Computer Vision
INM460/IN3060

Lecture 2
Digital images and image processing

Dr Giacomo Tarroni
Slides credits: Giacomo Tarroni, Sepehr Jalali

Recap from the previous lecture

Human visual system:

- How the eye works
- Different theories for colour perception
- How the signal is transmitted from the eye to the visual cortex
- Different functions of the visual cortex parts
- Simple/complex cells
- Advantages & limitations of the human visual sistem
- Gestalt laws: interpreting how perception is created
- Depth and motion perception

Overview of today's lecture

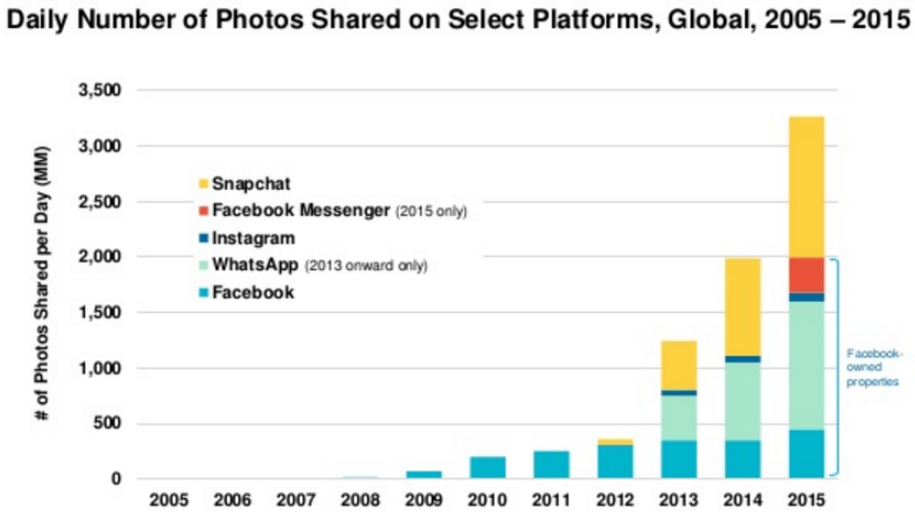
- Digital images:
  - Formation
  - Pinhole camera model
  - Digital representation
- Digital image processing:
  - Brightness and colour transformations
  - Geometric transformations
  - Filtering (next lecture)

Images and more images

- Images are produced at an extremely high rate these days

1990s

Today



Credits: http://www.kpcb.com/internet-trends

Imaging devices

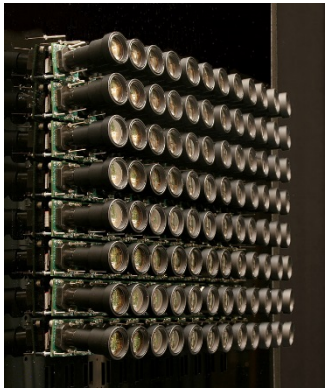
Digital cameras

Webcams

GoPro™

Nest™

360° cameras



Camera arrays

Smartphones

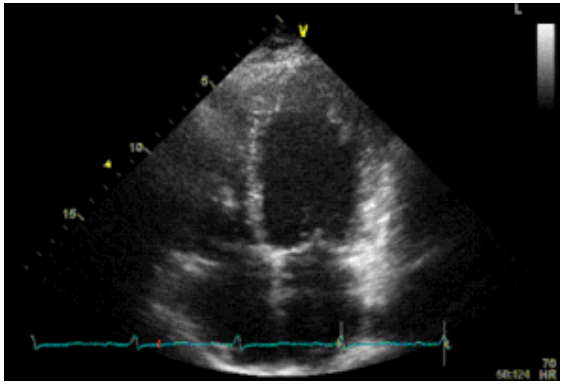
What is an image?

"An image is **multi-dimensional signal** that measures a **physical quantity**"

- **Multi-dimensional signal:**
  - 2D: Image (a function of x and y)
  - 3D:
    - Image (a function of x, y, and z, e.g. CT scan)
    - Video (a function of x, y and t)
- **Physical quantity:**
  - Typically visible light for standard photographs
  - Many other possibilities (temperature, acoustic properties, etc.)

Digital photo

Thermal image



Ultrasound video

2D Digital image

A 2D digital image is described by a **multi-dimensional function** (mapping) between **spatial coordinates** (x and y) and **image intensity** (for greyscale images) or colour channel values (for colour images)

Greyscale image

$$I(x, y): \mathbb{Z}^2 \rightarrow \mathbb{Z}$$

with

- (x, y) spatial coordinates
- I image intensity (typical range: 0÷255)

Colour image

$$I(x, y): \mathbb{Z}^2 \rightarrow \mathbb{Z}^3$$

with

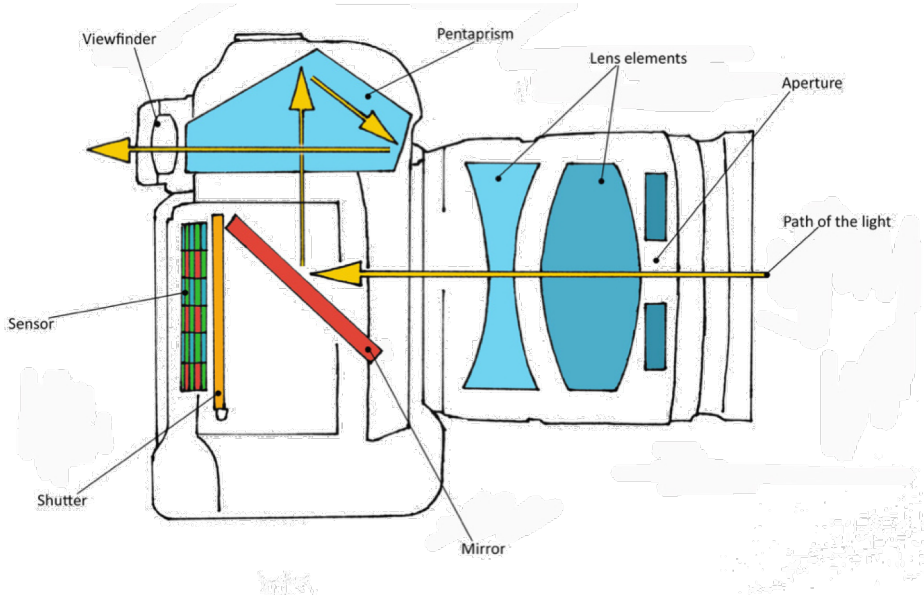
- (x, y) spatial coordinates
- $I = (I_R, I_G, I_B)$ colour channel values

Digital camera

Typical components of a DSLR:

- Aperture (to let light in)
- Lens
- Pentaprism (to direct light to viewfinder)
- Photosensitive image plane
- Housing to keep stray light out

https://www.youtube.com/watch?v=Ic0czeUJrGE



Credits: http://murrayparkphotography.weebly.com/inside-a-dslr.html

Pinhole camera model

Simplest camera model

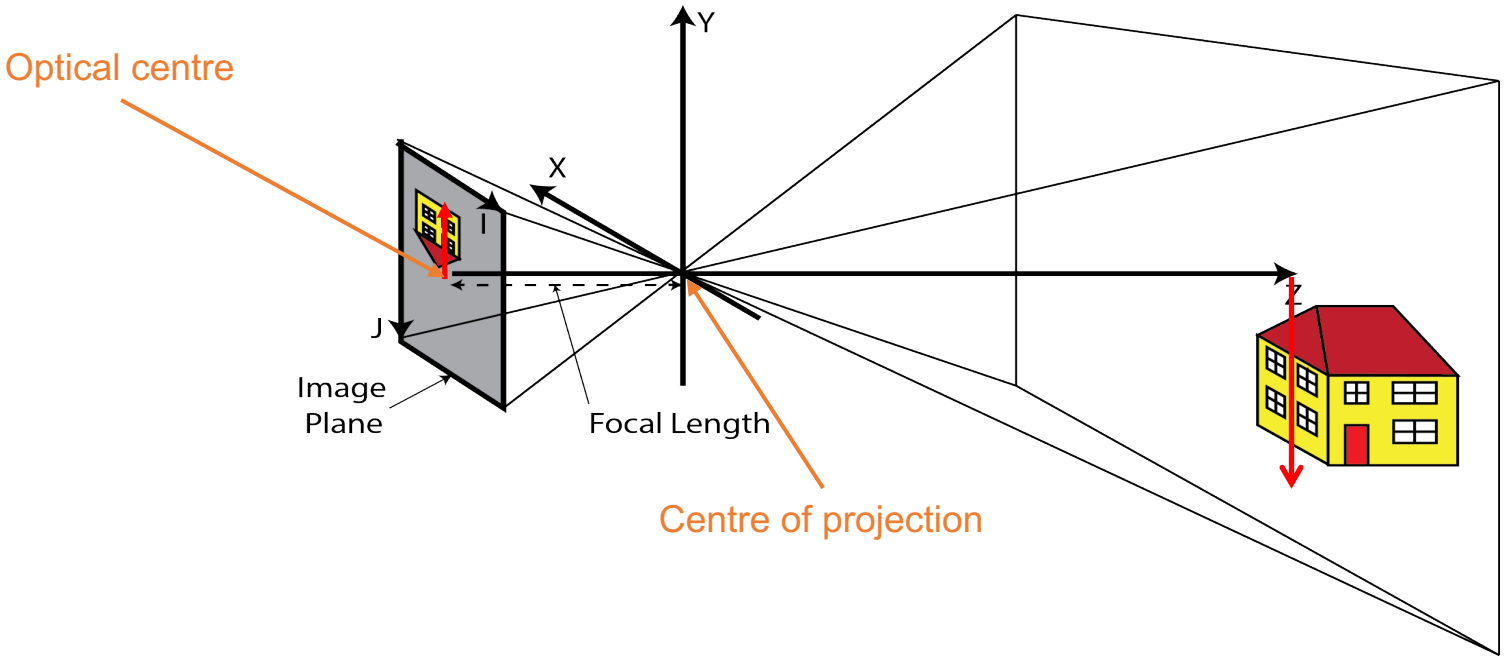
- Light passes through a single small hole, known as the **centre of projection**
- On the other side of the camera there is film that captures impinging light
- The point on the film plane directly in front of the hole is the **optical centre**
- The image is created on the film through a **perspective projection**

- Why small hole?
  This forces every point emitting light in the scene to form a small point on the film, so the image is sharp
- Why do we need lenses?
  A lens creates a much larger hole through which light can make it onto the film, meaning the film can be exposed faster

