Feature extraction

Outline

Basic feature extraction

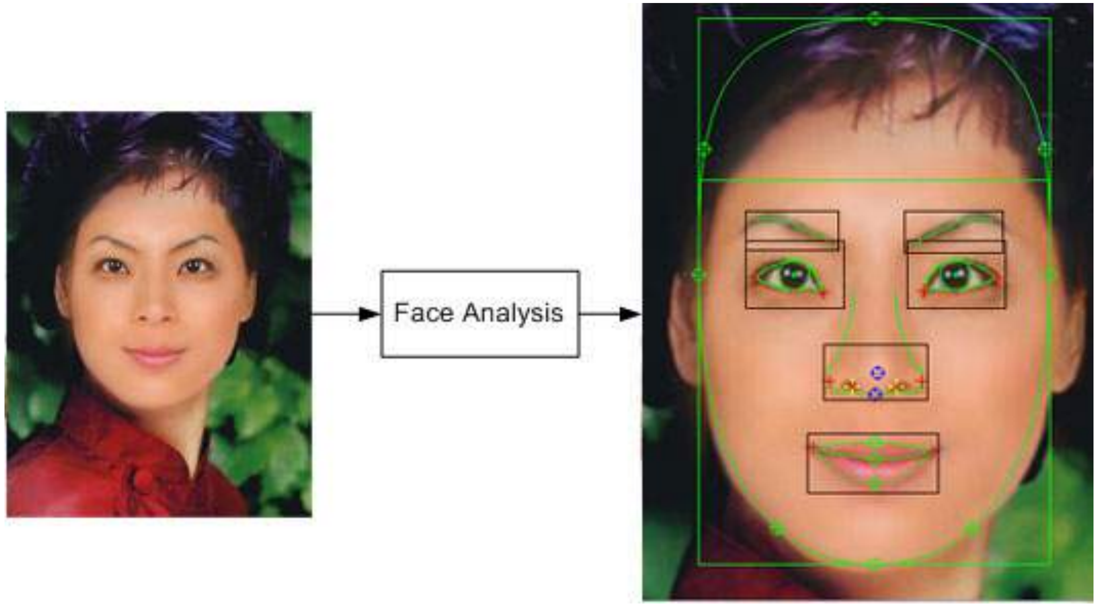
- *Points*, *lines* and *edges*
- How to compute *derivatives of images*
- *First-* and *second-derivatives* approaches
- Formulation as *spatial filters*

More advanced examples

- *Canny* edge detector
- Lines and circles by *voting procedures*
- *Known* shapes/objects

What is feature extraction?

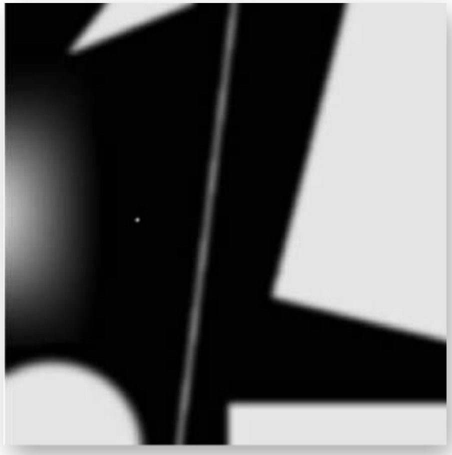
- Recognize and **extract specific features of an image** from the raw data that can be used as *input to a learning algorithm*



- Here we focus on **low level** feature extraction
  - **Basic features** that can be extracted **without any shape information**
    - e.g. *previous example* requires prior information on the objects to extract
  - Features of interest: *points*, *lines* and *edges*
  - Such features will be then used as **input to more advanced segmentation algorithms**

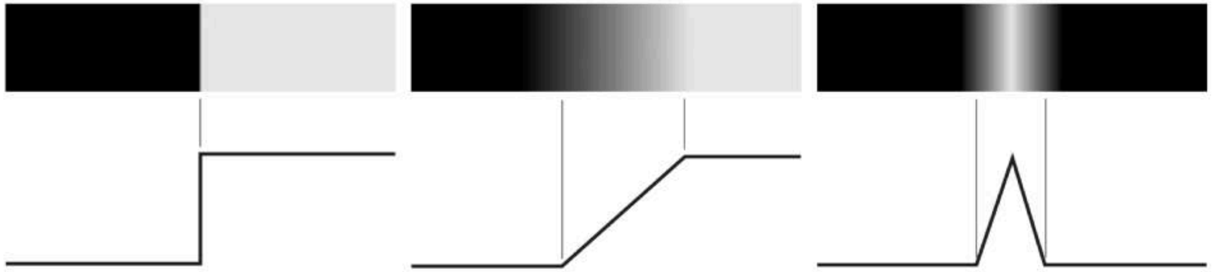
Features of interest (1/2)

- We are interested in **three basic types** of image features:
  - **Edges**
    - Pixels at which the intensity of an image changes abruptly
  - **Lines**
    - May be seen as *1-pixel-thick edges*
  - **Isolates points**
    - Line whose *length* and *width* are equal to *1 pixel*



- **Note**: all characterized by *sharp, local changes* in intensity

- **Typical edges** found in images



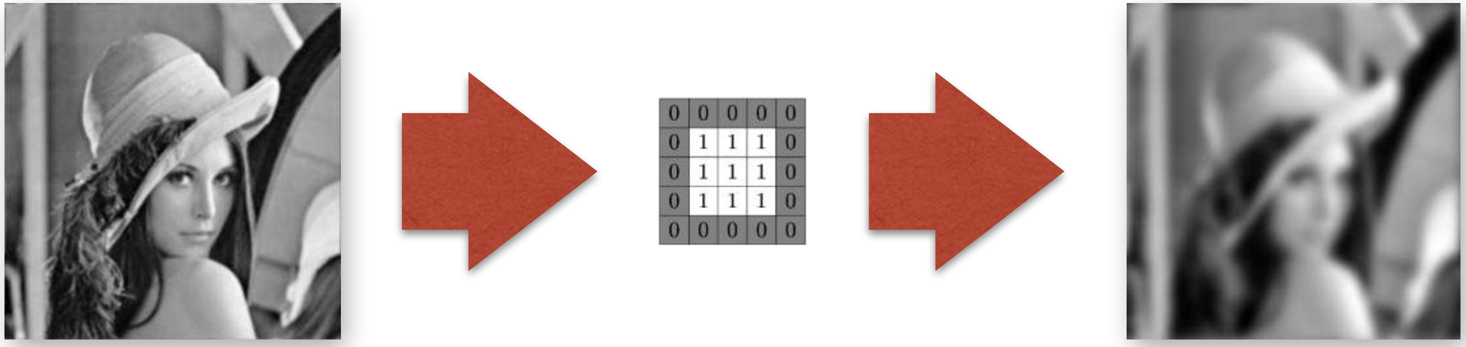
step edge (“ideal”)

