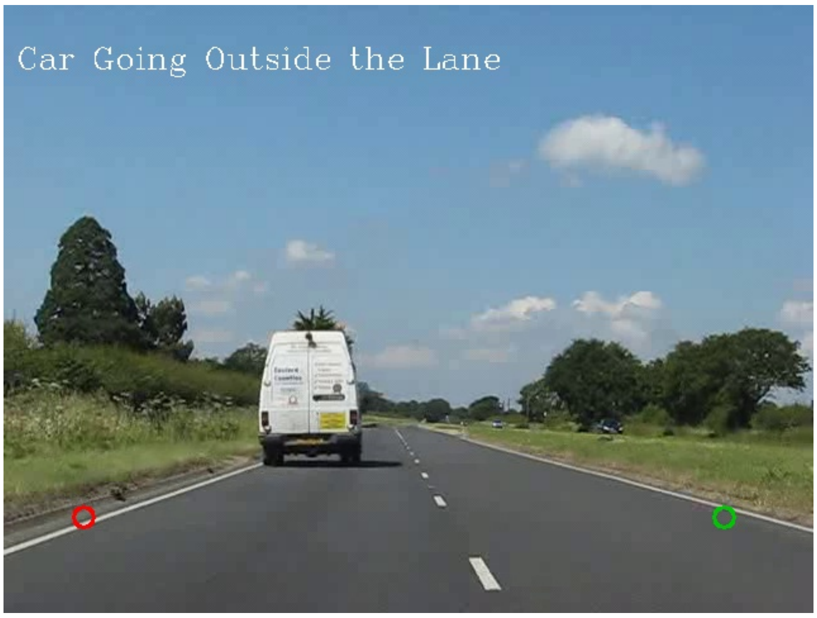


Figure 7.14: The Hough transform can be used in a lane-following system

A couple of footnotes are in order here. A line running diagonally across the middle of the image must contain more pixels than a short 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. Other minor improvements are also possible.

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.

7.7 Describing corners

Detecting corners may help us detect the boundaries of a feature (an obstacle in front of a robot, say) but that is of limited value. It is much more useful if one is able to relate the positions found in one frame of a video sequence to those in the next frame; or, with a pair of cameras looking at a scene, which feature found in the image from the left camera image matches a feature found in the right camera one. To be able to do that, one needs to be able to *describe* corners in some way, and then see how similar the descriptors are to those calculated from different frames or cameras.

The best identifying characteristic of a corner is its internal angle, and here at Essex we have shown [Kanwal et al., 2014] that this can be found from the values calculated as part of the Harris and Stephens corner detector. This forms the basis of a way of *describing* and *matching* corners — and our ARC descriptor is able to do so for thousands of corners at video rate on modest hardware. We found that its performance was similar to that of the BRIEF [Calonder et al., 2010] descriptor we shall consider below. However, neither ARC or BRIEF copes well with changes of orientation or scale, such as when a robot is moving rapidly towards an obstacle. For real-time vision, the current state of the art is the ORB ("oriented BRIEF") [Rublee et al., 2011] descriptor.

The way in which BRIEF works is surprisingly simple. Given a patch of size $S \times S$ encompassing a corner in the image I, some N locations (x_i, y_i) in the patch are chosen and a series of 'tests' are made for different combinations of i and j:

$$\tau = \begin{cases} 1 & \text{if } I(x_i, y_i) < I(x_j, y_j) \\ 0 & \text{otherwise} \end{cases} \tag{7.9}$$

Each of these comparisons yields a single binary number, so it is easy to combine N these into a string of bits — typical values of N are 128, 256 and 512. (You will see when we consider machine learning techniques that there are similarities between this approach and the one used by the WISARD 'neural network' discussed in Chapter 10.) Different approaches to choosing the N locations in the patch encompassing the corner yield descriptors with slightly different performances; it seems that selecting the positions randomly is as good a scheme as any other, though one has to be careful to ensure the same random positions are used whenever a descriptor is calculated.

Comparing the bit-strings from different patches is done by calculating the *Hamming distance*, the number of positions for which the bit-strings differ, and this can be done using the exclusive-OR (XOR) operator — which is why BRIEF descriptors can be matched at video rates.

The problem with BRIEF, like our ARC descriptor, is that it is affected badly by rotations in the image — the randomly-chosen places at which the comparisons take place do not lie in the same places relative to the rotated corner. This is circumvented to some extent in ORB: it positions the corner in the centre of the patch, then computes the intensity-weighted centroid of the patch. The direction of the line from this corner point to the centroid gives the orientation. To improve the rotation invariance, moments are computed at x and y positions which lie in a circular region of radius r, where $2r < S$ so that the circle lies within the patch, starting from the line from the corner to the centroid. Matching of ORB descriptors is also done by XOR. Both BRIEF and ORB are available within OpenCV.

7.8 SIFT and related techniques

David Lowe's *Scale-Invariant Feature Transform* (SIFT) [Lowe, 2004] detector actually preceded the development of the corner descriptors described above. SIFT was the first workable operator that was able to calculate and match consistent features that were reasonably *independent of scale and orientation* — in fact, it quickly became the *de facto* standard for feature detection and is still the yardstick against which we measure the performance of other feature detectors. SIFT is now part of OpenCV; a program to compute features on images and display the result is shown below:

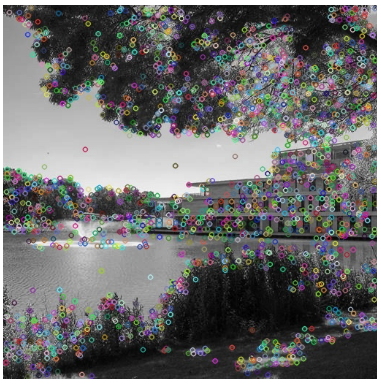


Figure 7.15: SIFT features marked on a natural image

⟨*Compute and display SIFT features*⟩ ≡

```
#/usr/bin/env python3
"Compute␣and␣display␣SIFT␣keypoints."
import sys, cv2

# Read the image and convert it to grey-scale.
im = cv2.imread (sys.argv[1])
gim = cv2.cvtColor (im, cv2.COLOR_BGR2GRAY)

# Create a SIFT detector and apply it to yield keypoints.
sift = cv2.SIFT_create ()
kp = sift.detect (gim, None)

# Mark the keypoints on the image and display the result.
img = cv2.drawKeypoints (gim, kp, im)
cv2.imshow ("SIFT␣keypoints␣of␣" + sys.argv[1], im)
cv2.waitKey (0)
```

The result of applying this program to our familiar image of the Essex campus is presented in Figure 7.15.

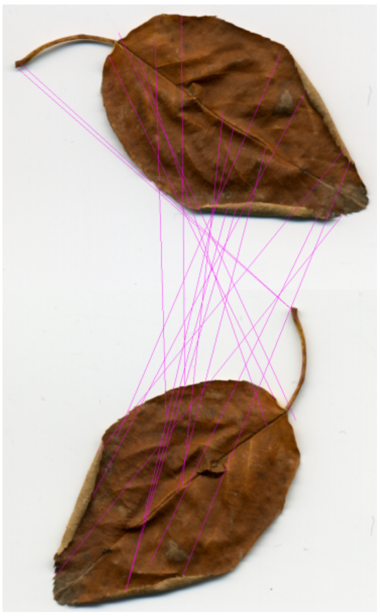


Figure 7.16: An example of feature matching using SIFT

SIFT finds characteristic features that are independent of scale and orientation and, for each of them, calculates a descriptor of (typically) 128 floating-point numbers. There may be several thousand such features in an image. Comparing objects in two images then comes down to finding groups of similar descriptors that have all undergone the same transformation. An example of matching SIFT features is shown in Figure 7.16.

How can one compare a pair of SIFT features, **A** and **B**? Several ways are in common use:

Euclidean distance: This is practically the same as the sum-squared difference we saw in earlier:

$$E = \sqrt{\sum_i (A_i - B_i)^2} \tag{7.10}$$

Manhattan distance: This is a quicker-to-calculate alternative to E:

$$M = \sum_i |A_i - B_i| \tag{7.11}$$

It is easy to show that $E \leq M$ (think of Pythagoras's theorem) and that is why E is a better measure than M.

Angle between vectors: If you have experience of the maths of 3D graphics or vector analysis, this approach will make sense. If you think of **A** and **B** as vectors, then their scalar product is given by

$$\mathbf{A} \cdot \mathbf{B} = \sum_i A_i B_i = |\mathbf{A}|\,|\mathbf{B}| \cos\theta$$

where $|\mathbf{A}|$ is the length of **A** *etc.* and θ is the angle between the vectors. Re-arranging, we obtain

$$\cos\theta = \frac{\sum_i A_i B_i}{|\mathbf{A}|\,|\mathbf{B}|} \tag{7.12}$$

The last of these alternatives is widely used in information retrieval, and the author's own experience is that it works best on the problems he has worked on.

Hence, to find matches between, say, objects in successive frames of a video, the first step is to calculate the SIFT features for each frame. Then, taking each feature of the first frame in turn, one compares it with *all* the features in the second frame and chooses the one with the lowest score from any of the criteria described above as being the best match. Clearly, the computation time scales as $O(N^2)$, so is slow to perform.

SIFT features can be computed quickly on systems equipped for general-purpose GPU calculations and good GPU-based implementations of the matching stage are starting to appear; but overall SIFT is quite slow to work with. In our research into reconstructing coral reefs, for example (Chapter 9), SIFT feature computation takes several hours to perform, even on machines having state-of-the-art GPUs, while feature-matching takes days. This restricts the size of the reef we are able to reconstruct.

There are a number of successor techniques to SIFT, such as Luc van Gool's SURF (*Speeded-Up Robust Features*) [Bay et al., 2008] which use slightly different approaches that allow the operator to run more quickly. In fact, in our research work, we recently compared about a dozen of these feature detectors [Bostanci et al., 2014] and found that Wolfgang Förstner's SPOF [Förstner et al., 2009] is the most effective overall.

The main problem with SIFT, SURF and most related techniques is that they tend to *avoid* identifying features in the vicinity of corners to ensure they are reproducible when there is motion between frames — but corners are often the features that are of most interest in some applications. For example, in our research into mobile robots and navigation aids for the visually impaired, we really need to know the boundaries of objects so that navigation around them is possible. So where are SIFT *etc*. used? They are extremely good for tasks such as stitching overlapping images into panoramas — if you have used *Photoshop* or related tools to do this, you have probably used SIFT — tracking features between images, and so on. They also form the mainstay of so-called "structure from motion" techniques, which allow 3D reconstructions to be calculated from (large numbers of) photographs, as discussed in Chapter 9.

7.9 Local binary patterns

We first considered textures in Chapter 5, where we looked at grey-level co-occurrence matrices and Laws' masks. We shall take another short look at it now, looking at an operator which uses similar ideas to those in BRIEF to described textured regions.

Local binary patterns (LBP) [Olaja et al., 1994] is a descriptor designed specifically to describe the texture of regions and is often used in conjunction with the HOG features we shall consider in Chapter 8. In its simplest form, the LBP descriptor is calculated as follows:

1. Divide the image, or the region of the image in which you're interested, into (usually) 16 × 16-pixel cells.
2. For each pixel in a cell, compare it with its eight neighbours, 'walking' through them either clockwise or anticlockwise. These do not have to be its nearest neighbours; Figure 7.17 shows neighbours at several distances from the centre pixel.
3. Where the centre pixel's value is greater than that of the neighbour, score it as `0`; otherwise score it as `1`. Walking round all the neighbours then gives an 8-bit number.
4. Over the whole cell, compute the histogram of the frequency of each 8-bit number.
5. Optionally normalise the histogram.
6. Concatenate all the (normalised) histograms of the cells to give a descriptor for the entire region of interest.

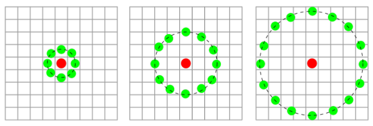


Figure 7.17: Local binary pattern (LBP) neighbours (image from the Wikipedia)

Just as with GLCMs and Laws' masks, the LBP descriptor does not in itself define the type of texture; instead, it provides evidence for subsequent processing. These days, that is almost always done using machine learning such as a support vector machine (Chapter 10) to classify image regions.

8
Looking at Humans

We review some techniques relating to the processing and analysis of human, and especially human faces, using computer vision. We start by examining face detection, showing why colour-based approaches are poor and explaining the popular technique due to Viola and Jones in some detail. Some applications that employ face- and face-feature location and outlined. We then consider face recognition, with emphasis on the 'eigenfaces' technique.

8.1 Introduction

Faces are interesting. Studies suggest that there is special 'circuitry' in the brain to locate faces in images, store them, and recognise them — that is one of the reasons we are able to 'see' faces in natural patterns such as clouds, tea leaves, and so on. In this chapter, we shall consider how faces are located in images, and then take a look at one of the techniques that attempts to do face recognition. Largely for amusement, we shall also look briefly at assessing the beauty of faces and a cute approach to communicate people speaking.

There are several steps in processing images of faces, and getting to grips with their nomenclature is a good first step. Processing an image to identify where faces are to be found is known, not unreasonably, as *face location* or *face detection*. Having found a face, a common requirement is to scale it so that the eyes and mouth appear at known locations within the image; this is known as *face normalisation*. Finally, determining who a particular face belongs to is *face recognition*.

8.2 Locating Faces by Colour

One way to identify where a face lies in an image is by its colour. We know that the colour of skin is determined by a compound called *melanin*: the more of this that is present, the better the skin protects against ultra-violet radiation and the darker it appears. So, the argument goes, finding skin-coloured regions in images and determining whether they are roughly oval is one way of identifying faces.

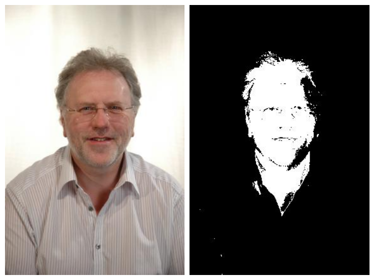


Figure 8.1: Successful detection of human skin by colour (hue 300°–30°, saturation 30%–70%)

As the amount of melanin differs from person to person, we cannot look for a little region in the RGB colour space. However, if we convert the image to the HSV colour space (Chapter 3), the fact that melanin is a specific colour should allow skin to be recognised by looking in a small range of hues and saturations. Implementing this is straightforward and Figure 8.1 shows that it can indeed find regions of skin.

However, there are major problems with this approach. Firstly, it clearly cannot work on monochrome ("black and white") images, while human vision obviously does. Secondly, the colour that skin appears is affected by the colour of its illumination, so the hue–saturation region changes as one moves from (say) daylight to fluorescent light. Thirdly and most crucially, there are compounds other than melanin that have similar colouring; in particular, some types of wood have rather similar hues to skin — Figure 8.2 gives an example failure. We are forced to conclude that using colour as a way of locating faces is far too naïve to be used in practice.

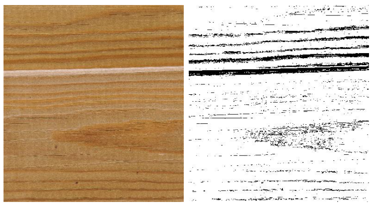


Figure 8.2: Skin by colour is easily confused; wood, for example, has a similar hue and saturation to skin (hue 300°–30°, saturation 30%–70%)

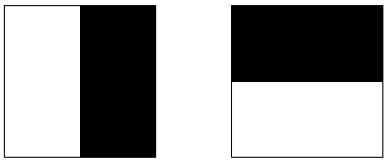
8.3 Viola-Jones: Haar Features and Adaptive Boosting

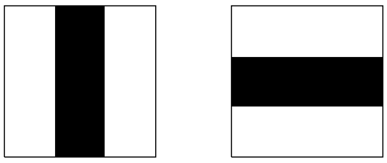
The face location technique due to Viola and Jones [Viola and Jones, 2004] is the *de facto* standard algorithm, present in practically every mobile 'phone and digital camera; the algorithm is patented, so anyone who uses it in commercial product has to pay a licence fee to Mitsubishi. As we shall see, the approach that Viola and Jones developed is actually general, applicable to other types of feature detection. It is implemented in OpenCV in a way that makes performing face location straightforward, yet retains the ability to recognise other types of object.

You should be aware that the Viola-Jones technique was developed specifically for camera manufacturers so that cameras could auto-focus on faces, which are the parts of photographs that people normally want to be sharp. Hence, the algorithm works only for full-face images; it does not work on profile views and performs fairly poorly on three-quarter views. As we shall see, the algorithm involves processing rectangular regions, and these will differ in landscape and portrait orientations — this is why cameras that do face recognition always have orientation sensors.

The most important thing to realise about this technique is that, like almost all systems that employ machine learning, it involves two phases. Firstly, a series of classifiers are trained how to detect faces using a database of positive (images containing faces at known locations) and negative (images without faces) examples, a slow process. When trained, these classifiers can be applied to any image to locate any faces that are in it, and this executes quickly. The slow learning is not a problem as, in principle, it need only ever be done once; but it is important that the trained classifiers execute quickly.

To train the classifiers, features need to be extracted from images. The approach that Viola and Jones took was to reduce any input image region down to a fixed size (they used 24×24 pixels). Then, all possible Haar features are found from this image — even for such small images, there are 162,336 possible features, and this is one of the reasons that learning is slow. A Haar feature is actually quite simple: it is the difference of the sum of pixels in rectangularly-shaped image regions. Figure 8.3 shows some example Haar features that correspond to edges and lines.


(a) Vertical and horizontal edges


(b) Vertical and horizontal lines

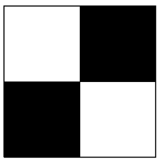

(c) Diagonal line

Figure 8.3: Some examples of Haar features. In each case, the feature is a single value calculated by subtracting the sum of pixels lying below the white rectangle from the sum of pixels lying below the black rectangle.

Most of the features calculated will be irrelevant but a few will help locate the face; for example, Figure 8.4 shows one that should help identify eyes. The learning algorithm finds which combination of features minimizes the number of mis-classifications of face and non-face image regions. Each feature individually does not perform particularly well (it is a *weak classifier* in machine learning jargon) but all the weak classifiers in combination may yield a strong classifier. This procedure is known as *adaptive boosting* or *Adaboost*; it is actually a bit more complicated than this in detail but this is its principle. Viola and Jones found that about 200 of these features classified faces correctly with about 95% accuracy, while about 6,000 classified all of them.

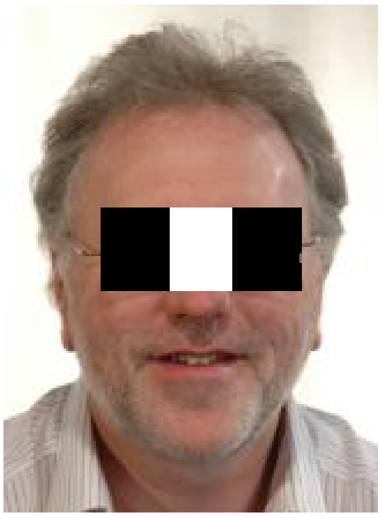


Figure 8.4: Haar feature that helps detect eyes

Although the time taken to calculate 6,000 features is substantially less than the time required for 160,000, the process can still be made more efficient. Most of a typical image will not contain a face, so a quick test to ascertain whether a region definitely does not contain a face means that the other Haar features do not have to be calculated; for example, completely uniform regions are not faces. Only when an image region might potentially contain a face is it worth calculating some of the other features. Hence, Viola and Jones introduced the idea of a *cascade of classifiers*, with images potentially being rejected at each step along the cascade. Their final face detection system contained 38 stages, the first five of which involve only 1, 10, 25, 25 and 50 features; the total number of features required by all 38 stages is a little over 6,000.

Speeding up Haar feature calculation using integral images

The computation of the rectangular regions used to form the Haar features can be speeded up significantly. This is done using a data structure known in the computer vision domain as an *integral image*; however the data structure was originally conceived for performing texture-mapping in computer graphics, where it is known as a *summed-area table*. Most graphics cards have the ability to calculate integral images.

An example of a image and its integral image is shown in Figure 8.5. Each pixel in the integral image is the sum of the values of it and all pixels lying above and to the left of it. The first row and column are special cases: on the first row, the value is the sum of that pixel and all those to its left, while on the first column it is the sum of that pixel and all those above it. The following code shows that it is fairly easy to calculate an integral image.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

(a) Original image

17	41	42	50	65
40	69	77	99	130
44	79	100	142	195
54	101	141	204	260
65	130	195	260	325

(b) Corresponding integral image

Figure 8.5: An image and its integral image

⟨*Integral image calculation*⟩ ≡

```
def calc_sat_slow (im):
    "Form␣an␣integral␣image␣from␣an␣image␣by␣the␣obvious␣approach."
    ny, nx, nc = im.shape
    sat = numpy.zeros ((ny,nx,1))

    # The first row and column are special cases.
    sat[0,0] = im[0,0]
    for x in range (1, nx):
        sat[0,x] = sat[0,x-1] + im[0,x]
    for y in range (1, ny):
        sat[y,0] = sat[y-1,0] + im[y,0]

    # For the remainder of the image, we add up all the pixels
    # within a region.
    for ylim in range (1, ny):
        for xlim in range (1, nx):
            regsum = 0.0
            for y in range (0, ylim+1):
                for x in range (0, xlim+1):
                    regsum += im[y,x]
            sat[ylim,xlim] = regsum
    return sat
```

You will see that there are four nested loops here, which means the computation is pretty slow.

However, the whole point of using an integral image is that it is fast to compute! The speed-up comes from the algebraic identity

$$ab = (x+a)(y+b) - (x+a)y - x(y+b) + xy \tag{8.1}$$

As ab is the area of a rectangle with sides a and b, this can be interpreted geometrically as shown in Figure 8.6. The area of the unshaded rectangle ab can be found by taking the area of the outer enclosing rectangle and subtracting from it the areas of the red and blue hatched regions. This means that the cross-hatched region has been subtracted twice, so it must then be added back in. Converting this into the manipulation of image pixels, if we call this integral image I, then the sum of all pixels in the rectangle with corners (x, y) and $(x+a, y+b)$ is

$$I(x+a, y+b) - I(x+a, y) - I(x, y+b) + I(x, y) \tag{8.2}$$

which involves precisely four additions or subtractions. This means that the sum of any rectangular region of an image, *irrespective of its size*, can be calculated in constant time. Constant-time computation is a really desirable property for any component of a real-time system and has led to integral images appearing in a number of other vision operators, *e.g.* SURF.