Perspective projection

This can be modelled mathematically by perspective projection:

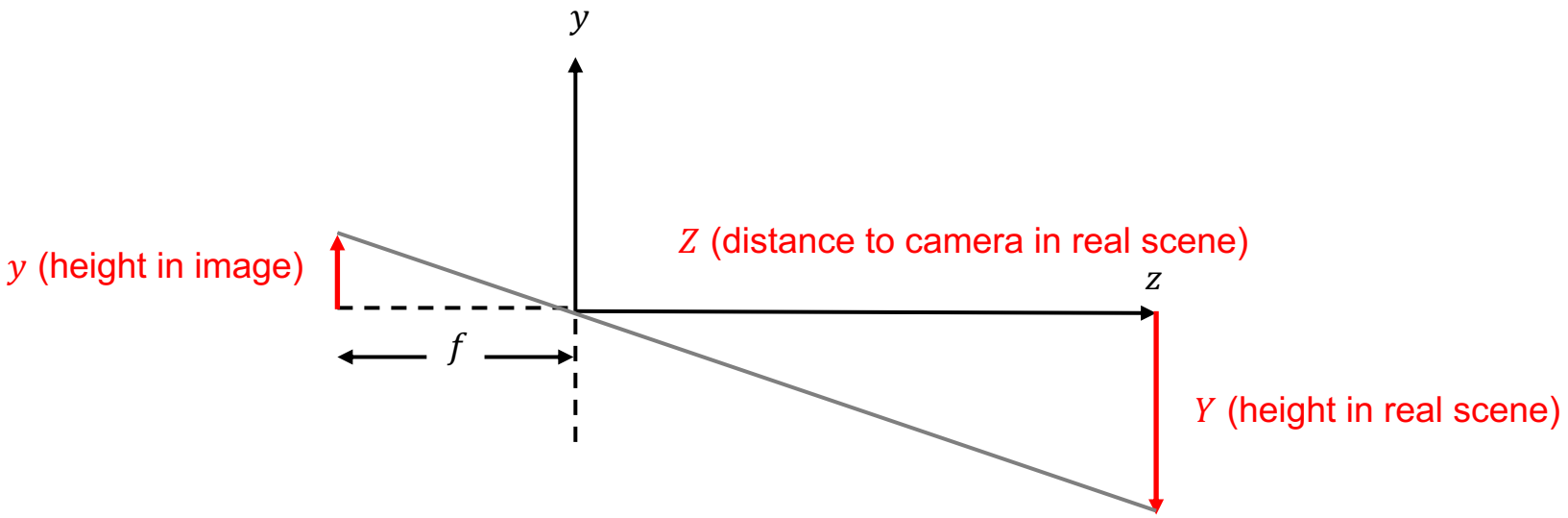
- Whilst simple, the model is reasonably realistic
- Deviations are introduced when using lenses



Perspective projection

Notation:

- X, Y, Z: coordinates of the points in the real scene
- x, y: coordinates of the points in the image plane (film or sensor)



By similar triangles:

$$\frac{y}{f} = \frac{Y}{Z} \quad \text{or} \quad y = \frac{fY}{Z}$$

As Z increases, y decreases (i.e. farther away objects appear smaller!)

Perspective projection

We can encode this projection using matrix/vector notation:

$$x = K \cdot X$$

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

With:

- f: the focal length
- c_x, c_y: the offset between optical centre and the centre of the image
- s: axis skew (equal to 0 in a true pinhole camera)
- (x, y, w) are homogeneous coordinates (basically, we will need to normalize vector so that $w = 1$)
- K is known as **intrinsic camera matrix**

Perspective projection

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} fX + sY + c_x Z \\ fY + c_y Z \\ Z \end{bmatrix} = \begin{bmatrix} \frac{fX + sY + c_x Z}{Z} \\ \frac{fY + c_y Z}{Z} \\ 1 \end{bmatrix}$$

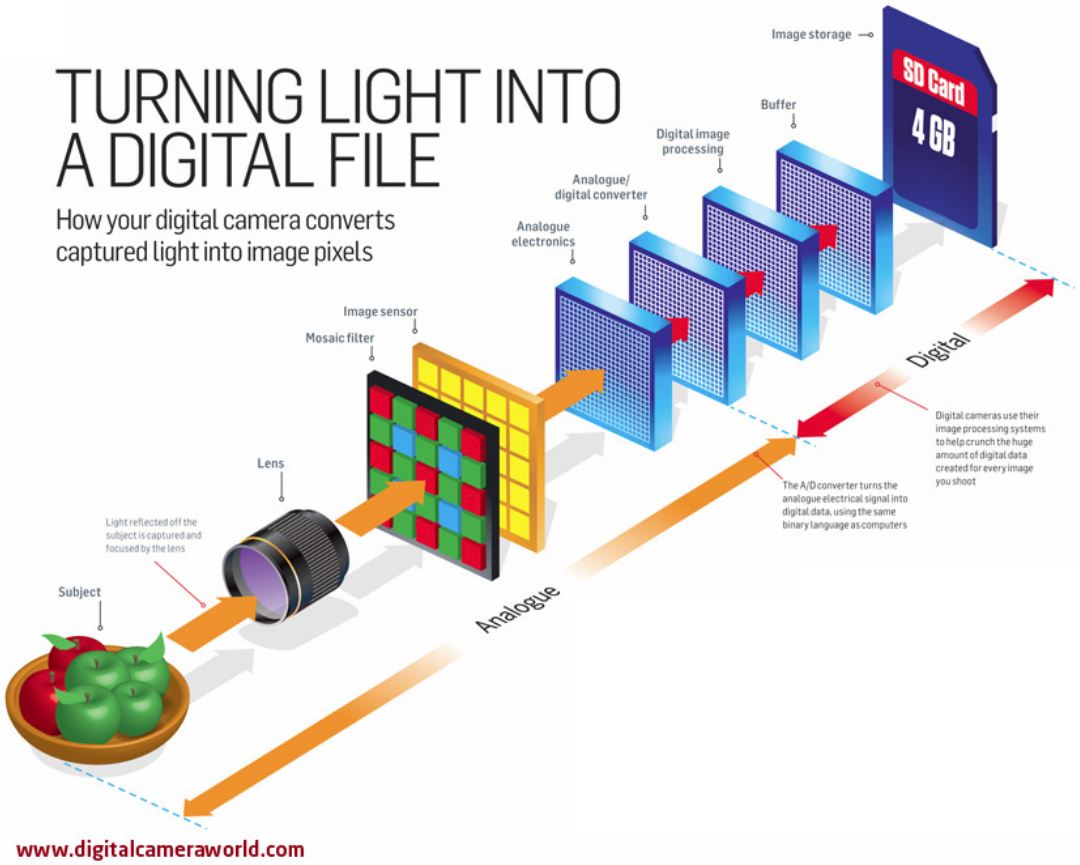
Fundamentally, perspective projection consists in both

- multiplying by f (focal length)
- dividing by Z (depth)

Further details (with interactive demo):
http://ksimek.github.io/2013/08/13/intrinsic/

From light to pixels

- The light impinging on the image sensor is a spatially continuous signal
- This signal is converted to a digital signal (i.e. a set of pixels) through **spatial sampling** and **quantisation**



Spatial sampling

- Sampling converts a continuous signal into a discrete one
- The light impinging the image plane is a continuous signal in x and y
- In a digital camera, this signal is sampled on a regular 2D grid
- One pixel per grid cell

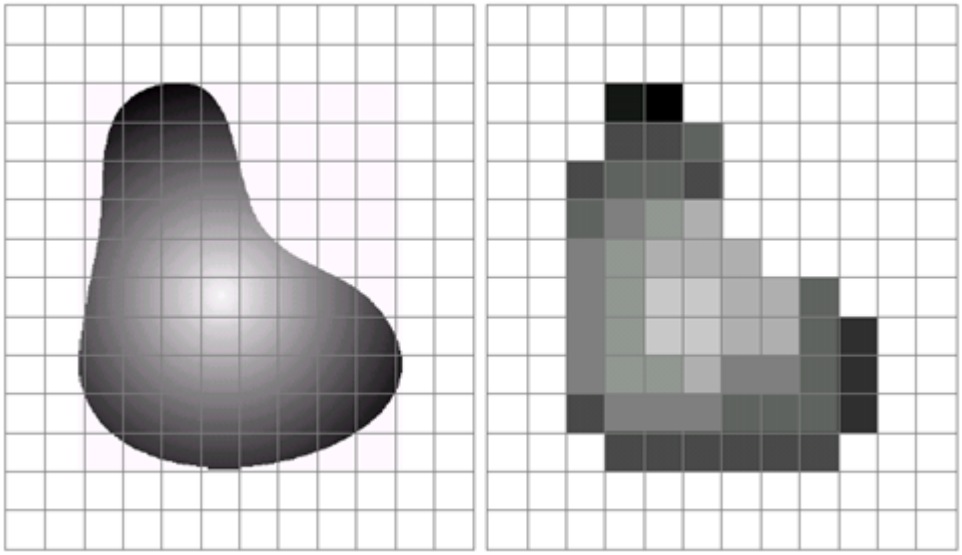
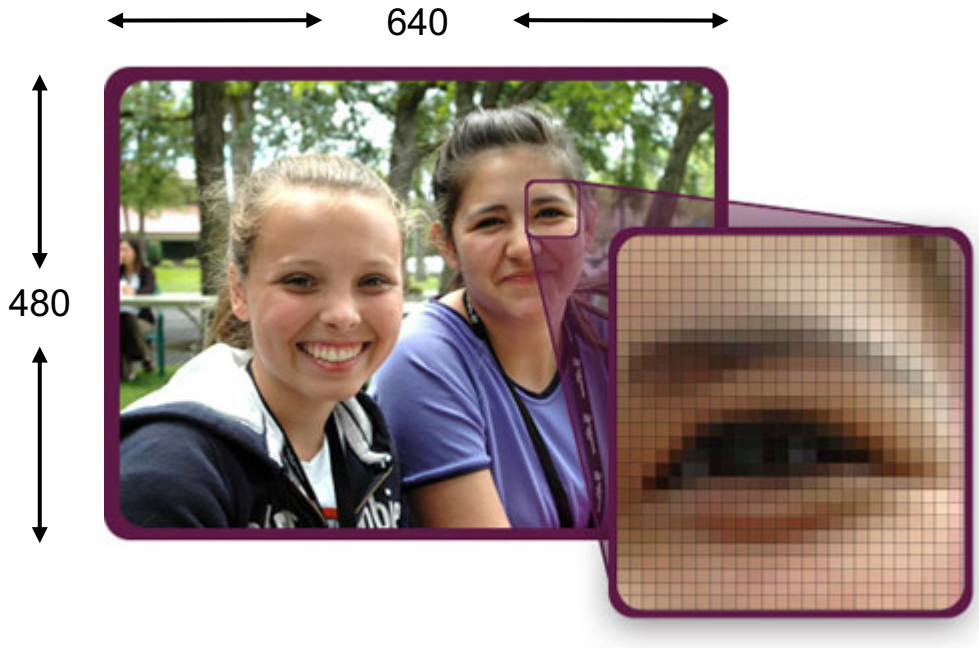
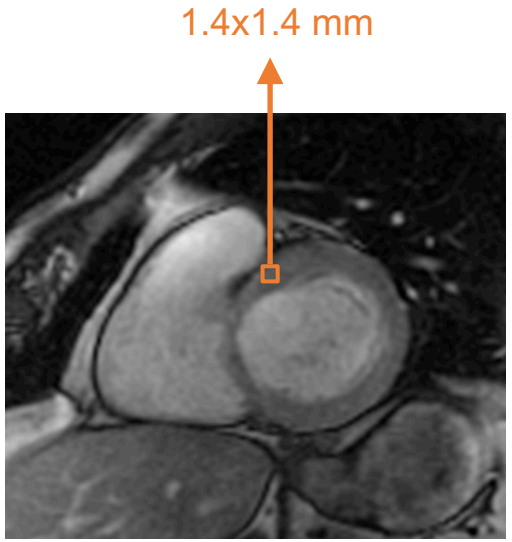


