CITY
UNIVERSITY OF LONDON
EST 1894

Computer Vision
INM460/IN3060

Lecture 3
Image filtering and edge detection

Dr Giacomo Tarroni
Slides credits: Giacomo Tarroni, Sepehr Jalali, Wenjia Bai

Admin

- Drop-in hours for Week 4: Wednesday 12/02, 11am – 1pm, A302B
- Others remain as default: Wednesdays 2 – 4pm, A302B

Recap from the previous lecture

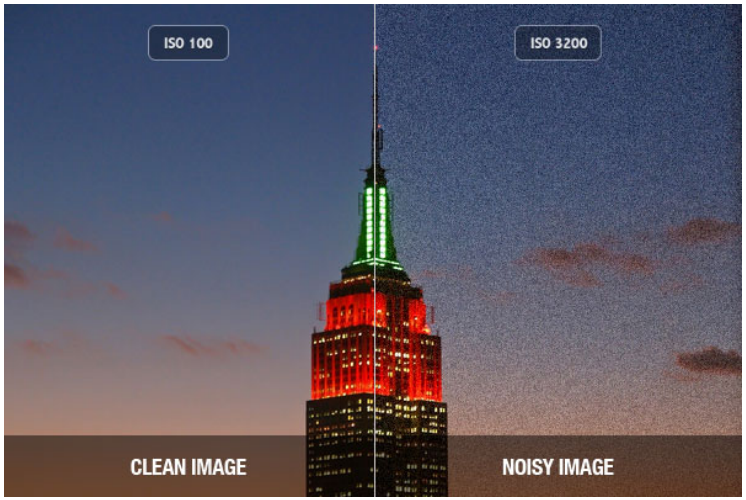
- Digital images:
  - Formation
  - Pinhole camera model
  - Digital representation
- Digital image processing:
  - Brightness and colour transformations
  - Geometric transformations
  - Filtering (not yet presented)

Overview of today's lecture

- Image filtering
- Linear filtering:
  - Convolution vs cross-correlation
  - Common filters:
  - Moving average
  - Gaussian
  - Sharpening
- Non-linear filtering: median filter
- Edge detection
  - Gradient-based
  - Laplacian-based
  - Non-maximum suppression
- Common CV tasks

Imperfections in images

Low resolution



Noise

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

Imperfections in images

Motion blur

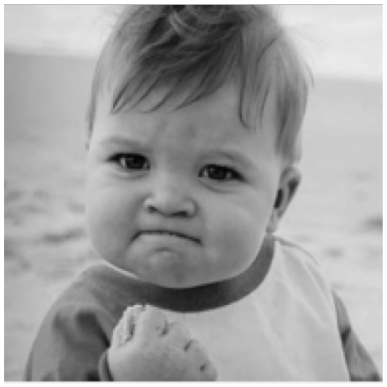
Poor contrast

Compression artefacts

Lens distortion

Image filtering

- The easiest way to improve image appearance is to perform image filtering
- In general, filtering allows to remove unwanted components from a given signal or to enhance some desired ones
- Image filtering can be used in a variety of contexts, including noise reduction and image enhancement
- Filtering can be applied to both grayscale and colour images. For colour images, the filtering is applied to each channel separately or to the V channel after a conversion to HSV colour space



Original image

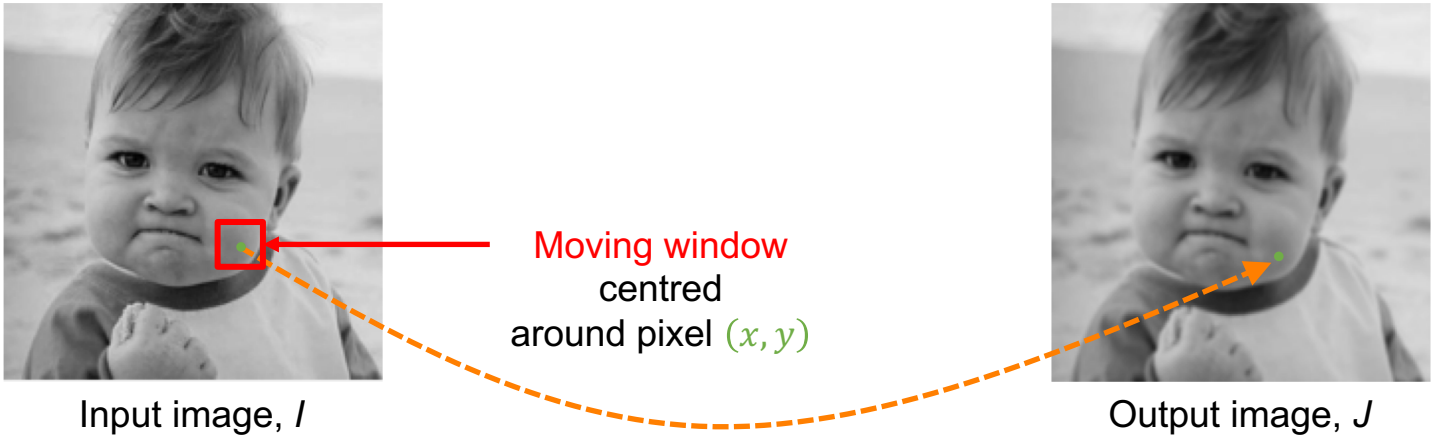
Smoothed
(-noise, -details)

Sharpened
(+details, +noise)

Credits: Laney and Sam Griner

Window-based image filtering

- In window-based image filtering, the value of each pixel (x, y) in the output image J is a function of the pixel values of the input image I in a neighbourhood of (x, y)
- The neighbourhood is defined by a window of a given size (e.g. N x N)
- This can be thought as sliding a N x N window on the input image, computing an output value at each position
- The type of filtering is determined by the function describing the relationship between input and output pixel values

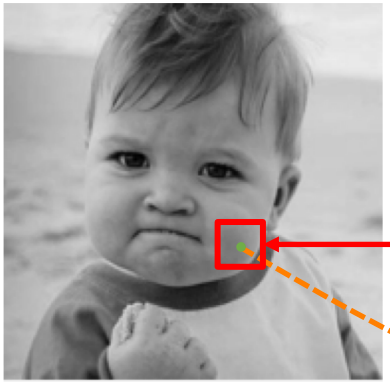


Linear filtering

- If the pixel values in the output image are a **linear combination** of pixel values in a neighbourhood of the input image, then all we need to represent the function between the input and the output are the weights
- The weights are usually represented as a **kernel**
- If, in addition, the weights are fixed for all positions, we are performing **linear filtering**

1	0	1
0	2	0
1	0	1

Kernel K



Input image, *I*

Moving window centred around pixel (x, y)

Output image, *J*

Linear filtering

- Our goal is to compute $J(x, y)$, the value of pixel (x, y) in the output image J
- To do this, we place the kernel K (a matrix of weights) on the input image I centred around a position (x, y). For every position (n, m) in K, we multiply the kernel value with the input image pixel, and sum up all the values:

$$J(x, y) = \sum_{n,m \in K} K(n, m)\, I(x + n, y + m)$$

- For example, consider the following 3x3 kernel, and a set of pixel values:

1	0	1
0	2	0
1	0	1

Kernel K

$*$

22	23	24
27	28	29
32	33	34

Input image I

$=$

	?	

Output image J

Linear filtering

- Our goal is to compute $J(x, y)$, the value of pixel (x, y) in the output image J
- To do this, we place the kernel K (a matrix of weights) on the input image I centred around a position (x, y). For every position (n, m) in K, we multiply the kernel value with the input image pixel, and sum up all the values:

$$J(x, y) = \sum_{n,m \in K} K(n, m)\, I(x + n, y + m)$$

- For example, consider the following 3x3 kernel, and a set of pixel values:

1	0	1
0	2	0
1	0	1

Kernel K

*

22	23	24
27	28	29
32	33	34

Input image I

=

	168	

Output image J

- In this case, the result would be (1*22 +1*24 +2*28 +1*32 + 1*34) = 168

Linear filtering

- The kernel is typically much smaller than the input image. The output image is generated by **sliding** the kernel on the input image, one pixel at a time

1	0	1
0	2	0
1	0	1

Kernel K

$*$

22	23	24	25	26
27	28	29	30	31
32	33	34	35	36
37	38	39	40	41
42	43	44	45	46

Input image I

$=$

X	X	X	X	X
X	168			X
X				X
X				X
X	X	X	X	X

Output image J

Linear filtering

- The kernel is typically much smaller than the input image. The output image is generated by **sliding** the kernel on the input image, one pixel at a time

1	0	1
0	2	0
1	0	1

Kernel K

$*$

22	23	24	25	26
27	28	29	30	31
32	33	34	35	36
37	38	39	40	41
42	43	44	45	46

Input image I

$=$

X	X	X	X	X
X	168	174		X
X				X
X				X
X	X	X	X	X

Output image J

Linear filtering

- The kernel is typically much smaller than the input image. The output image is generated by **sliding** the kernel on the input image, one pixel at a time

1	0	1
0	2	0
1	0	1

Kernel K

$*$

22	23	24	25	26
27	28	29	30	31
32	33	34	35	36
37	38	39	40	41
42	43	44	45	46

Input image I

$=$

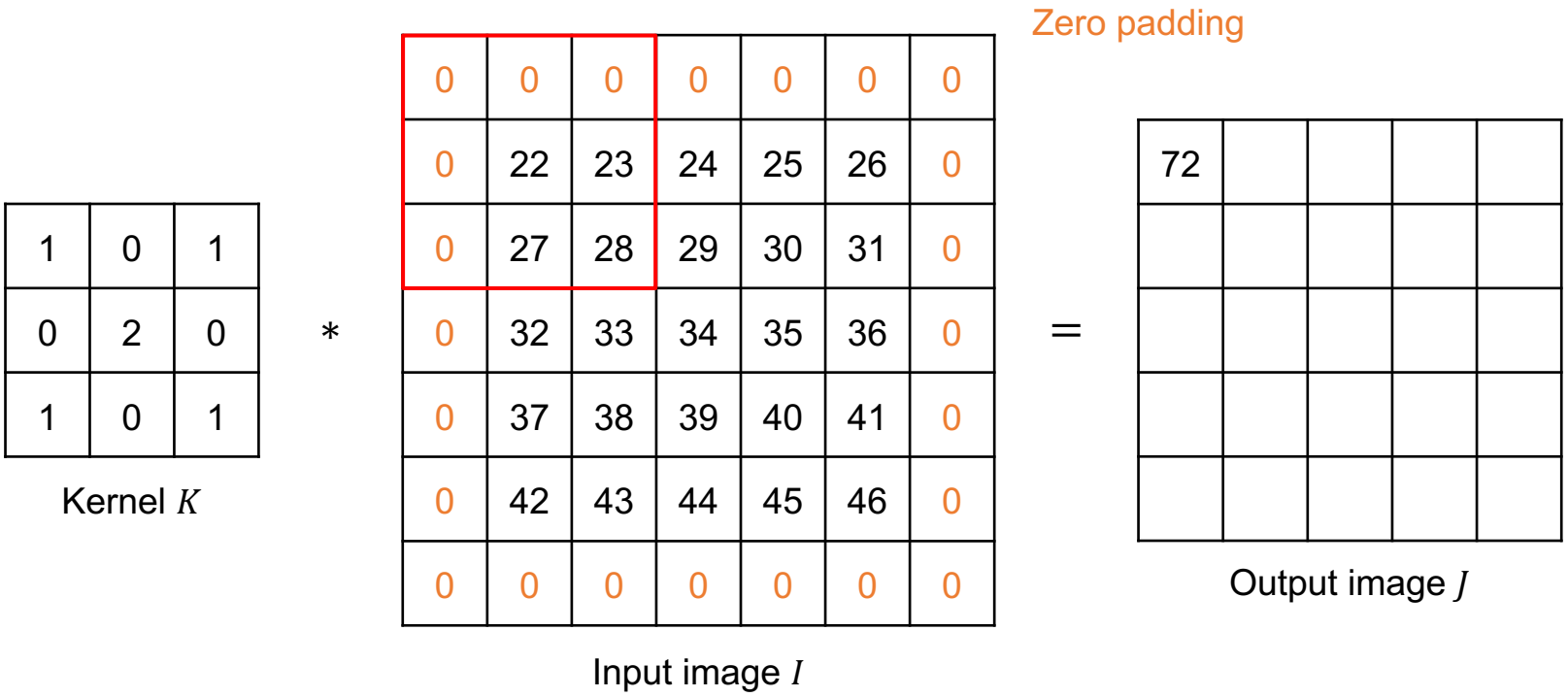
X	X	X	X	X
X	168	174	180	X
X	198	204	210	X
X	228	234	240	X
X	X	X	X	X