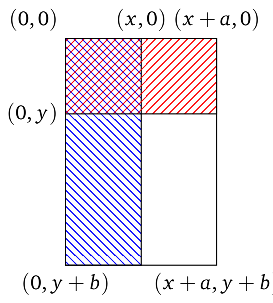


Figure 8.6: Principle of calculating an integral image: the unshaded rectangle is found by subtracting the areas of the red and blue hatched rectangles from the outer rectangle, then adding in the cross-hatched rectangle

Armed with this identity, coding up the calculation of an integral image becomes straightforward and avoids having so many nested loops:

⟨*Integral image calculation*⟩ +≡

```
def calc_sat (im):
    "Form␣an␣integral␣image␣from␣an␣image."
    ny, nx, nc = im.shape
    sat = numpy.zeros ((ny,nx,1))

    # The first row and column are special cases.
    sat[0,0] = im[0,0]
    for x in range (1,nx):
        sat[0,x] = im[0,x] + sat[0,x-1]
    for y in range (1, ny):
        sat[y,0] = im[y,0] + sat[y-1,0]

    # For the remainder of the image, we use the algebraic identity to
    # compute integral image values at pixels.
    for y in range (1, ny):
        for x in range (1, nx):
            sat[y,x] = im[y,x] + sat[y,x-1] + sat[y-1,x] - sat[y-1,x-1]
    return sat
```

You can confirm yourself that this yields identical results to `calc_sat_slow`.

Having formed the integral image, calculating the area of a rectangular region of the original image can be encapsulated in a short routine that also makes use of (8.2). This is regrettably not a single line of code because of the way that Python array subscripts work.

⟨*Integral image calculation*⟩ +≡

```
def get_sat (sat, ylo, yhi, xlo, xhi):
    """Return the area of a rectangular region of an image from its
    integral image."""
    # We have to do a little fiddling around with indices because the way
    # Python indices work does not quite match how an integral image is
    # most naturally indexed...so it is not quite constant in time.
Sigh.
    ylo -= 1
    xlo -= 1
    if ylo < 0:
        if xlo < 0:
           res = sat[yhi,xhi]
        else:
           res = sat[yhi,xhi] - sat[yhi,xlo]
    elif xlo < 0:
        res = sat[yhi,xhi] - sat[ylo,xhi]
    else:
        res = sat[yhi,xhi,0] + sat[ylo,xlo,0] - sat[yhi,xlo,0] - sat[ylo,xhi,0]
    return res
```

The following short program shows how the routines are used:

⟨*Integral image calculation*⟩ +≡

```
# Create and fill an image.
im = eve.image ((10,10,1))
eve.set (im, 1)
# Work out the correct sum of a rectangular region of it.
correct = im[4:6,1:3].sum()
# Form the integral image.
sat = calc_sat (im)
# Print out the correct sum and the value obtained from
# the integral image.
print correct, get_sat (sat, 4, 5, 1, 2)
```

Using Viola-Jones in OpenCV

Viola-Jones feature detection is built into OpenCV and is fairly straightforward to use. You have to give it the Haar cascade that it is to use when identifying whatever type of object that is to be detected; this is done by means of an XML file. Files are available for use with OpenCV for identifying full faces; the code below uses one of them. There are also cascades for detecting (left) eyes, smiles. . . and Russian number plates. There are loads of cascades produced by other people on the Web.

The following code is a self-contained face detection routine, though it could be made more efficient. An example of a face detected by this routine is shown in Figure 8.7.

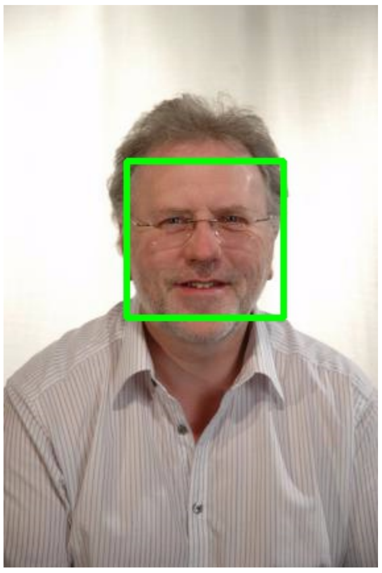
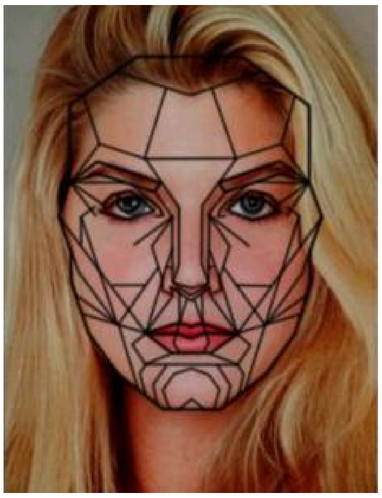


Figure 8.7: Viola-Jones face location in action

⟨*Viola-Jones face detection*⟩ ≡

```
def detect (im):
    "Detect␣a␣face␣using␣Haar␣cascades,␣as␣in␣Viola-Jones."

    cascade = c2.CascadeClassifier ("haarcascade_frontalface_alt.x
    faces = cascade.detectMultiScale (im, scaleFactor=1.1,
        minNeighbors=5)

    # Locate the faces and draw the surrounding rectangles on the
    # If we are to locate eyes too, do the same with them.
    for (x, y, w, h) in faces:
        print "␣␣", x ,y, x+w, y+h
        reg = im[y:y+h,x:x+w]
        cv2.rectangle (im, (x,y), (x+w,y+h), (255,0,0), 2)

    # Return the faces found and the marked-up image.
    return im, faces
```

(a) Original image


(b) Superimposed mask

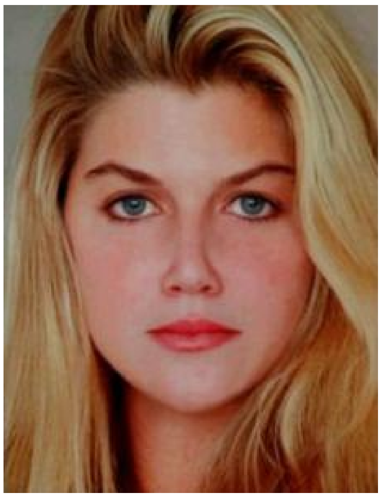

(c) "More beautiful" version according to the golden ratio

Figure 8.8: Face dimensions that accord to the golden ratio are supposed to be more attractive (from `http://www.goldennumber.net/beauty/`)

8.4 Processing Face Images

Measuring 'attractiveness'

'Beauty,' the old adage goes, 'is in the eye of the beholder' — meaning that a face found attractive by one person may not be by others. Nevertheless, psychologists and others have found some evidence that certain face shapes and arrangements of features tend to be considered more attractive than others, and this is supposed to be the case irrespective of gender and culture. There are two characteristics of attractive faces that are easily amenable to computer analysis and we shall consider them briefly.

Firstly, the more symmetric a face appears, the more attractive it is supposed to be. The difficulty here is identifying the axis of symmetry of a face, and one approach is to identify the locations of the eyes, nose and mouth and place the axis of symmetry mid-way between them. The degree of symmetry can be estimated by reflecting one half of the face and correlating it with the other half — though illumination needs to be fairly frontal for this to work.

Secondly, humans apparently find that things arranged according to the so-called *golden ratio*, ϕ, are pleasing. For $a > b > 0$, this is

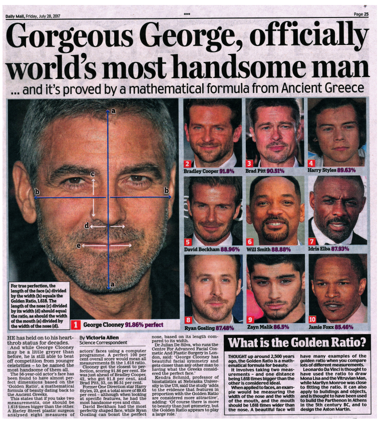
$$\phi \equiv \frac{a+b}{a} = \frac{a}{b} \tag{8.3}$$

In other words, two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities. A bit of algebra will let you determine that

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\cdots \tag{8.4}$$

(It is thought to be an irrational number.) Faces whose overall shape are in the ratio $\phi : 1$ are supposed to be the most attractive — see Figure 8.8 and Figure 8.9 for examples. The same idea is supposed to apply for the positions of features within the face: for example, if the eyes are placed at the golden ratio position within the height of the face, it is supposed to be the most attractive position. There are various software offerings that perform this ‘assessment’ on the Internet; Figure 8.8 is taken from the website of one of them, while Figure 8.9 is taken from a newspaper article found by the author.

Clearly, these measures should be regarded only as an amusement. If your own face or the face of a loved one is neither particularly symmetrical nor has their features arranged according to the golden ratio, don’t despair: Sophia Loren is reputed to be one of the most beautiful women ever born and her face apparently scores poorly according to these criteria. In any case, remember that there is another old adage that ‘beauty is more than skin deep.’


Daily Mail, Friday, July 28, 2017 Page 25

Gorgeous George, officially world’s most handsome man

... and it’s proved by a mathematical formula from Ancient Greece

For true perfection, the length of the face (a) divided by the width (b) equals the Golden Ratio, 1.618. The length of the nose (c) divided by its width (d) should equal the ratio, as should the width of the mouth (e) divided by the width of the nose (d).

HE has held on to his heart-throb status for decades.

And while George Clooney may be a little greyer than before, he is still able to beat off competition from younger celebrities – to be named the most handsome of them all.

The 56-year-old actor’s face has been found to have almost perfect dimensions based on the ‘Golden Ratio’, a mathematical formula of beauty dating back to the Ancient Greeks.

This states that if you take two measurements, one should be 1.618 times bigger than the other. A Harley Street plastic surgeon analysed eight measures of

By **Victoria Allen**
Science Correspondent

actors’ faces using a computer programme. A perfect 100 per cent overall score would mean all measurements fit the 1.618 ratio.

Clooney got the closest to perfection, scoring 91.86 per cent. He was just ahead of Bradley Cooper, 42, who got 91.8 per cent, and Brad Pitt, 53, on 90.51 per cent.

Former One Direction star Harry Styles, 23, got a total score of 89.63 per cent – although when looking at specific features, he had the most handsome eyes and chin.

David Beckham had the most perfectly shaped face, while Ryan Gosling can boast the perfect nose, based on its length compared to its width.

Dr Julian De Silva, who runs the Centre For Advanced Facial Cosmetic And Plastic Surgery in London, said: ‘George Clooney has beautiful facial symmetry and gets closer than any other man to having what the Greeks considered the perfect face.’

Kendra Schmid, professor of biostatistics at Nebraska University in the US, said the study ‘adds to the evidence that features in proportion with the Golden Ratio are considered more attractive’, adding: ‘Of course there is more to beauty than just geometry, but the Golden Ratio appears to play a large role.’

What is the Golden Ratio?

THOUGHT up around 2,500 years ago, the Golden Ratio is a mathematical formula for beauty.

It involves taking two measurements – and one distance being 1.618 times bigger than the other is considered ideal.

When applied to faces, an example would be measuring the width of the nose and the width of the mouth, and the mouth should be 1.618 times wider than the nose. A beautiful face will have many examples of the golden ratio when you compare lots of different measurements.

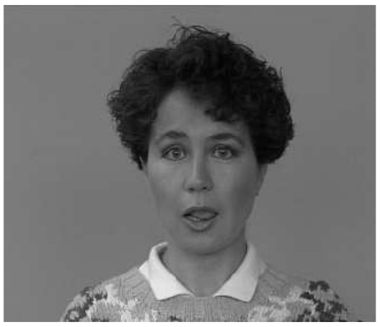
Leonardo Da Vinci is thought to have used the ratio to draw Mona Lisa and the Vitruvian Man, while Marilyn Monroe was close to fitting the ratio. It can also apply to buildings and objects, and is thought to have been used to build the Parthenon in Athens in the 5th century BC, and to design the Aston Martin.

Figure 8.9: Newspaper article from 2017 on beauty and the golden ratio

Model-based video coding

We have seen how it is possible to locate faces and facial features in images. With this capability, we can follow the motion of facial features from frame to frame of a video sequence and infer what the motion of a speaker’s head is: for example, if both eyes are moving horizontally, the head must be rotating and, as the diameter of the head can be estimated from the imagery, we can work out what the angle of rotation must be.

With this information, Münevver Köküer and the author were able to do video-telephony at *extremely* low data rates without having to attach markers to the face, as is usual in the movie and games industries. Rather than sending the actual video, we sent only the rotation angles *etc*. that the head is undergoing and then animated a 3D head model at the receiver — in our work, we used the Candide model discussed in Chapter 9. The appearance of the 3D model can be improved by texture-mapping the subject’s face onto it. Although this may seem a little bizarre, we were able to produce a system in the early 1990s that was accurate enough for deaf people to lip-read the animated model!


(a) Original image

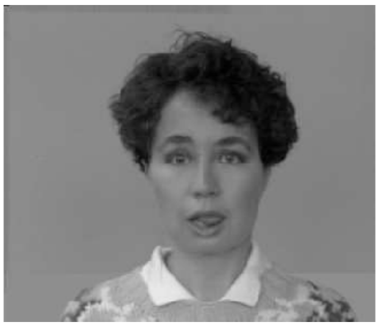

(b) Coded image

Figure 8.10: Model-based image coding. By analysing the motion of a human head in 2D over the frames of a video, it is possible to infer the motion in 3D from (a) and use that to animate a 3D model, as shown in (b).

8.5 Recognising Faces: Eigenfaces

You will recall that the standard deviation (or its square, the variance) gives us a single number that describes the variation present in an image. It would be useful if we could find a similarly straightforward formula that encapsulates the *similarity between two images* in a single number. An inspection of the equation defining the variance suggests something. Given two images, $P(x,y)$ and $Q(x,y)$, let us calculate

$$\mathrm{Cov}(P,Q) = \frac{1}{MN}\sum_{x,y}\left(P(x,y) - \langle P\rangle\right)\left(Q(x,y) - \langle Q\rangle\right). \qquad (8.5)$$

where $\langle\cdot\rangle$ represents the mean value. If $P(x,y) > \langle P\rangle$ and $Q(x,y) > \langle Q\rangle$, we will get a positive contribution to $\mathrm{Cov}(P,Q)$; likewise, we will get a positive contribution to $\mathrm{Cov}(P,Q)$ when $P(x,y) < \langle P\rangle$ and $Q(x,y) < \langle Q\rangle$. Conversely, when $P(x,y) - \langle P\rangle$ and $Q(x,y) - \langle Q\rangle$ have opposite signs, we will get a negative contribution to $\mathrm{Cov}(P,Q)$. So $\mathrm{Cov}(P,Q)$ is actually a fairly good measure of how well P and Q vary ‘in step’ with each other. Because of this and how similar it is to the variance, $\mathrm{Cov}(P,Q)$ is known as the *covariance* of P and Q. It is easy to show that if P and Q are uncorrelated or independent, then $\text{Cov}(P,Q) \approx 0$.

Covariance is useful if we have two images but what if we have more than two? The normal approach is to form a matrix of covariances; if we have N images $P_1, P_2, \ldots P_N$, the covariance matrix $\mathbf{C}$ is

$$\begin{pmatrix} \text{Cov}(P_1,P_1) & \text{Cov}(P_1,P_2) & \cdots & \text{Cov}(P_1,P_N) \\ \text{Cov}(P_2,P_1) & \text{Cov}(P_2,P_2) & \cdots & \text{Cov}(P_2,P_N) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(P_N,P_1) & \text{Cov}(P_N,P_2) & \cdots & \text{Cov}(P_N,P_N) \end{pmatrix} \tag{8.6}$$

In other words, each element represents the covariance between a pair of images. Diagonal elements, $\text{Cov}(P_i,P_i)$, are of course simply variances. A few moments' thought will tell you that all elements of the covariance matrix are non-negative and that the matrix is symmetric, as $\text{Cov}(P_i,P_j) = \text{Cov}(P_j,P_i)$.

For any set of images that are related in some way, for example different images of the same scene or a set of images of people's faces, off-diagonal elements will be non-zero if there is some similarity between images, which is usually the case. If we can transform the covariance matrix to remove this correlation (*i.e.,* make the off-diagonal elements zero), we can 'concentrate' the variation in a set of images into just a few images... and this leads us to the concept of *principal component analysis* (PCA), a powerful analysis technique for sets of related images. The technique is also known as the Hotelling transform and the Karhünen-Loève transform; the latter name is often used in, for example remote sensing.

The mathematics of PCA analysis is fairly abstract but the physical interpretation is easy to understand (Figure 8.11). If we have a cloud of points, each representing a single face, in an N-dimensional space, what PCA does is as follows. Firstly, it identifies the direction of maximum variation and this becomes the first *principal component*. The second principal component must be perpendicular to the first one; in Figure 8.11 this forces to lie in the direction shown but in three dimensions it could be anywhere on a circle perpendicular to the first principal component. The particular direction is again chosen to maximize the variance. The same approach is used for the remaining components.

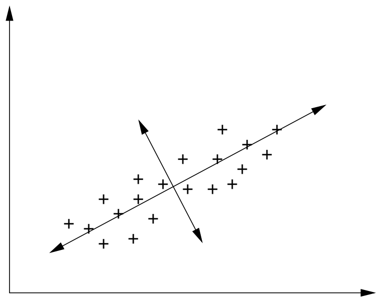


Figure 8.11: Illustration of principal component analysis

It is normally found that the first few principal components are much larger than the remainder. As these correspond to the variances (*i.e.,* importances of signals) of the various principal components, they show how the most common variation has been compressed into the first few of them, while the remainder contain smaller variations — the detail.

So why is PCA useful? Given an arbitrary image $R(x,y)$ which was part of the set of images used to determine the PCA, we can represent R as a weighted combination of the various principal component images:

$$R(x,y) = r_1 Y_1(x,y) + r_2 Y_2(x,y) + \cdots + r_n Y_n(x,y) \tag{8.7}$$

In other words, $r_1, r_2, \ldots r_N$ is a unique *feature vector* for the image $R(x,y)$ in terms of the principal components. For an image $R'(x,y)$ which was not part of the set of images that underwent PCA decomposition, the above relationship is usually approximately true. . . and this means it is possible to represent any image in terms of its feature vector. This is a significant finding, as it suggests that different images of the same face will end up in a small region of feature space — so we can apply any of the machine learning techniques of Chapter 10 to identify groups in PCA space corresponding to (say) a person's face.

A *caveat* is in order here, however. If an image is found which is significantly different to those that formed the set that underwent decomposition, it will not be represented well as a linear combination of the principal components. Put another way, PCA is good at interpolating between the various principal components but less good at extrapolating. Hence, it is important that the set of images used to obtain the principal components exhibits all of the possible modes of variation that are able to occur in practice.

There are several things that need to be done for this approach to work well. The first is that one needs to pre-process all the images so that images to be transformed are the same size with the eyes and mouth appearing at the same pixel locations — this is the *face normalization* process that was alluded to in the introduction to this chapter. If this is not done, the most significant principal component images encode this misalignment rather than any variation that is useful for face recognition (*e.g.,* nose shape).

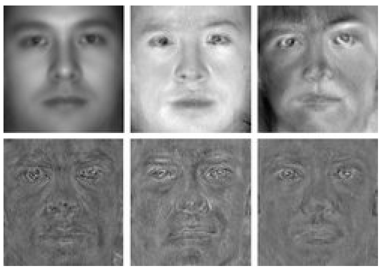


Figure 8.12: Some eigenfaces. Top left is the mean face image, and the other two images on the top row are the first two principal components ("eigenfaces"). The bottom row shows the three smallest principal components: these supposedly contain the least amount of information. [Scholarpedia]

To illustrate this, consider Figure 8.12 (taken from the Scholarpedia article on eigenfaces), which is a PCA decomposition of a well-known face database. The leftmost in the first row is the mean face, while the other two images on the top row are the top two eigenfaces; the second row shows eigenfaces with the three smallest eigenvalues. It is apparent that although the eyes have been aligned well (they do not appear blurred in the mean face), there is still significant variation around the eye region in the least significant principal components. The first principal component appears to encode the variation between asian and european face shapes; however, if this is the case, it is purely a consequence of the faces that form the database.

You might be pleased to learn that the eigenfaces approach has been implemented in OpenCV, and a web search will quickly turn up the documentation and example programs. There are also implementations of related techniques, for example the use of linear discriminant analysis to yield so-called "Fisherfaces" rather than eigenfaces — Fisherfaces is almost always better at recognition than eigenfaces. There are theoretical and practical advantages to some of these alternative techniques but exploring them would regrettably take us beyond the scope of this lecture course.

Expression recognition and affective computing

Having obtained some idea of how eigenfaces works, it is interesting to consider some other applications of the approach. There are many, as PCA is a general technique which is used in many subject domains; in vision, one of the more interesting is expression recognition.

We all know when a person is smiling — in fact, human examination of a smile is usually good enough to ascertain whether or not it is genuine, a subtlety well beyond computer vision. If software were able to identify when a computer user is happy by recognising their smile, then in principle one could make software adapt to this, and similarly with other emotions. This idea of having software adapt to the emotional state of its user is known as *affective computing* [Picard, 2000]. Of course, there are other ways that emotion can be measured: skin conductivity (as in a lie detector), alpha and other signals in brain–computer interfaces, jauntiness of walking as measured by accelerometers, and so on.

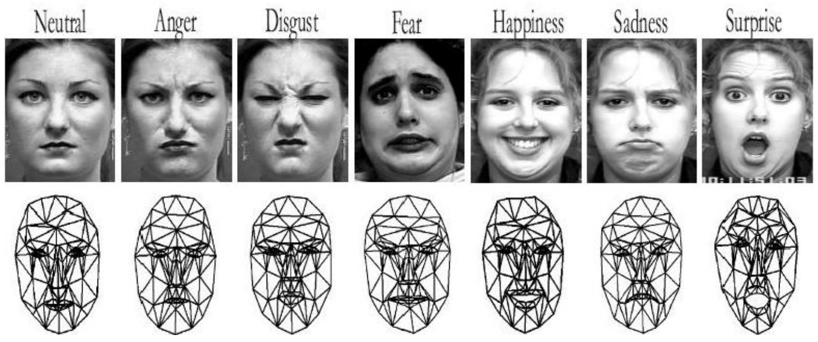


Figure 8.13: The six 'universal' expressions (from http://www.eecs.qmul.ac.uk/~ioannisp/ralis.htm)

Most expression work focusses on the so-called 'universal' emotions, illustrated in Figure 8.13 — though there is not universal agreement on them, several researchers adding 'contempt' to those shown in the figure.