Image resolution

- Image resolution measures the detail an image holds
- It can be described in many different ways:
  - **Pixel resolution**: image dimensions in pixels (e.g. 640x480, 0.3 MP)
  - **Pixel density**: number of pixels of sensor/screen per unit area (e.g. 1 MP/cm^2)
  - **Spatial resolution**: size of each pixel (for each dimension) in length (e.g. 1.4x1.4 mm). Used in specific fields (e.g. medical imaging)

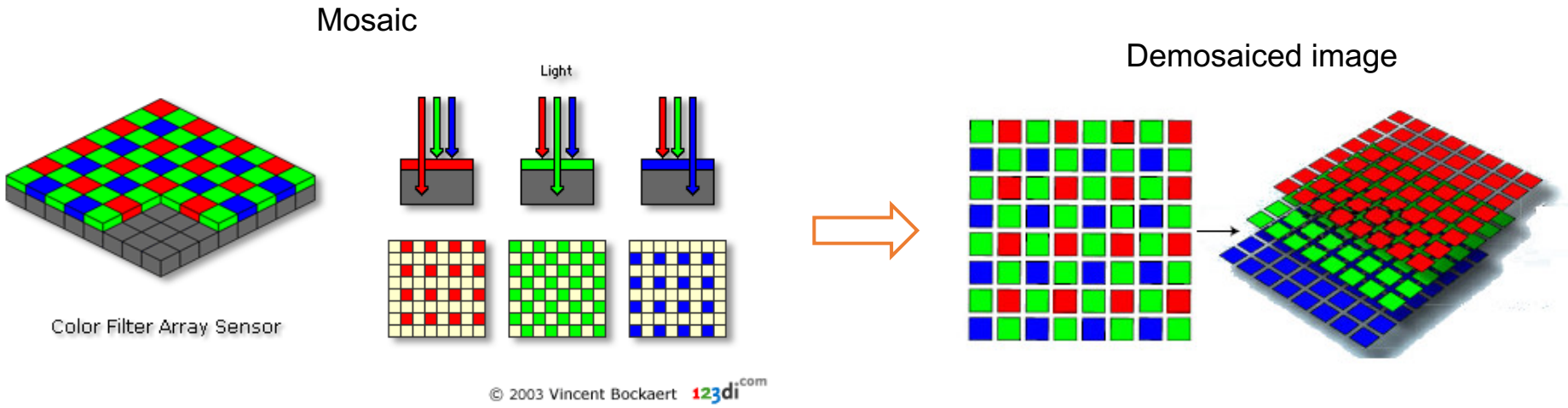




Credits: http://www.g2cs.org/media/interactives/image-compression/00.html

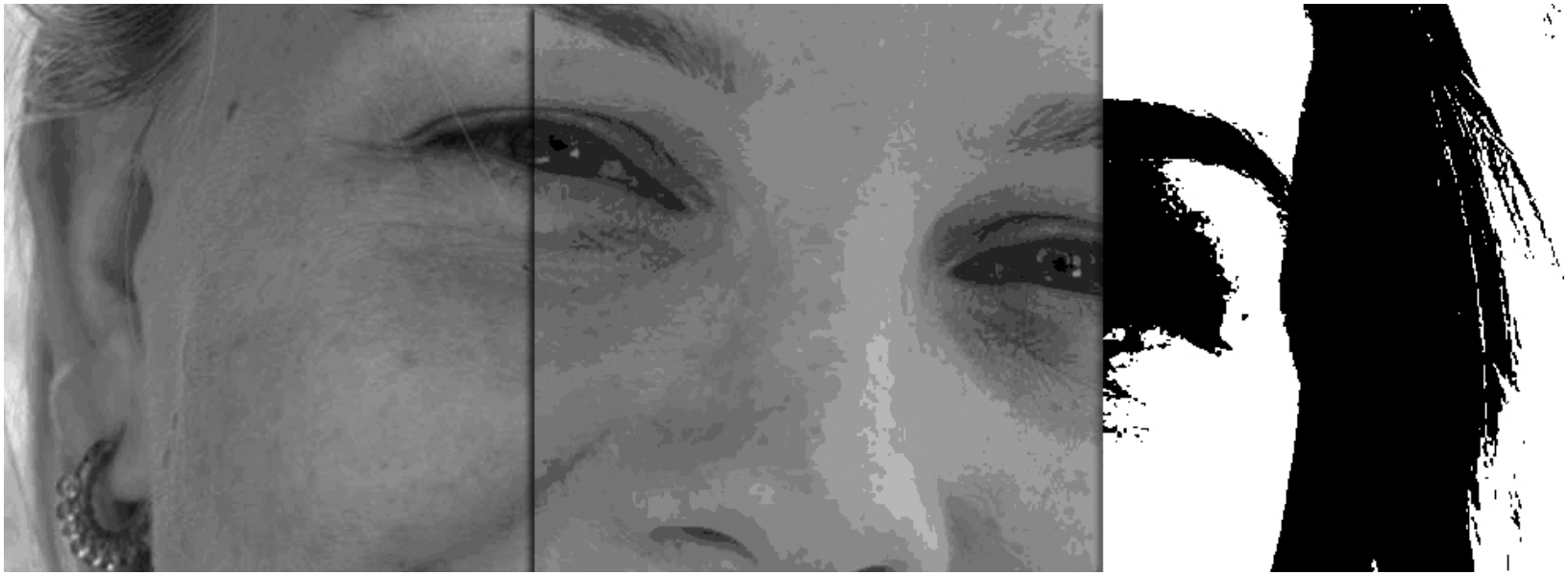
Colour filter array and demosaicing

- Light captured by digital cameras is binned into separate red, green, and blue values using a **colour filter array** (**CFA**)
- **Bayer pattern** is most common CFA design, based on GRBG: twice as many G bins than R or B because of higher sensitivity of human eye to green light
- A **demosaicing** algorithm **interpolates** the data so that each pixel has a red, green, and blue (RGB) value
  - Final pixel resolution preserved!
  - Raw file is what is collected before demosaicing
- https://www.youtube.com/watch?v=2-stCNB8jT8



Quantization

- Quantisation limits the values a pixel can have to a finite set
- In a greyscale image, if we use one byte (8 bits) per pixel, then we can represent $2^8 = 256$ different intensities (range 0÷255)



8 bits per pixel
$2^8 = 256$ shades

4 bits per pixel
$2^4 = 16$ shades

1 bit per pixel
$2^1 = 2$ shades
binary image

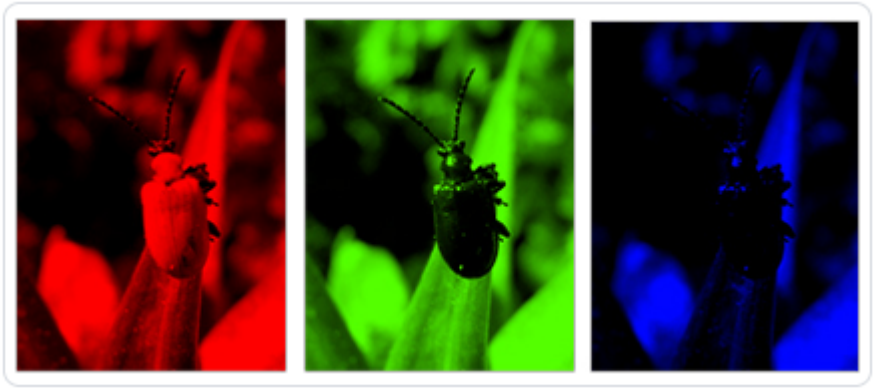
Credits: https://songbirdsings13.wordpress.com/year-12/new-media/unit-16/pmd1-unit-19/

Quantization

- In standard colour photographic images, we use 8 bits for each colour channel (red, green, blue), or 24 bits per pixel (i.e. 24-bit colour)
- That means there are 2^{24} possible colours for a pixel: ~16.7 million colours!
- Human eye can distinguish roughly ~10 million colours

Colour

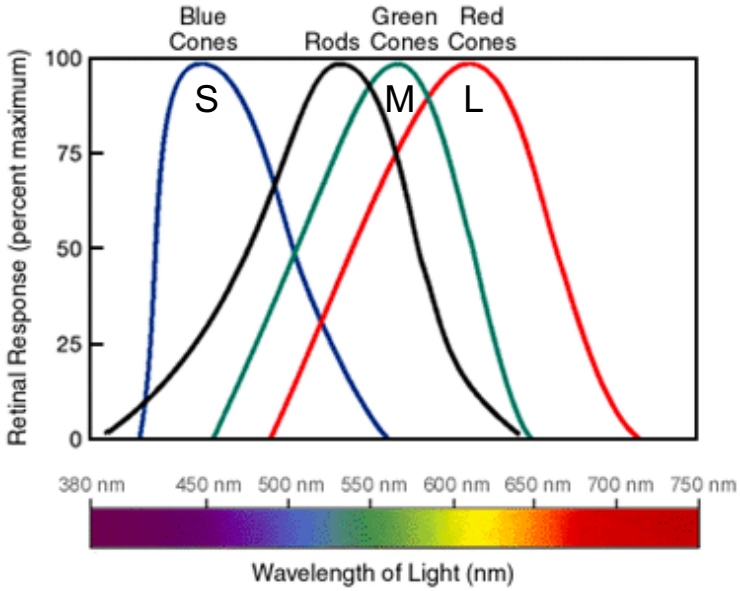
- Colour images typically have three colour channels corresponding to the amount of red, green, and blue present at each pixel
- Combining them together produces the final colour