Output image J

Linear filtering

Edge handling: how can we avoid losing the border pixels?

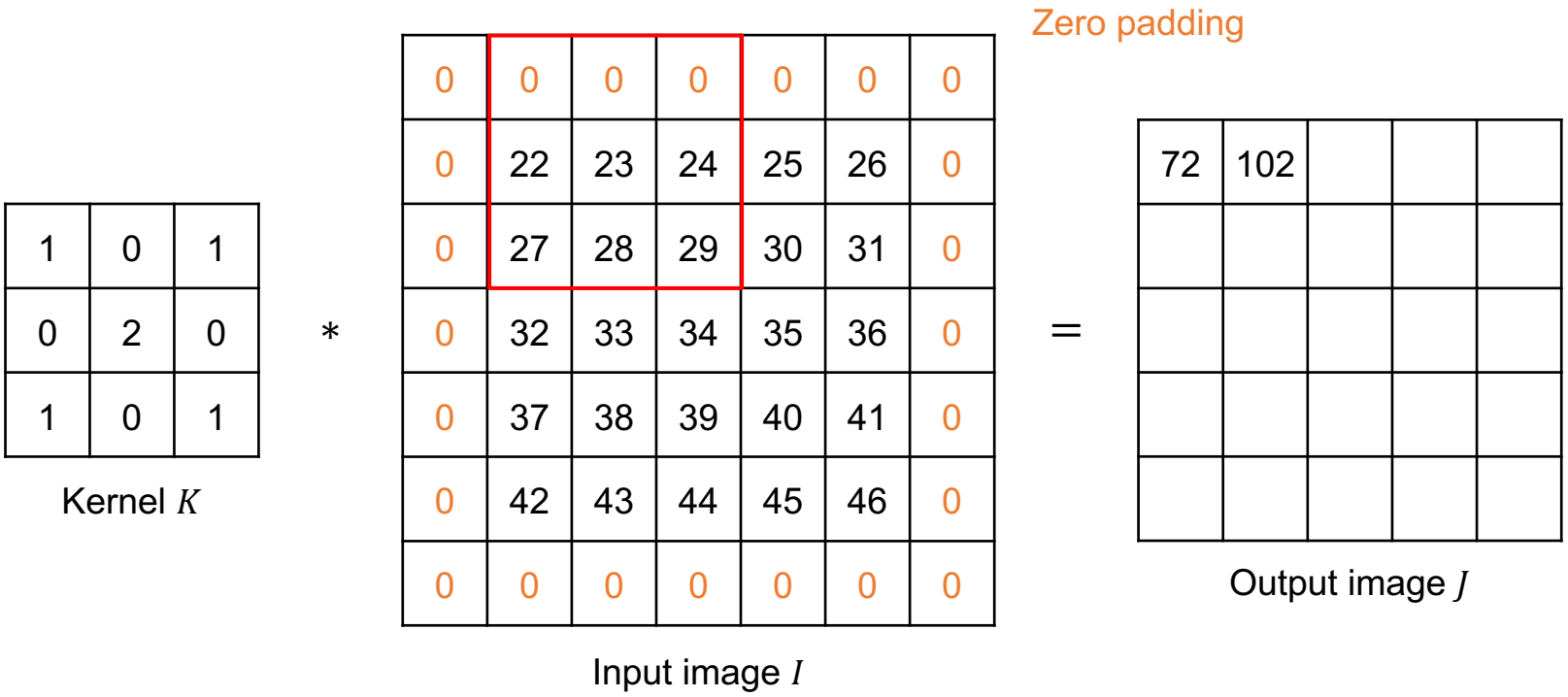
- Extension: the nearest border pixels are replicated as much as necessary to provide values for the kernel
- Wrapping: the pixels are extended using values on other side
- Zero padding: the pixels are extended using 0s (output of the filter on borders is however not reliable anymore)



Linear filtering

Edge handling: how can we avoid losing the border pixels?

- Extension: the nearest border pixels are replicated as much as necessary to provide values for the kernel
- Wrapping: the pixels are extended using values on other side
- Zero padding: the pixels are extended using 0s (output of the filter on borders is however not reliable anymore)



Linear filtering

- Kernel normalization:
  Each kernel element is **divided by the sum of all kernel elements (with their sign!)**, so that the sum of the elements of a normalized kernel is unity. This ensures that the intensity average for the whole image remains unchanged through filtering

- Clipping due to datatype ranges:
  If datatype is uint8, then range of values is 0÷255, which might cause unwanted clipping. When using filters, first convert your image to double, then do the calculations. Before converting it back to uint8 normalize the values

Cross-correlation vs convolution

- The act of moving a linear kernel on the input image to create the output one is called cross-correlation. Another very similar operation is convolution

- Cross-correlation:

$$J(x,y) = K \star I = \sum_{n,m} K(n,m)\, I(x+n, y+m)$$

- Convolution:

$$J(x,y) = K * I = \sum_{n,m} K(n,m)\, I(x-n, y-m) = \sum_{n,m} K(-n,-m)\, I(x+n, y+m)$$

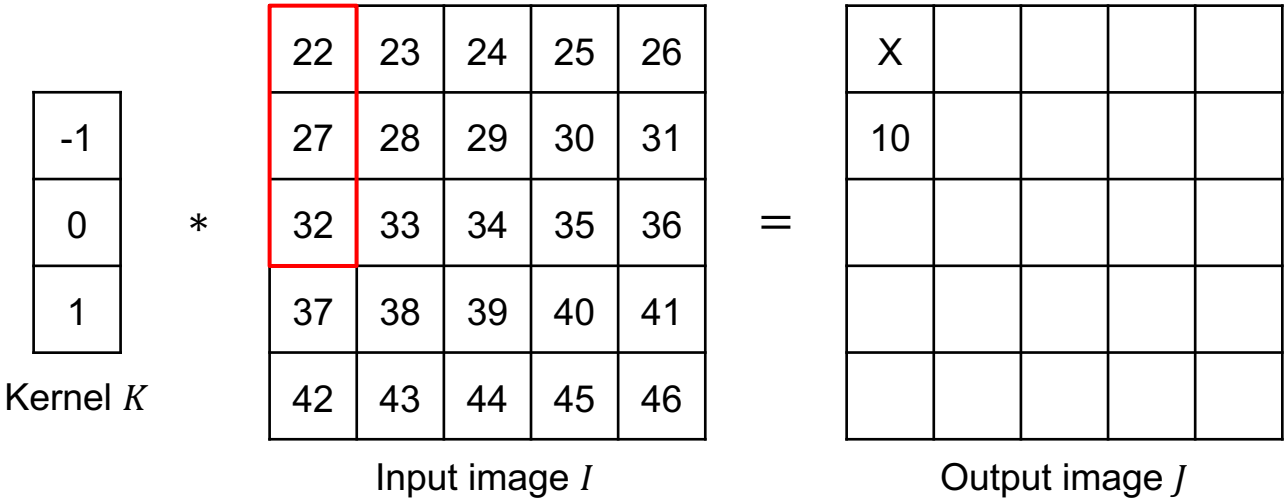
- Convolution is cross-correlation with a kernel flipped in both directions: www.youtube.com/watch?v=Ma0YONjMZLI
- They are the same for symmetric kernels
- In practice, the term used is “convolution” but the operation we will actually perform is cross-correlation

Convolution

Convolution:

$$J(x,y) = K * I = \sum_{n,m} K(n,m)\, I(x-n, y-m)$$

- Convolution is not restricted to 2D kernels:



Convolution

Convolution:

$$J(x,y) = K * I = \sum_{n,m} K(n,m)\, I(x-n, y-m)$$

- Convolution is a linear operator. Thus, these properties hold:
  - Commutative: $K * I = I * K$
  - Distributive: $(K_1 + K_2) * I = K_1 * I + K_2 * I$
  - Associative: $K_2 * (K_1 * I) = (K_2 * K_1) * I$
    The associative property is useful to decompose 2D kernels into 2, 1D ones:

1/3							1/9	1/9	1/9
1/3	*	1/3	1/3	1/3	=		1/9	1/9	1/9
1/3							1/9	1/9	1/9

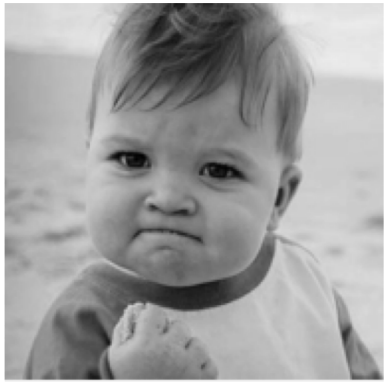
Identity

- The kernel of an identity filter is simply a set of zeros with a 1 in the centre
- The output pixel value will be equal to the input pixel value

0	0	0
0	1	0
0	0	0

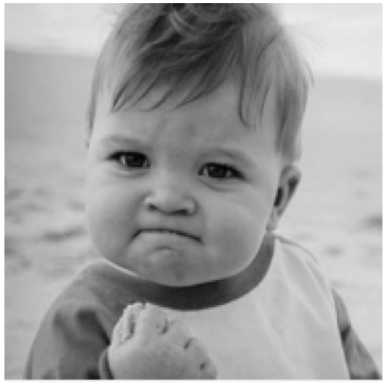
Kernel K
Identity

$*$



Input image I

$=$



Output image J

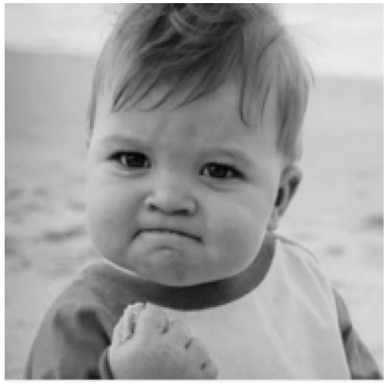
Moving average

- The kernel of a moving average (or box) filter is simply a set of ones divided by the number of kernel elements
- The output pixel value will be the **average** of the pixel values in the box
- Used for smoothing and blurring (noise reduction)

