Introduction to computer vision

Adrian F. Clark <alien@essex.ac.uk>
CSEE, University of Essex

What is an image?

```
       0    1    2    3
    +---+---+---+---+---
0   |   |   |   |   |
    +---+---+---+---+---
1   |   |   |   |   |
    +---+---+---+---+---
2   |   |   |   |   |
    +---+---+---+---+---
    |   |   |   |   |
```

```
Note that subscripts
normally increase
from the top-left
corner of the image
```

We normally work with images that are rectangularly sampled, like the one shown above but that is really a convenience for representing them in a computer

The light-sensitive cells (rods and cones) in the human eye are more like hexagonally packed, as in a honeycomb

The human vision system

Chapter 2 of the lecture notes goes into what we know about how the human vision system works — after all, it is the only working example of a complete vision system we have

I encourage you to read through it but **it is not examinable**

What is a pixel?

"A number in the range 0–255"... but why?

- humans can distinguish < 128 grey levels
- it fits into an unsigned byte

Decent cameras record > 8 bits (*e.g.*, my DSLR as a 16-bit sensor)

This is only for monochrome images — we shall consider colour images a little later

Image file formats

JPEG: best avoided because they use lossy compression, and the places where data are discarded are mostly around the edges of features. . . precisely the places most interesting in computer vision

PNG: a good choice

BMP: Windows only

GIF: just don't go there

TIFF: a "write-only" format as it's difficult to read all the different versions of the format

FITS: astronomers only

[there are many others too, of course]

Video file formats

MPEG-4

AVI

MOV

Almost all video formats involve compression, so any that you can read are fine

OpenCV works well with MPEG-4 and AVI

Formats such as H.264 are intended for video-conferencing; they are also fine but tend to have larger file sizes

Maths and code

There is a simple mapping between the maths you see in the lecture notes and the corresponding code; for example

$$S = \sum_{i=1}^{N} x_i$$

corresponds to

```
s = 0
for i in range (0, N):
   s = s + x[i]
```

One loop or three?

You'll often see a single sum like the one on the previous slide used as a shorthand; it really means you need to loop over all the pixels of an image:

$$S = \sum_{y=1}^{N_y} \sum_{x=1}^{N_x} \sum_{c=1}^{N_c} P_{yxc}$$

```
s = 0
for y in range (0, ny):
   for x in range (0, nx):
      for c in range (0, nc):
         s = s + P[y,x,c]
```

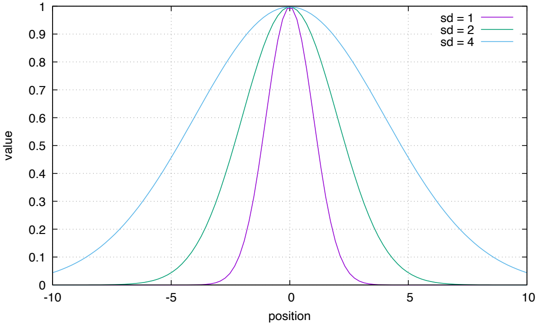
Calculating the mean

$$\bar{P} = \frac{1}{N}\sum_{i=1}^{N} P_i \rightarrow \frac{1}{N_y N_x N_c}\sum_y \sum_x \sum_c P_{yxc}$$

```
def mean (im):
    "Return the mean of image im"
    ny, nx, nc = im.shape
    sum = 0
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                sum = sum + im[y,x,c]
    return sum / ny / nx / nc
    # same as "sum / (ny * nx * nc)"
```

The standard deviation

This is the spread of a curve



Curves with different standard deviations

Calculating the standard deviation

$$\sigma^2 = \frac{1}{N} \sum_i \left(P_i - \bar{P} \right)^2$$

σ^2 is the *variance* and σ the *standard deviation*

```
def sd_slow (im):
   "Return the s.d. of image im"
   ny, nx, nc = im.shape
   sum = 0
   ave = mean (im)
   for y in range (0, ny):
      for x in range (0, nx):
         for c in range (0, nc):
         v = im[y,x,c] - ave
         sum = sum + v * v
   return math.sqrt (sum / ny / nx / nc)
```

A little mathematical manipulation

$$(P_i - \bar{P})^2 = (P_i - \bar{P})(P_i - \bar{P})$$

$$= P_i^2 - P_i\bar{P} - P_i\bar{P} + \bar{P}^2$$

$$= P_i^2 - 2P_i\frac{\sum P_i}{N} + \left(\frac{\sum P_i}{N}\right)^2$$

Note that $\sum P_i^2 \neq (\sum P_i)^2$. This gives us

$$\sigma^2 = \frac{1}{N}\left(\sum P_i^2 - \frac{1}{N}\left(\sum P_i\right)^2\right)$$

We can accumulate $\sum P_i^2$ and $\sum P_i$ in a single pass through the image, so we can compute the standard deviation without having to make two passes through the image as in in `sd_slow`

Minor manipulations of the maths like this can have a significant effect on the time taken to compute things from images or to operate on them

Colour representation

We normally think of images as having three "channels" of colour: red green and blue — the "RGB" colour model

This is because we are used to seeing images on monitors, where proportions of these *primary* colours *add* together

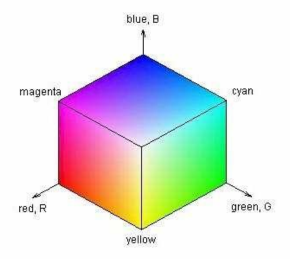
However, it is not the only way

When painting or mixing inks for printing, the colours mixed together *subtract* from what the white canvas the eye would otherwise see; the primary colours are then *cyan*, *magenta* and *yellow* — the "CMY" colour model

Black ink is cheaper than coloured ink, so printers use black for the dark component of colours and we have the "CMYK" colour model

The colour cube

Think of a cube with the red, green and blue values plotted along three axes



The colour cube

(from https://www.researchgate.net/publication/228719004_Human-centered_content-based_image_retrieval/figures?lo=1)

The subtractive primaries appear mid-way between the additive primaries

HSV — Hue, Saturation and Value

Hue

Rotate the red so that it lies at 0° (along the x-axis) and that gives us hue

Saturation is how far from the centre a colour lies, while value is how far it lies along the grey line of the colour cube

This often (though not always) makes segmenting by colour easier

Colour images and OpenCV

Pixels are stored in files as *RGBRGB*... so we expect

`im[y,x,0]` is red
`im[y,x,1]` is green
`im[y,x,2]` is blue

but this is not what happens in OpenCV! Instead we have

`im[y,x,0]` is blue
`im[y,x,1]` is green
`im[y,x,2]` is red

You can swap between colour models in OpenCV, including BGR to HSV