ramp edge

roof edge

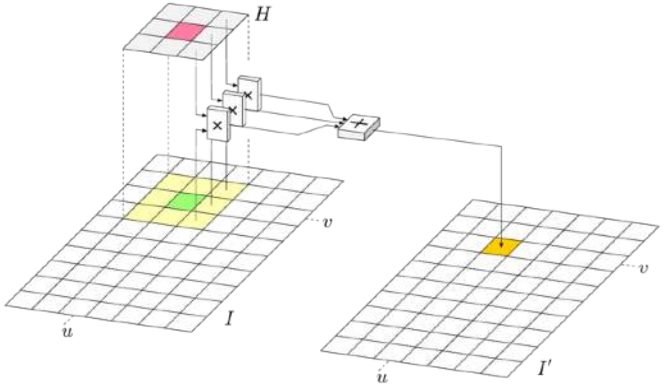
Features of interest (2/2)

- How to extract these features? **Analogy** with local averaging
  - Local averaging *smooths* an image



0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

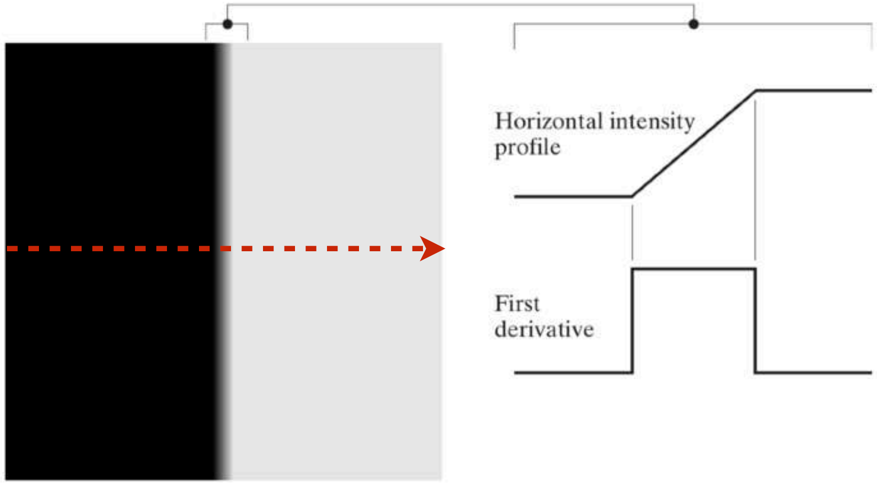
  - *Averaging* is analogous to *integration*



- No surprise that *abrupt, local changes* in intensity can be **detected using derivatives!**

Image features and derivatives (1/2)

- Analyze a **single line in a typical image**
  i.e. 1D signal

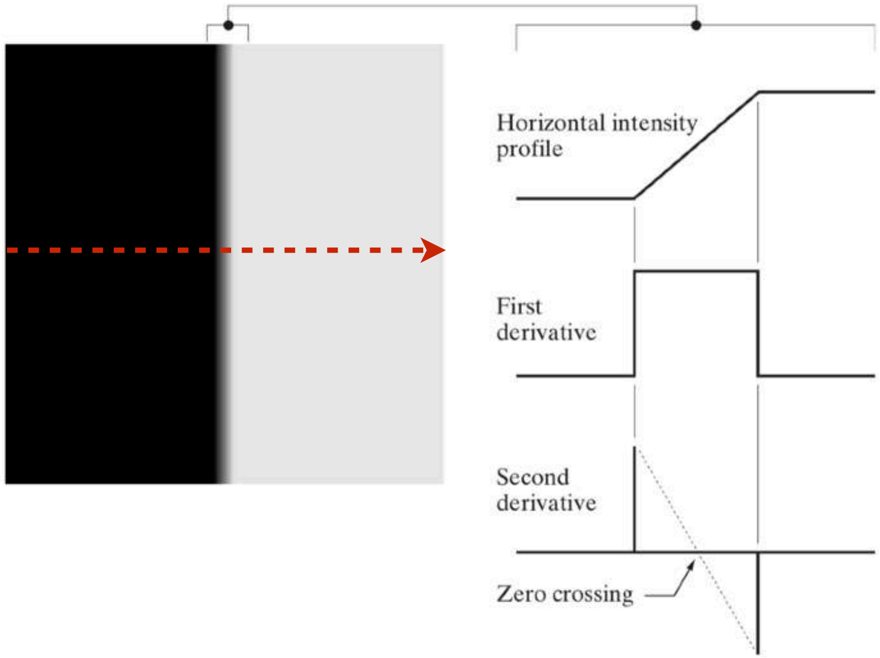


- **1st derivative,** $f'(x)$
  - **zero** in areas of constant intensity
  - **positive** at the onset of the ramp and **positive** on the ramp

- *Magnitude of* $f'(x)$ may be used to identify edge pixels
  - **NB**: an edge is detected with "thick" responses

Image features and derivatives (2/2)

- Analyze a **single line in a typical image**
  i.e. 1D signal



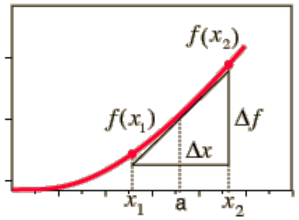
- **2nd derivative, $f''(x)$**
  - **zero** in areas of constant intensity and **zero** on the ramp
  - **positive** at the onset of the ramp but **negative** at the end (*double-edge* effect and *zero-crossing* point)

- *Sign change of $f''(x)$* can be used to identify edge pixels
  - **NB**: an edge is detected with "much thinner" responses
  - Higher sensitivity to finer details

How do we compute image derivatives? (1/5)

- For a **continuous 1D function**, the first-order derivative is

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$



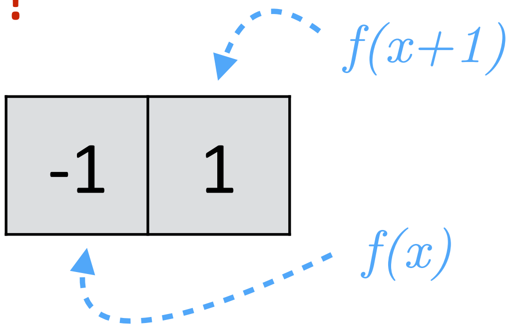
- For **discrete functions**, it can be approximated with *finite differences*