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Kernel K
Moving average

$*$



Input image I

$=$

Output image J

Moving average

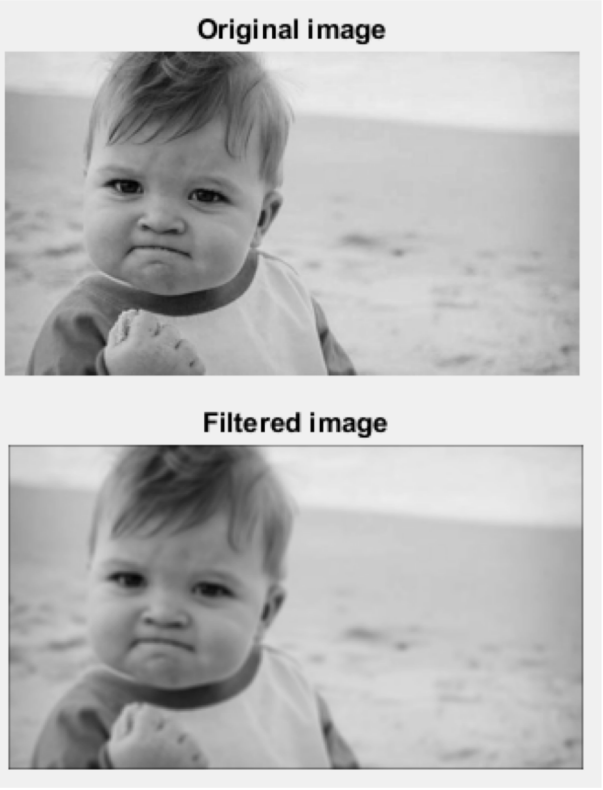
To implement this in Matlab, we must:

- Load the image I
- Define the kernel K
- Apply the filter to the image using the built-in function `imfilter`

```
% Load, resize, and show original image
I = double(imread('Success-Kid.jpg'));
I = imresize(I, 0.33);
figure; imshow(I/255); title('Original image');

% Create the kernel
K = (1/9) * ones(3,3);

% Apply to each colour channel and display result
J = imfilter(I, K);
figure; imshow(J/255);  title('Filtered image');
```

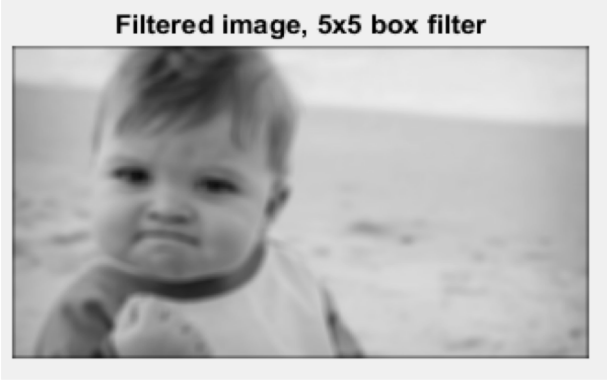


- By default imfilter performs zero padding
- Other edge handling options are available

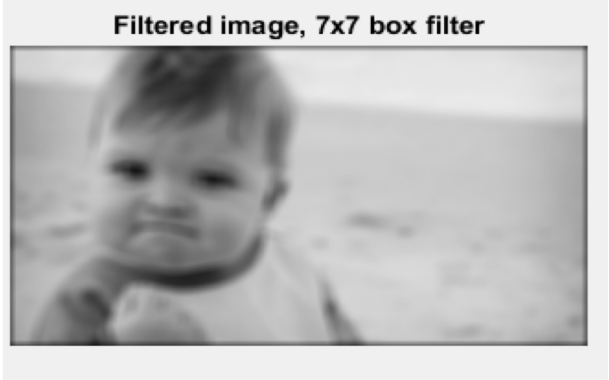
Moving average

What is the impact of the kernel size on the input?


```
K = (1/9) * ones(3,3);
```




```
K = (1/25) * ones(5,5);
```

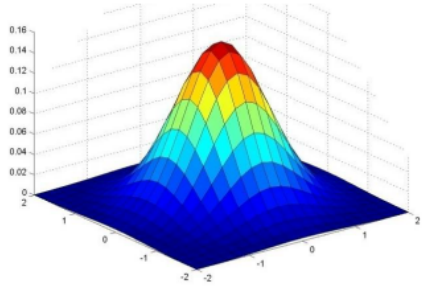


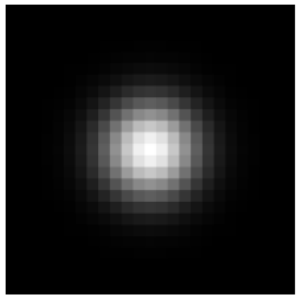

```
K = (1/49) * ones(7,7);
```

Gaussian filter

- The kernel consists of the values sampled from a 2D Gaussian distribution centred on the centre of the kernel: the central pixel has the highest weight, and the weights decrease from the centre outwards
- The standard deviation σ is a parameter controlling the amount of blurring

$$K_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$





0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

5 x 5, σ = 1

Built-in Matlab functions:

- `fspecial` creates a Gaussian kernel
  `K = fspecial('gaussian', 5, 1);`
- `imgaussfilt` directly performs Gaussian filtering

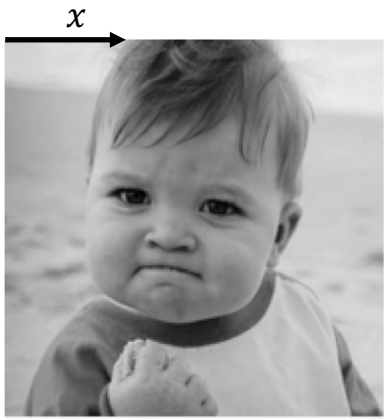
Sharpening filter

- A sharpening filter makes edges more pronounced
- Built-in Matlab function: `imsharpen`

-1/9	-1/9	-1/9
-1/9	17/9	-1/9
-1/9	-1/9	-1/9

Kernel K
Sharpening

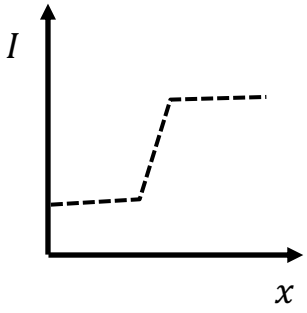
$*$

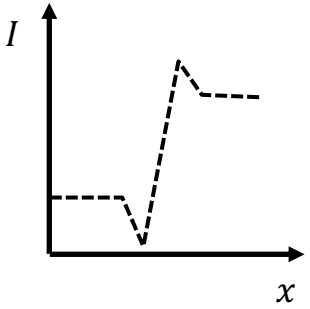


Input image I

$=$

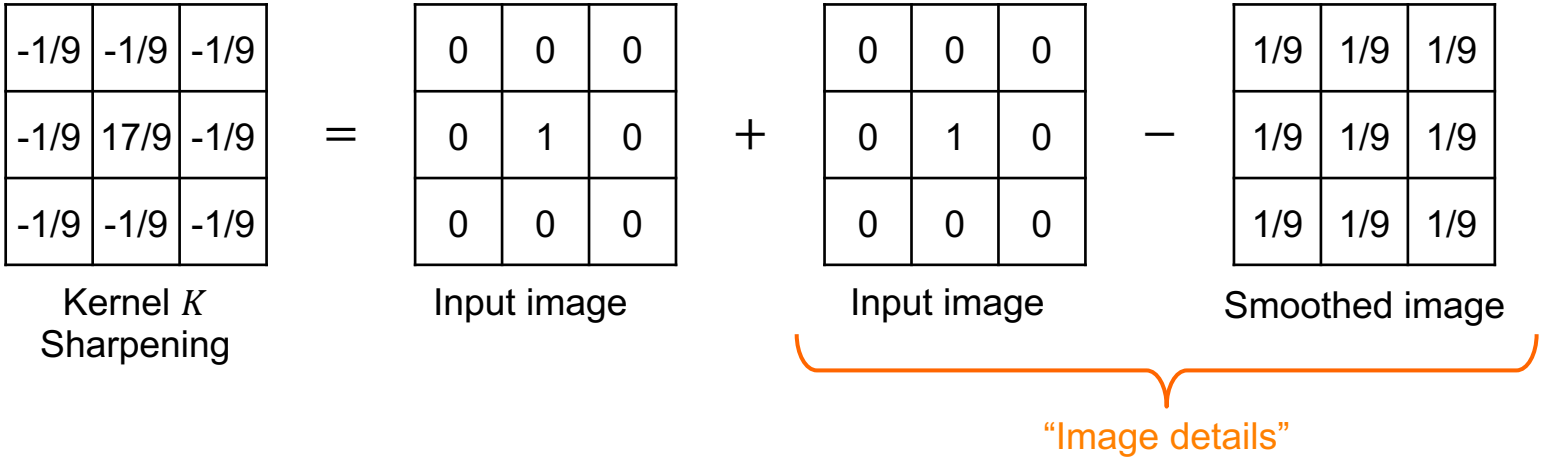
Output image J





Sharpening filter

Using the distributive property of convolution, we can see that the sharpening filter adds “image details” (i.e. differences from local average)



Non-linear filtering

- So far, we have described linear filtering operations, which are mathematically represented by the convolution of a kernel with the image
- A different class of filtering methods are instead non-linear, and cannot be represented through convolution
- An example is the median filter

Median filter

- The median filter is solely determined by the size of the moving window (e.g. 3x3, 5x5, etc.)
- The window is shifted on the image as in linear filtering. The output pixel will be the **median** of the pixel values in the window
- Used for smoothing (noise reduction), especially in the presence of “salt & pepper” noise

22	23	24	25	26
27	28	29	30	31
32	33	34	35	36
37	38	39	40	41
42	43	44	45	46

Input image I

=

X	X	X	X	X
X	28			X
X				X
X				X
X	X	X	X	X

Output image J

Median filter

Built-in Matlab function: `medfilt2`

Original image
(with added “salt & pepper” noise)

Median filtered image

Hand-crafted vs learned filters

- The filters seen so far are hand-crafted: their design (including the choice of parameters like the size of the kernel) is usually based on heuristic
- Machine learning allows to “learn” the filter parameters from a set of input and output images (e.g. noisy images as input and smoothed images as output)
- This aspect is particularly evident in deep learning techniques