Expression recognition works in essentially the same way as face recognition: a training set of images of each expression is captured and its principal components found. An image whose expression is to be determined is processed by the same PCA kernel and its best match identifies the expression class to which it belongs.

Some reasonably impressive results have been reported in the literature, though this author treats them with some caution because the datasets he has examined seem to have ludicrously exaggerated expressions. Perhaps no-one he has seen has been quite as surprised as the people pictured in the datasets, who presumably have just been told that expression recognition works... However, please do take a look at the online resources and decide for yourself.

As a footnote to this brief foray into affective computing, it is worth noting that the vast majority of published work is expended towards recognising emotional state; there seems to be little, if any, work towards adapting interactions based on knowledge of the perceived state — a good topic for a PhD, perhaps?

8.6 HOG, the histogram of oriented gradients

Having concentrated on the human face for the majority of this chapter, let us conclude it by looking at techniques that has been applied to tracking humans as they move. The first of these is the *histogram of oriented gradients* or HOG. This is a descriptor of local regions of an image which works by counting the occurrences of gradient orientation in each region [Dalal and Triggs, 2005]. The underlying idea is that appearance and shape within an image can be described by the distribution of intensity gradients or edge directions. A nice property of the HOG descriptor is that it is largely invariant to geometric and grey-scale changes, though not to changes in orientation.

To calculate HOG, the image is partitioned into a set of connected regions called cells (Figure 8.14) and, for each cell, a histogram of gradient directions is computed. This is done using the masks

$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

The outputs from the masks are used to calculate edge direction just as for the Canny edge detector, and the resulting angle (in the range 0–180°) is used to increment one of about 10 bins, usually by the corresponding gradient magnitude (Figure 8.15).

To accommodate changes in illumination and contrast, the gradient strengths are locally normalised, and this involves grouping several cells into a 'block;' blocks typically overlap. The HOG descriptor is formed by concatenating the normalised cell histograms from all the blocks.

HOG was designed to detect human movement, and has proven to be particularly useful for detecting pedestrians and people running.

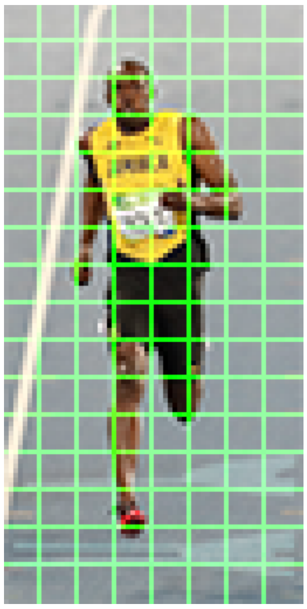


Figure 8.14: Histogram of oriented gradients (HOG) cells (image from Satya Mallick's website)

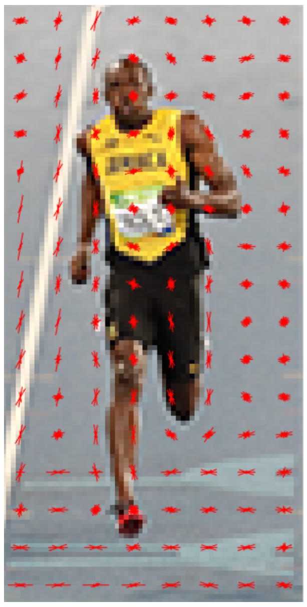


Figure 8.15: Histogram of oriented gradients (HOG) gradients (image from Satya Mallick's website)

8.7 OpenPose and friends

Our discussion of HOG suggests that detecting human pose and tracking the motion of limbs is a difficult thing to do — which indeed it is. If you had asked me in the early 2000s about doing a PhD on tracking human motion, I would have told you that it needs stereo cameras (Chapter 9) or special hardware such as the depth camera on a Microsoft Kinect. If you had said you wanted to do it from a single camera, I would have laughed and said something like "not in my lifetime." How wrong I was!

The game changed with CMU's OpenPose in 2017 [Cao et al., 2019]. This provides marker-less, real-time (given enough GPUs) tracking of human pose from a potentially moving camera; see Figure 8.16. Not only can it identify the poses of many individuals, it is able to do so at a range of sizes, something that also came as a surprise. Almost inevitably, the pose recognition in OpenPose is based around a deep neural network — though, to be honest, the real innovation was probably not in the machine learning part itself but in the way the research team captured the huge amount of data needed for training. Figure 8.16 is a still from a video which you are strongly encouraged to watch.

As its name suggests, OpenPose is open-source and freely available from GitHub. It is built on top of Caffe (there is a port to TensorFlow) and so needs a machine equipped with a graphics card having a reasonable number of GPUs. Many gamers will have beefy enough graphics cards to be able to run OpenPose; the author can certainly use it on a fast Dell laptop in something close to real time.

OpenPose returns a class instance for each detected skeleton which gives its estimates of the locations of the joints within a frame. The underlying library is written in C++ and interfaces to OpenCV but there are Python wrappers to make our lives easier, just as with OpenCV itself. Calculating joint angles from the joint locations is easy, and it is usually possible to infer 3D pose by tracking 2D motions over a few frames. Skeletons are numbered from the top-left corner of a frame. This can cause difficulties because people's motions might mean that they have inconsistent skeleton identifiers from frame to frame.

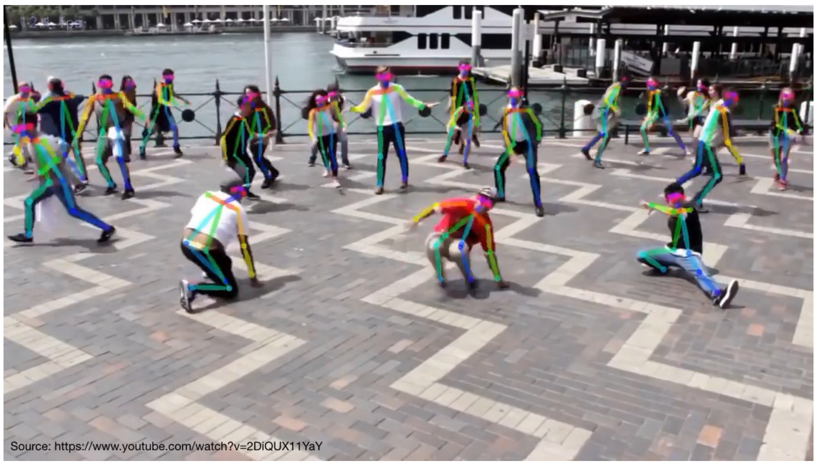


Figure 8.16: Tracking human motion with OpenPose

You should be aware that building software on top of the capabilities of TensorFlow is very much trying to hit a moving target. A project running in the 2022–23 academic year looked at porting OpenPose to the most recent TensorFlow and struggled to get it working outside Google's Colab environment, so be warned. Thankfully, alternatives exist and the one built on top of MediaPipe seems to be the most reliable one at the time of writing.

9
Vision in a 3D World

With images from two carefully-aligned cameras, it is possible to calculate the distance to objects with some easily-derived mathematics. With images from many cameras and some more sophisticated mathematics, one can reconstruct 3D objects in the scene or navigate through a 3D world. We first consider how images are formed in a single camera and, building on that, find that it is possible to calculate the distance of objects using a simple analytic expression under certain constraints. This is illustrated on computer-generated imagery of a 3D model. Following on from that, the principles of the techniques known as visual structure from motion *and* visual SLAM *are considered.*

9.1 Introduction

We know that images from multiple cameras can in principle be processed to determine the locations of objects in 3D space because that is what the human visual system does. This is actually a difficult problem computationally as it involves using 2D information (images) to determine things in 3D; indeed, without ancillary information, the problem cannot be solved. This 'additional information' is the locations and orientations of the cameras that capture the images, and this is often obtained by some kind of *calibration*: imaging objects of known size and shape and calculating what the cameras' geometry must be for them to appear where they do. Humans, of course, do not do this but instead learn to judge distances implicitly while they are babies and infants, which is why it seems so easy to us. However, about 10% of humans cannot see in stereo for some reason or other and they learn to judge distances using other cues.

The process of calculating positions from images captured by two cameras is usually known as *computational stereo*. The principles can also be applied to larger numbers of cameras ("multi-camera stereo") or to moving cameras, leading us techniques such as *visual structure from motion* and *visual SLAM*: the former attempts to construct 3D objects with some detail while the latter allows a robot to navigate through the world or for real-time augmented reality.

The underlying principle of computational stereo is fairly easy to demonstrate. Close your right eye and point at an object like a doorknob or the corner of a window. Then, without moving your hand, close your left eye and open your right one. The place you are pointing at will no longer be the feature you originally chose, and this is simply because your left and right eyes are in different places. This discrepancy in position is known as *parallax* in physics and as *disparity* in computer vision.

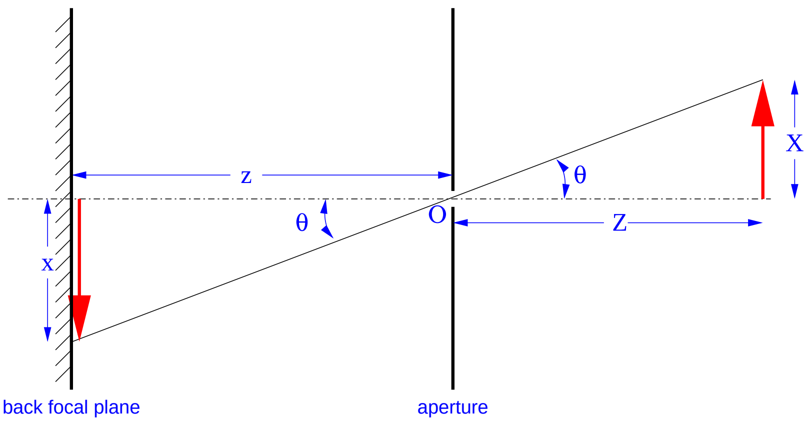


Figure 9.1: Imaging geometry of a pin-hole camera

9.2 The geometry of imaging

Before we can proceed, we need some idea of how a camera works in geometrical terms. The simplest camera is a 'pin-hole' one (Figure 9.1) — many children make them by taking a box, cutting out a large square hole at one end and sticking tracing paper over it, then putting a small hole in the other end. If one goes into a darkened room and points the camera through a gap in the curtains, it is just about possible to distinguish an (upside down) image on the tracing paper.

It is easy to see that the triangles OZX and Ozx in Figure 9.1 are similar, so

$$\tan\frac{x}{z} = \tan\frac{X}{Z} \Rightarrow \frac{x}{z} = \frac{X}{Z}.$$

The reason that people rarely use pin-hole cameras in practice is that the aperture is so small, and hence the amount of light admitted is minuscule. If the tracing paper screen is replaced by photographic film or a modern imaging sensor, exposure times can be several minutes rather than the fractions of a second achieved with conventional cameras. The author does this with a SLR camera, replacing its lens with a body dust cap with a pin-hole in it.

So how does a real camera differ from a pin-hole camera? In terms of imaging geometry, the principal difference is that the pin-hole is replaced by a lens, whose sole purpose is to focus light impinging on the larger aperture onto the *back focal plane*, the place where the tracing paper is stuck on a simple pin-hole camera (Figure 9.2). The ability to capture much more light via the lens means that exposure times are greatly reduced. The way that lenses form images means that when $Z = \infty$, $z = f$, the focal length of the lens. We shall generally assume in the following that $Z \approx \infty$, *i.e.,* that $z = f$. This is quite reasonable in practice.

For those readers who are photographers, a perfect pin-hole camera has an infinite depth of field (*i.e.,* objects at all distances from the camera are in focus); it is the finite diameter of the aperture that makes the depth of field finite in real cameras. If you think about this, it is entirely consistent with

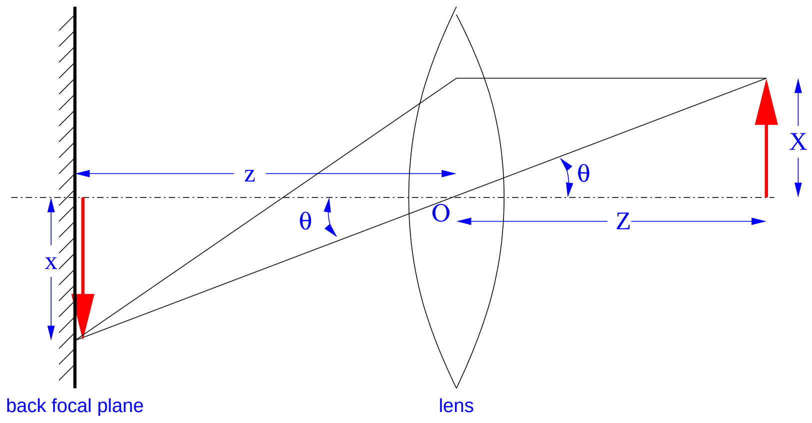
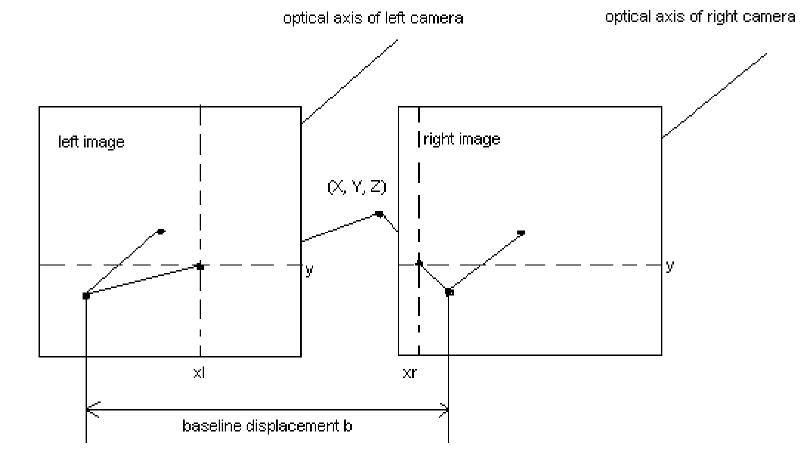
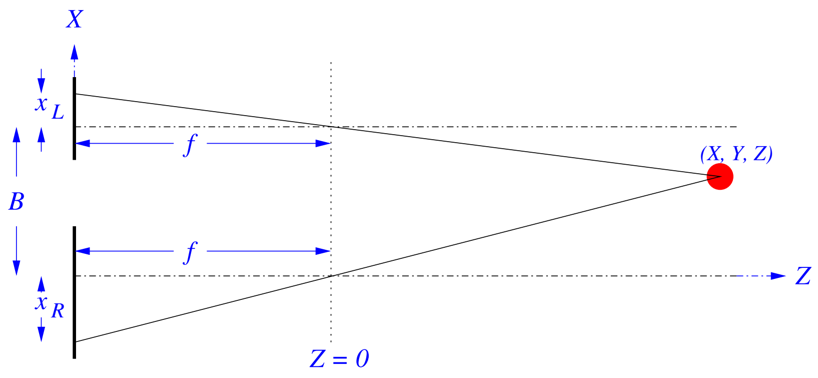


Figure 9.2: Imaging geometry of a real camera

the photographers' 'rule' that the depth of field increases as the diameter of the aperture decreases (or the f-number increases).



(a) Stereo configuration



(b) Plan view

Figure 9.3: The geometry of computational stereo

9.3 Computational stereo

The maths involved in the images from multiple cameras to calculate the 3D locations of markers is a little tricky but we can see the general way in which it is done by considering just a pair of cameras (Figure 9.3). To make the maths easier, we shall make some simplifying assumptions:

- two pinhole cameras are used, with identical focal lengths;
- the two cameras have parallel optical axes;
- the two cameras are aligned so that disparities always lie in the same row of the images.

Much of the 'trickiness' in the maths alluded to above is in circumventing these constraints.

Let us put the origin of our coordinate system at the front of the lens of the right camera. A point (X,Y,Z) on an object in the scene is projected onto image point (x_L, y) in the left image and onto point (x_R, y) in the right image. From similar triangles in the left camera in the plan view of Figure 9.3, we have

$$\frac{x_L}{f} = \frac{B-X}{Z} \tag{9.1}$$

and in the right image

$$-\frac{x_R}{f} = \frac{X}{Z} \tag{9.2}$$

Formally, we should proceed by re-arranging (9.1) and (9.2) to make X the subject, then equate them and perform some manipulation of the result. However, a more direct approach is simply to add them:

$$\frac{x_L - x_R}{f} = \frac{B-X+X}{Z} = -\frac{B}{Z} \tag{9.3}$$

where the minus sign is due to the different direction of the angle. This can be re-arranged to give the depth (distance from the camera), Z, in terms of the disparity measured between the two images, $x_L - x_R$:

$$Z = \frac{fB}{x_L - x_R}. \tag{9.4}$$

A little care is needed with these quantities. The image coordinates (x_L, y) and (x_R, y) in above equations have their origin at the centre of the images with their axes running in the same directions as the 3D coordinate system. However, we typically place the origin of an image at its upper-left corner with the x and y axes left-to-right and top-to-bottom respectively. This means that positions in images have to be transformed to conform with the 3D coordinate system before using them in calculations. The image coordinate transformation is

$$x' = \text{image } x \text{ location} - \text{half the image size in } x \tag{9.5}$$

$$y' = \text{half of the image size in } y - \text{image } y \text{ location} \tag{9.6}$$

If the image size is odd in x or y, a fractional value results. Furthermore, one needs to be able to convert the disparity from pixels to the same units as the focal length. That can in principle be done by consulting the manufacturer's data sheet; but in practice, it is better to find this quantity by calibration. The same calibration process can identify (some) aberrations in the lenses used too, making the overall result more accurate. There are both camera calibration and computational stereo routines in OpenCV, the latter more general than the approach described here.

9.4 Computational stereo in action

A good way to prove that this kind of computational stereo works is with simulated data for which the answer is known — and a very convenient way to simulate images of 3D objects is by computer graphics. Using POV-ray, a ray-tracing program that accepts a description of a scene in something resembling a programming language and generates realistic-looking, geometrically-accurate 2D images of it, it is relatively easy to define the imaging geometry of objects and cameras and hence generate stereo pairs for use in computational stereo.

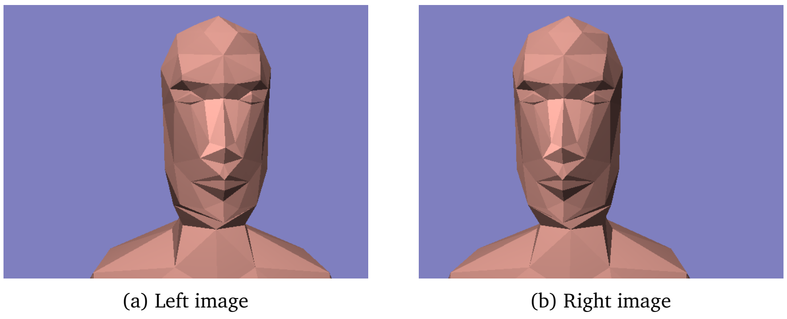

(a) Left image (b) Right image

Figure 9.4: Stereo pair of the Candide 3D head model

A stereo pair of a simple head model known as *Candide* is shown in Figure 9.4. With a little effort, it is possible for most people to look at this pair of images and see the object in 3D. The trick is to get your eyes to look straight ahead rather than trying to focus on each image separately; and the easiest way to do that is to hold the page close to your face and stare at the stereo pair, then slowly move the page away from your face. You should find that a third image which shows depth appears between the two real ones. (However, recall that about 10% of people are not able to see stereo at all; you might be one of the unlucky ones if you can't make this work, or you might just need to practise.)

The stereo pair was generated using the following pair of POV-ray camera definitions:

```
// Left camera                  // Right camera
camera {                        camera {
  location <75, 300, 800>         location <-75, 300, 800>
  look_at  <75, 300, 280>         look_at  <-75, 300, 280>
}                               }
```

i.e., with a baseline $B = 150\,\text{mm}$. (All measurements on the model are in millimetres, so the camera definitions *etc.* are too.) By calibration for 640 × 480-pixel images, it is found that these POV-ray cameras have $f = 477.35\,\text{mm}$. Hence, from (9.4) we can calculate the distance to a point on the head as

$$Z = \frac{150 \times 477.35}{x_L - x_R} = \frac{71\,602.5}{x_L - x_R}\text{mm.}$$

Using an image display program, we can find the locations of (say) the end of the nose or the corner of an eye in the left and right images, then use them to calculate Z. We can then compare the calculated Z with its true value in the Candide model.

If we choose vertex number 5, the tip of the nose, the calculations proceed as follows. Its x positions in the left and right images are 389 and 252 respectively. Transforming them according to (9.5) gives $x_L = 69$ and $x_R = -68$. This gives us

$$Z_5 = \frac{71\,602.5}{69 + 68} = 522.6\,\text{mm}$$

The true value of Z_5 is 520 mm — reasonably close.

Even though all this experiment is performed on a computer — the images are calculated from specific numbers according to well-understood equations, and practical problems such as mis-alignment of the optical axes of the camera do not arise — the results from the computational stereo calculation are not identical to their true values. This is not due to bugs in the software, and you might like to reflect on possible causes before reading the next paragraph.

So why are the results not accurate? There are two main reasons. First, the camera used by POV-ray must be calibrated in some way, just like a real camera, and this introduces inaccuracy. Secondly, the feature matches in left and right images are accurate to the nearest pixel, as illustrated in Figure 9.5: the actual position of the sharp point of the polygon can be measured only as being within the pixel at $(x, y) = (1, 3)$ even though we can see it is not right in the middle of it in this expanded view. It is important to realize that *taking measurements is never perfectly accurate*, even when everything is done on a computer.

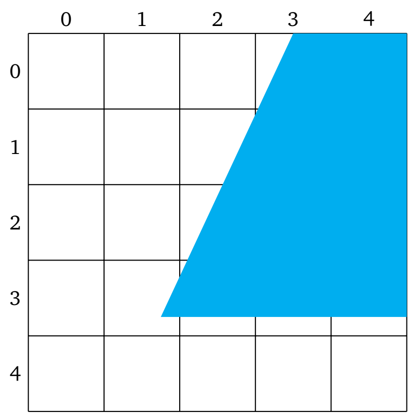


Figure 9.5: Locations are accurate only to the nearest pixel

9.5 Propagating uncertainty

You might be surprised to learn that *if we know how uncertain* some things are, we can carry that uncertainty forward through calculations and, on some occasions, even reduce it — that is why roboticists and control engineers use a Kalman filter to combine measurements. We should actually write that the position of a feature in the left or right image is something like $x_L = (69 \pm 1)$ pixels: in other words, we quote the actual measurement and its associated uncertainty. Why ± 1? It is because we cannot measure the position of a feature in an image, such as the tip of Candide's nose, to less than a pixel. The jargon word for this uncertainty is *experimental error* or just *error*, though you should not take this word in the sense of it being a mistake.