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

  - Δx is dictated by the discretization step (sampling) in x
  - NB: Δx = 1 pixel

- It can be implemented as a **linear spatial filter**!

$$f'(x) = \frac{df}{dx} \approx f(x+1) - f(x)$$



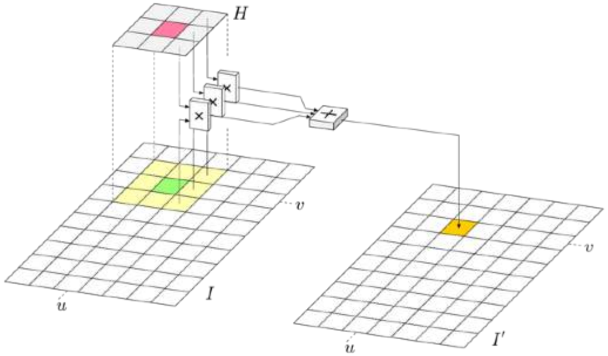
-1	1

How do we compute image derivatives? (2/5)

- **Recall** the definition of *spatial filtering*

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

$$R = w_1 z_1 + w_2 z_2 + \cdots + w_9 z_9 = \sum_{k=1}^{9} w_k z_k$$



  - A *filter* H is defined as a 3x3 matrix
  - The *values* w_k define the function implemented by the filter
  - *Response of the filter* (R) at the central pixel computed by *weighted average*

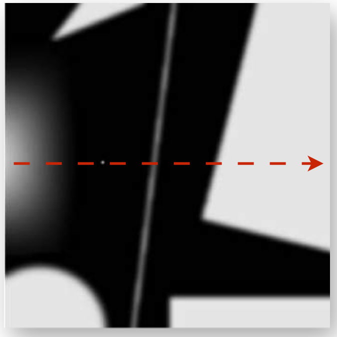
- According to our *previous conventions*, the filter for $f'(x)$ can be written as

-1	1

0	0	0
0	-1	1
0	0	0

How do we compute image derivatives? (3/5)

- Example



-1	1
x	$x+1$

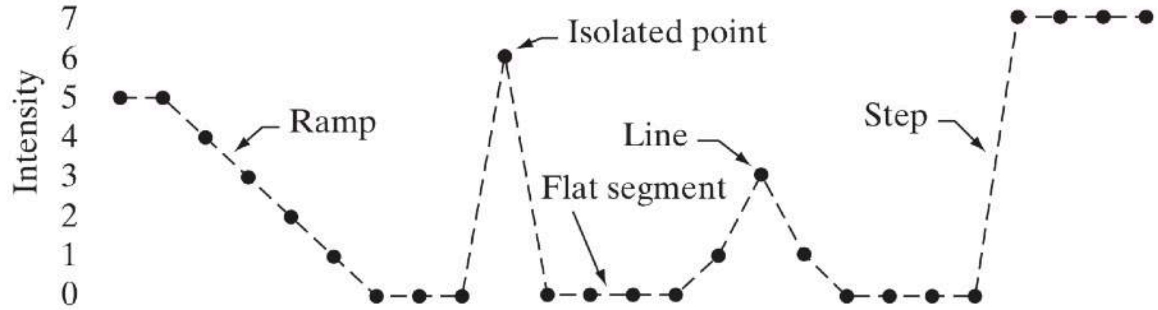


Image strip	5	5	4	3	2	1	0	0	0	6	0	0	0	0	1	3	1	0	0	0	0	7	7	7	7	•	•
First derivative		−1	−1	−1	−1	−1	0	0	6	−6	0	0	0	1	2	−2	−1	0	0	0	7	0	0	0			
Second derivative		−1	0	0	0	0	1	0	6	−12	6	0	0	1	1	−4	1	1	0	0	7	−7	0	0			

- Similarly for the **second-order derivative**

$$
\begin{aligned}
f''(x) = \frac{df'(x)}{dx} &= f'(x+1) - f'(x) \\
&\approx f(x+2) - f(x+1) - f(x+1) + f(x)
\end{aligned}
$$

  - **NB:** this expansion is about point $x+1$, we need to center it at x

$$f''(x) = \frac{d^2 f}{d^2 x} \approx f(x+1) - 2f(x) + f(x-1)$$

1	-2	1

How do we compute image derivatives? (4/5)

- With **discrete functions**, *left* and *right derivatives* don't match
  - *Forward* differences

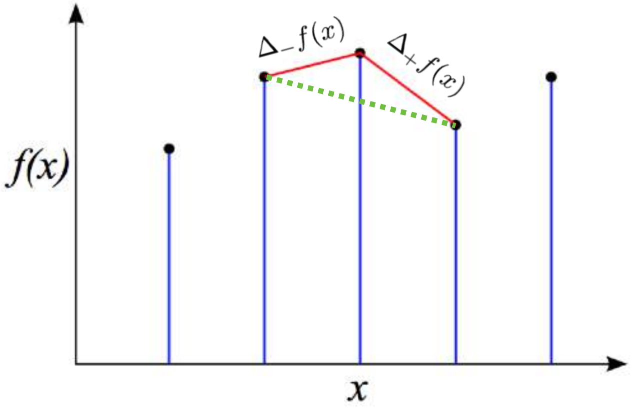
$$\Delta_+ f(x) = f(x+1) - f(x)$$

(right slope)

  - *Backward* differences

$$\Delta_- f(x) = f(x) - f(x-1)$$

(left slope)



- **Solution**: take the average
  - *Central* differences

$$\Delta f(x) = \frac{1}{2}\Big(\Delta_+ f(x) + \Delta_- f(x)\Big) = \frac{1}{2}\Big(f(x+1) - f(x-1)\Big)$$

(average slope)

- Central differences as **spatial filter**

$$H = \frac{1}{2}\,[-1 \;\; 0 \;\; 1]$$

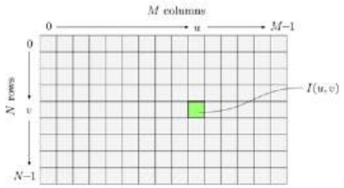
$f(x{+}1)$

$f(x{-}1)$

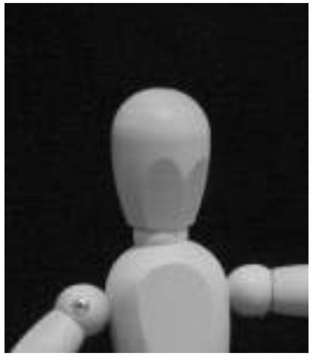
How do we compute image derivatives? (5/5)

Derivatives of **images**

- Images have **two dimensions**
- We can take derivatives of $I(u,v)$ with respect to u or v