Credits: http://digital.pho.to/

Colour theory: human retina

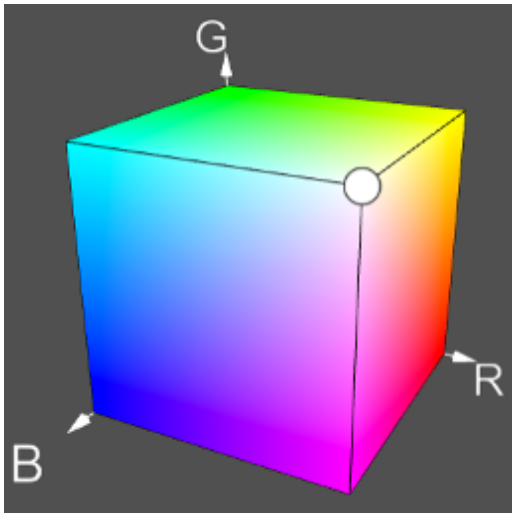
- Trichromatic theory of vision:
  - Human eyes perceive colour through stimulation of three different types cones (S, M, L) in the retina
  - Three types have peak sensitivity roughly around to red, green and blue
- RGB colour model was defined based on the physiology of human eye (Exact choice for wavelengths was practical/arbitrary)



Spectral sensitivity

RGB colour model

- First introduced in 1800s, now used in displays (including phones)
- **Additive mixing**: red, green, blue **lights** are summed to form the final colour
- RGB colour model:
  - One axis per colour channel
  - Range from 0 (no light) to 255 (full intensity) along each axis
  - A colour is a point in this space and is represented as a vector: (I_R, I_G, I_B)

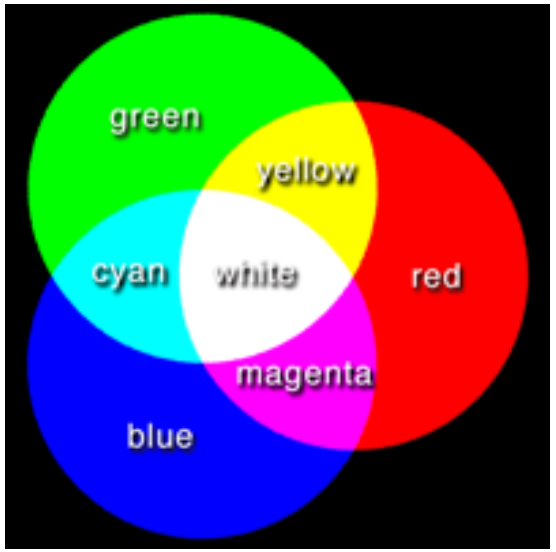


Credits: http://web.cs.uni-paderborn.de

RGB colour model

- Additive colour mixing: RGB describes what kind of light needs to be **emitted** to produce a given colour
- Light is added together to move from black to white

Colour components			Common colour name
R	**G**	**B**	
0	0	0	
0	0	255	
0	255	0	
0	255	255	
255	0	0	
255	0	255	
255	255	0	
255	255	255	

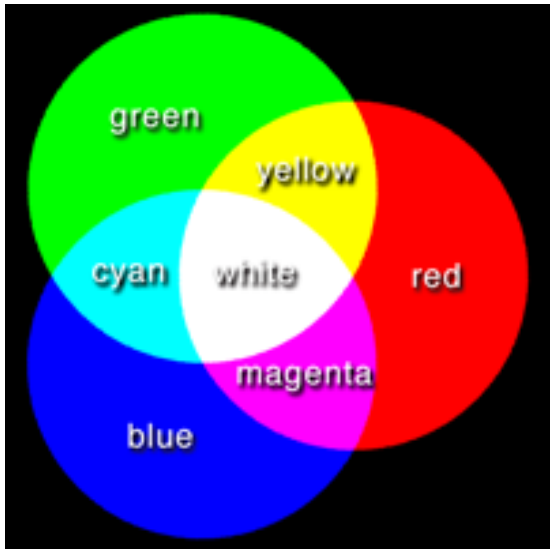


Credits: http://web.cs.uni-paderborn.de

RGB colour model

- Additive colour mixing: RGB describes what kind of light needs to be **emitted** to produce a given colour
- Light is added together to move from black to white

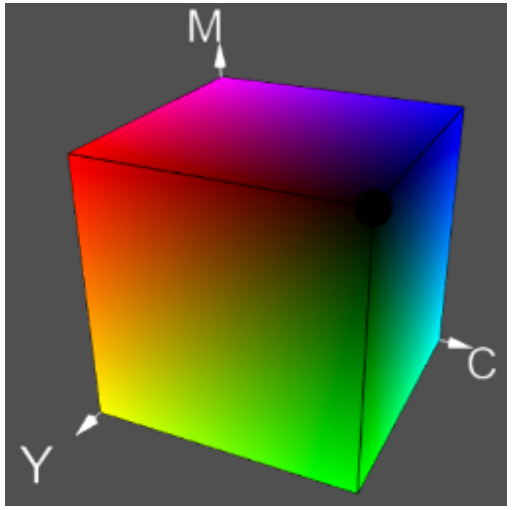
Colour components			Common colour name
R	**G**	**B**	
0	0	0	Black
0	0	255	Blue
0	255	0	Green
0	255	255	Cyan
255	0	0	Red
255	0	255	Magenta
255	255	0	Yellow
255	255	255	White

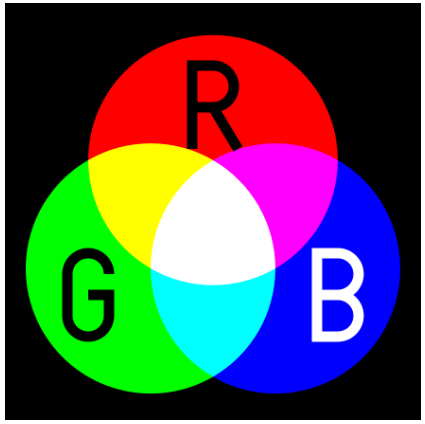


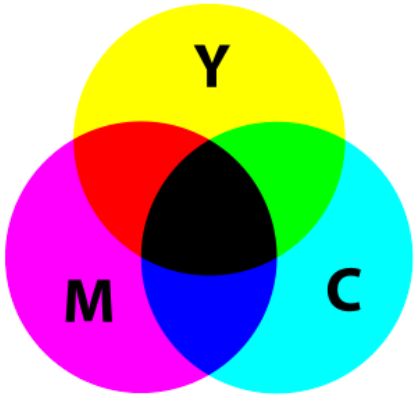
Credits: http://web.cs.uni-paderborn.de

CMY(K) colour model

- Developed for printing
- **Subtractive mixing**: cyan, magenta, yellow (and black) **inks** are summed to form the final colour. Each ink subtracts a portion of the light that would otherwise be reflected from a white background
- RGB colours are subtracted from white light (W):
  W-R = C; W-G = M; W-B=Y
- Black (K) helps to create additional colours (otherwise dark green instead of black)



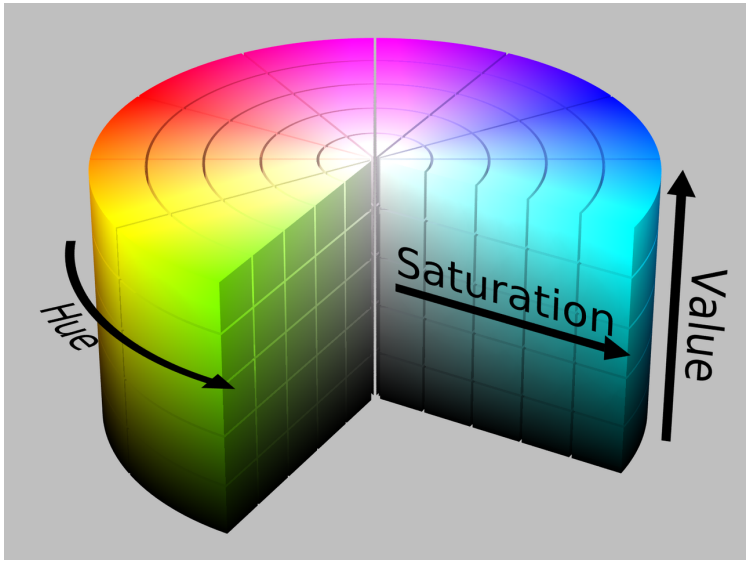




Credits: http://web.cs.uni-paderborn.de
http://wikipedia.org

HSV colour model

- More intuitive and **more perceptually relevant** than RGB
- Cylindrical coordinate system with
  - Hue (H): discernible colour based on the dominant wavelength
  - Saturation (S): vividness of the colour (zero saturation means a greyscale colour in the centre of the cylinder)
  - Value (V): brightness
- Example: a bright red colour has a red hue, full saturation, and high value

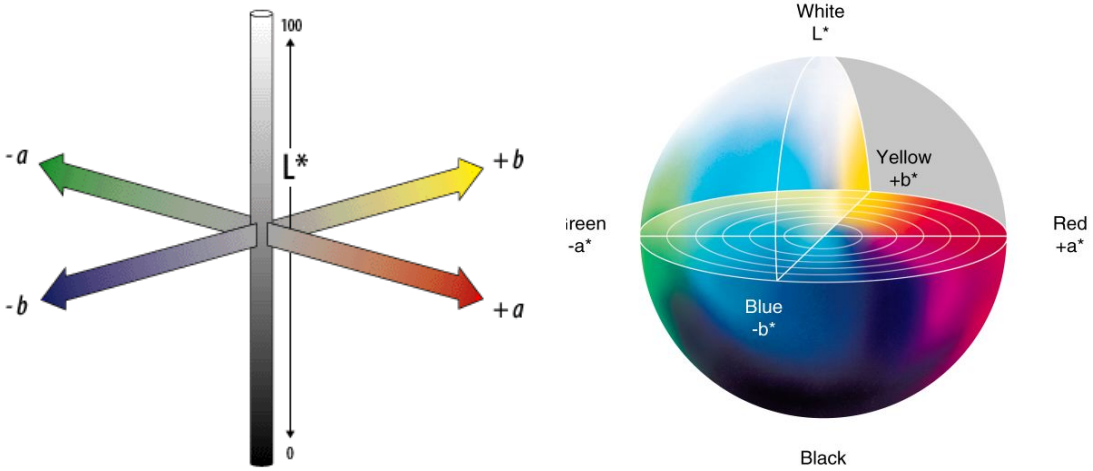


"How to halve the saturation in RGB space?"