Edge detection

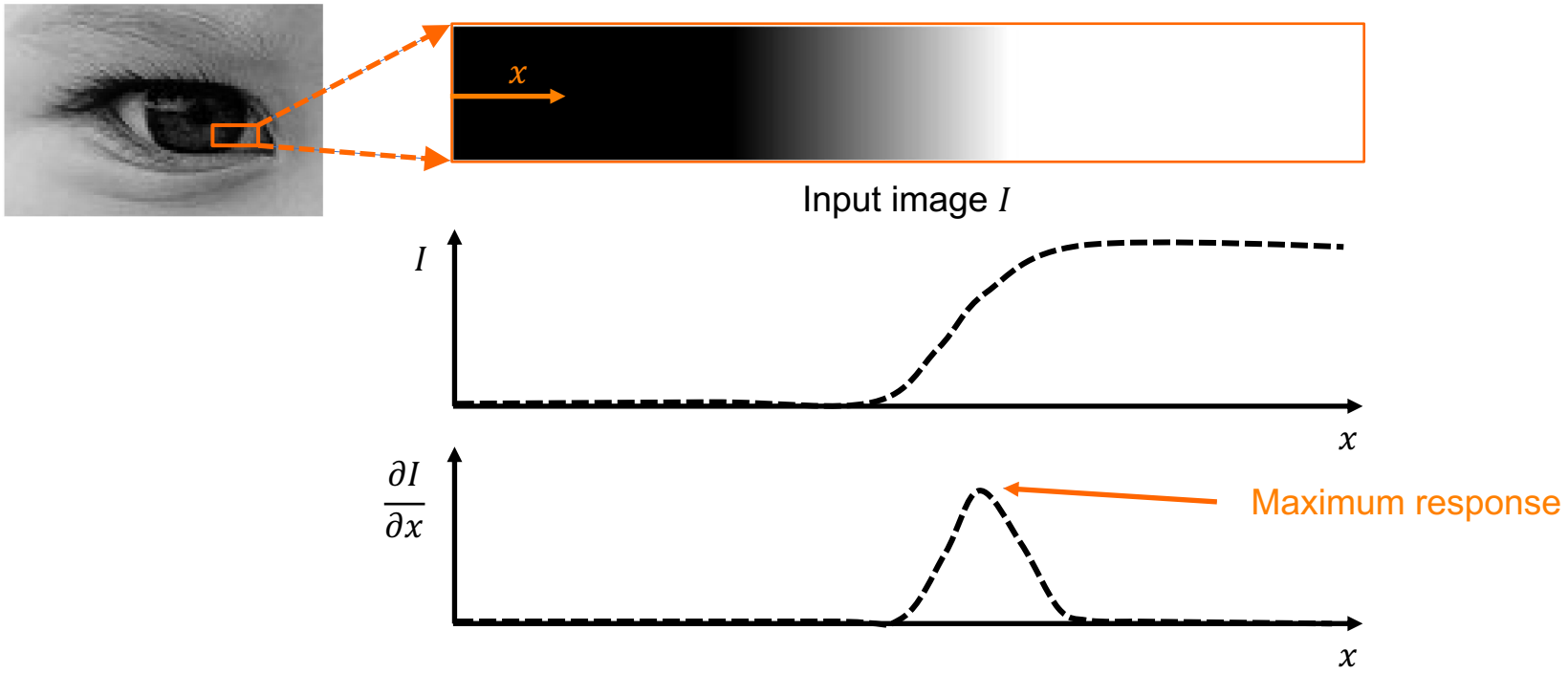
- Edge detection is a fundamental problem in CV. It consists in locating the edges, i.e. strong discontinuities in pixel intensity and/or colour in the image
- This can be useful for distinguishing objects from their surroundings (both for humans and for computers)

Edge detection

- A naïve edge detection algorithm is as follows. For each pixel in the image:
  - Analyse each of the 8 pixels surrounding it
  - Detect both the darkest and the lightest pixels
  - If $(I_{lightest} - I_{darkest}) > threshold$, then set the output pixel to 1; else set it to 0
- There are many techniques to perform it, including
  - gradient-based
  - Laplacian-based
- Usually performed on grayscale images

Gradient-based edge detection

- Edge detection can be performed using first spatial derivatives of the image intensity, i.e. evaluating the (**absolute**) **slope of image intensity** at each pixel (and for each dimension, x and y)
- High value of the first derivative along x for one pixel
  → high change in intensity along x
  → high chances of having an edge along x in that pixel



First derivative

- The first derivative can be approximated using a Taylor expansion:

$$I(x+h,y) = I(x,y) + h\frac{\partial I(x,y)}{\partial x} + O(h^2)$$ Forward scheme difference

$$I(x-h,y) = I(x,y) - h\frac{\partial I(x,y)}{\partial x} + O(h^2)$$ Backward scheme difference

- If we subtract the two, and consider the limit for $h \to 0$, we can write

$$I(x+h,y) - I(x-h,y) = 2h\frac{\partial I(x,y)}{\partial x}$$

$$I_x \equiv \frac{\partial I(x,y)}{\partial x} = \frac{I(x+h,y) - I(x-h,y)}{2h}$$ Central scheme difference

- The smallest h we can take is one pixel: to implement the derivative of $I(x,y)$ with respect to x at a pixel (x,y), we simply take the intensity of the pixel to the right, subtract the intensity of the pixel to the left, and divide by 2
- For y, we have

$$I_y \equiv \frac{\partial I(x,y)}{\partial y} = \frac{I(x,y+h) - I(x,y-h)}{2h}$$

First derivative kernels

- A simple first derivative kernel with respect to x is

0.5 x

-1	0	1

First derivative kernel
(x direction)

```
K = 0.5*[-1 0 1];
Ix = imfilter(I,K);
imshow(Ix,[]);
```

I_x

First derivative kernels

- A simple first derivative kernel with respect to y is

0.5 x

-1
0
1

First derivative kernel
(y direction)

Transpose

```
K = 0.5*[-1 0 1]';
Iy = imfilter(I,K);
imshow(Iy,[]);
```

I_y

Image gradient

- The image gradient ∇I is a vector consisting of the two first derivatives computed at each point in the image

$$\nabla I(x,y) \equiv (I_x, I_y) = \left(\frac{\partial I(x,y)}{\partial x}, \frac{\partial I(x,y)}{\partial y}\right)$$

- Built-in Matlab function: `imgradient`

- It is usually represented with two images:
  - Magnitude:

$$|\nabla I(x,y)| = \sqrt{I_x^2 + I_y^2}$$

  - Direction/angle:

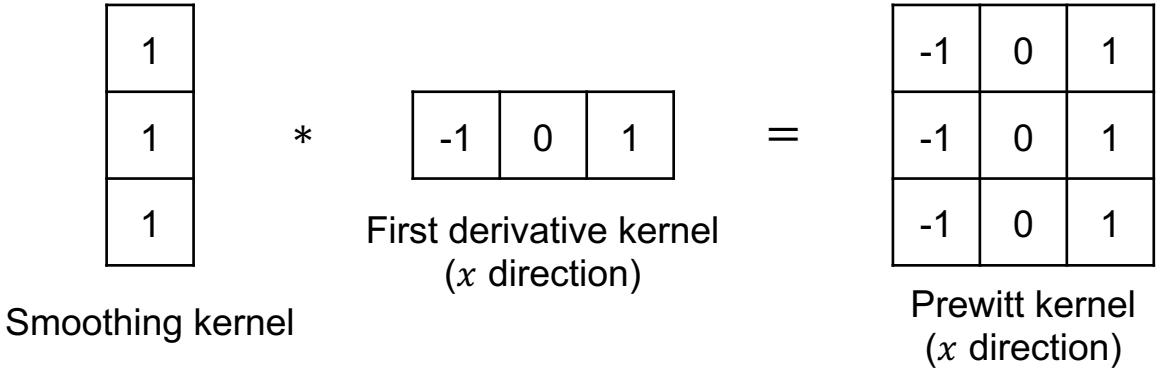
$$\theta(x,y) = \arctan(I_y, I_x)$$

$|\nabla I(x,y)|$

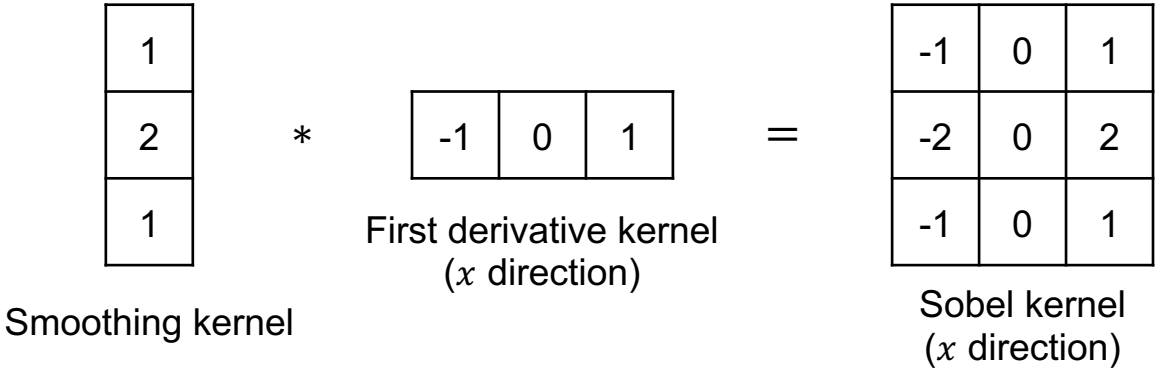
$\theta(x,y)$

Common first derivative kernels

- Derivatives are highly sensitive to noise in the images
- It is common to first smooth the image and then apply the derivative kernel:
  - Prewitt filter:



  - Sobel filter:



Derivative of Gaussian

- The Gaussian filter is also a common choice for smoothing
- Using the associative property of convolution

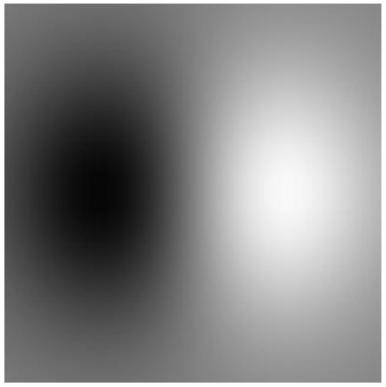
$$K_2 * (K_1 * I) = (K_2 * K_1) * I$$

we can create a single kernel that performs two operations:

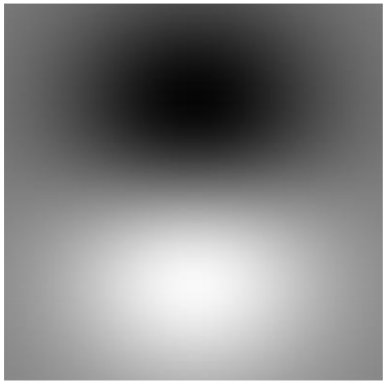
$$K_1 = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

$$K_2 = \frac{\partial}{\partial x}$$

- $K_3 = K_2 * K_1$ is the Derivative of Gaussian kernel (along x)