Let us go back to the calculation of Z_5 above. Rather than having everything known perfectly accurately, let us ascribe uncertainties to each of the quantities: $B = (150 \pm 0.1)$ mm, $f = (477.35 \pm 1.2)$ mm, $x_L = (69 \pm 1)$ pixels as above and $x_R = (-68 \pm 1)$ pixels. The uncertainties in B and f are fictional here but can be found by measurement in real experiments. We can work out the uncertainty in Z_5 by applying two rules:

- when adding or subtracting values, add the errors;
- when multiplying or dividing values, add the fractional errors.

For mathematically-inclined readers, these are consequences of Taylor's theorem, *i.e.* they involve the derivative of the equation for combining the quantities. The *fractional error* in $(Q+\delta Q)$ is $\frac{\delta Q}{Q}$, pretty much what you would expect from its name. Because this calculation is based around a simulation, we have ignored an important practical point, the conversion of the disparity in pixels to a real unit of length, say millimetres in this case. Let's say the sensor was 1 cm (10 mm) square; then each of the 640 pixels would occupy

$$\frac{10}{640} = 0.015625\,\text{mm}$$

and we would write $x_R = (1.078125 \pm 0.015625)$ mm and $x_L = (-1.06250 \pm 0.015625)$ mm.

To find the error in Z_5, we have to add up the fractional errors in B, f and the disparity

$$\frac{\delta Z_5}{Z_5} = \frac{0.1}{150} + \frac{1.2}{477.35} + \frac{0.003125}{2.140625}$$

which works out to give $\delta Z_5 \approx 1.7$ mm, so we would write its value as being (520.0 ± 2.5) mm.

As an aside, if you ever see a graph with little I-shapes drawn at each point, these are indicating the sizes of the uncertainties of the values drawn — as a physics student, handling uncertainties was pretty much the first thing I was taught in my degree.

9.6 3D coordinate geometry

In order to progress further in solving 3D problems using computer vision, a better grasp of the mathematics underlying the 3D world is needed. However, the mathematical knowledge is not especially difficult and is the same as that underlying much of 3D graphics — the author used to teach it to second-year undergraduates. Let us start by considering the three basic types of transformation that retain the basic geometry of an image: translation, scaling and rotation. Other types of 3D transformation are obtained by combinations of these primitive operations.

Translation

Consider the movement of the point $\mathbf{P} = (x, y)'$ to $\mathbf{P}' = (x', y')'$ in Figure 9.6(a). We have the two equations

$$x' = x + dx$$
$$y' = y + dy$$

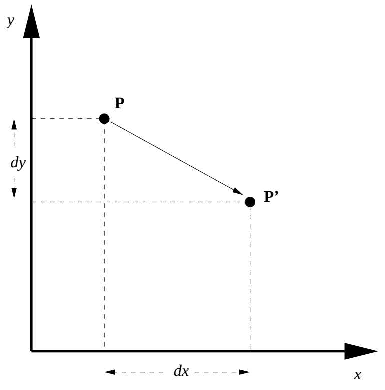
We can write this in matrix form as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix}$$

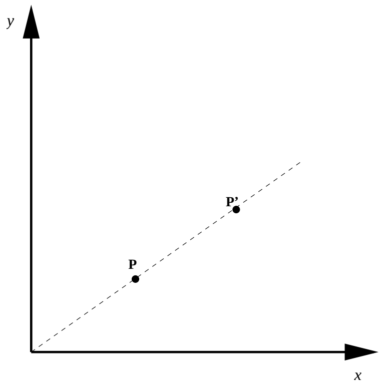
or as

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

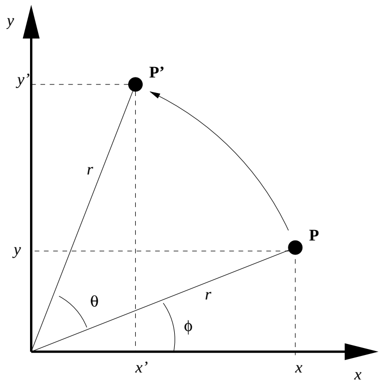
where $\mathbf{T}$ represents the translation.



(a) Translation of a point



(b) Scaling of a point



(c) Rotation of a point

Figure 9.6: The fundamental 2D transformations

Scaling

If we have $x' = xS_x$ and $y' = yS_y$ as in Figure 9.6(b), we have *scaled* the point **P** to **P**$'$. Note that the scale factor may be different in x and y: this would turn a circle into an ellipse, for example. We can write scaling as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

or as

$$\mathbf{P}' = \mathbf{SP}$$

where **S** is the matrix that represents the scaling.

A *shear*, which distorts a shape along one direction only, is a kind of scaling. Shears parallel to the x- and y-axes are given by

$$\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix} \text{and} \begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}$$

Rotation

From Figure 9.6(c), we have $x = r\cos\phi$ and $y = r\sin\phi$ and we can see that

$$\begin{aligned} x' &= r\cos(\theta+\phi) &= r\cos\theta\cos\phi - r\sin\theta\sin\phi \\ y' &= r\sin(\theta+\phi) &= r\cos\theta\sin\phi + r\sin\theta\cos\phi \end{aligned}$$

Substituting for x and y, we get

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

We can write these in matrix form as

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

or as

$$\mathbf{P}' = \mathbf{RP}$$

where **R** is the matrix that represents the rotation.

Transformations such as these have several important properties:

- points on a straight line remain on a straight line after transformation;
- parallel lines remain parallel;
- intersecting lines remain intersecting — we can show this by drawing a line across the diagonal of a square and then rotating the square.

Combining transformations using homogeneous coordinates

If we have, say, $\mathbf{P}' = \mathbf{SP}$ to perform a scaling and $\mathbf{P}'' = \mathbf{RP}'$ to perform a rotation, then we can also say

$$\mathbf{P}'' = \mathbf{RSP}.$$

In other words, we can introduce an additional scaling or rotation by *pre-multiplying* by the relevant transformation matrix. (The order in which we apply transformation is important, as matrix multiplication is not commutative ($\mathbf{M}_1\mathbf{M}_2 \neq \mathbf{M}_2\mathbf{M}_1$).) However, we cannot represent translation in the same way, as it is *additive* rather than multiplicative. We get around this by using so-called *homogeneous coordinates* in which the point (x, y) is represented by the *three* numbers $(x, y, 1)$. The 2D point is found by dividing all elements by the last one. In other words, $(x, y, 1)$ and (hx, hy, h) represent the *same* homogeneous coordinate. This allows us to apply translation in exactly the same way as rotation and scaling, simply by pre-multiplying:

$$\mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Knowing these transformation matrices, we can work out where a pixel will move to if the camera is subject to rotation, scaling and translation — in other words, we can work out what will happen to an image if we have a camera mounted on a moving robot, car, person or aircraft. However, the rotation matrix works only if we rotate around the top-left corner of the image, which is pretty unlikely to happen in practice, so we need to stack up some transformations that let us apply it.

Let us assume that the image rotates around the point $\mathbf{C} = (x_c, y_c)$ in the image. If we move that point to the origin and then apply the rotation matrix from above, we will rotate the image around the centre; and we then have to translate the resulting image to the right place. In other words, the combination of rotation, translation and scaling on a pixel $\mathbf{P} = (x, y)$ in the image will be

$$\mathbf{P}^{'} = \mathbf{STRT}_{-c}\mathbf{P} \tag{9.7}$$

where $\mathbf{T}_{-c}$ represents the shifting of $\mathbf{C}$ to the origin. Carrying out this series of multiplications, one ends up with the composite transformation

$$x' = s(x - x_c)\cos\theta - s(y - y_c)\sin\theta + t_x \tag{9.8}$$
$$y' = s(x - x_c)\sin\theta + s(y - y_c)\cos\theta + t_y \tag{9.9}$$

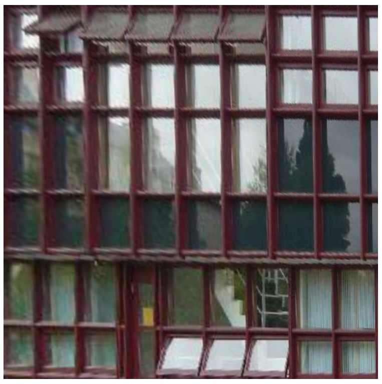
where s represents the scale factor, θ the angle of rotation and (t_x, t_y) the translations.

What this tells us is that, knowing the values of these transformations, one can calculate where each pixel in the image will end up. However, it is normally the *converse* of this that one is interested in: knowing where some pixels in one image end up in another, how does one work out the transformations between them?

9.7 Correcting perspective effects

We shall consider only a special case, the stage in the Sudoku capture program discussed by Chris Greening in Chapter 1, where he straightens the grid. In fact, the solution to this problem turns out to be useful in many other places: the author uses it for removing 'converging verticals' from his photographs and for preparing photographs of buildings for texture-mapping in computer graphics. This latter process is shown in Figure 9.7, which shows foreshortening with distance due to perspective; our aim is to make the four marked points become the corners of a straightened image, a process often called 'rectification.' The technique was first reported in [Criminisi et al., 1997] and expounded in full in [Hartley and Zisserman, 2003], the standard work on multi-view geometry in computer vision.

(a) A photograph showing the normal foreshortening due to distance, with four points marked on features that lie on the side of the building


(b) The region between the four points, extracted in a way that removes the perspective distortion

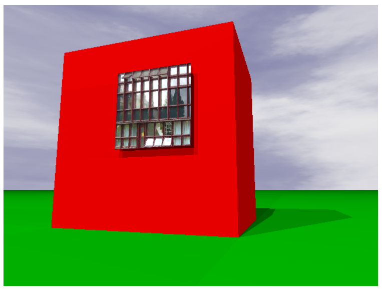

(c) An image of a 3D model with the undistorted image texture-mapped onto one face

Figure 9.7: An image showing perspective distortion and a region of it, 'rectified' to remove the perspective effect. These rectified images have a variety of uses, such as texture-mapping in computer graphics.

Let the coordinates of a marked point in the photograph be (x, y) and the location we want to map it onto in the straightened image (x', y'). We can write the transformation in homogeneous coordinates as

$$\begin{pmatrix} x' \\ y' \\ k \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where the last entry in $(x', y', k)^T$ is because the values of H_{ij} might not be normalized. Converting this to inhomogeneous coordinates involves calculating

$$\frac{x'}{k} = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}; \qquad \frac{y'}{k} = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{9.10}$$

which are essentially the equations given in the introductory lecture. Simply re-ordering these equations as

$$h_{11}x + h_{12}y + h_{13} = x'(h_{31}x + h_{32}y + h_{33}) \tag{9.11}$$

$$h_{21}x + h_{22}y + h_{23} = y'(h_{31}x + h_{32}y + h_{33}) \tag{9.12}$$

shows that they are linear in h_{ij}. This pair of equations is for one matched point, so the four matched points in Figure 9.7 give us 8 such equations and this is enough to solve for h_{ij} up to a multiplicative scaling factor, providing no three points chosen lie in a straight line — if they do, solving the equations ends up dividing by zero. Writing code to solve these equations leads to a fairly compact routine to calculate the transformation.

⟨*Undoing perspective effects*⟩ ≡

```
def find_perspective_transform (src, dst):
    """This routine calculates the coefficients of the
    perspective transformation which maps (xi,yi) to
    (ui,vi), (i=1,2,3,4)."
    n = len (pos)
    if n != 4: raise Exception ("Wrong number of matches.")
    a = numpy.zeros ((8,8), dtype=numpy.float32)
    b = numpy.zeros ((8), dtype=numpy.float32)
    m = numpy.zeros ((3,3), dtype=numpy.float32)

    for i in range (0, 4):
        a[i,0] = a[i+4,3] = src[i,0]
        a[i,1] = a[i+4,4] = src[i,1]
        a[i,2] = a[i+4,5] = 1
        a[i,3] = a[i,4] = a[i,5] = a[i+4,0] = a[i+4,1] = a[i+4,2] = 0
        a[i,6] = -src[i,0] * dst[i,0]
        a[i,7] = -src[i,1] * dst[i,0]
        a[i+4,6] = -src[i,0] * dst[i,1]
        a[i+4,7] = -src[i,1] * dst[i,1]
        b[i] = dst[i,0]
        b[i+4] = dst[i,1]
    # Solve the set of equations and copy the solution into a transformation
    # matrix, which we return.
    x = numpy.linalg.solve (a, b)
    m[0,0] = x[0]
    m[1,0] = x[3]
    m[2,0] = x[6]
    m[0,1] = x[1]
    m[1,1] = x[4]
    m[2,1] = x[7]
    m[0,2] = x[2]
    m[1,2] = x[5]
    m[2,2] = 1.0
    return m
```

This would be used in a program such as the following one.

⟨*Finding the perspective transform*⟩ ≡

```
#!/usr/bin/env python
'''Image rectification using OpenCV.'''
from __future__ import division
import sys, numpy, cv2

# Define the image containing the region to extract and the boundaries
# of the region.
pos = numpy.zeros ((4,2), dtype=numpy.float32)
fn =  "psychology.jpg"
pos[0,:] = [249, 337]  # ULx, ULy
pos[1,:] = [238, 483]  # LLx, LLy
pos[2,:] = [548, 501]  # LRx, LRy
pos[3,:] = [557, 190]  # URx, URy

# Define the size of the output image and the region of it we want to fill.
n = m = 512
opos = numpy.zeros ((4,2), dtype=numpy.float32)
opos[0,:] = [0, 0]
opos[1,:] = [0, n-1]
opos[2,:] = [m-1, n-1]
opos[3,:] = [m-1, 0]

# Work out the transformation from the match-points.
xform = find_perspective_transform (pos, opos)

# Invoke OpenCV routines to extract the region.
im =  cv2.imread (fn)
warp = cv2.warpPerspective (im, xform, (m, n))
cv2.imwrite ("rectified.jpg", warp)
cv2.waitKey ()
```

One important feature of this technique is that it does not involve any knowledge of the camera, whereas the most general techniques involve either calibrating the camera first or recording ancillary data about focal length or sensor size with the photographs. Of course, it does require knowledge that the four match-points are coplanar, and only a human is really able to determine that.

9.8 Visual structure from motion

How far can one extend this general approach? If one takes a few *hundred* images, then one can calculate the SIFT or other features of each image and match them up between images. These matches allow one to build up a set of equations that can be solved to yield the 3D positions that must have generated the images — see Figure 9.8.

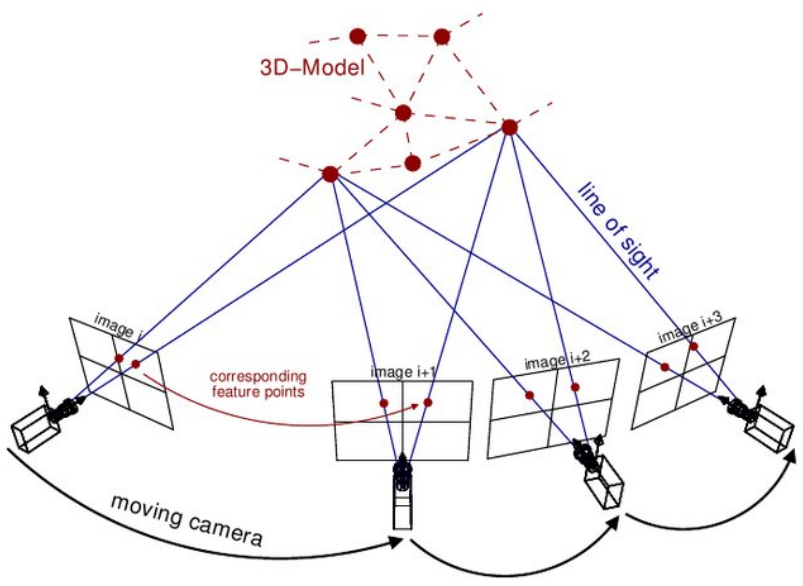


Figure 9.8: The principle of 3D reconstruction by structure from motion

The solution can take hours or even days, depending on the number of features and images, but one ends up with a cloud of individual coloured points that can be manipulated to show the scene in 3D. If the focal length of the camera is known for each image (which is usually recorded in the EXIF header of a JPEG image), then it is possible to scale the 3D reconstruction to match real-world features. This technique is known as *visual structure from motion* (VSFM). There have been open-source VSFM tools around for a few years now. The leading commercial product is currently Agisoft's Photoscan Pro. There are also very impressive free tools such as AliceVision's Meshroom, which we have used to construct and 3D-print a version of the University's Mace.

Here at Essex, we use these 3D reconstructions as part of our research. Views of a model of Square 2 produced in this way are shown in Figure 9.9, while a local church is shown in Figure 9.10. We are active in reconstructing coral reefs using VSFM and a reconstruction of about 100×50 m is shown in Figure 9.11.

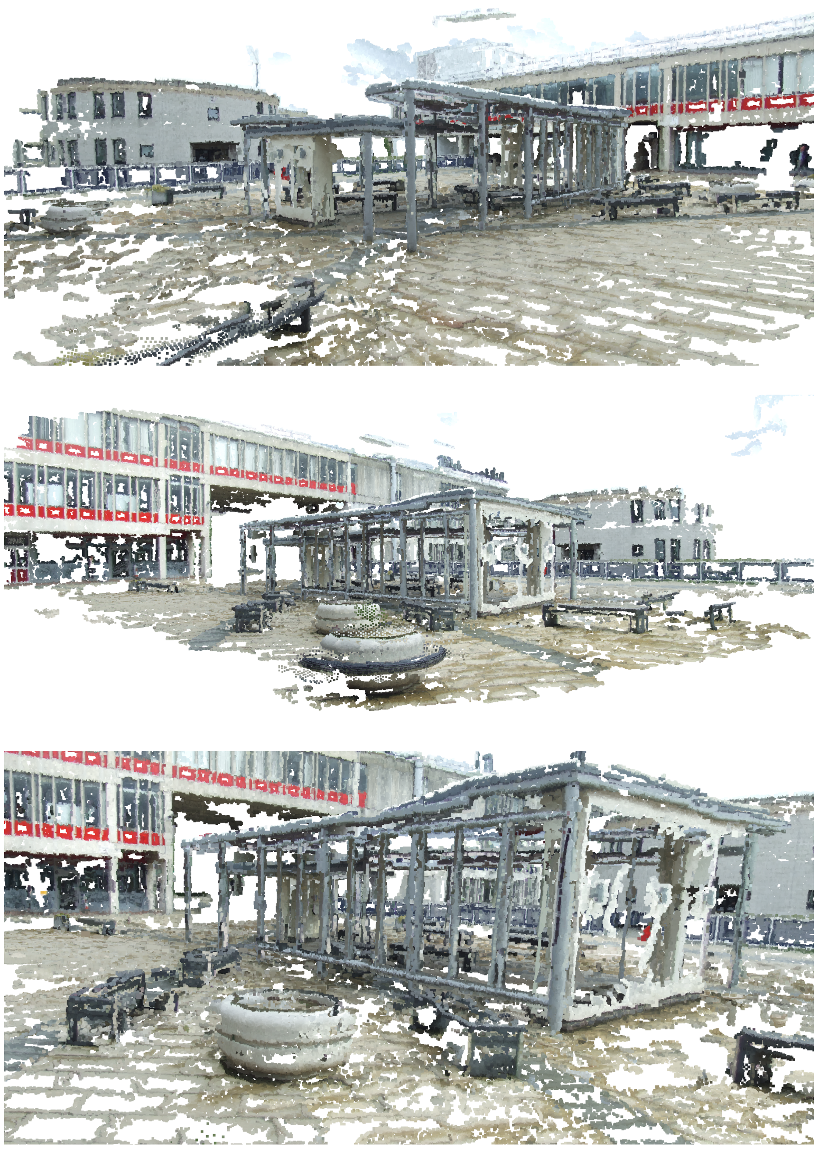


Figure 9.9: Views of a reconstruction of Square 2 using a structure from motion technique