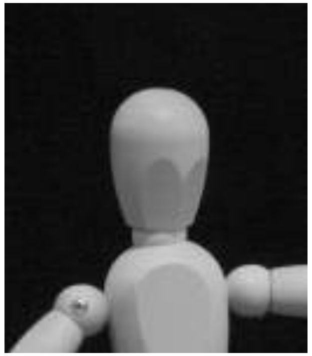
derivative along u



$\otimes \; \frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} =$

$\frac{\partial f}{\partial u}$ or $\nabla_u I(u,v)$

derivative along v



$\otimes \; \frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}^T =$

$\frac{\partial f}{\partial v}$ or $\nabla_v I(u,v)$

NB: the results are <u>two images</u>

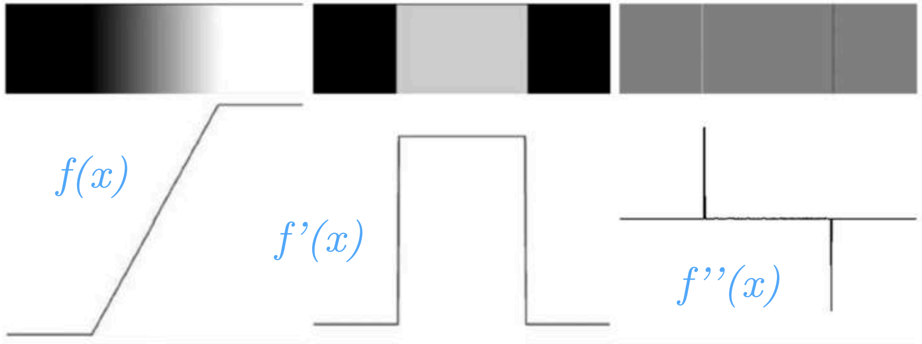
- The *value in each pixel* is the corresponding partial derivative

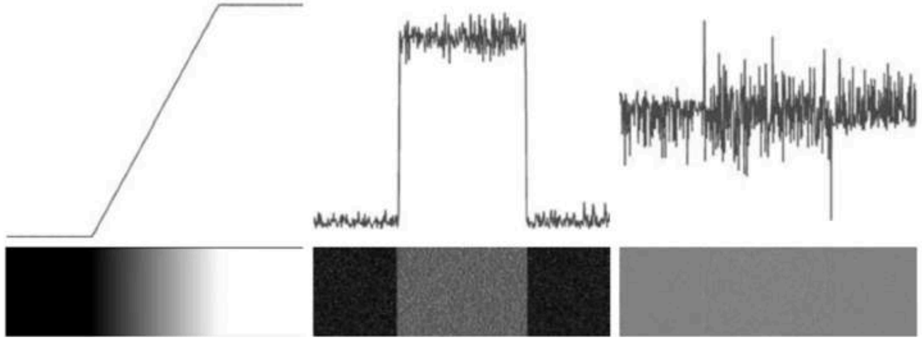
Effects of noise (1/3)

- Finite difference filters **respond strongly to noise**

“ideal” case

(no noise)