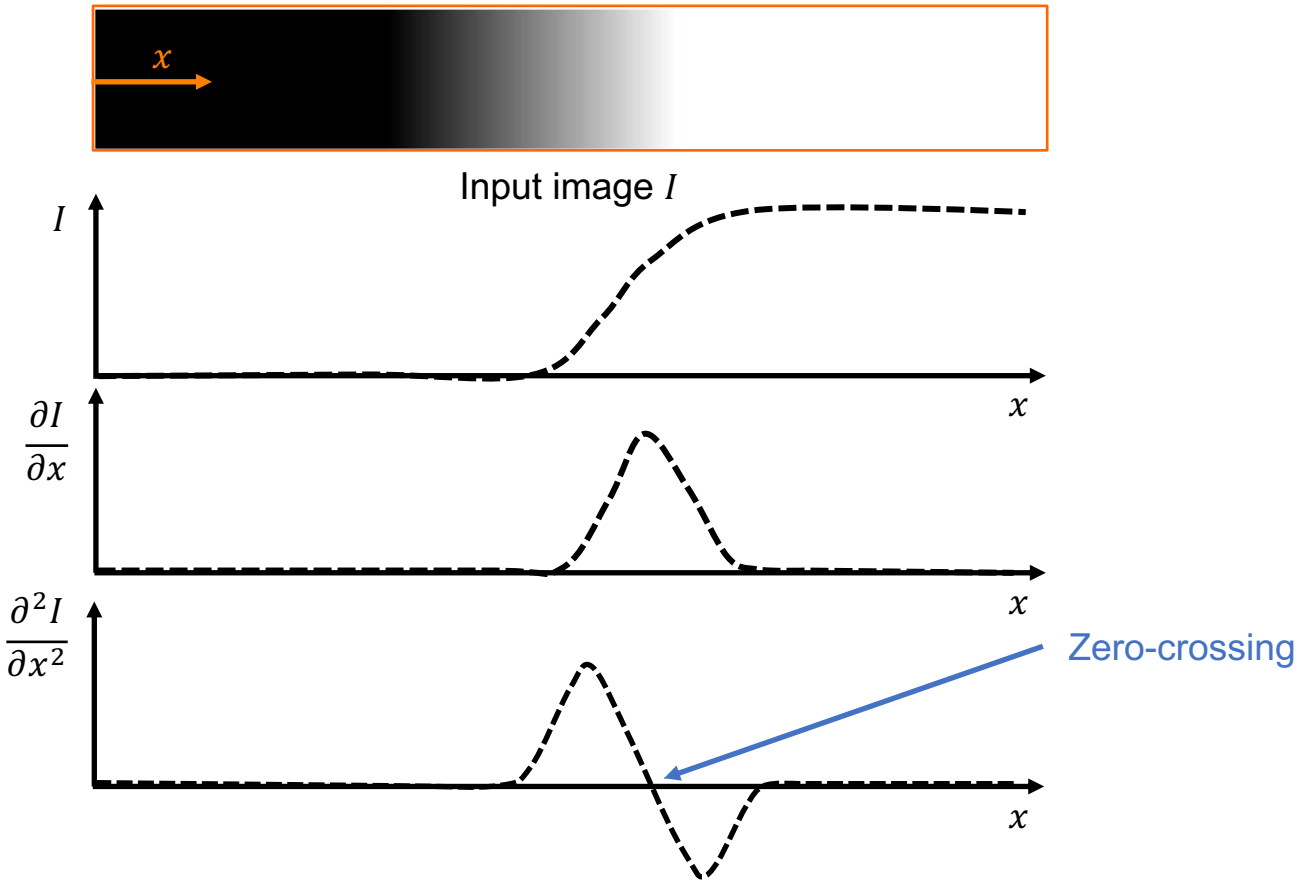
Derivative of Gaussian
(x direction)



Derivative of Gaussian
(y direction)

Laplacian-based edge detection

Edge detection can be performed using second spatial derivatives of the image intensity, i.e. looking for their **zero-crossings** at each pixel (along both dimensions)



Second derivative kernels

- If we consider the limit for $h \to 0$, the second derivative can be approximated with the following:

$$\frac{\partial^2 I(x,y)}{\partial x^2} = \frac{I(x+h,y) - 2I(x,y) + I(x-h,y)}{h^2}$$

- Since we are looking for the zero-crossings, we can sum the two derivatives
- The sum of the two second derivatives is the **Laplacian** ΔI of the image:

$$\Delta I(x,y) = \frac{\partial^2 I(x,y)}{\partial x^2} + \frac{\partial^2 I(x,y)}{\partial y^2}$$

Image Laplacian

The kernel can be implemented as

0	0	0
1	-2	1
0	0	0

Second derivative kernel (x direction)

$+$

0	1	0
0	-2	0
0	1	0

Second derivative kernel (y direction)

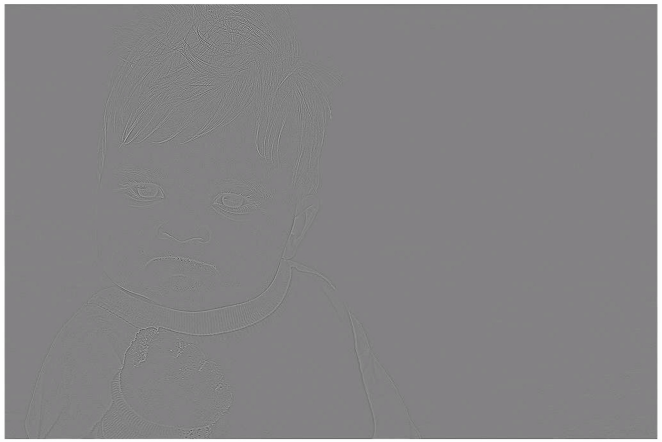
$=$

0	1	0
1	-4	1
0	1	0

Laplacian kernel

```
K = fspecial('laplacian',0)
J = imfilter(I, K);
imshow(J,[]);
```

ΔI



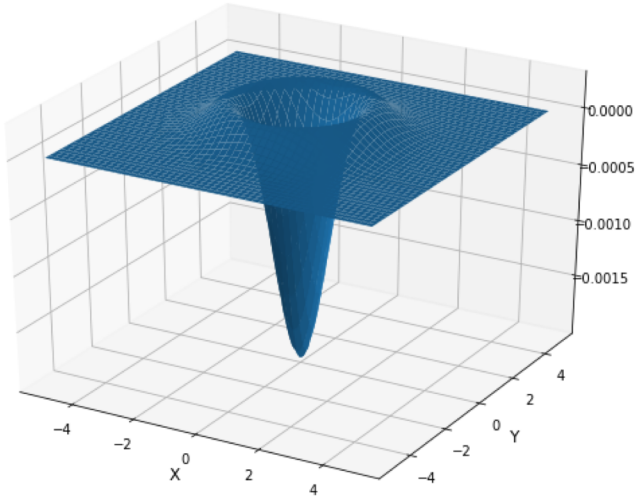
Laplacian of Gaussian

- The second derivative is more sensitive to noise than the first derivative
- We can also smooth the image using a Gaussian kernel, which results in a Laplacian of Gaussian filter:

$$K_1 = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

$$K_2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

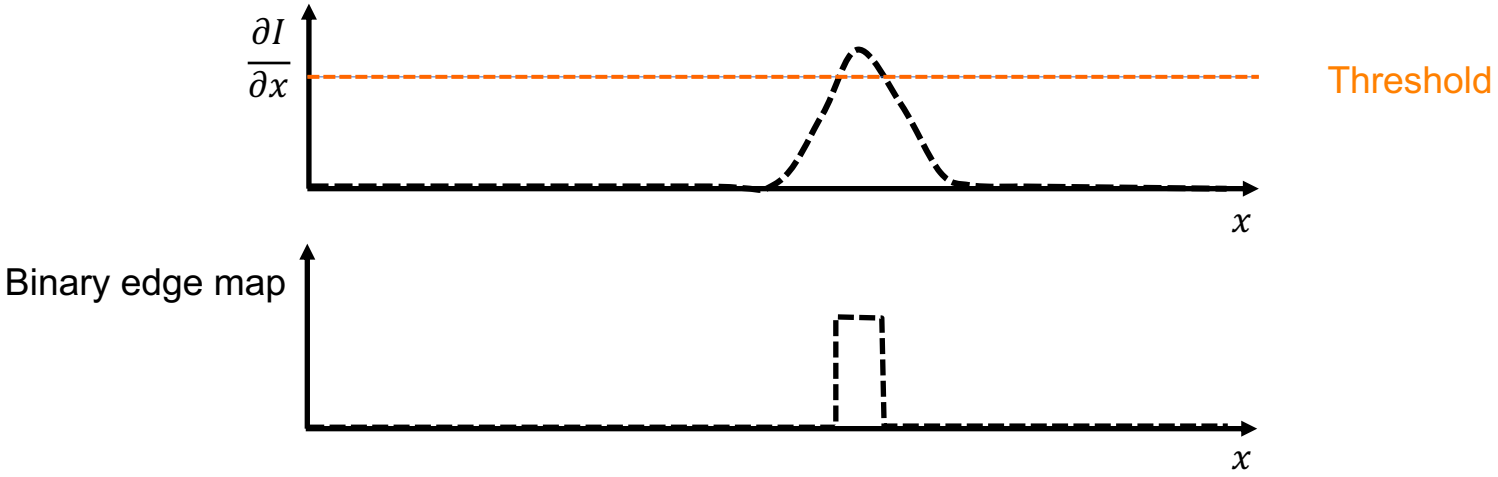
- $K_3 = K_2 * K_1$ is the Laplacian of Gaussian (LoG) kernel
- Built-in Matlab function: `K = fspecial('log', hsize, sigma)`