Credits: http://wikipedia.org

LAB colour model

- RGB and HSV are **not perceptually uniform**:
  the distance between colours in the colour space does not match human visual perception of colour difference
- CIE LAB attempts to be perceptually uniform
- It is based on **opponent process theory**: observation that for humans, retinal colour stimuli are translated into distinctions between
  - blue versus yellow
  - red versus green
  - black (dark) versus white (light)
- Larger gamut than both RGB and CMYK



Other common image types

- **RGBD**
  In addition to a colour image (RGB), acquires a depth image (D), which represents the distance from each pixel from the camera
  $$I(x, y): \mathbb{Z}^2 \rightarrow \mathbb{Z}^4$$

- **Volumetric images**
  - Image data stored as voxels (i.e. volume elements: small cubes or 3D pixels)
  - Example: computed tomography (CT)
    $$I(x, y, z): \mathbb{Z}^3 \rightarrow \mathbb{Z}$$

- **Video**
  A sequence of images over time
  $$I(x, y, t): \mathbb{Z}^3 \rightarrow \mathbb{Z}^3$$

Image file formats

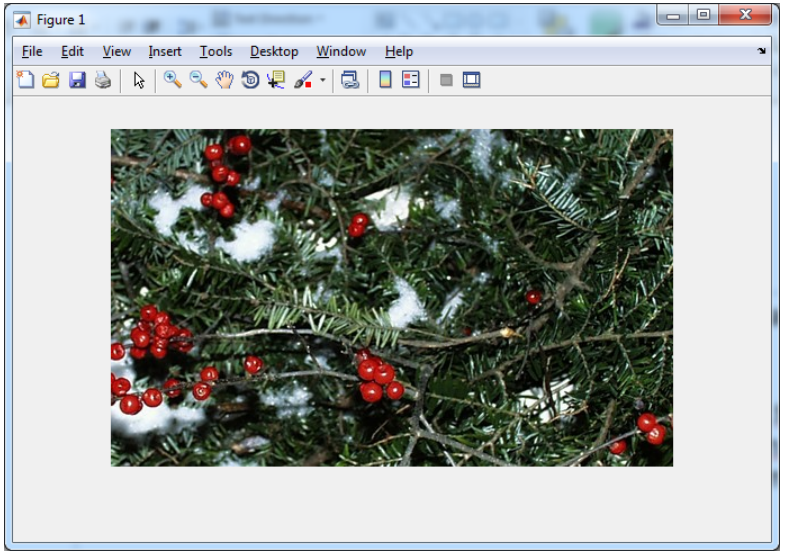
- **BitMap** (BMP): uncompressed, therefore large and lossless. If 24-bit colour format, then:
  - a 12MP image requires 36MB of storage
  - a 30 fps video generates 30 x 36MB = 1.008GB per second
  - a 2 hour video (image only) requires 7257.6GB (7.26TB) of storage
- **JPEG**: Joint Photographic Experts Group is a lossy compression method
- **TIFF**: Tagged Image File Format is a flexible format (both loss and lossless)
- **GIF**: Graphics Interchange Format is usually limited to an 8-bit palette (256 colours). The GIF format is most suitable for storing graphics with few colours, such as simple diagrams, shapes, logos and cartoon style images
- **PNG**: the PNG (Portable Network Graphics) file format was created as a free, open-source alternative to GIF

Images in Matlab

Matlab (aka **Mat**rix **Lab**oratory) is well suited for images, as it was originally designed to manipulate 2D arrays (matrices)

- Matlab can load images of a variety of common formats (.tif, .jpg, .bmp, .png). It also comes with some images built-in (e.g. ‘cameraman.tif’, ‘coins.png’)
- The command to load an image is `imread`
- To show an image, use the command `imshow`

```
I = imread('greens.jpg');
imshow(I);
```



Images in Matlab

Check `I` in the Workspace:

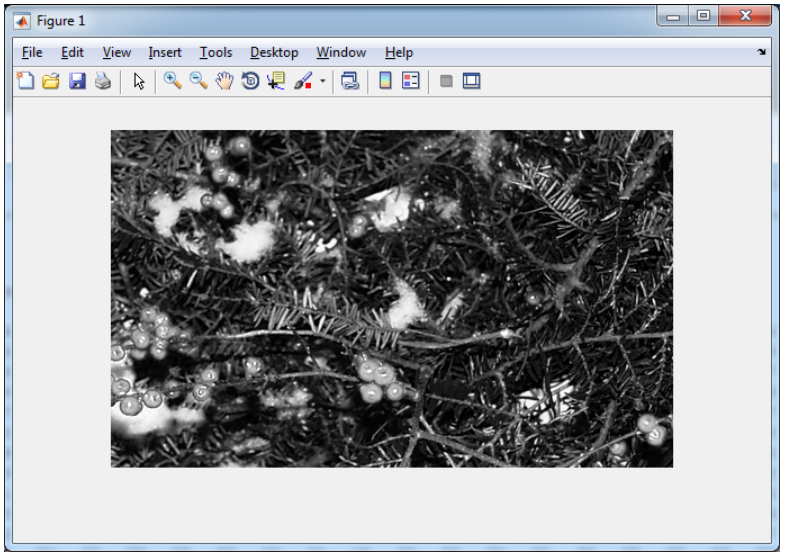
- It's a variable of size 300 x 500 x 3. This means the image has a height of 300, width of 500, and 3 colour channels (I_R, I_G, I_B)
- Its type is **uint8**, i.e. each value in each colour channel is an unsigned integer with 8 bits of precision. This means there are 2^8 = 256 values possible, in the range [0, 255]

To look at the red channel, show only the first colour channel:

red channel

```
imshow(I(:, :, 1));
```

all rows all columns

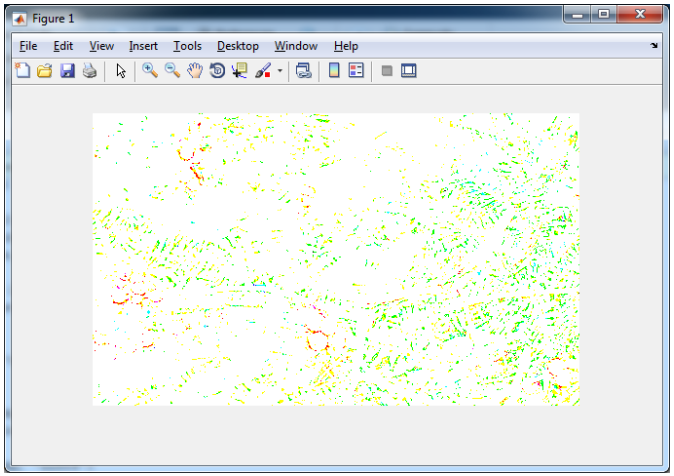


uint8 and double

- Images are typically uint8 when loaded. However, sometimes it may be convenient to convert them to double (e.g. for filtering). Matlab supports type conversions:

```
I = imread('greens.jpg');
J = double(I);
```

- The values of `I` (going from 0 to 255 in each colour channel) are converted to double. However if you call `imshow` on `J`, a funny image is shown, since **for double variables, Matlab expects values between 0 and 1** (everything else clipped)




```
imshow(J);
```

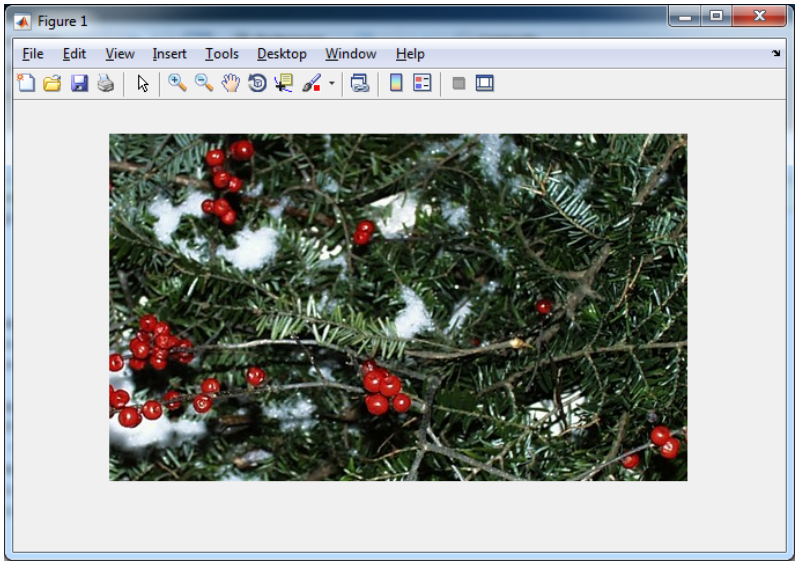
Visualizing double data

- This can be resolved by visualising the data divided by 255:

```
I = imread('greens.jpg');
J = double(I);
imshow(J/255);
```

- The function `im2double(I)` does the division for you:

```
I = imread('greens.jpg');
J = im2double(I);
imshow(J);
```



Conversions to greyscale

- In Matlab, it is easy to convert a colour image into a greyscale one.
  Built-in Matlab function: `rgb2gray`

```
I = imread('Holiday.png');
imshow(I);
I1 = rgb2gray(I);
imshow(I1);
```

- When dealing with **double** greyscale images with custom ranges (different from [0,1], it is often best to let `imshow` adapt its range:
  `imshow(I1,[]);`

- Otherwise, we can use `imagesc`, which does it automatically. However, we need to specify axis/colormap information:
  `imagesc(I1), axis image, colormap gray;`