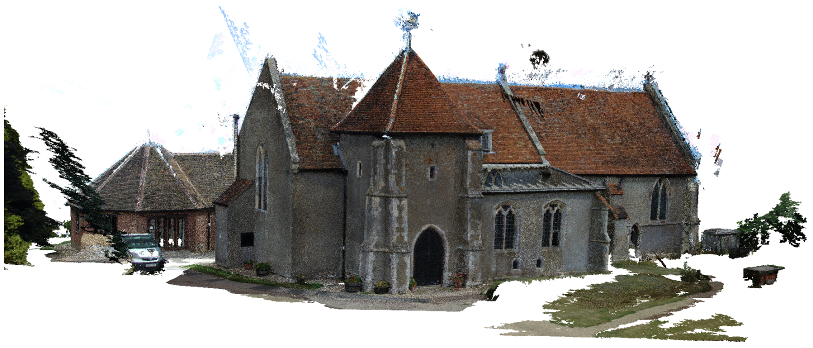


Figure 9.10: A reconstruction of Elmstead's church

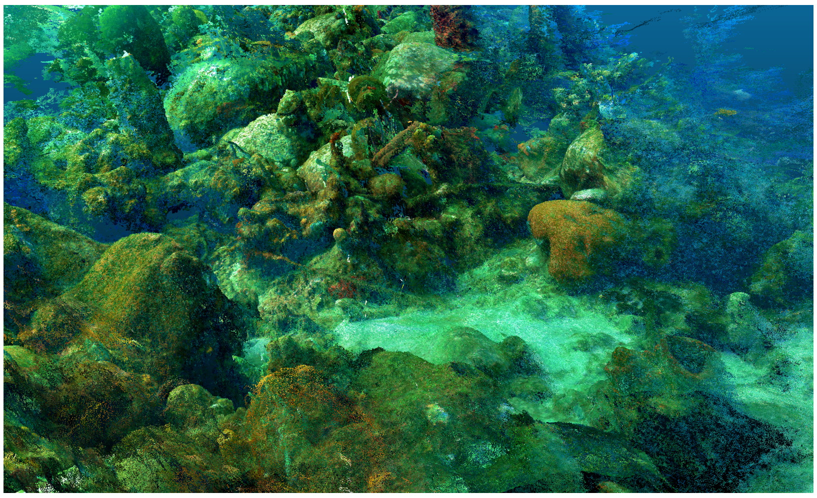


Figure 9.11: A coral reef reconstruction

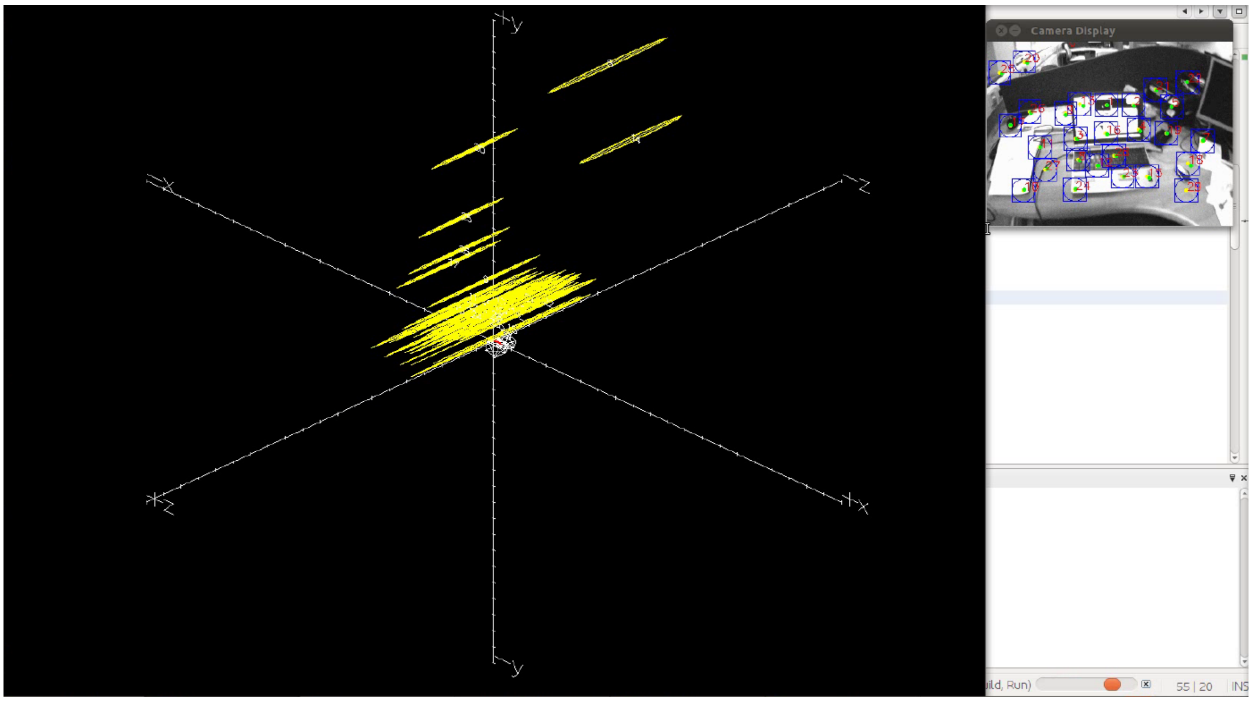
9.9 Visual SLAM

When we combine the pose estimation techniques outlined above with a temporal filter such as a Kalman filter, we end up with a camera-based version of *simultaneous location and mapping* (SLAM). SLAM was originally devised in robotics research, and allows a robot to explore an environment, identifying 'landmarks' and navigating by them in a way that makes both its position relative to those landmarks *and* the locations of the landmarks relative to it improve in precision the further it explores. Visual SLAM is essentially a vision-based version of the original (radar- and sonar-based) SLAM algorithm.

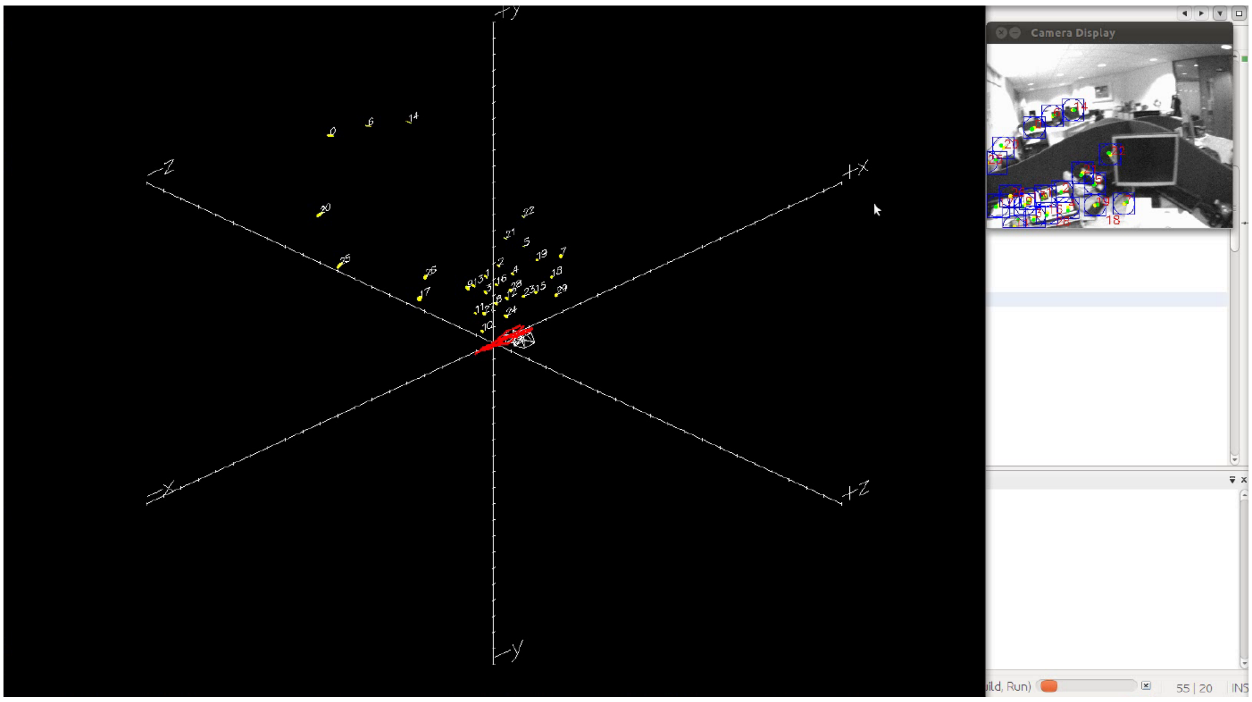
Visual SLAM has proven to be very effective for real-time augmented reality in moderately small environments, say a desktop or part of a room. With careful implementation and a fast graphics processor, it is able to run in real time. A major challenge currently being explored by vision researchers is making these techniques scale up so that, say, a car equipped with a camera can be driven around a town and 'know' when it returns to a place it has seen before. Some experiments to do this have been performed in Australia (where SLAM originated) and at Oxford but they are currently non-real-time and the researchers are struggling to make everything work well enough.

Here at Essex, we have also explored visual SLAM. Figure 9.12 shows two screen captures from our development system. The 3D view draws yellow ellipsoids showing the error in 3D of each feature tracked. As the feature is tracked successfully from frame to frame, these ellipsoids shrink, showing that the software is more confident of where that feature lies in 3D coordinates. The red line shows the trajectory of the camera.

The camera display to the top right shows what is happening in the video frames. Predicted feature locations from the extended Kalman filter are shown with squares. The circles within the squares indicate the covariance (essentially a measure of uncertainty) of the predicted positions. The exact predicted position is shown with yellow circles. When a prediction matches the true location of a feature, it is shown in green; when unsuccessful, it



(a) Shortly after initialization; note the size of the error ellipsoids



(b) After 700 frames, the error ellipsoids are much smaller

Figure 9.12: Visual SLAM in operation

is shown in red. This version of the software is using normalized cross-correlation to perform feature matching as SIFT *etc*. are all too slow to operate in real time. There are now a couple of Visual SLAM toolkits freely available: one is from Andy Davison's group at Imperial College and the other from the University of Zaragoza in Spain; the latter is based around matching ORB features and is reported to be very good.

We are currently exploring visual SLAM use for an augmented reality tour guide of local archaeological sites, in which a person equipped with a wearable computer and head-mounted display roams around the site, seeing 3D reconstructions of the buildings from the right orientation all the time.

10
High-Level Vision with Machine Learning

An introduction to machine learning in computer vision is provided. We start by discussing the important distinction into supervised and unsupervised approaches, and emphasise the importance of providing useful information to the machine learning algorithm to use.

A simple unsupervised technique is presented, and then a series of supervised schemes of increasing complexity are presented, in most cases covering the underlying principles and how they can be used in practice. Indicative performances are presented on some typical datasets.

10.1 Introduction

To conclude our exploration of computer vision, we shall consider the rôle that machine learning (ML) plays in it. Vision researchers were some of the first to make large-scale use of ML, and different techniques have been 'hot topics' at various times in recent years. In the early 2000s, the support vector machine (SVM) was the technique of choice; but recent years have seen the rise of 'deep learning' (neural networks with many hidden layers, which we shall consider in Chapter 11) in general and the convolutional neural network (CNN) in particular. The CNN is currently the most effective ML technique on many problems in computer vision, so much so that it seems that every other vision paper is concerned with its application. This furore will eventually dissipate and some more sensible science will hopefully ensue.

There are many ML techniques, and they fall into two broad categories:

Unsupervised. Given a set of unlabelled data, these techniques try to infer a way of partitioning them into classes with no knowledge of the data or underlying problem.

Supervised. These techniques 'learn' how a set of data is partitioned into classes from training examples.

As training data for supervised learning include the 'ground truth' labels for the problem, they generally produce solutions that are better able to classify unseen data. The majority of ML techniques that are widely known are supervised ones. Unsupervised techniques are more useful in exploratory data analysis, when the properties of the data or problem are not yet known.

ML algorithms are often not trained directly on images but rather on numbers extracted from images or image feature — a *feature vector*, which we first encountered in Chapter 5.

Although ML techniques can produce amazing results, the way in which they are trained is critical to their ultimate performance. If a developer trains any ML algorithm with data that do not encode useful information for distinguishing class labels, or do so poorly, it will not learn well. As mentioned in an earlier chapter, this is often stated succinctly as "garbage in, garbage out."

The final thing to be aware of is that supervised ML is slow — and as a rule of thumb, the more sophisticated the ML algorithm, the longer it takes to train. Given a top-of-the-range, multi-core, PC-class machine, training times of tens of minutes are achievable for some of the datasets we shall look at here using techniques such as SVM and multi-layer perceptron (MLP); for CNN, the equivalent training time is of the order of an hour — and as the dataset size increases, so does the training time. For this reason, researchers have expended a fair amount of effort in producing implementations of the most popular ML techniques on GPUs, the high-performance but dumb processors found on some graphics cards. Highly-optimised GPU code can run literally hundreds of times faster than conventional CPU-based code; but the author's experience using some of the libraries described here is that the speed-up is much more modest, about 3–10. Your mileage may vary, of course. Before you become enthused about writing GPU-based implementations yourself, let me warn you that programming parallel computers is *difficult*. Even with GPUs chugging away, training a convolutional neural network on state of the art image databases can take literally weeks!

10.2 Unsupervised learning using k-means

k-means is a clustering technique closely related to *vector quantization* in signal processing. It partitions N feature vectors into k clusters in which each feature vector belongs to the cluster with the nearest mean. A feature vector is considered to be in a particular cluster if it is closer to that cluster's centroid (mean) than any other centroid. The problem is formally computationally difficult ("NP-hard").

The most popular variant of the algorithm finds the best centroids by alternating between:

- assigning data points to clusters based on the current centroids; and
- choosing centroids (cluster centres) based on the current assignment of data points to clusters.

The algorithm has converged when the assignments no longer change. Six example runs of the algorithm are shown in Figure 10.1. In each case, ten points are randomly generated in each of six regions equally spaced around a circle. k-means is run for 100 iterations and the resulting classification of the points is shown in different colours.

Note that the algorithm has to be told the number of clusters to look for. Furthermore, there is no guarantee that the global optimum is reached, and

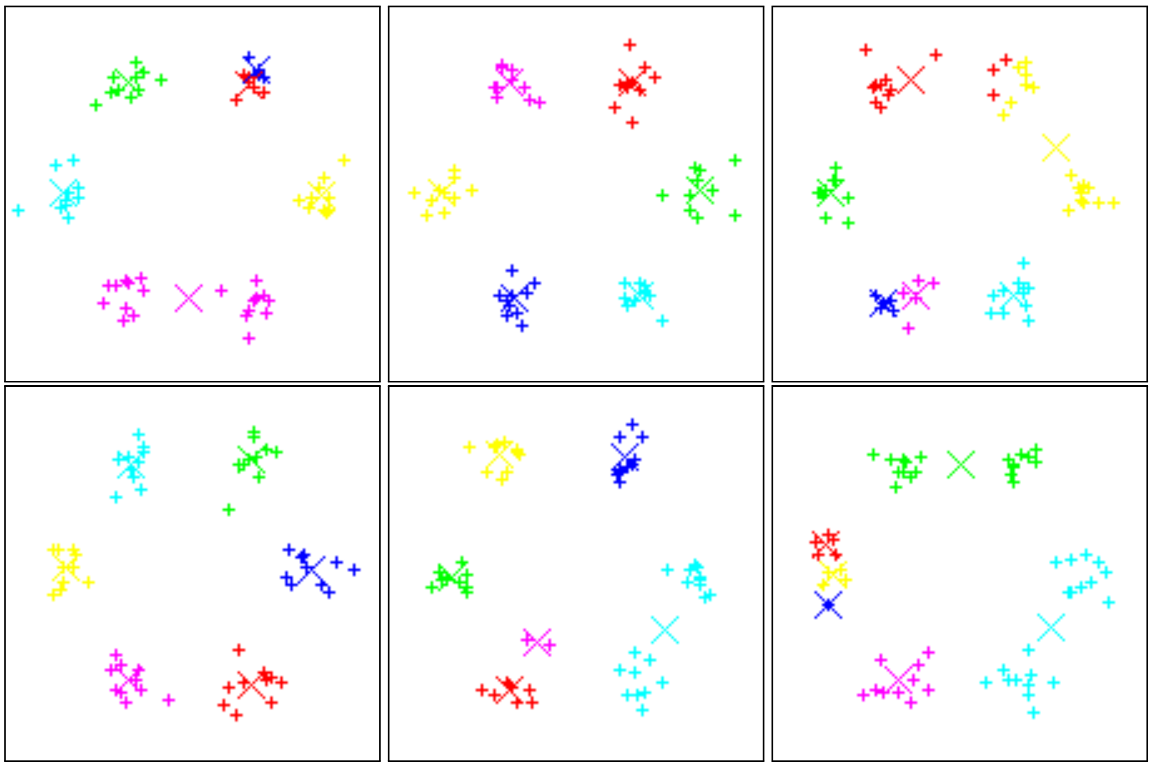


Figure 10.1: Examples of running the k-means algorithm. Training examples are shown as pluses and cluster centroids as crosses. The clusters identified are shown in different colours.

the local optimum found depends on the order in which the feature vectors are presented. However, it runs so quickly that many researchers execute it many times and take the best solution. There are implementations of the algorithm in Scipy and Scikit-learn, as well as in stand-alone packages.

People tend to confuse the technique known as *k nearest neighbours* ("k-NN") with k-means but they are very different. Given a set of known clusters, k-NN classifies a feature vector into a class based on the class of the majority of its k nearest neighbours; it is common for $k = 1$, *i.e.* to use the class of a feature vector's nearest neighbour.

10.3 Simple supervised learning with WISARD

WISARD is a simple pattern recognition scheme, devised in the 1970s by Wilkie, Stonham and Aleksander; the name stands for *Wilkie, Stonham and Aleksander's Recognition Device* [Aleksander et al., 1984]. It is sometimes described as a neural network, though many do not regard it as so because it is weightless and can be trained by a single data presentation. It is occasionally described as an "associative memory," which is closer to how it actually works, and as an "n-tuple" network. However, we don't need to get ourselves bogged down in such pedantry here; WISARD does a simple kind of ML.

The operation of WISARD is illustrated in Figure 10.2. It is intended to operate on binary images, *i.e.* ones whose pixels may take only the values zero and unity. The system consists of a frame buffer, a register or *discriminator element* and a 1-bit RAM — the simplicity of the architecture is because WISARD was designed to be implemented in hardware. In fact,

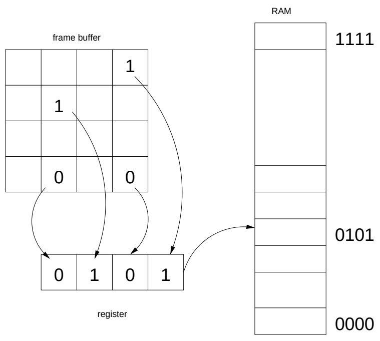


Figure 10.2: Illustration of the operation of WISARD

the original hardware was on display in the Science Museum in 2017, and Figure 10.3 shows an image of it.

Initially, every element of the RAM is cleared (set to zero). Several pixel locations, four in Figure 10.2, are chosen at random[1] and connected to the bits of the register. When an image is loaded into the frame buffer, the values held in those four locations determine the value in the register. What happens next depends on whether WISARD is being trained or tested (used for recognition):

- when being trained, the value in the RAM addressed by the register is set;
- when being tested, the value in the RAM addressed by the register is compared with unity.

Fairly obviously, when the test pattern is identical to the training pattern, at least in the selected locations, a match will be found.

Of course, if WISARD used only four samples from an image, it would not be particularly robust; hence, several other sets of four randomly-chosen pixels are used in the same way. The number of comparisons from groups of four pixels that produce a match are added up, resulting in a "score" that determines how close the overall match is.

An implementation of WISARD is available in one of the associated lab experiments. Note that a single WISARD network is able to recognize only one pattern. To be able to distinguish several patterns, it is necessary to train a network for each of them, then run the test pattern through them all and ascertain which gives the largest score.

Figure 10.3: WISARD on display in the Science Museum

[1] Clearly, these random locations must be the same during the training and testing phases discussed below. This is done by recording the locations used for training in the file that results from it, and then examining those locations when testing.

10.4 The MNIST test case

To explore how well WISARD and some of the other techniques described below work, we shall again use one of the best-known 'standard' datasets, the MNIST database of handwritten digits considered in Chapter 6. There are some 60,000 training images and 10,000 test images in it, arranged as shown in Table 10.1; Figure 10.4 shows some typical examples. Each image is only 28 × 28 pixels in size.

digit	train	test
0	5923	980
1	6742	1135
2	5958	1032
3	6131	1010
4	5842	982
5	5421	892
6	5918	958
7	6265	1028
8	5851	974
9	5949	1009
total	60000	10000

Table 10.1: Numbers of training and test examples for each digit in the MNIST database

Testing the above implementation of WISARD by hand on 10,000 test images would be both error-prone and tiresome, so it is best to resort to a test harness; we shall use HATE, the big brother of the FACT test harness which you have used in your laboratories.

Results from training and testing on the complete MNIST database are shown in Table 10.2 (accuracies) and Table 10.3 (class confusion matrix). You will see that the results are pretty awful because a large proportion of the matches are ambiguous (more than one network returns the same score), which causes the 'reject' class to be returned. From this we conclude that WISARD is poor at encoding the appearance of its training data.

Additional evidence for this conclusion can be gleaned by training and testing WISARD on a subset of the MNIST database. Using only 50 images of each class for training and about 140 other images for testing, we obtain the results in Table 10.4 (accuracy) and Table 10.5 (class confusion matrix).

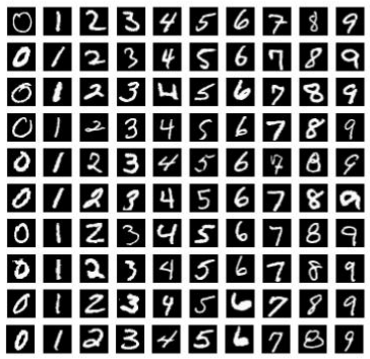


Figure 10.4: Typical images from the MNIST database

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
980	118	0	20	842	0.1204	0.1229	0.8551	0.0000	0
1135	41	0	5	1089	0.0361	0.0363	0.8913	0.0000	1
1032	220	0	47	765	0.2132	0.2234	0.8240	0.0000	2
1010	64	0	39	907	0.0634	0.0659	0.6214	0.0000	3
982	133	0	15	834	0.1354	0.1375	0.8986	0.0000	4
892	79	0	18	795	0.0886	0.0904	0.8144	0.0000	5
958	263	0	16	679	0.2745	0.2792	0.9427	0.0000	6
1028	99	0	19	910	0.0963	0.0981	0.8390	0.0000	7
974	63	0	34	877	0.0647	0.0670	0.6495	0.0000	8
1009	36	0	8	965	0.0357	0.0360	0.8182	0.0000	9
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	reject
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	fail
10000	1116	0	221	8663	0.1116	0.1141	0.8347	0.0000	overall

Table 10.2: MNIST WISARD error rates

true	class returned by algorithm											
class	0	1	2	3	4	5	6	7	8	9	reject	fail
0	118	0	4	9	1	2	3	0	1	0	842	0
1	1	41	0	0	0	0	1	0	3	0	1089	0
2	4	2	220	21	4	3	3	1	8	1	765	0
3	5	1	22	64	1	3	0	3	3	1	907	0
4	2	0	3	4	133	2	2	1	1	0	834	0
5	7	0	3	1	2	79	1	1	3	0	795	0
6	2	0	3	1	1	6	263	1	2	0	679	0
7	1	0	6	8	1	0	0	99	0	3	910	0
8	5	1	4	6	6	6	5	0	63	1	877	0
9	0	0	1	1	0	2	1	2	1	36	965	0
reject	0	0	0	0	0	0	0	0	0	0	0	0
fail	0	0	0	0	0	0	0	0	0	0	0	0

Table 10.3: Class confusion matrix for WISARD on MNIST

These results are significantly better than those from the full database, so we can conclude that WISARD's ability to learn has been overwhelmed by the volume of data in the full MNIST database.

The author's view is that the underlying approach of WISARD has more potential than these results suggest: there are similarities between it and the BRIEF and ORB descriptors, for example, and the number of places in which the image is sampled is probably too few here. It would be interesting to get to grips with it properly and see what improvements can be made.