LoG kernel

Towards binary edge maps

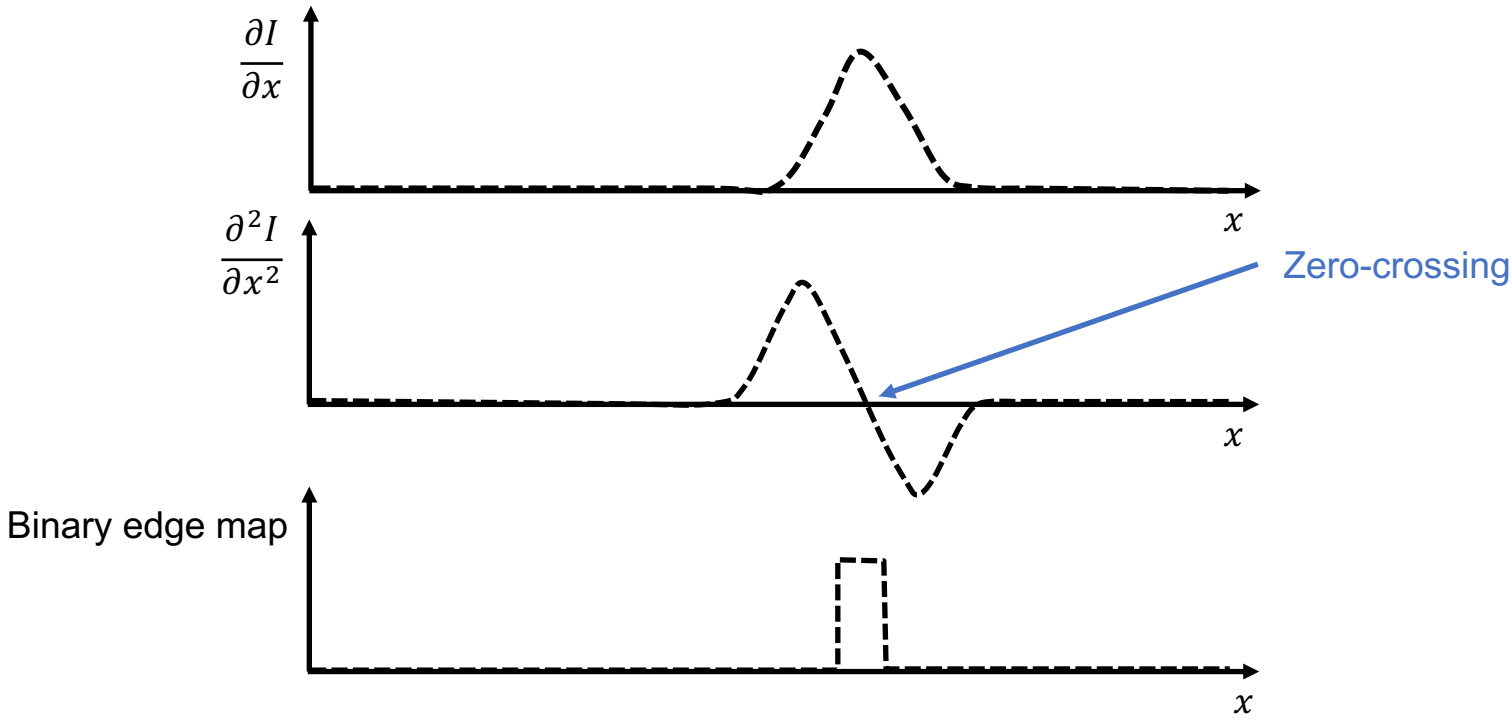
- In most applications, a binary edge map is required:
  0s everywhere, and 1s in correspondence of an edge
- **Edge-based methods** look for the points of maximum-response: the higher the value of the gradient, the more likely it is for the pixel to belong to an edge
- To create a binary edge map, a threshold is usually used:



- This is not a straightforward process:
  - A low threshold will be susceptible to noise and detect irrelevant edges
  - A high threshold will miss subtle edges or result in fragmented ones

Towards binary edge maps

- Laplacian-based methods, on the other hand, look for zero-crossings to identify edges:



- Theoretically, there is no need to decide a threshold
- However, second derivatives are very sensitive to noise, so in the end some manual thresholds can still be needed

Canny edge detector

Canny [1] proposed the following criteria for an edge detector:

- **Good detection**
  There should be a low probability of failing to mark real edge points, and low probability of falsely marking non-edge points
- **Good localisation**
  The points marked as edge points by the operator should be as close as possible to the centre of the true edge
- **Single response**
  Only one response to a single edge

[1] J. Canny. A Computational Approach to Edge Detection. IEEE TPAMI, 1986

Canny edge detector

Canny [1] proposed a gradient-based edge detection algorithm with the following main steps:

1. Perform Gaussian filtering to suppress noise
2. Calculate the gradient magnitude and direction
3. Apply non-maximum suppression (NMS) to get a single response for each edge
4. Perform hysteresis thresholding to find potential edges

[1] J. Canny. A Computational Approach to Edge Detection. IEEE TPAMI, 1986

Canny edge detector

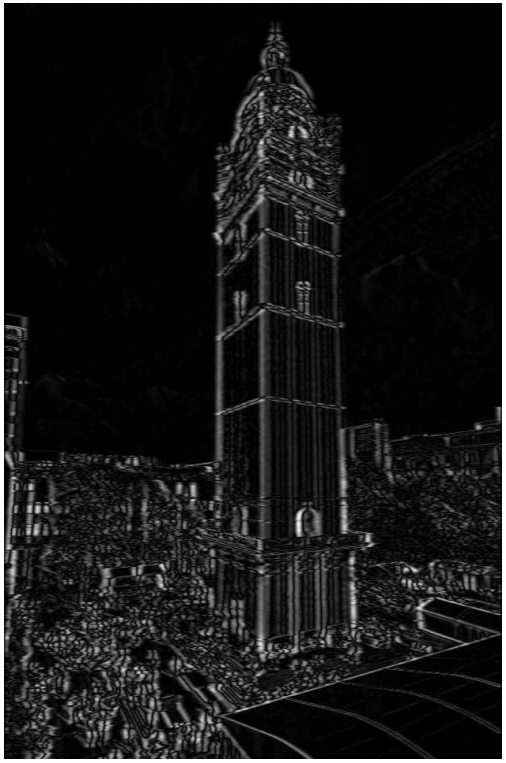
1. Perform Gaussian filtering to suppress noise:
   The choice for the parameter σ in the Gaussian kernel has a major impact on the result of the detector:
   - High $\sigma \rightarrow$ detect large scale structures
   - Low $\sigma \rightarrow$ detect fine details
2. Calculate the gradient magnitude and direction

These two steps can be done in two ways:

- In two separate steps: Gaussian filter + Prewitt or Sobel filter
- In one step using the derivative of Gaussian filters

Canny edge detector

Input image



Gradient magnitude

Gradient direction

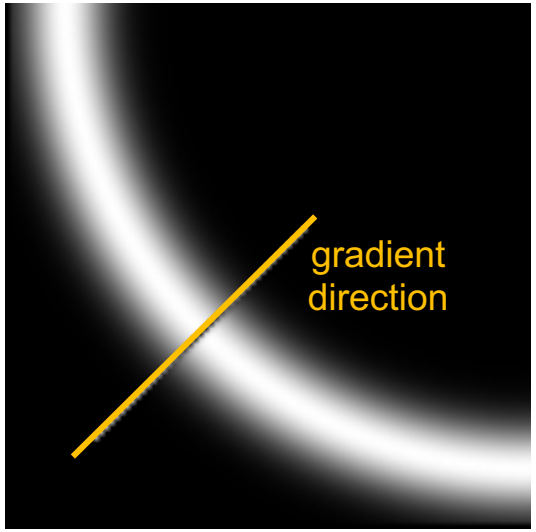
Canny edge detector

3. Apply non-maximum suppression (NMS) to get a single response for each edge:
   NMS is an edge thinning technique: suppress all the gradient values (by setting them to 0) except the local maxima, which indicate locations with the sharpest change of intensity value

Steps for NMS:

- For each pixel q, check whether the gradient magnitude is a local maximum along gradient direction
- If it is, leave magnitude untouched; if it is not, set magnitude to 0

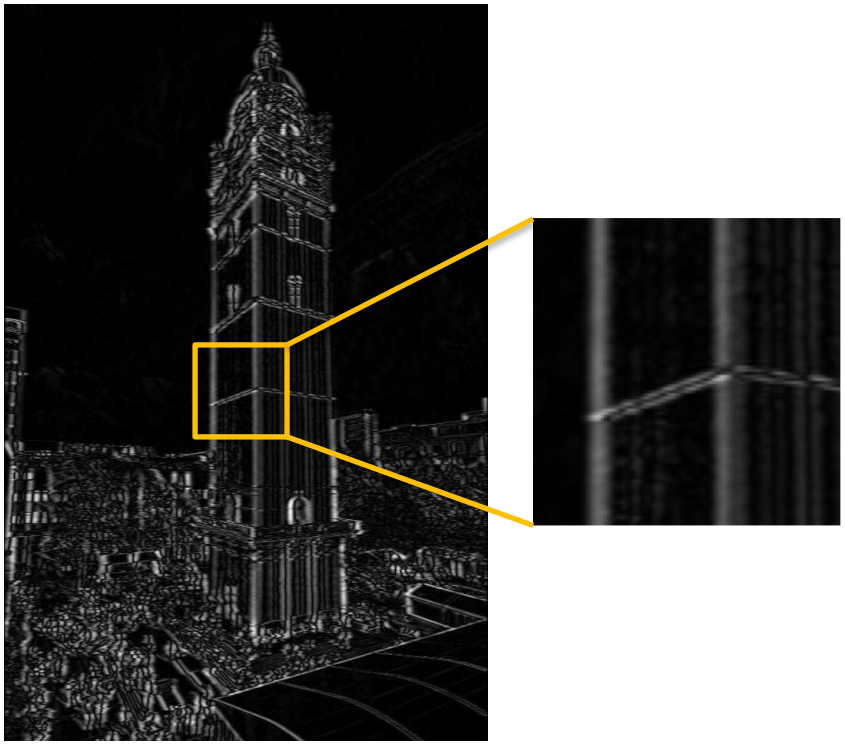
Usually, the gradient directions are quantized into a set of discrete ones:
horizontal, vertical, diagonal and anti-diagonal