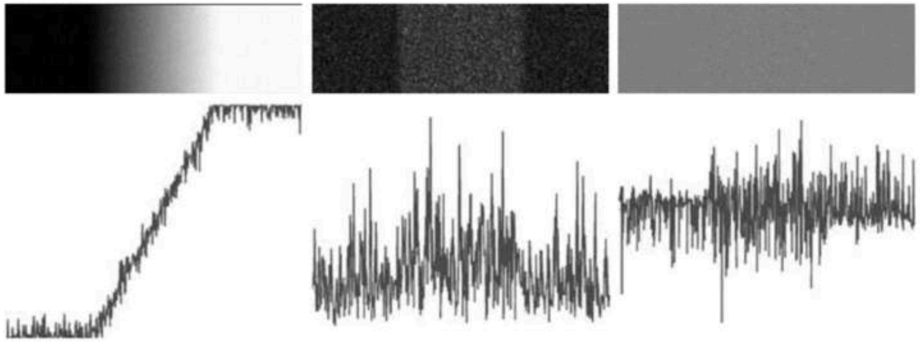


little noise added

NB: *2nd derivative* more sensitive to noise than *1st derivative*

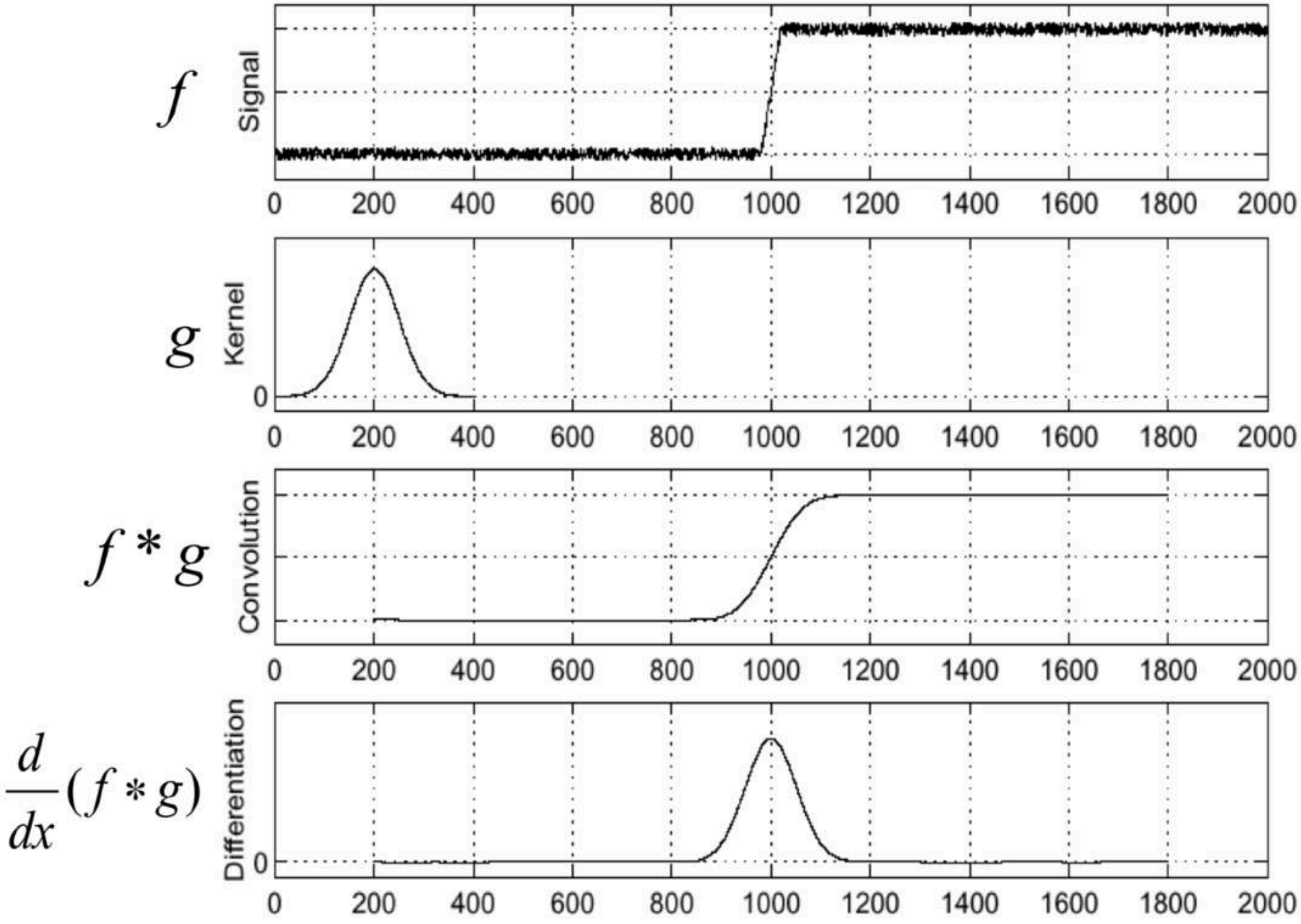
real case

Where is the edge??



Effects of noise (2/3)

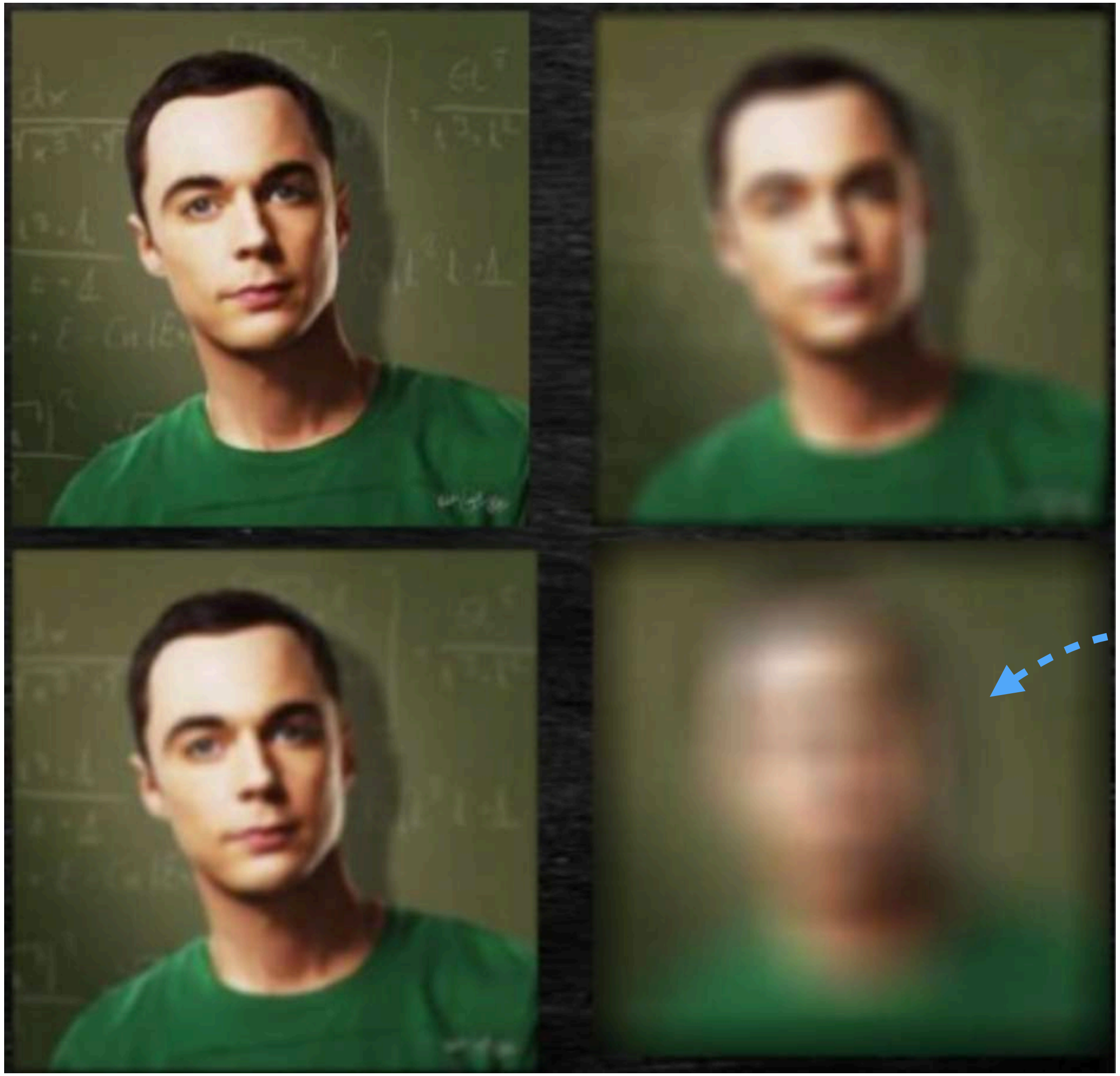
- **Very important**: *smooth the image* before detecting edges



- To find edges, we **look for peaks** in the derivative images

Effects of noise (3/3)

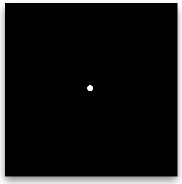
- NB: do not **exaggerate**!



Where are the edges??