10.5 Support vector machines

The SVM [Cortes and Vapnik, 1995; Alpaydin, 2014] has been widely used by the research community for supervised machine learning. Unlike all of the techniques we shall consider below, the SVM is *deterministic* rather than stochastic and so returns exactly the same result every time it is run.

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
139	80	0	9	50	0.5755	0.6154	0.8989	0.0000	0
148	82	0	13	53	0.5541	0.6074	0.8632	0.0000	1
145	102	0	9	34	0.7034	0.7500	0.9189	0.0000	2
149	77	0	17	55	0.5168	0.5833	0.8191	0.0000	3
136	78	0	15	43	0.5735	0.6446	0.8387	0.0000	4
137	79	0	9	49	0.5766	0.6172	0.8977	0.0000	5
145	97	0	6	42	0.6690	0.6978	0.9417	0.0000	6
151	66	0	20	65	0.4371	0.5038	0.7674	0.0000	7
130	51	0	25	54	0.3923	0.4857	0.6711	0.0000	8
154	58	0	27	69	0.3766	0.4567	0.6824	0.0000	9
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	reject
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	fail
1434	770	0	150	514	0.5370	0.5997	0.8370	0.0000	overall

Table 10.4: Table of error rates for WISARD on a subset of MNIST

true	class returned by algorithm											
class	0	1	2	3	4	5	6	7	8	9	reject	fail
0	80	0	0	2	1	1	2	0	1	2	50	0
1	0	82	5	0	1	0	1	0	3	3	53	0
2	1	0	102	0	0	2	0	2	4	0	34	0
3	0	0	6	77	0	1	0	0	1	9	55	0
4	0	5	0	0	78	1	1	3	2	3	43	0
5	0	1	2	3	1	79	1	0	0	1	49	0
6	5	1	0	0	0	0	97	0	0	0	42	0
7	0	0	0	10	0	1	0	66	1	8	65	0
8	0	4	3	4	0	0	5	4	51	5	54	0
9	4	5	0	6	1	2	0	6	3	58	69	0
reject	0	0	0	0	0	0	0	0	0	0	0	0
fail	0	0	0	0	0	0	0	0	0	0	0	0

Table 10.5: Class confusion matrix for WISARD on a subset of MNIST

As a consequence, it is also much quicker to train.

The mathematics behind the SVM is too messy to cover here, but the principle behind it is easy to understand (Figure 10.5). Suppose some feature vectors each belong to one of two classes and we need to determine which class a new data point belongs to. If the feature vector has p numbers in it, we are working in a p-dimensional space and need to determine whether we can separate such points with a $(p-1)$-dimensional hyperplane. If this is possible, we have a so-called *linear classifier*. There are many hyperplanes that might classify the feature vector correctly. Arguably the best maximises the separation (the *margin*) between the two classes, so we choose the one for which the distance from it to the nearest feature vector on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum margin classifier. The SVM is essentially an algorithm for calculating this maximum margin classifier.

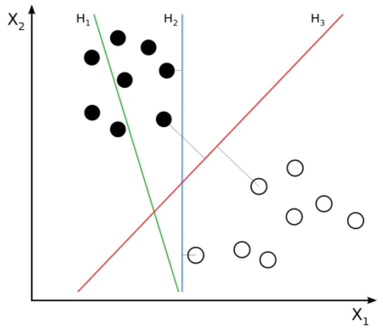


Figure 10.5: Principle of the support vector machine (image from the Wikipedia). H_1 does not separate the classes. H_2 does, but only with a small margin. H_3 separates them with the maximum margin.

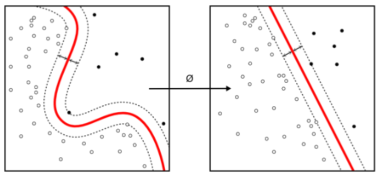


Figure 10.6: The 'kernel trick' for performing non-linear classification with an SVM (image from the Wikipedia)

The ability to produce only linear classifiers mean that the SVM cannot be used directly on problems where the classes cannot be separated by a hyperplane. However, the people who dreamt up the SVM proposed the use of a 'kernel trick' (Figure 10.6) which scales feature vectors non-linearly so that the SVM is applicable (and then undoes the scaling).

How well does SVM perform on MNIST? To be able to ascertain that, one needs an implementation and, unusually, almost everyone uses the same software, `libSVM` [Chang and Lin, 2011], which is available in C++ and Java. Just as with OpenCV though, people have produced 'wrappers' for the library for Python. The particular one that will be employed here is `Scikit-learn` [Pedregosa et al., 2011]. A program for training and testing SVM on MNIST is shown below, though without its support routines; you will find the complete program in one of the laboratories.

⟨*train-and-test-svm.py*⟩ ≡

```
#!/usr/bin/env python3
"Train␣up␣a␣SVM␣and␣run␣it␣on␣its␣test␣dataset."
import sys, numpy, os, struct, array, time, datetime

<<Support routines for training on MNIST>>

# Ensure we were invoked with the database name.
if len (sys.argv) != 3:
    print ("Usage:", sys.argv[0], "<database>␣<transcript>", file=sys.stderr)
    exit (1)

# Import the machine learning technique and report its version.
import sklearn
print ("Using␣Scikit-learn␣version:", sklearn.__version__)
from sklearn import svm

# Read in the training and testing datasets and pull out the classes.
train_dataset = load_dataset (sys.argv[1], "train")
test_dataset = load_dataset (sys.argv[1], "test")

print_summary (sys.argv[1], train_dataset, test_dataset)
```

```

# Convert the training and test datasets into the formed needed by scikit-learn
# and do some sanity-checking.
train_images, train_labels = dataset_to_scikit (train_dataset)
test_images, test_labels = dataset_to_scikit (test_dataset)

assert len (train_images) == len (train_labels)
assert len (test_images) == len (test_labels)

# Create a classifier with the parameter settings from
# http://www.trungh.com/2013/04/digit-recognition-using-svm-in-python/
classifier = sklearn.svm.SVC (kernel="rbf", C=2.8, gamma=0.0073)

# Train the SVM.
start = time.time ()
try:
    classifier.fit (train_images, train_labels)
except ValueError:
    print ("Alas,␣training␣failed!", file=sys.stderr)
    exit (1)
duration = time.time() - start
print ("Training␣took␣%.1f␣seconds." % duration)

# Evaluate the trained SVM on the test data and save the results in a form
# we can analyse with FACT.
start = time.time ()
results = classifier.predict (test_images)
duration = time.time() - start
output_transcript (sys.argv[2], results, test_labels, sys.argv[1], duration)
```

Learning time on the author's (pretty fast) laptop is about 4.5 minutes. Testing takes about 1.5 minutes. We obtain the results in Table 10.6 (accuracies) and Table 10.7 (class confusion matrix). You will see that its performance is a significant improvement on that of WISARD.

Although SVM can produce high accuracies, the author's own experience of it is that the parameters that tune how it learns need to be twiddled with quite a lot in order to obtain good results. This can take a long time and makes the technique less suitable for use in an autonomous vision system.

10.6 The genetic algorithm

The genetic algorithm (GA) [Goldberg, 1989] starts with a population of N individuals, often termed 'chromosomes.' Each chromosome has a random set of the P values that need to be optimised, with the equivalent location in each chromosome representing the equivalent value (Figure 10.7). It works by altering the values in the chromosomes in a way modelled loosely on biological evolution, using operators that mimic sexual reproduction.

The population goes through a set of 'generations' and in each generation, some individuals 'mate' to produce 'children' by exchanging values using principally a mechanism called *crossover*. As originally conceived, a GA works with integers values only. That is not convenient for many real-world problems, so an attractive alternative is a *real-valued* GA [Mühlenbein and Schlierkamp-Voosen, 1993]. To combine individual 'chromo-

tests	TP	TN	FP	FN	accuracy	recall	precision	specificity	class
980	972	0	8	0	0.9918	1.0000	0.9918	0.0000	0
1135	1126	0	9	0	0.9921	1.0000	0.9921	0.0000	1
1032	1013	0	19	0	0.9816	1.0000	0.9816	0.0000	2
1010	993	0	17	0	0.9832	1.0000	0.9832	0.0000	3
982	963	0	19	0	0.9807	1.0000	0.9807	0.0000	4
892	867	0	25	0	0.9720	1.0000	0.9720	0.0000	5
958	944	0	14	0	0.9854	1.0000	0.9854	0.0000	6
1028	997	0	31	0	0.9698	1.0000	0.9698	0.0000	7
974	949	0	25	0	0.9743	1.0000	0.9743	0.0000	8
1009	973	0	36	0	0.9643	1.0000	0.9643	0.0000	9
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	reject
0	0	0	0	0	0.0000	0.0000	0.0000	0.0000	fail
10000	9797	0	203	0	0.9797	1.0000	0.9797	0.0000	overall

Table 10.6: Table of error rates for SVM on MNIST

true	class returned by algorithm											
class	0	1	2	3	4	5	6	7	8	9	reject	fail
0	972	0	1	0	0	3	1	1	2	0	0	0
1	0	1126	3	1	0	1	1	1	2	0	0	0
2	5	1	1013	0	1	0	1	7	3	1	0	0
3	0	0	2	993	0	2	0	6	5	2	0	0
4	0	0	5	0	963	0	3	0	1	10	0	0
5	3	0	0	10	1	867	4	1	4	2	0	0
6	5	2	1	0	2	3	944	0	1	0	0	0
7	1	7	10	2	2	0	0	997	1	8	0	0
8	3	0	2	6	5	2	2	2	949	3	0	0
9	3	3	1	7	10	1	1	7	3	973	0	0
reject	0	0	0	0	0	0	0	0	0	0	0	0
fail	0	0	0	0	0	0	0	0	0	0	0	0

Table 10.7: Class confusion matrix for SVM on MNIST

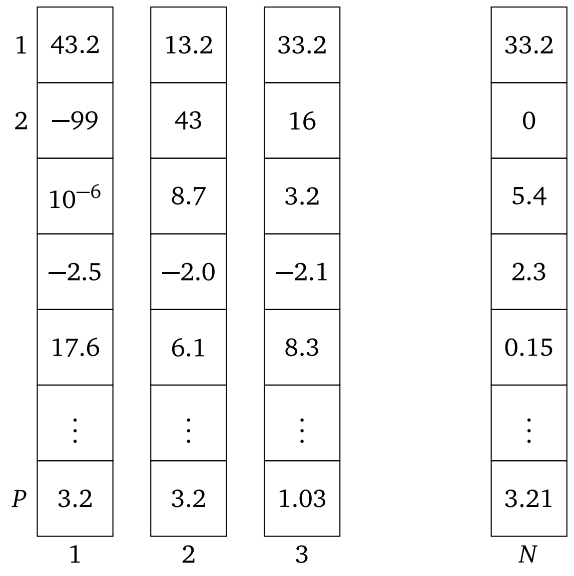


Figure 10.7: A population of N chromosomes, each having P values

somes' in a real-valued GA, one simply draws an imaginary straight line between equivalent locations in individuals and moves a random distance α along that line to produce a 'child' value, as illustrated in Figure 10.8.

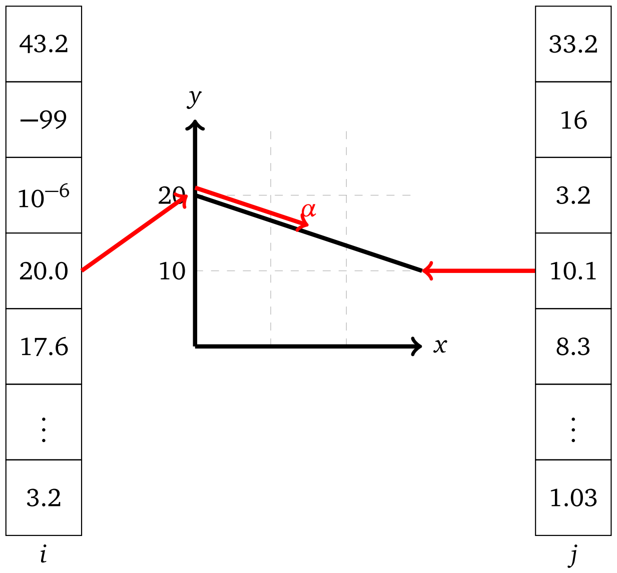


Figure 10.8: Real-valued crossover

If this procedure is carried out on the whole population equally, there would be little progress towards finding an optimum (the best set of all p values). However, if one allows only the best chromosomes (how these are determined will be explained shortly) to mate, the individuals produced by crossover tend towards the optimum values over a reasonable number of generations. This procedure is helped if the best individuals in a particular generation are allowed to 'live' into the next generation too, a feature called *elitism*.

How well each individual performs is measured using a so-called *cost function* or *fitness function*. This uses the values stored in each chromosome of the population in turn and computes via some procedure how well the chromosome solves the problem. In a cost function, lower values mean better and a cost of zero corresponds to a perfect solution; conversely, in a fitness function, larger values mean better and a perfect solution often corresponds to a value of unity or 100. Here, we shall use cost functions only.

From the above discussion, there remains the possibility that the population becomes trapped in the vicinity of a local minimum of the cost function. Hence, a GA normally also performs *mutation* on some individuals, randomly changing the values in randomly-chosen locations to random values. This will often 'kick' the population out of a local minimum.

Hence, to define a problem to a GA, all one needs to know in principle are how many unknown parameters there are and a cost function. In practice, a GA will work towards an optimum solution better if all the parameters are bounded (*e.g.,* an angle always lies in the range 0–2π or 0–360).

10.7 Genetic programming

As with the GA, most ML techniques operate on numeric values. Genetic programming (GP) is different in that it delivers instead *a program* that solves a problem. The origins of the technique date back to the mid-1980s but it was the ground-breaking work of [Koza, 1992] that established the technique as it is known today, showing that many then standard problems in artificial intelligence could be solved using it. Koza subsequently applied the technique to many other problems, showing for example that it could both reproduce patented antenna designs and devise novel ones. A good modern text that describes genetic programming (GP) is [Poli et al., 2008].

GP is similar in concept to a GA but rather than working with sets of numbers, it uses *parse trees*. A parse tree is an internal representation of a piece of computer code; for example, the one illustrated in Figure 10.9 corresponds to the program fragment

```
(if (= corners 4)
   (if (= asprat 1)
      "square"
      "rect")
    "other")
```

```
if (corners == 4) {
    if (asprat == 1) {
        return "square"
    } else {
        return "rect"
    }
} else {
    return "other"
}
```

The code on the left above is presented in the Lisp programming language, which is especially well-suited to this kind of task because it is easy to write out a parse tree in this form and equally easy to convert Lisp into a parse tree. It is also pretty compact. Equivalent C-like (Java-like, ...) code is to the right above.

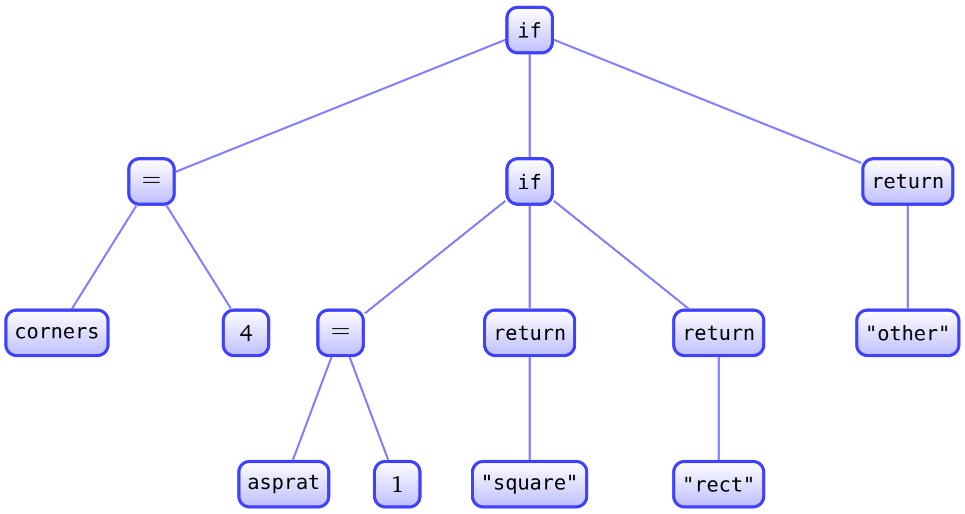
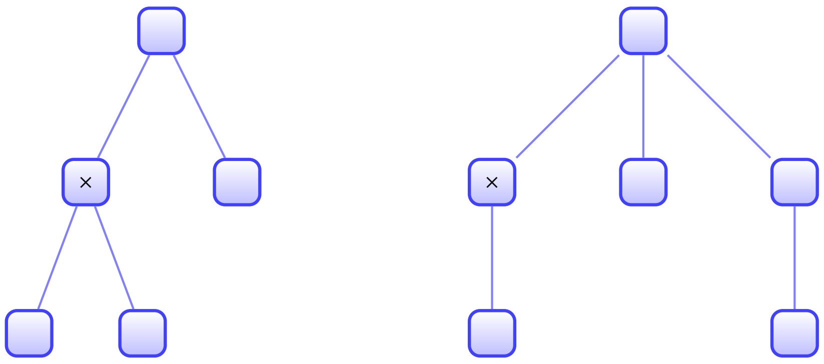
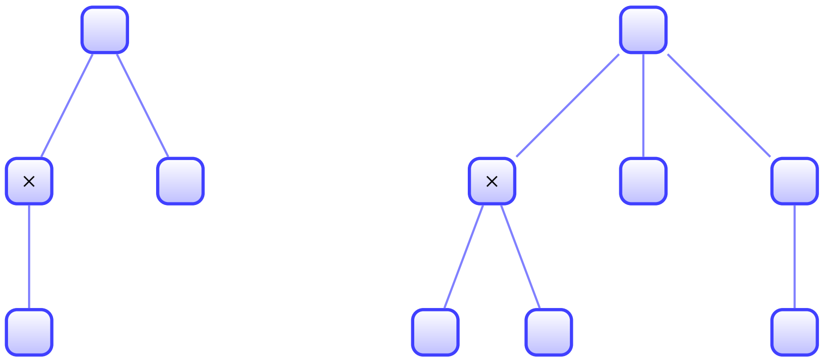


Figure 10.9: Representation of program code as a parse tree

Just as with a GA, GP 'evolves' the population using elitism, crossover and mutation. The latter two are somewhat different from the GA case, however. Crossover moves entire sub-trees from one parent into another to create a child (Figure 10.10), while mutation replaces a sub-tree with a randomly-generated new one. Early research into GP found that programs grew quickly, so it is common to limit the depth of programs when they are created or modified using crossover or mutation.



(a) Parents with the crossover points indicated by ×



(b) Children showing the exchanged sub-trees

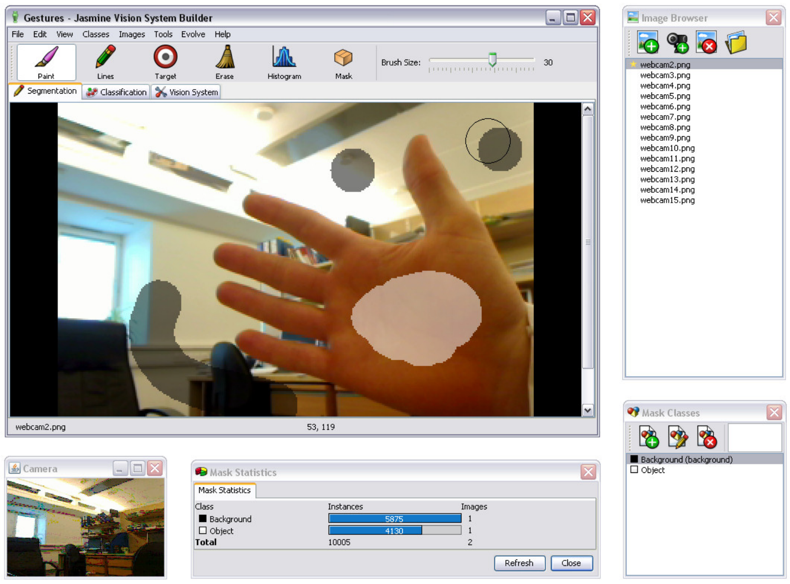
Figure 10.10: Crossover exchanges sub-trees between parse trees

The individuals chosen to produce children programs are normally chosen in GP using a mechanism called *tournament selection*. For a tournament of size N, N individuals are randomly selected from the population and the best-performing of them (*i.e.,* the one with the lowest cost) is chosen as one parent. The same procedure is followed to select the second parent, and then random nodes are chosen in both parents to select the sub-trees that form the new individuals.

A cost function is used to determine how effective an evolved program is. In the GP case, this is done by executing the program on known input and determining how close the corresponding outputs are to the correct ones. In our work (see below), our cost function counts the number of pixels (for segmentation) or regions (for classification) that agree with the training data; these are exactly the same criteria used for evaluating their performance — this is *not* usually the case with machine learning techniques such as neural networks.

The example parse trees discussed above employ numbers and logical tests. These map well onto the arithmetic and logical types that all computers represent. However, there is nothing in principle that prohibits GP from evolving programs that make use of other operators, as long as there is an implementation of those operators available when calculating the cost function. Useful functions would be, for example, `sqrt` for calculating square roots.

Working with PhD students Olly Oechsle, Panitnat Yimyam and now



webcam2.png
webcam3.png
webcam4.png
webcam5.png
webcam6.png
webcam7.png
webcam8.png
webcam9.png
webcam10.png
webcam11.png
webcam12.png
webcam13.png
webcam14.png
webcam15.png

■ Background (background)
☐ Object

Class	Instances	Images
■ Background	5875	1
☐ Object	4130	1
Total	10005	2

Figure 10.11: Jasmine, a GP-based vision system construction tool

Julian Forrester, the author's own research uses GP to build vision systems. The starting point is a 'library' of vision components that perform low-level but useful tasks, many inspired by capabilities of the human visual system, then uses GP to combine them into programs. The machine-generated programs are then executed on real-world imagery, with the cost function measuring how well the programs perform in a way analogous to a test harness.

Olly's research resulted in a system called Jasmine, and Panitnat enhanced and extended it (Figure 10.11). Rather than chasing the ultimate in performance, we tried to show the versatility of the approach. Jasmine was *without any modification whatsoever* able to produce solutions to the problems shown in Figure 10.12. Only reading car number plates took more than an hour to evolve the solution on a single desktop PC.