Other colour space conversions

- Conversion from RGB to HSV can be done with `rgb2hsv`

```
I = imread('Holiday.png');
imshow(I);
J = rgb2hsv(I);
imshow(J(:,:,1)); % H
figure;
imshow(J(:,:,2)); % S
figure;
imshow(J(:,:,3)); % V
```

and back with `hsv2rgb`

```
K = hsv2rgb(J);
```

H

S

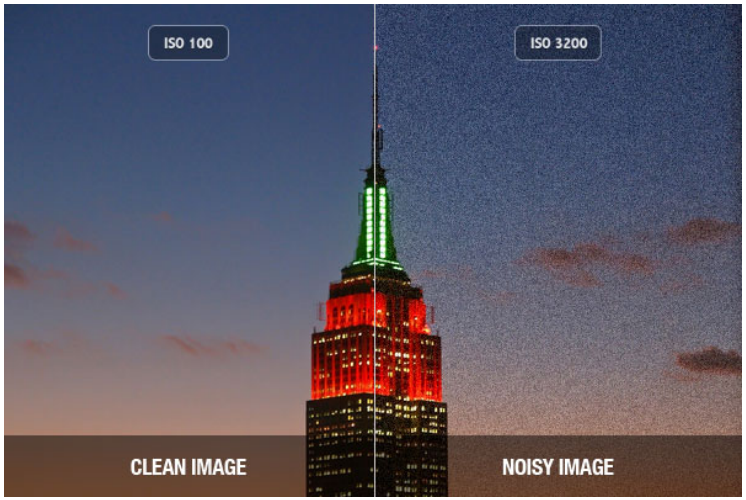
V

(just for illustration purposes…)

- There is also `rgb2lab`, and `lab2rgb`

Imperfections in images

Low resolution



Noise

Bloom
(i.e. light bleeding on darker background)

Imperfections in images

Motion blur

Poor contrast

Compression artefacts

Lens distortion

Digital image processing

- Digital image processing is the use of computer algorithms to transform an image
- Example transformations:
  - Brightness and colour transformations
  - Geometric transformations
  - Filtering (next lecture)

Brightness transformations

- Brightness transformations are **position-independent** and take the form

$$J = f(I)$$

where
  - $I(x, y)$ is the intensity (or colour channel value) of original image
  - $J(x, y)$ is the intensity (or colour channel value) of transformed image
  - f describes the transformation between them (e.g.: $J = I + 20$)

- The function f is independent of position in the image, and therefore the (x, y) arguments are dropped in the equation $J = f(I)$ above: f operates on the pixel values
- Capping/clipping:
  - if $J(x, y) > 255$, $J(x, y) = 255$
  - if $J(x, y) < 0$, $J(x, y) = 0$

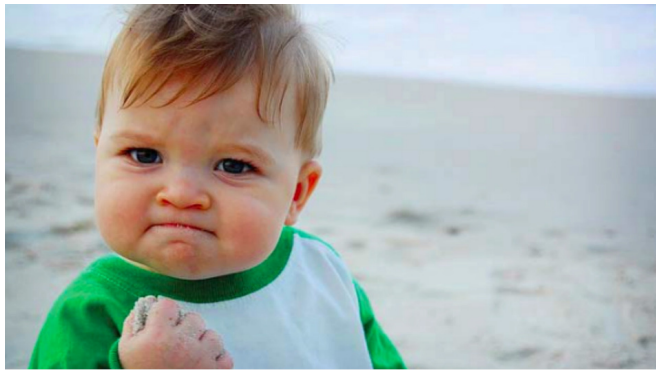
Negative

- The negative of an 24-bit colour image can be formed simply as

$$J = 255 - I$$

- In Matlab, this can be implemented as

```
I = imread('success-Kid.jpg');
figure; imshow(I);
J = 255 - I;
figure; imshow(J);
```



Original image

Negative

Tinting

- Tinting (and colour balancing) applies an adjustment to the colours, normally as a multiplication

$$\begin{bmatrix} I_R{}' \\ I_G{}' \\ I_B{}' \end{bmatrix} = \begin{bmatrix} s_R I_R \\ s_G I_G \\ s_B I_B \end{bmatrix}$$

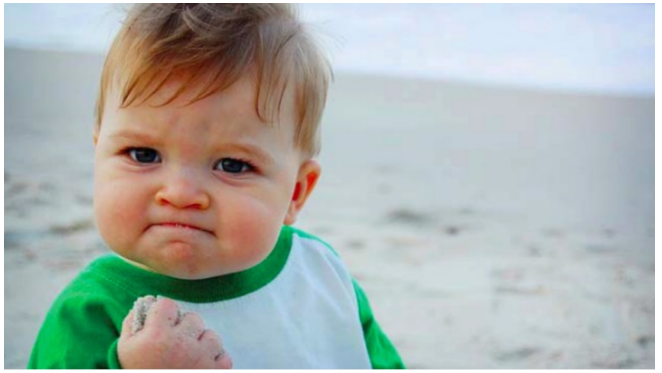
where s_R, s_G, and s_B scale the red, green, and blue colour of each pixel

Tinting

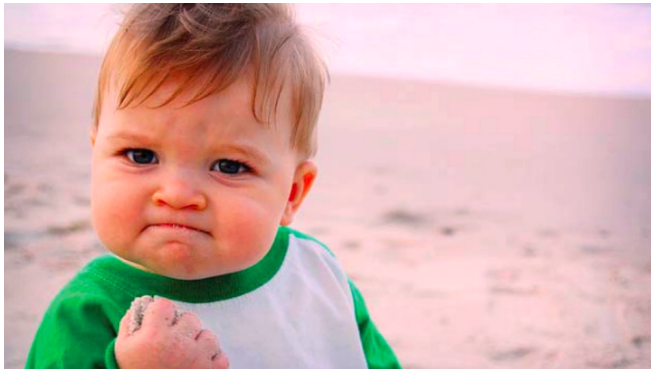
Example: increasing redness of an image

$s_R = 1.25$, $s_G = 1$, and $s_B = 1$:

```
I = imread('success-Kid.jpg');
R = I(:, :, 1);
G = I(:, :, 2);
B = I(:, :, 3);
s_r = 1.25; s_g = 1; s_b = 1;
J(:, :, 1) = s_r * R;
J(:, :, 2) = s_g * G;
J(:, :, 3) = s_b * B;
figure; imshow(J);
```



Original image

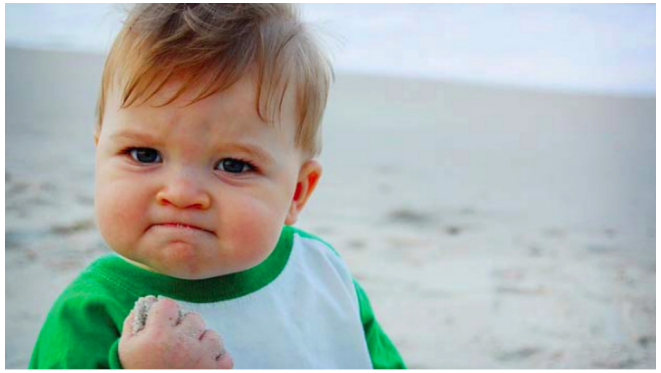


Transformed image

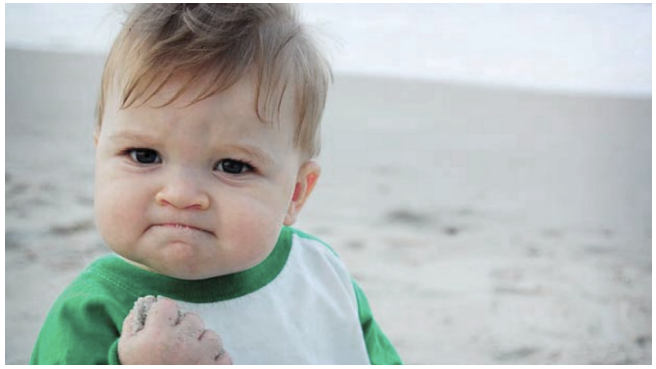
Transformation in HSV space

This can be done in other colour spaces too.
Example: decreasing saturation

```
I = imread('success-Kid.jpg');
HSV = rgb2hsv(I);
H = HSV(:, :, 1);
S = HSV(:, :, 2);
V = HSV(:, :, 3);
s_h = 1; s_s = 0.5; s_v = 1;
J(:, :, 1) = s_h * H;
J(:, :, 2) = s_s * S;
J(:, :, 3) = s_v * V;
J = hsv2rgb(J);
figure; imshow(J);
```



Original image



Transformed image

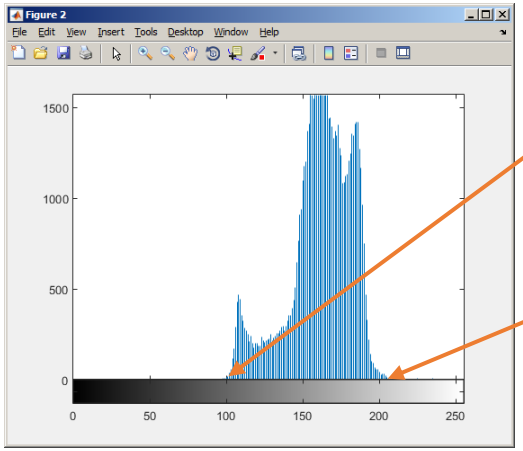
Histogram

- A histogram gives the **count** of pixel values in an image
- The full available range is divided into bins for easier interpretation
- Can be computed both for each colour channel or grayscale version
- Built-in Matlab function: `imhist`

```
I = imread('cars.png');
figure; imshow(I);
figure; imhist(I, 256);
```

Num of bins

Image



Histogram

$\min(I) = 103$

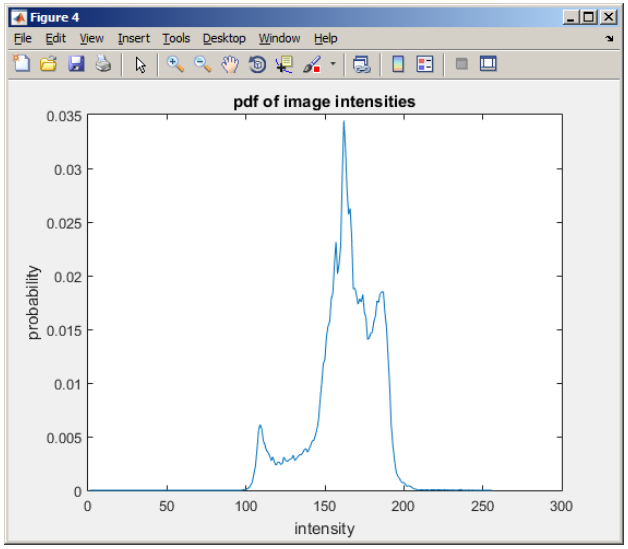
Dynamic range

$\max(I) = 210$

Histogram

- A histogram can be normalised and interpreted as a **probability density function** (pdf), representing the probability of a pixel having a particular brightness

```
[rows, cols] = size(I);
pdf = h / (rows*cols);
figure;
plot(pdf);
title('pdf of image intensities');
xlabel('intensity');
ylabel('probability');
```



pdf

- In the example in this slide, we could say that a pixel has a higher probability of having an intensity of 160 than an intensity of 200