Detecting isolated points (1/3)

- Isolated points are **very small features**…



- …thus, **2nd derivative** should be an appropriate choice
  (recall previous discussion)

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Laplacian of $f(x,y)$

- Compute the **partials about pixel** (x,y) as described before

$$\frac{\partial^2 f(x,y)}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y)$$

1	-2	1

$$\frac{\partial^2 f(x,y)}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y)$$

1
-2
1

Detecting isolated points (2/3)

- The **Laplacian** is then

$$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

- Written as a *spatial filter*
  - **Measures the weighted differences** between a pixel and its 4-neighbors
  - This formulation can be *extended* to include the *diagonal terms*

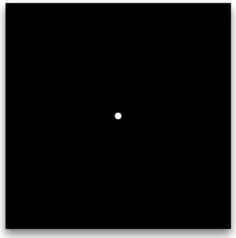
4-neighbors

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

8-neighbors

  - **NB**: coefficients *sum to zero* ➜ response $R=0$ in constant areas



- **Intuition:** intensity of isolated point is *quite different from its neighbors* ➜ this filter will have **high response at that point**

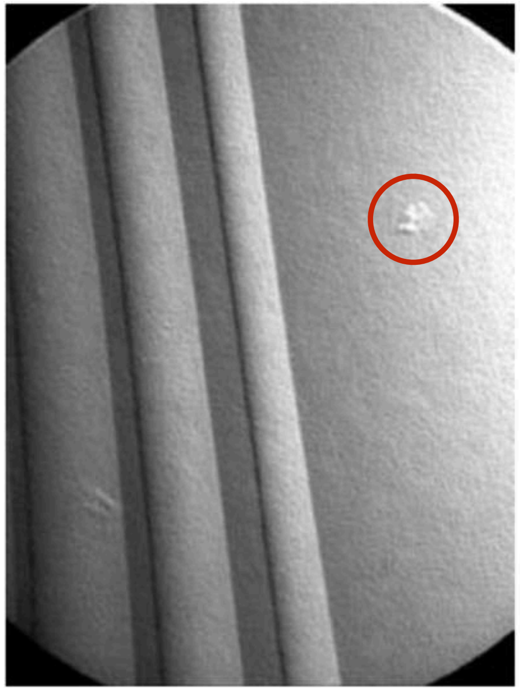
Detecting isolated points (3/3)

- Usually a **threshold** T is used to claim that a feature, e.g. point, has been detected at pixel (x, y)

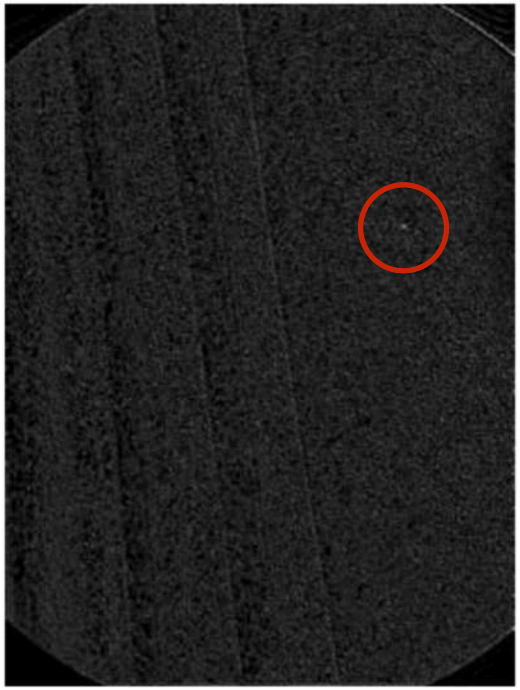
$$g(x, y) = \begin{cases} 1 & \text{if } |R(x, y)| \geq T \\ 0 & \text{otherwise} \end{cases}$$

R is the response to the detector mask

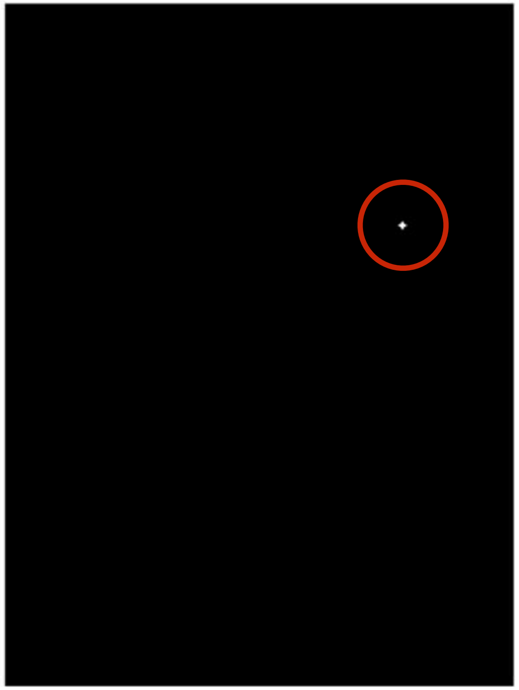
- Example



Input image



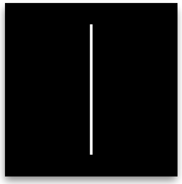
Filtered with **point detector mask**



T=90% of max value

Detecting lines (1/4)

- As lines are **very small features** (i.e. 1 pixel wide) we could use again the **Laplacian mask**