Over the summer of 2017, thanks to some external funding, Julian Forrester (at the time an undergraduate student), Christine Clark and the author re-implemented and extended the functionality of Jasmine in Python to produce a package called ELVS (for *Evolutionary Learning Vision System*); we are now using that to solve a number of real-world image recognition problems. Our approach requires a minuscule number of training images by comparison to neural networks and trains in the order of ten minutes. Moreover, no reconfiguration of our system or customization of the cost function is required to adapt it to different problems, you simply present it with different training data. You don't have to identify all the pixels in a region in the training data; ELVS uses only what you have marked. Moreover, the number of training images is normally small: in most of our work, we use no more than five — compare this with the number required to train a typical neural network!

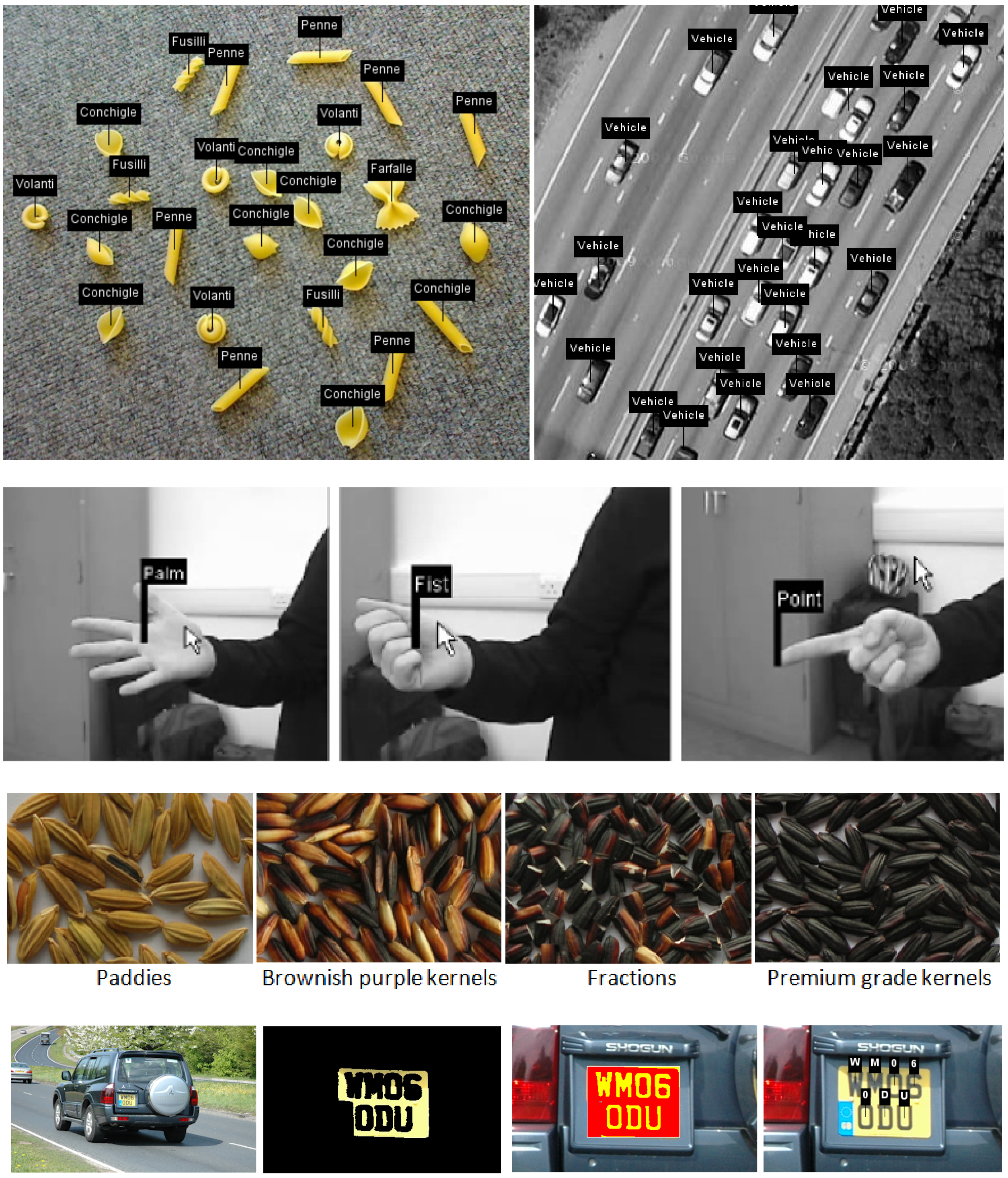


Figure 10.12: Some of the problems for which Jasmine has evolved solutions. Top row: pasta shape identification, identifying vehicles from aerial images. Second row: gesture recognition in videos. Third row: rice grading. Bottom row: locating and reading car number plates.

To illustrate how to use ELVS, consider the rather simple problem shown in Figure 10.13, which has two classes of fairly easily distinguished object. The following “task file” specifies the image input, the ground truth and the annotation used in the latter for the various classes of output:

```
# Adrian F Clark <alien@essex.ac.uk> 2023-03-16

name: test01
type: vision
purpose:
  Segmentation and classification of a synthetic test image.

class:
  unknown    #7f7f7f
  background #ffffff
  circle     #0000ff
  rectangle  #ff0000

property:
  annotation marked

dataset: train
   im-000.png gt-000.png
```

where the two images in the last line are those in Figure 10.13.

As explained above, ELVS initially combines image operators into programs randomly, evaluates them on the data and subsequently ‘mates’ the best of them in each generation. Evolution is directed via a cost function which simply counts the number of pixels the program gets wrong; if two programs do equally well, it prefers the shorter one. ELVS evolves a complete solution in two stages, first evolving a segmenter and then a classifier. For the problem shown above, a short, perfect segmenter is

```
  (CHrange input)
```

which calculates at each pixel of its `input` the difference between its largest and smallest colour channels. Other perfect segmenters include

```
  (Otsu (0-im1 (im1-i (im1+im1 (im> (Otsu (CHrange input)) 256i 0i) (CHmin
    (im3*im3 (im3-i (im3*im3 (im3-i input 150i) (i-im3 5i input)) 25i)
    (im3+im3 (im3/im3
    (i-im3 25i input) (im3/im3 input input)) (im3+i (im3+i input 4i)
    (i+ 4i 100i)))))) 2i)))
```

which is harder to read but can nevertheless be understood. The best segmenter the author has seen evolved is

```
  (Otsu (grey input))
```

as it is exactly what he would write. ELVS comes up with this reasonably often.

ELVS can evolve classifiers in different ways. The best is arguably to evolve one for each class of output, in which case perfect ones for the two classes are:

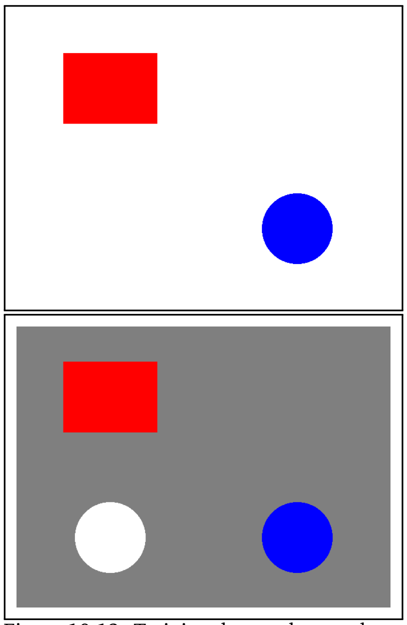


Figure 10.13: Training data and ground truth for a simple vision task

```
(f< 1f (occupancy input))          # circle
(f> (height input) (width input))  # rectangle
```

`occupancy` is the ratio of the area of the region to that of its bounding box, which is unity for a rectangle and < 1 for a circle. For this task, the height of a region being larger than its width is enough to identify it as a rectangle. This is clearly not going to work for every rectangle but that is a consequence of the choice of training data rather than the machine learning process.

If we tell ELVS to return the label of a region, a good solution is:

```
(Label.if (f> (f/ 128f (occupancy input)) (f- (f+ (mean3 (im3+im3
  (HSV Region.RGB) Region.RGB)) 8f) (i->f 150i))) rectangle)
```

which we can see is a good bit more complicated. A solution closer to the one-classifier-per-class result is

```
(Label.ifelse (! (f> (width input) (occupancy input)))
  circle rectangle)
```

However, this takes longer to run and finds a perfect solution less often, a consequence of needing to evolve a more complicated program.

We can improve on the one-classifier-per-class approach shown above by *discarding* the training data for the first class recognized once its classifier has been evolved, those for the second class once its classifier has been evolved, and so on; see Figure 10.14. This means that ELVS evolves a series of binary classifiers, leaving the most difficult until last. We have found that this approach lets us evolve solutions to large multi-class problems fairly quickly, we just have to remember the order in which they were evolved and run them on test images in the same order. We call this a *cascade* as it is similar to a Haar cascade in the Viola-Jones algorithm.

A final point for you to note is that we have been able to read the evolved solutions from ELVS and understand precisely how they work, a desirable property known as *explainable AI*. As things currently stand, *this is not possible with any other machine learning technique.*

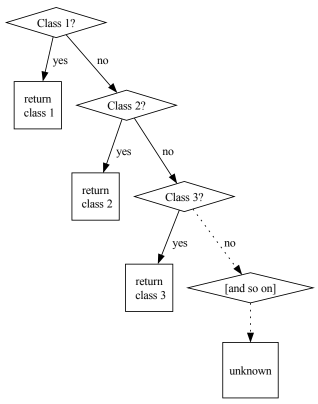


Figure 10.14: A cascade of binary classifiers

11
Deep Learning and Neural Networks

Neural networks, especially those with many hidden layers, represent the current state of the art in machine learning applied to computer vision. We review two neural network approaches, the well-established Multi-Layer Perceptron *and the more recent* Convolutional Neural Network*, and touch on two interesting current research themes, Yolo and GAN.*

11.1 Introduction

The brains of animals consist of large numbers of simple neurons which have many connections to other neurons. A neural network is a computer model of this: an individual neuron contains little information but is connected to many other neurons. They were first devised in the 1950s but came to prominence in the 1980s, when they were shown to be able to solve some problems that other techniques could not. Interest waned in the 1990s because techniques such as SVM out-performed them, but has been re-invigorated in recent years because networks with many layers (so-called 'deep learning' networks) out-perform other mainstrea ML techniques. (This kind of 'arms race' between machine learning techniques will undoubtedly continue.)

We shall start by considering a 'traditional' neural network, the MLP, because that establishes the fundamental idea of how neurons are modelled and how training proceeds. We then move on to the CNN, looking at its basic idea and some common instances of it in recent research.

11.2 The multi-layer perceptron

An MLP is a network of neurons that maps a set of inputs onto a set of outputs — for example, with the MNIST dataset in mind, the pixels of an image onto the 10 classes of digit. It consists of a number of layers of *neurons* (simple computational elements) as illustrated in Figure 11.1, where the circles represent neurons and the lines between them *interconncetions*. An MLP is a *fully-connected* network as each node in one layer connects with some weight w to every node in the following layer. The data are presented at the *input layer*, which passes them to a single *hidden layer* in Figure 11.1, and that in turn passes them to an *output layer*. The layers can of course be organised in two dimensions to accept images as input, and 'deep learning' architectures have more than one hidden layer.

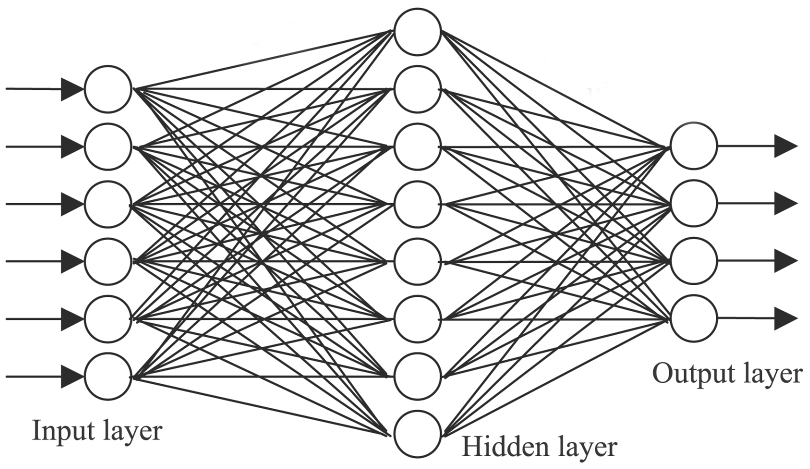


Figure 11.1: Illustration of a multi-layer perceptron. (Image from http://www.slideshare.net/steveomohundro/tedx-talk-whats-happening-with-artificial-intell

Neural networks of the 1980s normally had a single hidden layer as training a network took a long time; the longest time the author has allowed a network to train was 88 cpu days — about five months of elapsed time! (Thankfully, it worked well.) Recent years have seen computational performance increase to the point that a dozen or more hidden layers are not uncommon, though training times remain long.

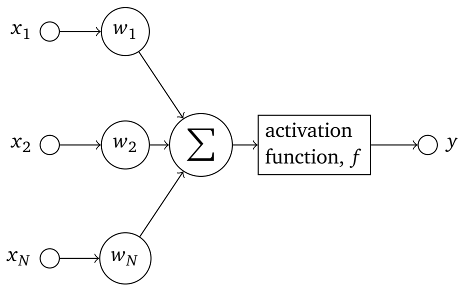


Figure 11.2: Illustration of a single neuron

The simplest neuron sums its N weighted inputs and uses the result as the argument of a non-linear function f:

$$y = f\left(\sum_{i=1}^{N} x_i w_i\right) \tag{11.1}$$

where x_i are the inputs to a neuron and w_i the corresponding connection strengths (Figure 11.2). Strictly speaking, to be a *perceptron*, the output of the neuron should be zero or unity; but it has become customary to use smoother functions, and popular choices are 'sigmoid' functions such as

$$y = \tanh\left(\sum_{i} x_i w_i\right) \tag{11.2}$$

and

$$y = \frac{1}{1+\exp(-\sum_i x_i w_i)} \tag{11.3}$$

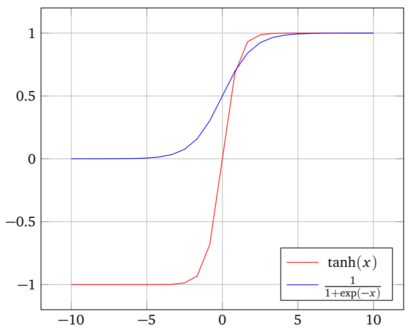


Figure 11.3: Common choices for the activation function of a neuron

as shown in Figure 11.3. More sophisticated choices include various different *radial basis functions* such as Gaussians. The activation function needs to be differentiable.

Training, or learning, rules specify an initial set of connection weightings and indicate how these weights should be altered as training proceeds. In supervised training, the neural network is supplied with a target output for each input pattern. The quality of the output may be measured by computing the RMS error, and one of several iterative schemes may be employed to modify the weights so that this error is reduced. When the error has been minimised (or, more commonly, a given number of iterations or *epochs* have been done), the network is said to be trained. An acceptable RMS target error is normally allowed so that, if the convergence error is less than or equal to this target error, the neural network is said to have found the 'correct' result. It is also possible to train a network in an unsupervised way. Here, the network adjusts its weights in response to input patterns without any target answers and classifies the input into similarity classes.

The learning rule is at the heart of a neural network as it determines how the connection weights are adjusted as the neural network learns. The error in output node j in the n^{th} training example is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{11.4}$$

where d_j is the target value and y_j the value produced by the neuron. We then make corrections to the weights of the nodes based on those corrections that minimize the error in the entire output, given by

$$\mathcal{E}(n) = \sum_j e_j^2(n). \tag{11.5}$$

Using gradient descent, the change in each weight is

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n) \tag{11.6}$$

where y_i is the output of the previous neuron and η is the *learning rate*, which has to be selected to ensure that the weights converge to a response fast enough without producing oscillations.

For an output node, this derivative can be simplified to

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n)) \tag{11.7}$$

where ϕ' is the derivative of the activation function. Calculating the change in weights to a hidden node is more difficult but it can be shown that the relevant derivative is

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial v_k(n)} w_{kj}(n) \tag{11.8}$$

which depends on the change in weights of the k^{th} nodes, the output layer. So to change hidden layer weights, one must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a *back-propagation* of the activation function; hence, this learning algorithm is known as *back-propagation*. (Note that there are quite a few refinements beyond this set of equations to improve the speed of learning.)

The training process requires many iterations and often takes considerable time. A single example of each input pattern may in principle be applied repeatedly but that tends to lead to 'over-training,' where the network recognises only that pattern. More effective training uses a large set of training data; where that is not available, it is common to present noisy, shifted or rotated versions of the images, and to vary the order of presentation. This is known as *data augmentation*. To give an idea of the numbers of images involved, the 6,000 training images in each class of MNIST is regarded as being just about enough. Networks trained in this way tend to be more effective on unseen data because they have learned the types of variations that input of the different classes contain.

The trained network may be 'run' on test data simply and with great speed: the data are applied to the input layer and the outputs of each layer, calculated using the previously-determined weights, are propagated forwards until the final outputs are found. The strength of the outputs indicates the degree of resemblance between the input and the previously-learned patterns.

11.3 MLP in software

There are quite a few pieces of software freely available that you can download and use to explore different neural networks. The one that slots most easily into the Python framework used in these notes is *Scikit-learn* [Pedregosa et al., 2011]. However if you need to use machine learning seriously in your work, you may wish to explore Google's `TensorFlow` or Berkeley's `Caffe`; TensorFlow is certainly able to spread computation over your CPUs or use the GPUs of your graphics card. It can be a bit of a faff to make this work but the speed-up achieved can be dramatic.

With `Scikit-learn` installed, a program that applies an MLP to the MNIST dataset is very similar to the SVM one presented in Chapter 10 — thanks to Scikit-learn, the only significant change is in the creation of the classifier.

⟨*train-and-test-mlp.py*⟩ ≡

```
#!/usr/bin/env python3
"Train␣up␣a␣SVM␣and␣run␣it␣on␣its␣test␣dataset."
import sys, numpy, os, struct, array, time, datetime

<<Support routines for training on MNIST>>

# Ensure we were invoked with the database name.
if len (sys.argv) != 4:
    print ("Usage:", sys.argv[0], "<database>␣<epochs>␣<transcript>",
           file=sys.stderr)
    exit (1)

# Import the machine learning technique and report its version.
import sklearn
print ("Using␣Scikit-learn␣version:", sklearn.__version__)
from sklearn.neural_network import MLPClassifier

# Read in the training and testing datasets and pull out the classes.
train_dataset = load_dataset (sys.argv[1], "train")
test_dataset = load_dataset (sys.argv[1], "test")

print_summary (sys.argv[1], train_dataset, test_dataset)

# Convert the training and test datasets into the formed needed by scikit-learn
# and do some sanity-checking.
train_images, train_labels = dataset_to_scikit (train_dataset)
test_images, test_labels = dataset_to_scikit (test_dataset)

```

```
assert len (train_images) == len (train_labels)
assert len (test_images) == len (test_labels)

# Create a classifier.
epochs = int (sys.argv[2])
classifier = MLPClassifier (hidden_layer_sizes=(50,), max_iter=epochs,
                            solver="sgd", alpha=1.0e-4, tol=1e-4,
                            learning_rate_init=0.1, verbose=False)

# Train the MLP.
start = time.time ()
try:
    classifier.fit (train_images, train_labels)
except ValueError:
    print ("Alas,␣training␣failed!", file=sys.stderr)
    exit (1)
duration = time.time() - start
print ("Training␣took␣%.1f␣seconds." % duration)

# Evaluate the trained MLP on the test data and save the results in a form
# we can analyse with FACT.
start = time.time ()
results = classifier.predict (test_images)
duration = time.time() - start
output_transcript (sys.argv[3], results, test_labels, sys.argv[1], duration)
```

You will see that there is one hidden layer of 50 neurons. If you run this for 400 epochs:

```
python3 train-and-test-mlp.py mnist 400 mlp-mnist.res
```

you should find by running FACT on the transcript (see Chapter 6) that the resulting accuracy is about 98%. Training for 400 epochs takes 21.8 seconds on the author's laptop — we shall compare this with the time taken to train a CNN below.

11.4 Convolutional neural networks

A CNN is not totally different to an MLP, more a kind of refinement of it. Firstly, it is assumed that its inputs are images, and this allows the network to be simplified somewhat. For a conventional MLP, even with the minuscule images in MNIST (only 28×28 pixels), there are 784 weights to be determined. Instead, a CNN consists of layers that can be thought of as having a 3D arrangement of neurons, so they have a height and width that correspond to the dimensions of the image and a 'depth' that corresponds to an *activation volume*. Within each layer, the neurons are connected only to a small region of the layer preceding it. CNNs are conventionally constructed from a few different types of layers:

INPUT: this receives the raw pixels of the data; for colour data, the red, green and blue (or HSV) values are normally presented to different input neurons;

CONV: a convolutional or CONV layer computes the output of neurons that are connected to local regions of in the input — in other words, it performs a convolution with coefficients that are learned from the data;

RELU: this applies an element-wise activation function, which may be as simple as `max(0,x)` to threshold at zero — clearly, this leaves the width and height of the network unchanged;

POOL: this down-samples or averages regions of its input, so that the overall width and height of the network is reduced;

FC: the final layer is usually fully-connected, just as in an MLP; it computes the class scores and hence returns the calculated class of the pattern presented at the network's input.

You will see that a CNN transforms its input image, layer by layer, from the original pixel values to the final class scores. CONV and FC layers transform the data as a function of their inputs according to the weights and biases of the neurons, which means they must be trained. Conversely, RELU and POOL layers implement a fixed function and do not require training.

At the time of writing, Scikit-learn does not support CNNs so we shall implement one using TensorFlow and a more human-friendly wrapper for it known as Keras. Even so, a program that builds a CNN and applies it to MNIST is fairly similar to the MLP one above.