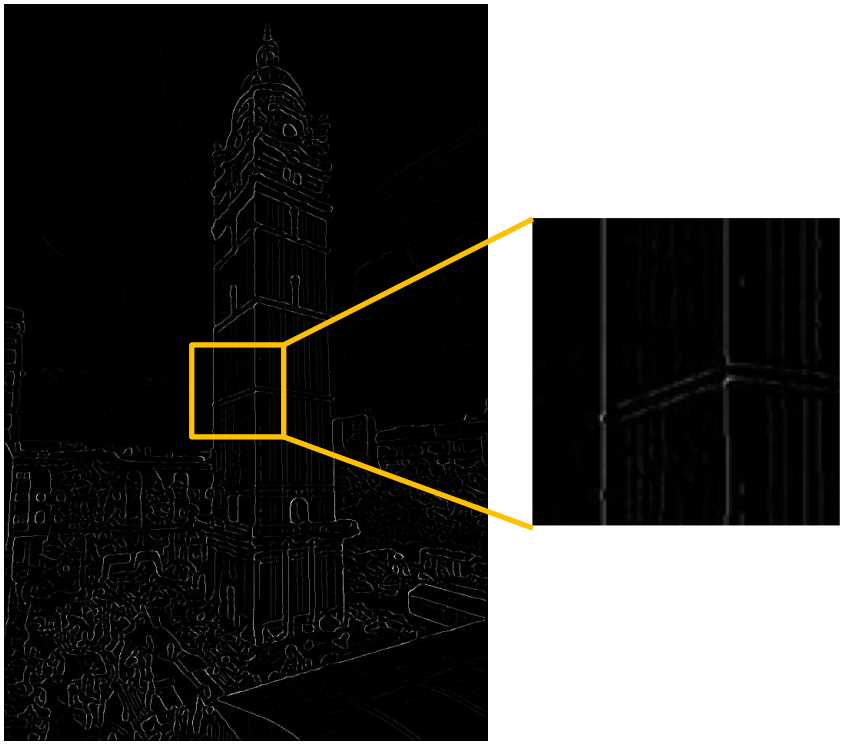
Gradient magnitude

Canny edge detector

3. Apply non-maximum suppression (NMS) to get a single response for each edge:



Before NMS

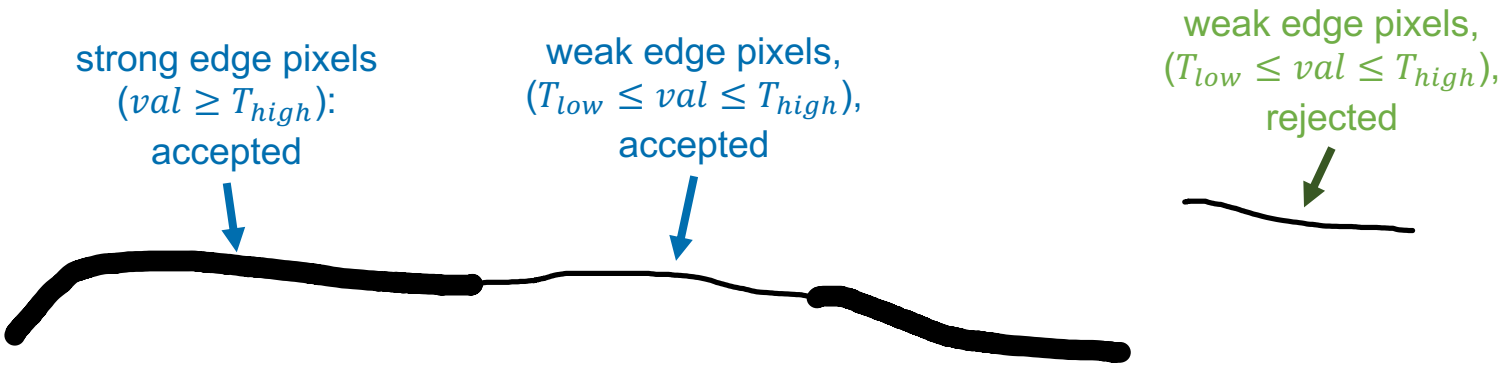


After NMS

Canny edge detector

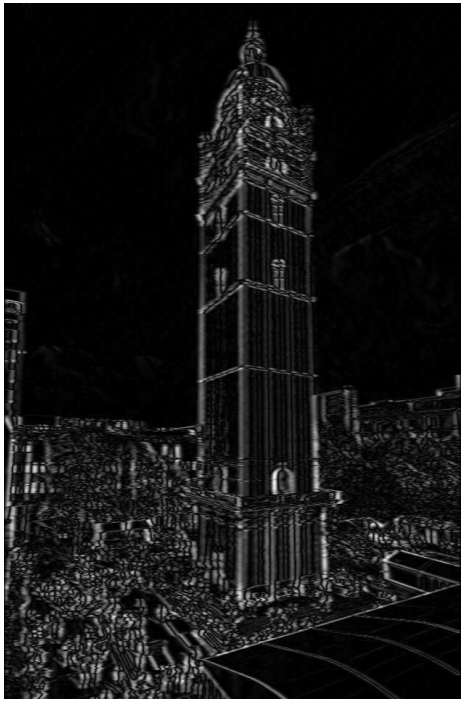
4. Perform hysteresis thresholding to find potential edges

- Hysteresis is the lagging of an effect, an inertia
- Hysteresis thresholding requires the definition of two thresholds: T_{high}, T_{low}
- In edge detection, hysteresis thresholding means that
  - If a pixel gradient magnitude is larger than T_{high}, it is accepted as an edge
  - If a pixel gradient magnitude is lower than T_{low}, it is rejected
  - If in between, it will be accepted if it is connected to an edge

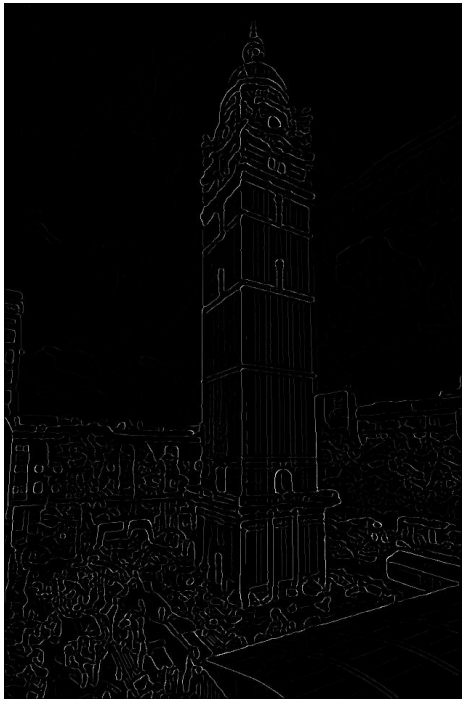


Canny edge detector

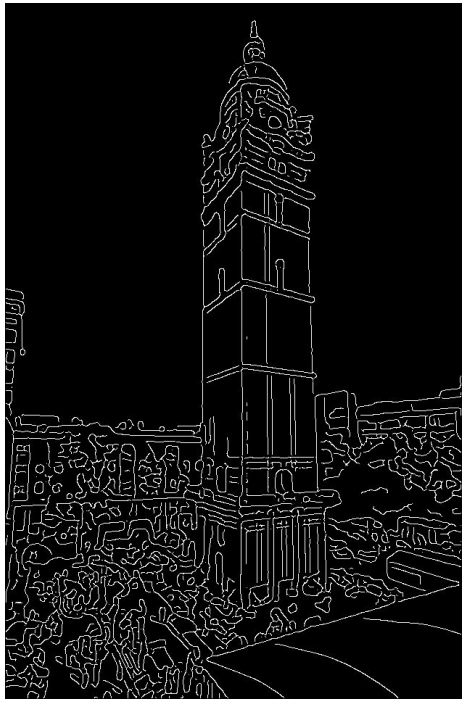
Input image



Gradient magnitude



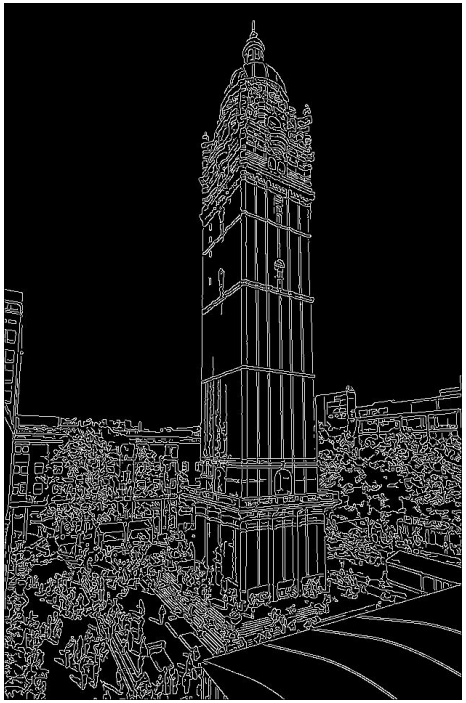
NMS



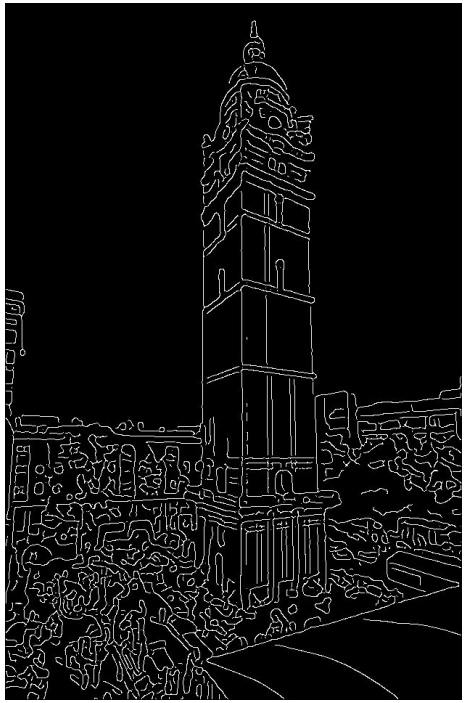
Hysteresis thresholding

Canny edge detector

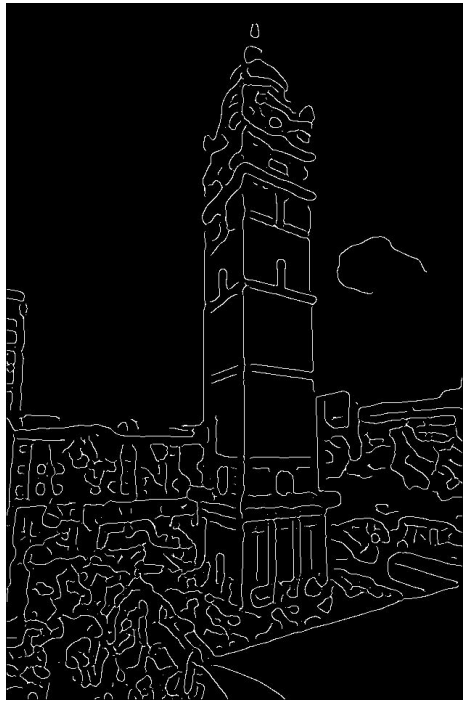
Effect of the Gaussian kernel parameter:



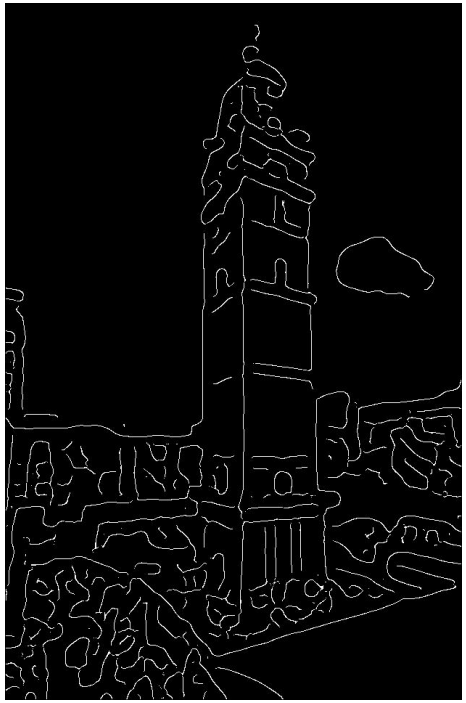
$\sigma = 1$



$\sigma = 3$



$\sigma = 5$



$\sigma = 7$

- High $\sigma \rightarrow$ detect large scale structures
- Low $\sigma \rightarrow$ detect fine details

Additional resources

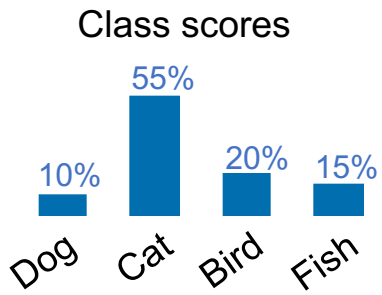
All of the presented edge detection strategies are implemented in a single function in Matlab: `edge`

Common CV tasks

Image classification

- Identify the class of the object shown in the image
- Often (but not necessarily) based on class scores estimation

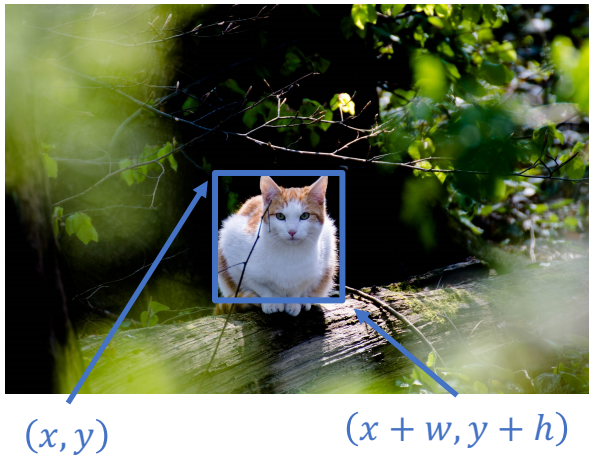
Class label: “Cat”



Common CV tasks

Image classification & localization

- Identify the class and the location of the object (regression of bounding box coordinates)
- Assume only one object per image



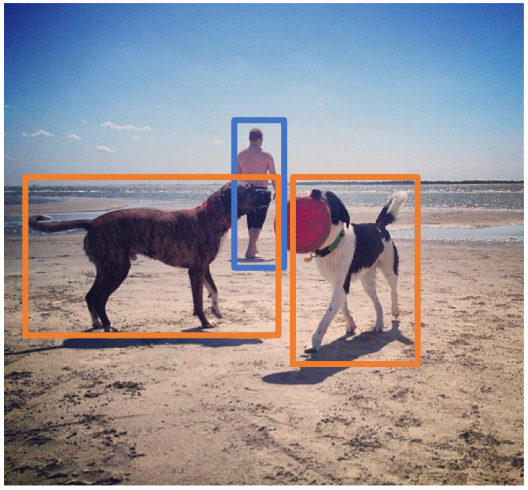
Class. & local.