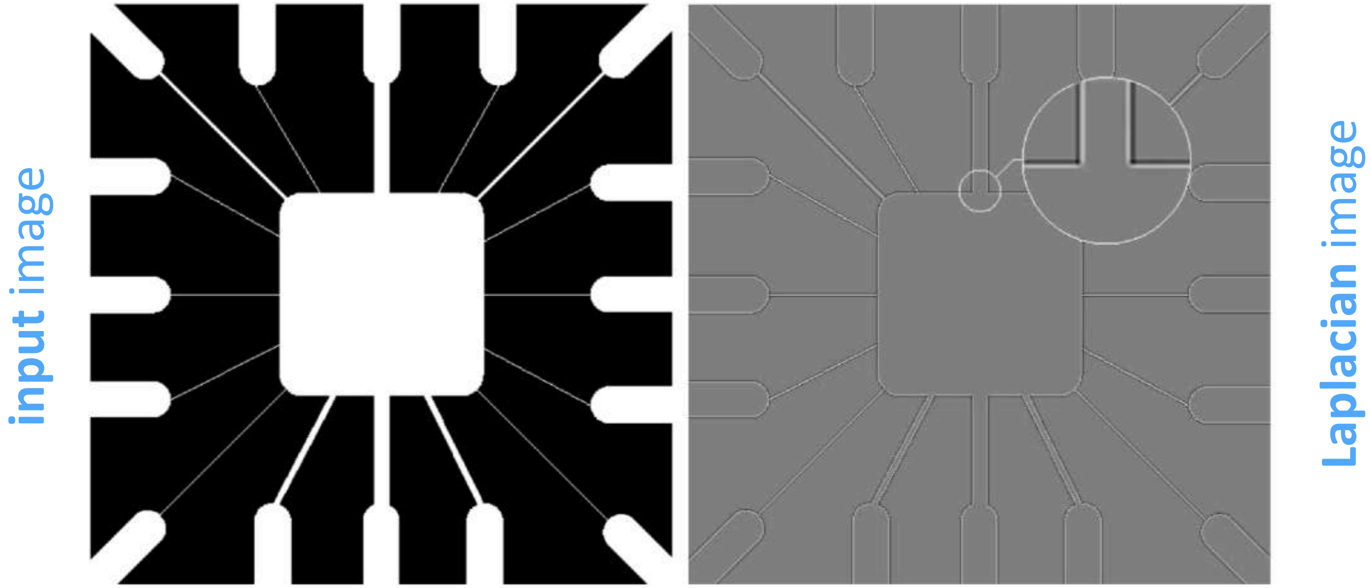


Laplacian of $f(x,y)$

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

1	1	1
1	-8	1
1	1	1

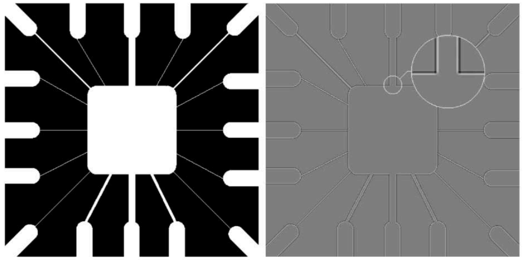
- Example



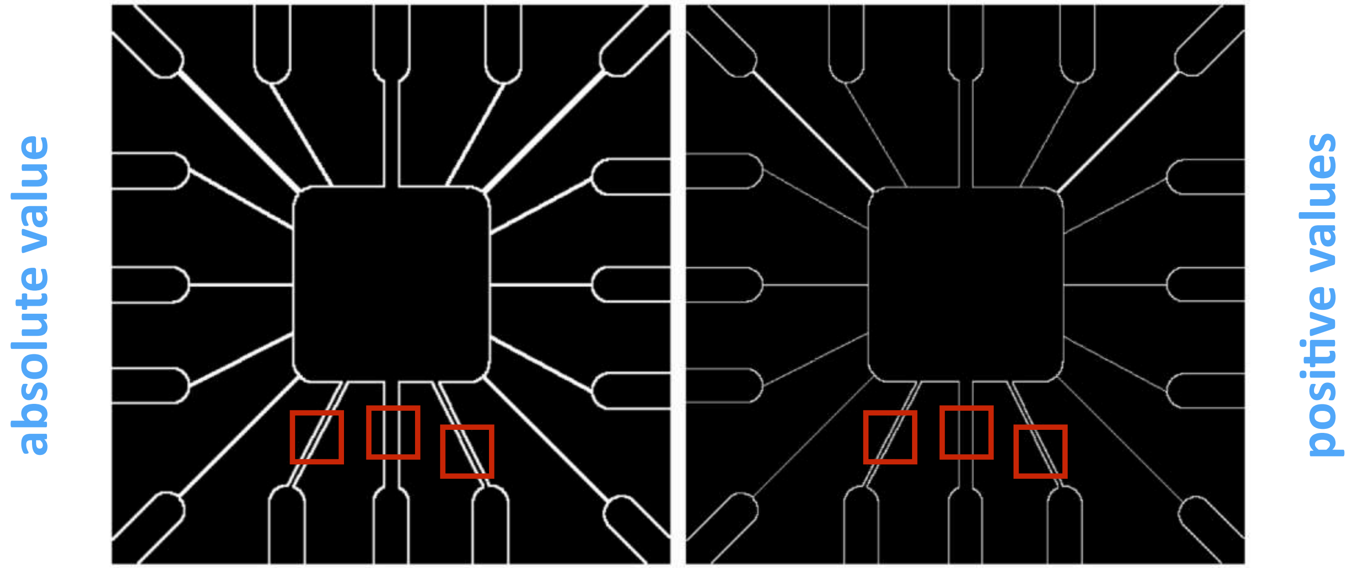
▶ $R=0$ in areas of *constant intensity*, and $R\neq 0$ close to *lines* (and *thin edges*)

▶ Note the **double-edge effect**

Detecting lines (2/4)



- How to handle **positive/negative values?**
  - Taking the **absolute value** (*left*) ➜ *thick* lines
  - Keeping **positive values** (*right*) ➜ *thinner* lines



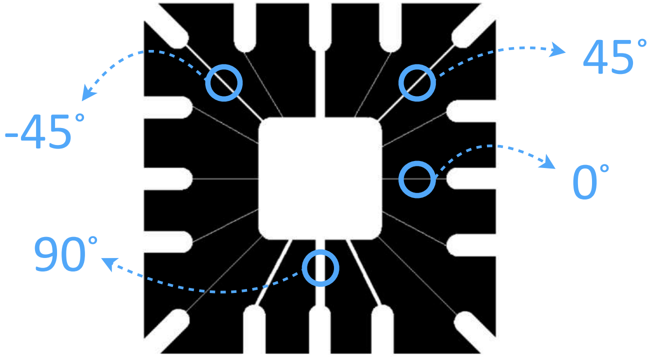
- **Note**: when the lines are wide w.r.t. mask size, lines are separated by a "**zero valley**"
  - Imagine a *3x3 mask* in a white region at least *5 pixels wide* ➜ $R=0$
  - Line detection **assumes thin lines** (otherwise they are *edges*)

Detecting lines (3/4)