⟨*train-and-test-cnn.py*⟩ ≡

```
#!/usr/bin/env python3
"Train␣up␣a␣SVM␣and␣run␣it␣on␣its␣test␣dataset."
import sys, numpy, os, struct, array, time, datetime

<<Support routines for training on MNIST>>

# Ensure we were invoked with the database name.
if len (sys.argv) != 4:
    print ("Usage:", sys.argv[0], "<database>␣<epochs>␣<transcript>",
           file=sys.stderr)
    exit (1)

# Import the machine learning technique and report its version.
import keras, tensorflow
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

# Read in the training and testing datasets and pull out the classes.
train_dataset = load_dataset (sys.argv[1], "train")
test_dataset = load_dataset (sys.argv[1], "test")

print_summary (sys.argv[1], train_dataset, test_dataset)

# Convert the training and test datasets into the formed needed first by
# Scikit-lean and from that to Keras.
train_images, train_labels = dataset_to_scikit (train_dataset)
test_images, test_labels = dataset_to_scikit (test_dataset)

ny = 28
nx = 28
nc = 1
nclasses = 10
```

```
train_images.resize (60000,28,28,1)
train_labels = keras.utils.to_categorical (train_labels, nclasses)
test_images.resize (10000,28,28,1)

# Create the CNN.
classifier = Sequential ()
classifier.add (Conv2D (32, kernel_size=(3, 3), activation="relu",
                        input_shape=(ny, nx, nc)))
classifier.add (Conv2D (64, (3, 3), activation="relu"))
classifier.add (MaxPooling2D (pool_size=(2, 2)))
classifier.add (Dropout (0.25))
classifier.add (Flatten ())
classifier.add (Dense (128, activation="relu"))
classifier.add (Dropout (0.5))
classifier.add (Dense (nclasses, activation="softmax"))

# Train the CNN.
epochs = int (sys.argv[2])
start = time.time ()
try:
    classifier.compile (loss=keras.losses.categorical_crossentropy,
                        optimizer=keras.optimizers.Adadelta(),
                        metrics=["accuracy"])
    classifier.fit (train_images, train_labels, batch_size=120, epochs=epochs,
                    verbose=0)
except ValueError:
    print ("Alas,␣training␣failed!", file=sys.stderr)
    exit (1)
duration = time.time() - start
print ("Training␣took␣%.1f␣seconds." % duration)

# Evaluate the trained CNN on the test data and save the results in a form
# we can analyse with FACT.
start = time.time ()
#results = classifier.predict_classes (test_images)
results = numpy.argmax (classifier.predict (test_images), axis=-1)
duration = time.time() - start
output_transcript (sys.argv[3], results, test_labels, sys.argv[1], duration)
```

You should be able to see what the layers of the CNN are from the way `classifier` is constructed. Running this program with

```
python3 train-and-test-cnn.py mnist 100 cnn-mnist.res
```

takes 50 minutes to train without GPU support, with the trained network achieving about 95% accuracy on the MNIST test set.

Popular CNN architectures

LeNet. The first CNN was *LeNet*, developed by Yann LeCun, the inventor of the CNN [LeCun et al., 1998] (Figure 11.4). This was used to read handwritten digits, such as the numbers on cheques or the zip-code on envelopes in the USA; indeed, LeCun had a large rôle in popularising MNIST as a standard task for comparing machine learning algorithms. It has the structure

```
INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => RELU => FC
```

which is a pair of `CONV => RELU => POOL` triplets followed by a pair of fully-connected layers with thresholding.

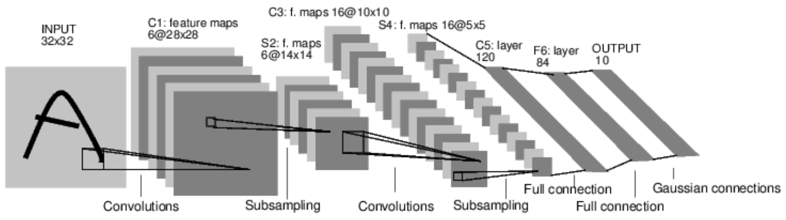


Figure 11.4: LeNet (image from http://eblearn.sourceforge.net/beginner_tutorial2_train.html)

AlexNet. The paper that popularised CNNs in computer vision was AlexNet [Krizhevsky et al., 2012], which presented a solution to the ImageNet Large Scale Visual Recognition Challenge in 2012 and won by a substantial margin. AlexNet is similar to LeNet but features successive CONV layers; previously, it was common to have a single CONV layer followed by RELU and POOL layers. The architecture is shown in Figure 11.5, accompanied by the types of feature that the different layers are purported to help identify.

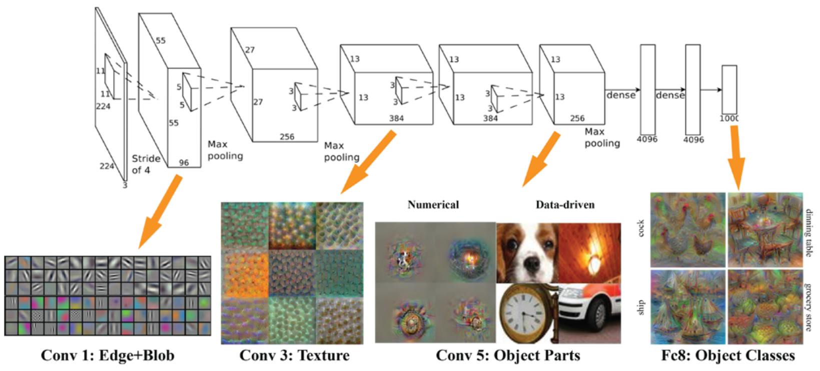


Figure 11.5: AlexNet (image from http://www.cc.gatech.edu/~hays/compvision/proj6/)

GoogLeNet. The ILSVRC 2014 winner was a CNN from Google [Szegedy et al., 2015]. Its main contribution was an *inception module* that dramatically reduced the number of parameters in the network. Additionally, GoogLeNet uses average pooling instead of FC layers at the output stage of the network, eliminating many parameters that seem to have little effect. There are also several followup versions to GoogLeNet, most recently Inception-v4.

VGGNet. The runner-up in ILSVRC 2014 was the one that became known as VGGNet [Simonyan and Zisserman, 2015]. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final network contained 16 CONV/FC layers and featured an homogeneous architecture, with only 3×3 convolutions and 2×2 pooling stages — but at the cost of being expensive to evaluate, using more memory and having more parameters.

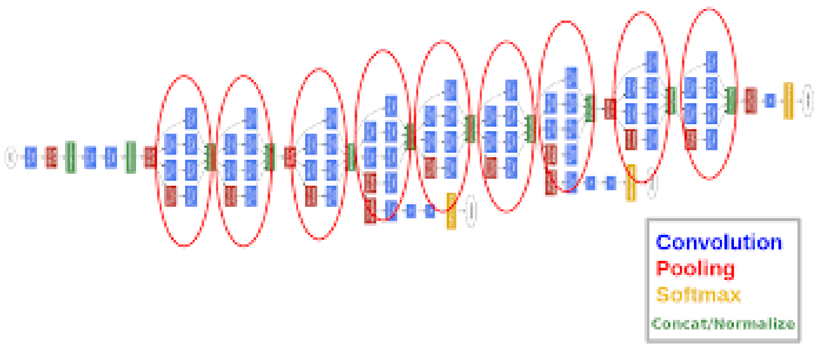


Figure 11.6: GoogLeNet (image from `https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/googlenet.html`)

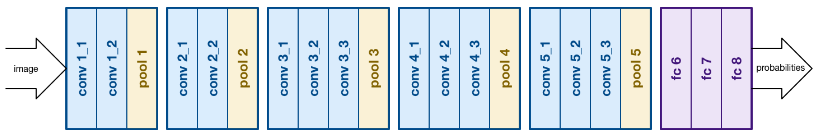


Figure 11.7: VGGNet (image from `http://matthijshollemans.com/2016/08/30/vggnet-convolutional-neural-network-iphone/`)

ResNet. The *Residual Network* of [He et al., 2016] was the winner of ILSVRC 2015. Its designers observed that adding more layers to a neural network tends to increase its training error, and hypothesized that it might be effective to encourage the network to learn the residual error instead of the original mapping. As well as having a ludicrous number of layers, some 200, it features special 'skip' connections and a heavy use of batch normalization in training. The architecture also omits fully connected layers at the end of the network. Nevertheless, due to their superior performance, ResNets are currently regarded as being pretty much the state of the art in performance on vision tasks.

YOLO. YOLO is an acronym for "you look only once." It is a popular, real-time real-time object detection system which performs pretty well on benchmark tests against other forms of machine learning, including other deep networks [Redmon and Farhadi, 2018]. The basic approach is to apply a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities. Considering the entire image allows the network to consider the context around potentially interesting regions, and it makes predictions in a single network evaluation rather than the per-region evaluations used on most other networks.

Another reason for the popularity of YOLO is that it is easy to install, coming down to a few commands entered into a Unix terminal *and* produces decent visualizations of its results with little effort. This comes with a default pre-trained network but its authors make it fairly straightforward for users to train their own. There is much to commend YOLO, so if you have an opportunity to investigate it, do create a Python virtual machine and try it out within it.

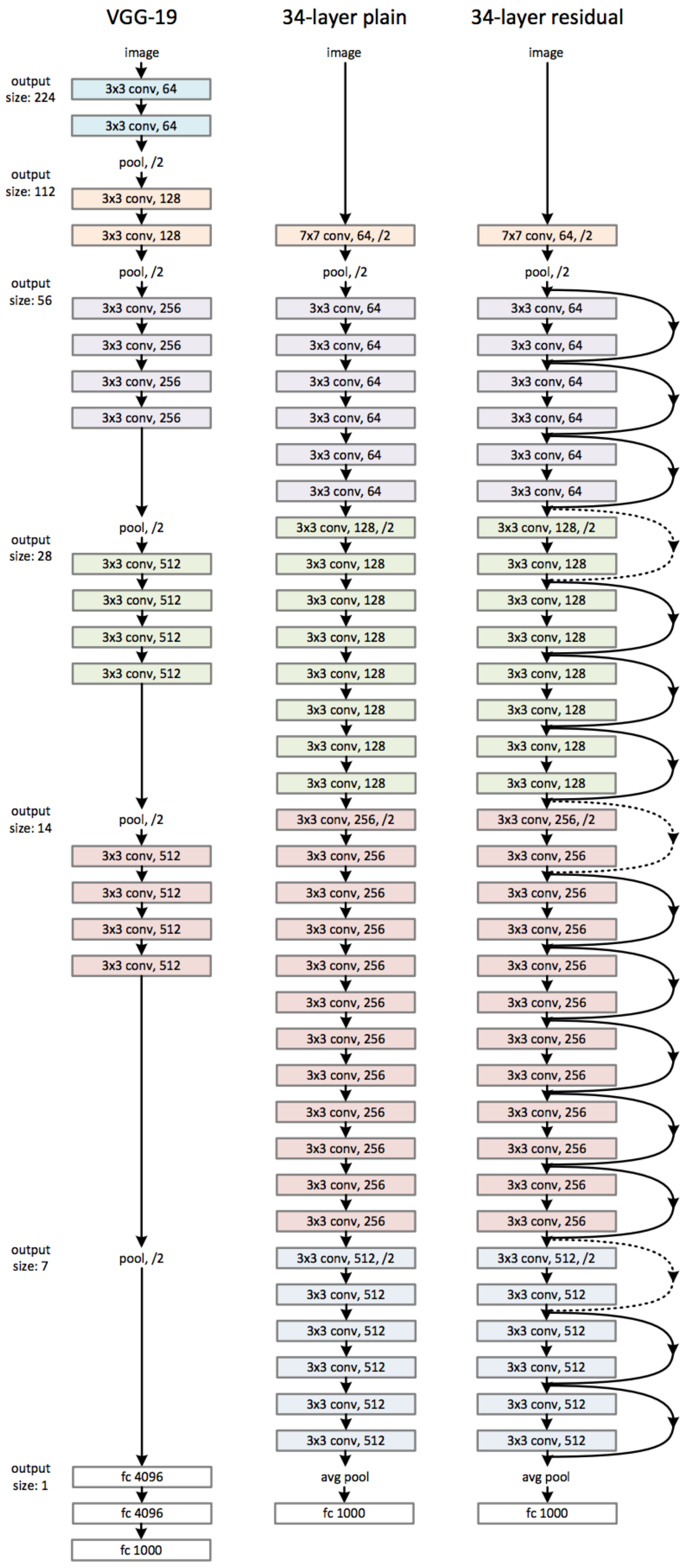


Figure 11.8: ResNet (image from `https://arxiv.org/pdf/1512.03385v1.pdf`)

11.5 Transfer learning.

There is a trend towards *transfer learning*, “tapping off” the results of some CNN stages — for example Figure 11.5 shows a stage that appears to distinguish different textures — and using these as generic feature detectors. The outputs from these stages are then used as input to (say) a SVM or MLP, which is trained up on a specific problem. People have reported some successes using this approach. The problem with it, in the author’s opinion, is that the stages are not deliberately trained to exhibit these types of behaviour; researchers are avoiding having to find a really good solution to the individual stages by hoping that the neural network will do it for them.

You will see that all these architectures are variations on a theme. The author’s view of this is that the research community is still exploring what these types of architectures do and how they do it. When there is a little more understanding, I fully expect that someone will present a series of stages that are better tailored to performing texture analysis, feature extraction *etc*. and are able to be trained in isolation — as things stand, very few research groups have the resources to train 200-layer CNNs on the million of so images in the ImageNet dataset.

11.6 Discussion

Hardly a week seems to go by without a TV news programme announcing that some problem or other has been “solved” using “deep learning.” Many of these problems are in the vision domain and the machine learning technique used is usually a CNN. They genuinely do represent the state of the art in machine learning and computer vision.

However, you now have enough knowledge and insight to ask some pertinent questions about these news articles. Firstly, what is meant by “solved?” Is the result human-competitive — and to what extent has the system been tuned to perform well on the benchmark? Is there evidence that the system performs well on unseen data? You should be able to come up with several other questions.

Research into CNNs proceeds at a furious pace but there are three significant intellectual shortcomings. The first is that there is no real methodology for designing them: designs are a combination of expediency (due to the image size) and experience. A theory, or at least set of guidelines, needs to emerge for them to become more accepted to those researchers who care how things work.

Visually identical inputs to a deep network can yield totally different classes as output, and both with high confidence values, as shown in Figure 11.9. As discussed in Goodfellow et al. [2015], this example adds an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, changing GoogLeNet’s classification of the image. Their ϵ of 0.007 corresponds to the magnitude of the smallest bit of an 8-bit image encoding after GoogLeNet’s conversion to real numbers.

This problem has given rise to an approach called *Generative Adversarial*

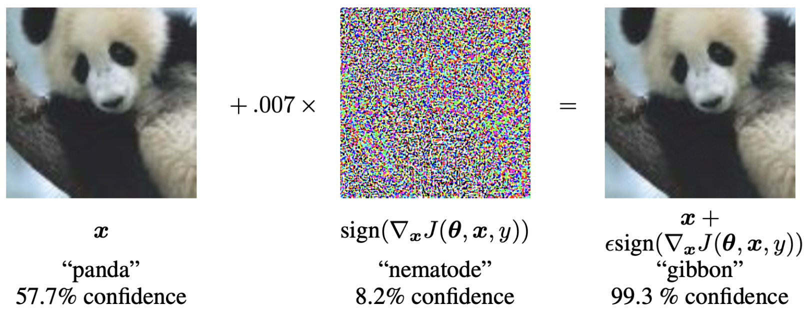

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 11.9: Problematic classification with deep networks (reproduced from Goodfellow et al. [2015]

Networks (GANs), in which two neural networks are pitted against each other to generate new, synthetic images that can pass for real ones; you may have heard of "deepfakes" — though they can be used to mimic any distribution of data. A *generator* network produces input to a *discriminator* one; both are trained concurrently, with the generator trying to produce examples that the discriminator cannot distinguish from real images, while the discriminator tries to spot the generator's fakes. The process runs until the discriminator can no longer distinguish real images from fakes. The discriminator resulting from this procedure tends to be much better at distinguishing classes of input. Knowing what you know now, you might like to reflect on comments earlier in these notes about the inclusion of negative examples when training, and on what you already know about augmenting training data — there is much to commend in this approach.

As always seems to be the case, other types of machine learning have already explored this notion; for example, in genetic programming it is known as *co-evolution*. In both disciplines, the act of generating more difficult cases to train on improves the effectiveness of the trained system to classify difficult inputs correctly, so there seems to be much to commend it.

Ultimately, the most difficult problem is that CNNs in particular and neural networks in general remain "black boxes"[1] techniques, giving the user no insights into what is happening within them. If all you need to do is solve a particular problem, this isn't really a problem; but if you need to understand how a system works — which can be the case for medical diagnosis, for example — a black box is not necessarily good enough. The author regards these deep networks as *excellent engineering but poor science*.

[1] Meaning you cannot see inside them.

An unexpected consequence of this opaque nature of deep networks is the rise of *explainable AI*, in which a human is able to interpret and explain how the trained system operates in some detail — and you have already seen that GP is an explainable AI technique.

If CNNs represent the state of the art, does this mean that computer vision has been solved? Not in this author's view. A single CNN architecture does not span a range of problems and a human always seems to be necessary to prepare the data or decide what is to be used for transfer learning. If the aim is to mimic the generality of human vision, learning to solve new visual tasks simply by looking at images — which is what the author researches — this is not the solution.

Epilogue

The material presented here has given you an overview of computer vision but not really much depth. There are many techniques that we have not considered, either because they do not fall neatly into the lecture series or because they are too complicated to present in the time available; and I have been economical with the mathematical detail — my aim has been to give you an understanding of how and why techniques work. Of course, there is an awful lot of exciting stuff that can be done using only the computer vision techniques we have considered. In particular, the machine learning approaches outlined in the last couple of chapters hold much promise — but this is an area that is evolving rapidly, so expect much of the detail to change.

Whenever you look at the world around you, think of the cues you are using to work out an object's shape, or distance, or texture. Not until researchers have worked out how to emulate these processes and assimilate them in some system does computer vision stand any real chance of competing with the human visual system. Trying to get there is a fascinating problem.

Colophon One of the sadnesses of modern publishing is that authors give very little information regarding how their documents appear or were prepared; it is as though they take little pride in their work. The author's research area is image processing and computer vision, and one aspect of image processing is controlling just how the little dots of toner appear on a printed page. Some care has therefore been expended to make these lecture notes easy to read online and in printed form, and the author would welcome feedback where you do not find this to be the case.

This document was prepared using `Emacs`, the One True Editor, and typeset using pdfLaTeX. The basic document style is `tufte-book`, which supports the large number of marginal figures and notes that you see. The body font is Bitstream Charter and the sans serif font Droid Sans. The `microtype` package is used to enhance TeX's already good letter- and word-spacing algorithms. British hyphenation is used, thanks to the hard work of Dominik Wujastyk and Graham Toal. The graphs and many of the figures were prepared using a combination of `gnuplot` and TikZ; I have unashamedly stolen others from the Web (with acknowledgements, a good lesson for when you write your project final report). Hyperlinks within the document were generated using the excellent `hyperref` package due to the late Sebastian Rahtz and Heiko Oberdiek. Sebastian was instrumental in encouraging the development of pdfLaTeX and CTAN; I am proud to have known him.

Bibliography

I. Aleksander, W. V Thomas, and P. A. Bowden. WISARD: a radical step forward in image recognition. *Sensor Review*, 4(3):120–124, 1984.

Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, third edition, 2014.

Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded up robust features. *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

G. E. Bostanci, N. Kanwal, and A. F. Clark. Spatial statistics for image features for performance comparison. *IEEE Transactions on Image Processing*, 23(1):153–162, January 2014.

Gary Bradski and Adrian Kaehler. *Learning OpenCV — Computer Vision with the OpenCV Library*. O'Reilly, 2008.

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

Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary robust independent elementary features. In *Proceedings of the 11th European Conference on Computer Vision*, pages 778–792, 2010.

John F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. OpenPose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2: 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

R. W. Connors and C. A. Harlow. A theoretical comparison of texture algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(3):204–222, May 1980.

Tom N. Cornsweet. *Visual Perception*. Academic Press, 1970.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20 (3):273–297, 1995.

A. Criminisi, I. Reid, and A. Zisserman. A plane measuring device. In *Proceedings of the British Machine Vision Conference*, pages 699–708, September 1997.

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, June 2005.

E. Roy Davies. *Computer Vision: Principles, Algorithms, Applications, Learning*. Academic Press, 5^{th} edition edition, 2017.

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

Ronald A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, September 1936.

James D. Foley, Andries van Dam, Steve K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition edition, 1990.

W. Förstner, T. Dickscheid, and F. Schindler. Detecting interpretable and accurate scale-invariant keypoints. In *Proceedings of the 12th International Conference on Computer Vision*, pages 2256–2263, 2009.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, January 1989.

Rafael C. Gonzalez and Richard E. Woods. *Digital Image Procecssing*. Addison-Wesley, 1992.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the International Conference on Learning Representations*, May 2015.

R. M. Haralick, K. Shanmugam, and I. H. Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man and Cybernetics*, pages 610–621, 1973.

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

Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition edition, 2003.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*. 2016.

Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

N. Kanwal, G. E. Bostanci, and A. F. Clark. Matching corners using the informative arc. *IET Proceedings on Computer Vision*, 8(3):245–253, June 2014.

Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2): 97–111, May 1984.

John R. Koza. *Genetic Programming*. MIT Press, 1992.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

K. L. Laws. Texture energy measures. In *Proceedings of the Image Understanding Workshop*, pages 47–51, 1979.

K. L. Laws. Textured image segmentation. Technical Report 940, University of Southern California, 1980a.

K. L. Laws. Rapid texture identification. *Proceedings of the SPIE*, 238: 376–381, 1980b.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf.

H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

T. Olaja, M. Pietikäinen, and D. Harwood. Performance evaluation of texture measures with classification based on kullback discrimination of distributions. In *Proceedings of the International Conference on Pattern Recognition*, pages 582–585, 1994.

Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, 1979.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, and David Cournapeau. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Rosalind W. Picard. *Affective Computing*. MIT Press, 2000.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008. ISBN 9781409200734.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

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

Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An efficient alternative to SIFT or SURF. In *Proceedings of the 2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

Scholarpedia. Scholarpedia article on eigenfaces, December 2009. `http://www.scholarpedia.org/article/Eigenfaces`.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceeedings of the International Conference on Learning Representations*, 2015.

A. R. Smith. Color gamut transform pairs. In *Proceedings of SIGGRAPH 1978*, pages 12–19, 1978.

Stephen M. Smith and J. Michael Brady. SUSAN—a new approach to low level image processing. *Int. J. Comput. Vision*, 23:45–78, May 1997.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, 2015. URL `http://arxiv.org/abs/1409.4842`.

Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010. Available online at `http://szeliski.org/Book/`.

Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.