- Class label: "Cat"
- Bounding box coordinates: (x, y, w, h)

Common CV tasks

Object detection

- Identify the class and the location of the objects in the image
- Assume **more than one object** per image



Object detector

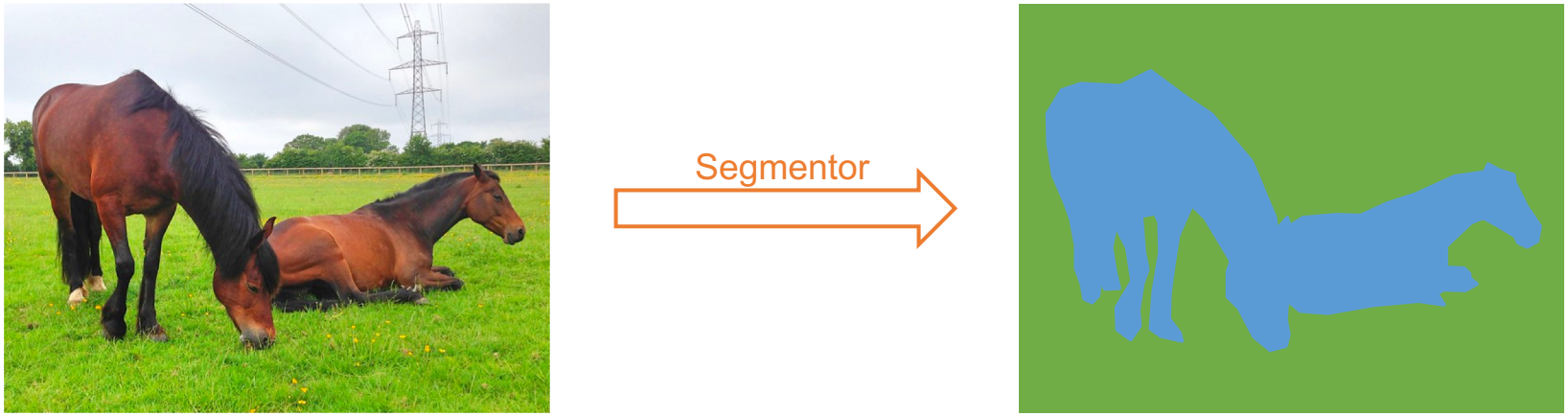
For each object in the image:

- Class label: "Dog"
- Bounding box coordinates: (x, y, w, h)

Common CV tasks

Semantic segmentation

- Assign a label to each pixel

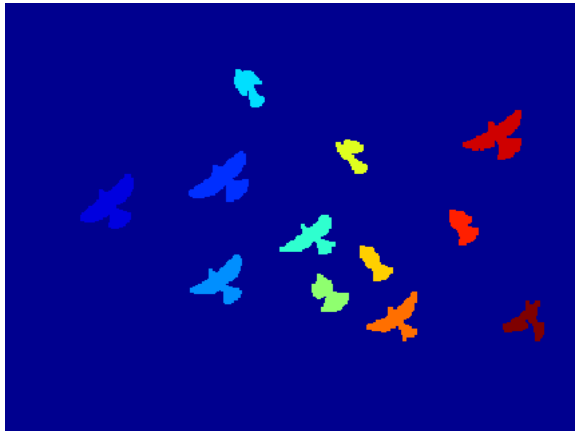


Common CV tasks

Instance segmentation

- Assign a label to each pixel
- **Distinguish** between different **instances**
- Combination of object detection and semantic segmentation

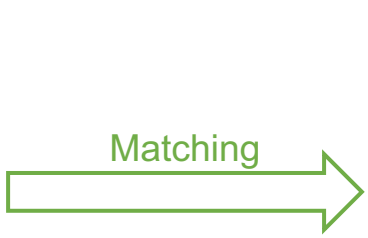
Instance segm.



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

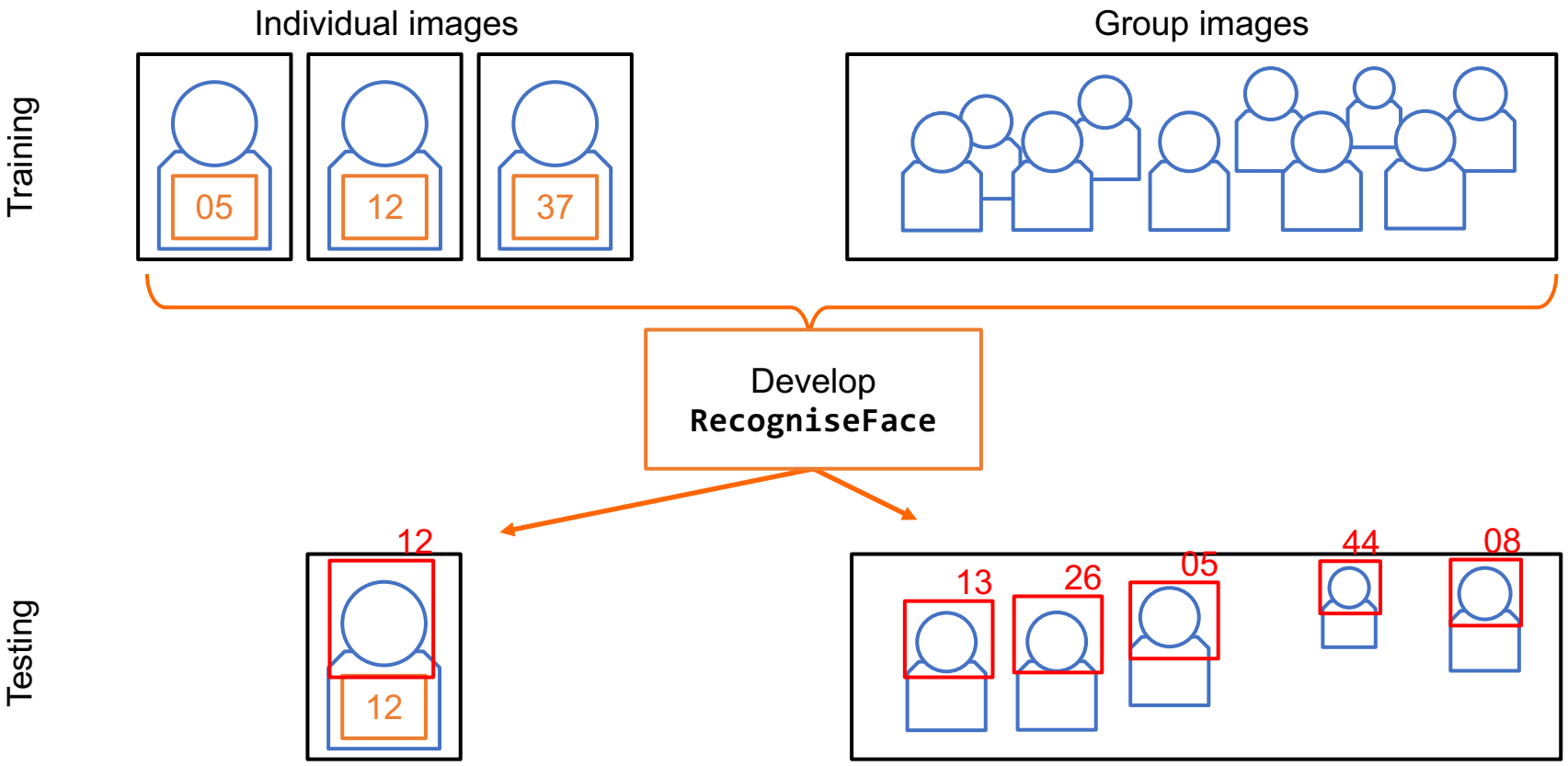
Overview of next week's lecture

Image segmentation:

- Intensity-based methods
  - Thresholding
  - K-means clustering
- Watershed
- Active contours
  - Snakes
  - Level-sets
- Graph cuts
- Superpixels
- Quantifying results: how to tell if we achieved a good segmentation

Coursework: face recognition

- Deadline (UGs & PGs): **Thursday 30 April 2020, 5pm**
- Coding: **Matlab** (strongly recommended). Python solutions will also be accepted (but support from academic staff limited to Matlab)
- Develop a function **`RecogniseFace`** that identifies faces and assigns IDs



Coursework: face recognition

- **`RecogniseFace`** must include different solutions, i.e. combinations features-classifiers
- The training and evaluation procedures will be carried out by you, starting from the data split
- Short videos (i.e. “Live Photos”) have been added too: extract frames from them if you believe you need additional training/evaluation data
- At least for the best implemented combination of features-classifier, provide performance metrics (e.g. % of accurately classified faces in a portion of the evaluation set)
- Your **`RecogniseFace`** will be tested on unseen individual/group images (not available on the folders on Moodle)
- **Additional task for PGs**: if prompted, allow for **`RecogniseFace`** to modify the detected faces (e.g. cartoonify, add content e.g. beard, etc.)

Coursework: face recognition

- You will need to submit:
  - **`RecogniseFace`** and other developed code (commented and referenced!)
  - Readme.txt to run your code
  - Trained models
  - Report describing your approach and showing results
  - Short (**max 3 mins**) screen capture video showing your function running on images from your evaluation set
- You **must not** submit the training/evaluation dataset
- Package all the listed items in a zip and submit to Moodle. Given the 200MB limit, the **models alone** can be uploaded on personal cloud space (e.g. Dropbox) and **linked in the readme.txt file**
- **Your models will not be retrained**. Your **`RecogniseFace`** function must run as is, so package it properly and test it before submission
- Don’t freak out if you don’t get very high accuracy in some cases!