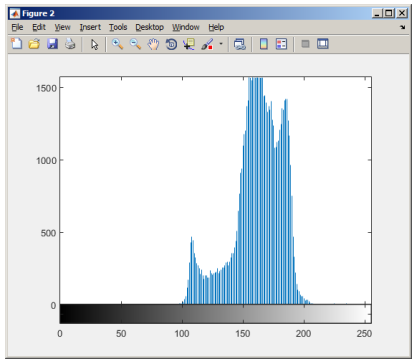
Contrast stretching

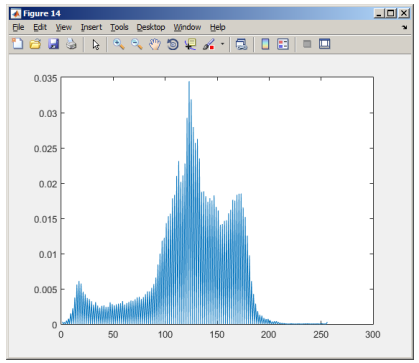
- If most of the intensities are clustered in the 100÷200 range, the image looks “washed out” (lack of darker and brighter values)
- Contrast stretching is a **linear transformation** that aims at extending the dynamic range of the image
- We can transform the values using a function

$$J = \alpha I + \beta$$

```
alpha = 2; beta = -200;
J = uint8(alpha * double(I) + beta);
figure; imshow(J);
```

- Built-in Matlab function: `imadjust`





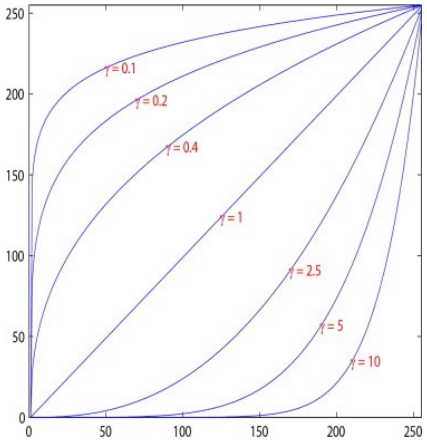
Gamma correction

- Gamma correction applies a **non-linear transformation** of pixel values
- It tries to take advantage of the non-linear manner in which humans perceive light (greater sensitivity to differences between darker tones than lighter ones)

- The mathematical form is

$$J = AI^{\gamma}$$

where γ is a parameter and usually $A = 255^{(1-\gamma)}$ for an image with range 0÷255



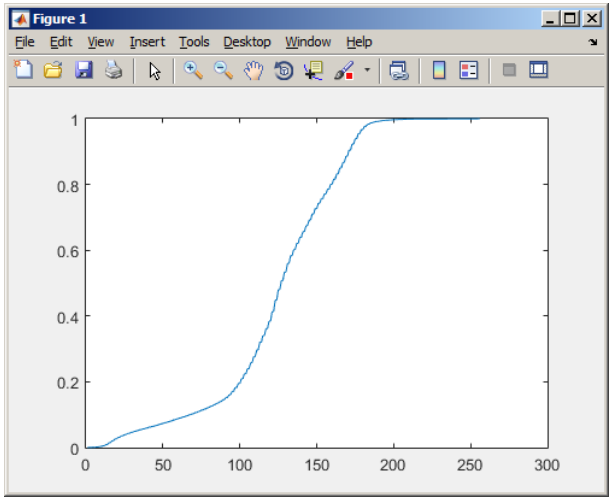
Credits: www.medium.com

```
gamma = 2;
J = 255^(1-gamma)*double(I).^gamma;
figure;
imshow(uint8(J));
```

Histogram equalization

- Histogram equalisation applies a **non-linear transformation** that tries to uniformly distribute the pixel values in the dynamic range
- Non-linear: differently from contrast stretching, relative distances between pixel values can both increase and decrease for different ranges
- Usually uses the (normalized) cumulative histogram function, which gives, for a given intensity value, the percentage of pixels in the image that have that value or a lower one
- Cumulative histogram function of the output will be roughly a straight line

```
h = imhist(I, 256);
[rows, cols] = size(I);
cdf = cumsum(h) / (rows*cols);
plot(cdf);
```



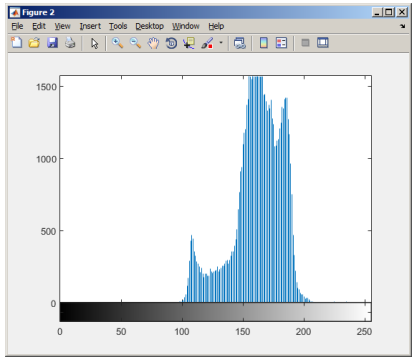
cdf

Histogram equalization

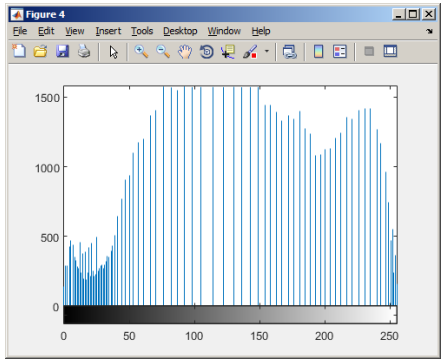
- Actual transformation:

```
J = uint8(255*cdf(I+1));
figure; imshow(J);
figure; imhist(J, 256);
```

- Built-in Matlab functions: `histeq`, `adapthisteq`
  https://stackoverflow.com/questions/26818568/whats-the-difference-between-histeq-and-adapthisteq



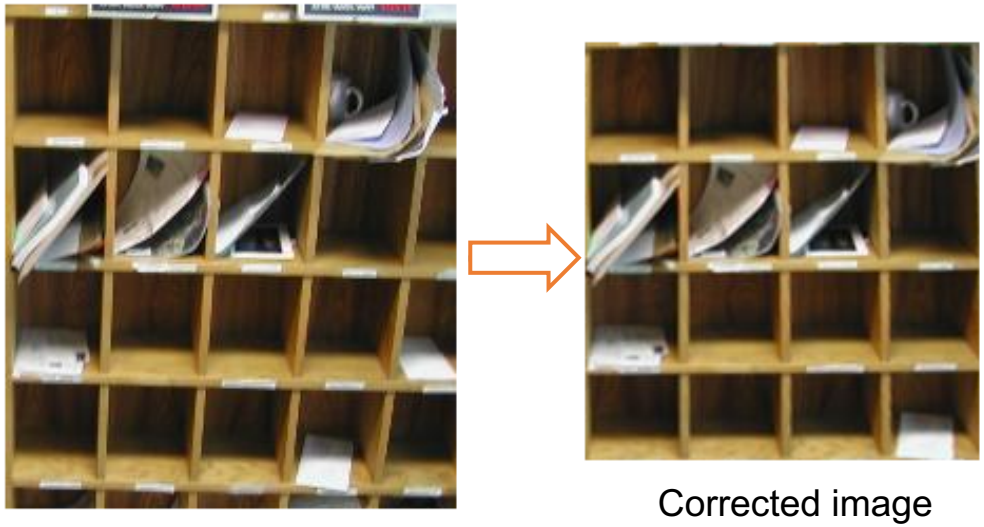
Original



After histogram equalization

Geometric transformations

- Geometric transformations **change the spatial position** of pixels in the image
- Geometric transformations (generically called image warpings) have a variety of practical uses, including
  - Registration: aligning different (but similar) images
  - Removing distortion
  - Simplifying further processing



Distorted image

Corrected image

Geometric transformations

- The positions of pixels in the image are transformed
- Mathematically, this is expressed as

$$\boldsymbol{x}' = T(\boldsymbol{x})$$

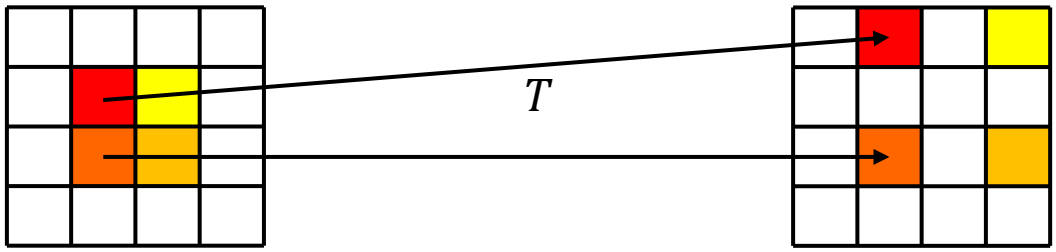
where:

  - $\boldsymbol{x} = (x, y)$ is the position of a point in the distorted image
  - $\boldsymbol{x}' = (x', y')$ is the position of a point in the corrected image
  - $T(\boldsymbol{x})$ is a mapping function

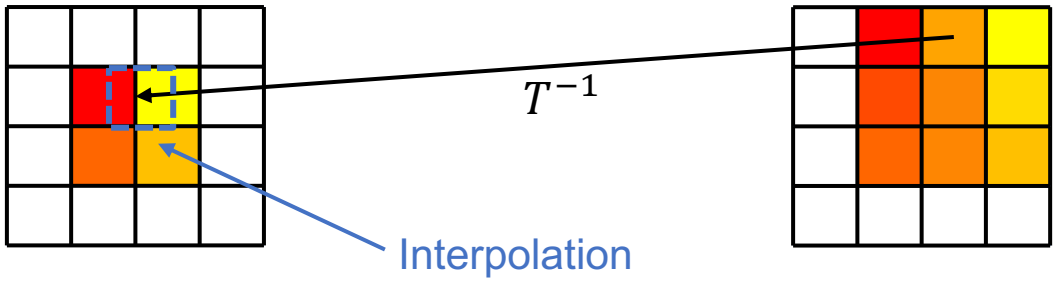
- The easiest way to implement an image warping from I to J is as follows:
  For every pixel position $\boldsymbol{x}'$ in the corrected image:
  - Using T^{-1}, determine $\boldsymbol{x}$ (i.e. where $\boldsymbol{x}'$ came from in the distorted image)
  - Interpolate a value from $I(\boldsymbol{x})$ to produce $J(\boldsymbol{x}')$

Geometric transformations

- You may notice this is applied somewhat backwards: rather than using a (forward) mapping T to transform pixels from the distorted image to the corrected image, we use the (inverse) transform T^{-1}



Forward mapping may result in gaps



Inverse mapping ensures no gaps

- This ensures that all the pixels in the corrected image will be filled. (Gaps can be filled in the corrected image too, but more complicated)
- However, it's often necessary to **interpolate** pixels from the distorted image

Affine transformation

- Affine transformations are transformations that preserve (among others):
  - Collinearity (i.e. aligned points will remain aligned)
  - Parallelism (i.e. parallel lines will remain parallel)
- It can be fully expressed through matrix product:

$$\boldsymbol{x}' = A\boldsymbol{x}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13} \\ a_{21}x + a_{22}y + a_{23} \\ 1 \end{bmatrix}$$

- In Matlab, you can define an affine transformation using

```
A = affine2d([a11 a12 a13; a21 a22 a23; 0 0 1]');
```

and apply the transformation to an image using `imwarp`

Matlab wants a transposed input

Special cases

- Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

- Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ 1 \end{bmatrix}$$

- Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ 1 \end{bmatrix}$$

- Interpolation will then be (usually) needed
- Built-in Matlab functions: `imtranslate`, `imrotate`, `imresize`

Special cases

```
I = imread('Holiday.png');
imshow(I);
```

```
J = imtranslate(I, [100, 0]);
imshow(J);
```

```
J = imrotate(I, 45);
imshow(J);
```

```
J = imresize(I, 0.5);
imshow(J);
```

Special cases

- Skew

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \tan\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + y\tan\theta \\ y \\ 1 \end{bmatrix}$$

or

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + sy \\ y \\ 1 \end{bmatrix}$$

Exercise 1

Test: manually transform image in order to have well-aligned digits

```
I = imread('LicensePlate.png');
figure; imshow(I);
```


```
% Rotate clockwise 15 degrees to align base
J1 = imrotate(I, -15, 'bilinear');
figure; imshow(J1);
```

Type of interpolation


```
% Now apply a skew
tform = affine2d([1 .3 0; 0 1 0; 0 0 1]');
J2 = imwarp(J1, tform);
figure; imshow(J2);
```

Exercise 2

Test: match code to transformed image

```
R = imref2d([700,700]);
```
← Spatial referencing object (for better visualization)

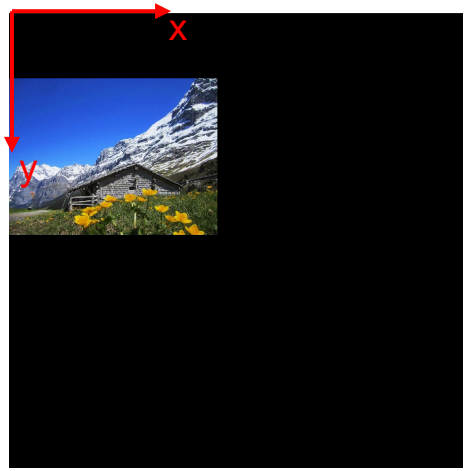
```
% Transformation 1
A = affine2d([.5 0 0; 0 .5 100; 0 0 1]');
[J, RJ] = imwarp(I, A, 'OutputView', R);
imshow(J, RJ);
```

```
% Transformation 2
A = affine2d([.5 0 100; 0 .5 0; 0 0 1]');
[J, RJ] = imwarp(I, A, 'OutputView', R);
imshow(J, RJ);
```

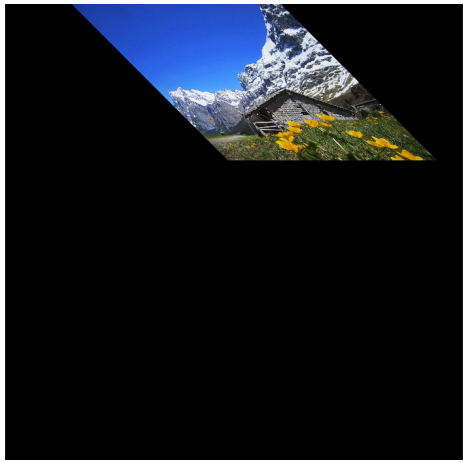
```
% Transformation 3
A = affine2d([.5 .5 100; 0 .5 0; 0 0 1]');
[J, RJ] = imwarp(I, A, 'OutputView', R);
imshow(J, RJ);
```

```
% Transformation 4
A = affine2d([0 .5 0; .5 .5 100; 0 0 1]');
[J, RJ] = imwarp(I, A, 'OutputView', R);
imshow(J, RJ);
```

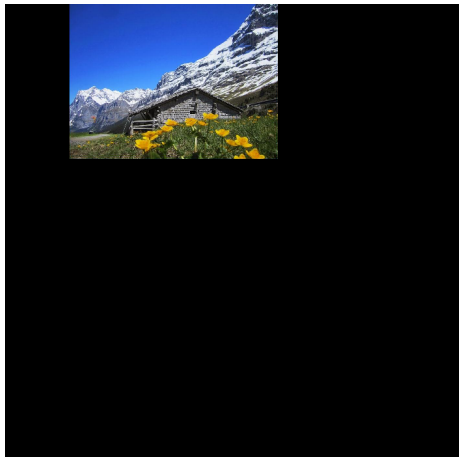
A



B



C

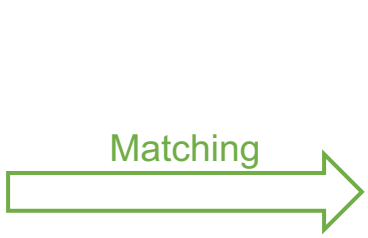


D

Common CV tasks

Image matching/alignment/registration

- Identify the (rigid? affine? non-rigid?) transformation which aligns two or more images
- Used for multi-modal comparison, image stitching, etc.)



M. Brown. Automatic panoramic image stitching using invariant features, IJCV 2007

Image matching

Test: find the affine transformation that aligns two images

- The goal is to determine the six coefficients in the A matrix
- This can be achieved by finding **at least 3 correspondences** (e.g. plate corners) in the two images

Let's assume that we have manually selected the following correspondences:

$$\boldsymbol{p}_1 = [18, 47]^T \longrightarrow \boldsymbol{q}_1 = [48, 50]^T$$
$$\boldsymbol{p}_2 = [15, 100]^T \longrightarrow \boldsymbol{q}_2 = [48, 100]^T$$
$$\boldsymbol{p}_3 = [178, 6]^T \longrightarrow \boldsymbol{q}_3 = [212, 50]^T$$

Image matching

Remembering that for each point we have

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13} \\ a_{21}x + a_{22}y + a_{23} \\ 1 \end{bmatrix}$$

Then for three correspondences we can write

$$\begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix}$$

Or in matrix form

$$q = Ma$$

- The 6 coefficients of the affine transformation can be solved with $a = M^{-1}q$
- If we had more than 3 correspondences, we could have solved this using least-squares

Image matching

Built-in Matlab function: `estimateGeometricTransform`

```
% Estimate affine transformation through correspondences
I = imread('LicensePlate.png');
p1 = [18, 47]; p2 = [15, 100]; p3 = [178, 6];
q1 = [48, 50]; q2 = [48, 100]; q3 = [212, 50];

P = [p1; p2; p3];
Q = [q1; q2; q3];
tform = estimateGeometricTransform(P, Q, 'affine');

% Now apply the warp to the image
J = imwarp(I, tform);
figure; imshow(J);
```

Type of transformation

Projective transformation

- Images normally acquired by photographic cameras are formed by perspective projection, as described earlier
- A rectangular surface not parallel to the image plane will be projected into a trapezoid
- If we want to transform it into a rectangle, an affine transformation will not be enough
- Instead, we must use a projective transformation:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- To perform image matching with a projective transformation, at least **4 correspondences are needed**
- This is due to the eight unknowns p_{ij}, which correspond to the degrees of freedom (DOF) of this type of projective transformations

Projective transformation

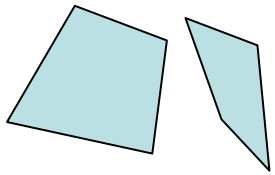
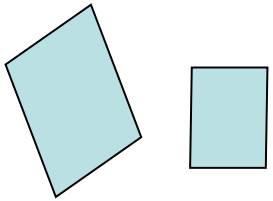
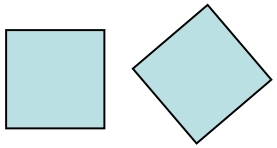
Test: transform the pinball surface into a square

Original

Affine

Projective

Hierarchy of 2D transformations

Transformation	Matrix	Transformed squares	Properties preserved
Projective 8 DOF	$\begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & 1 \end{bmatrix}$		Collinearity
Affine 6 DOF	$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$		+ Parallelism of lines
Rigid 3 DOF	$\begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$		+ Lengths, angles, areas

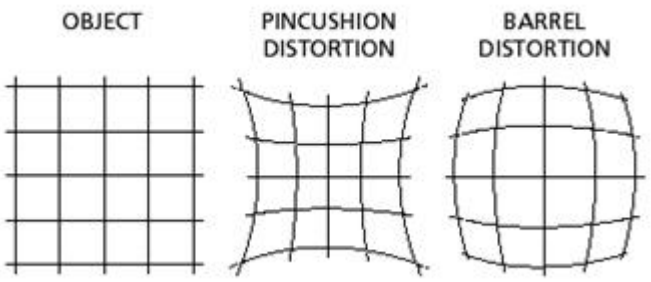
Credits: Marc Pollefeys

Non-linear transformations

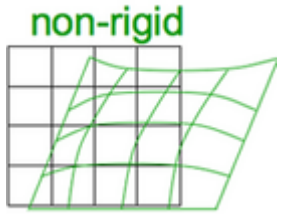
- Non-linear transformations cannot be represented as a matrix multiplication
- For them, we need to use the general function that transforms pixel locations independently:

$$\boldsymbol{x}' = T(\boldsymbol{x})$$

- Examples of common non-linear transformations include:



Radial lens distortion



Non-rigid transformation

Overview of next week's lecture

- Image filtering:
  - Kernel-based methods (with convolution)
- Common filters:
  - Moving average
  - Gaussian
  - Sharpening
  - Median
- Edge detection
  - Gradient-based
  - Laplacian-based
  - Non-maximum suppression
- Common CV tasks