- The Laplacian mask is **isotropic**
  - i.e. insensitive to *line direction*

1	1	1
1	-8	1
1	1	1

- Often, it is interesting to **detect lines in specified directions**



- To achieve this, we can filter the image with **specialized masks**

0°

-1	-1	-1
2	**2**	**2**
-1	-1	-1

-45°

2	-1	-1
-1	**2**	-1
-1	-1	**2**

90°

-1	**2**	-1
-1	**2**	-1
-1	**2**	-1

45°

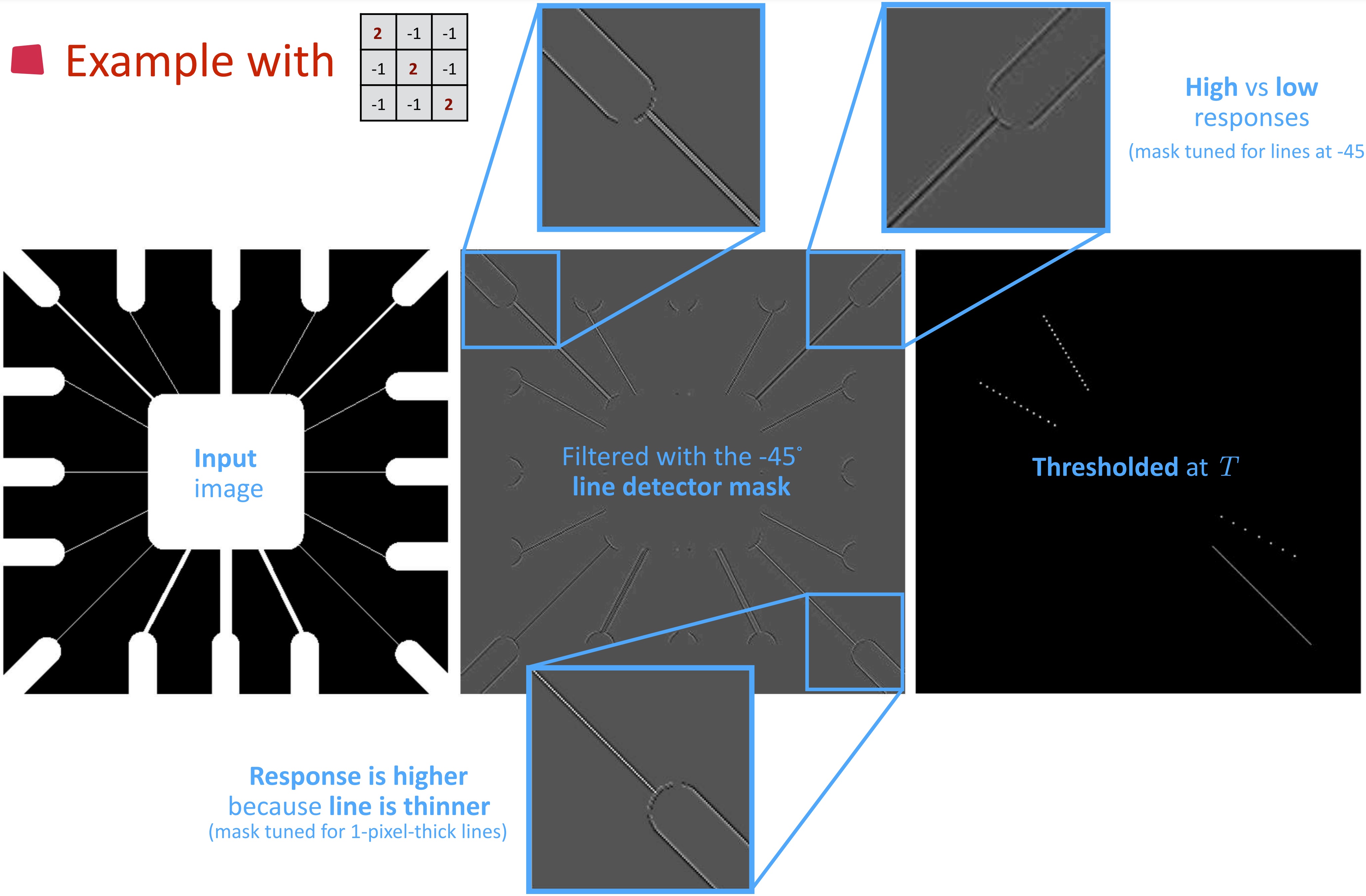
-1	-1	**2**
-1	**2**	-1
2	-1	-1

  - The **response of each filter** is maximum when line is aligned with it

Detecting lines (4/4)

- Example with

2	-1	-1
-1	**2**	-1
-1	-1	**2**



High vs **low** responses
(mask tuned for lines at -45°)

Input image

Filtered with the -45°
line detector mask

Thresholded at T

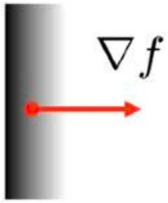
Response is higher because **line is thinner**
(mask tuned for 1-pixel-thick lines)

Basic edge detection (1/2)

- Based on the **image gradient**
  - The **discrete gradient** of $I(u, v)$ is the 2D vector

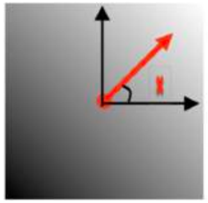
$$\nabla I(u,v) = \begin{bmatrix} \nabla_u I(u,v) \\ \nabla_v I(u,v) \end{bmatrix}$$

  - **Important geometrical property**: it points in the direction of *most rapid increase* in intensity



$$\nabla f = \left[\frac{\partial f}{\partial x}, 0\right]$$

$$\nabla f = \left[0, \frac{\partial f}{\partial y}\right]$$



$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

- The **gradient magnitude** is

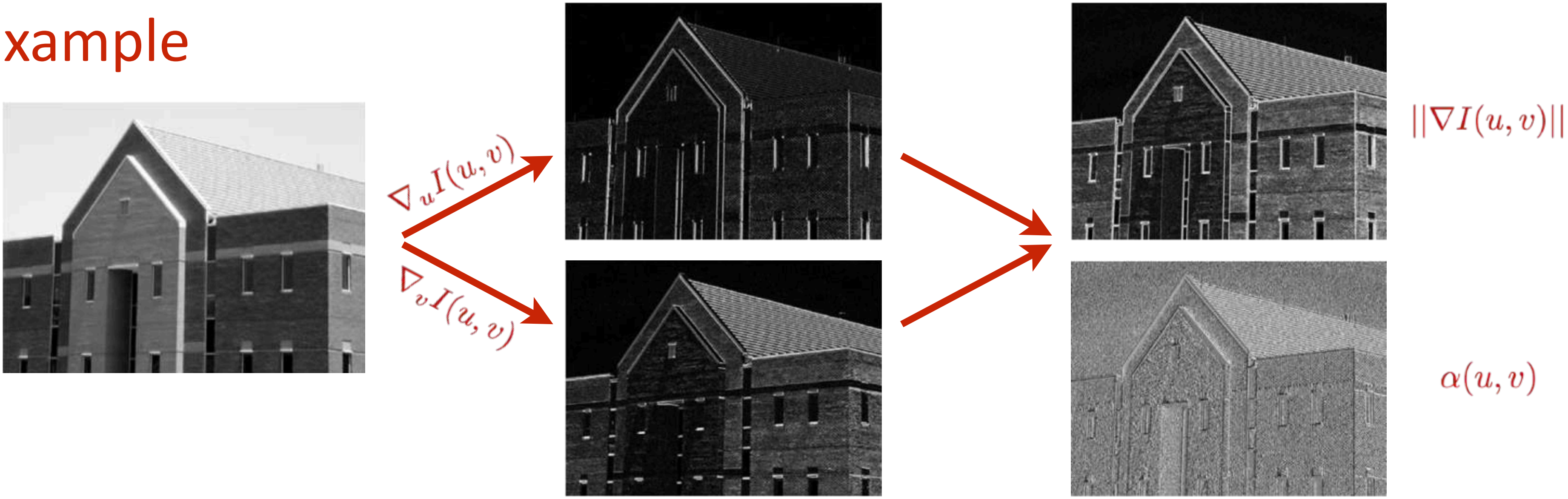
$$||\nabla I(u,v)|| = \sqrt{(\nabla_u I(u,v))^2 + (\nabla_v I(u,v))^2}$$

- The **gradient direction** is

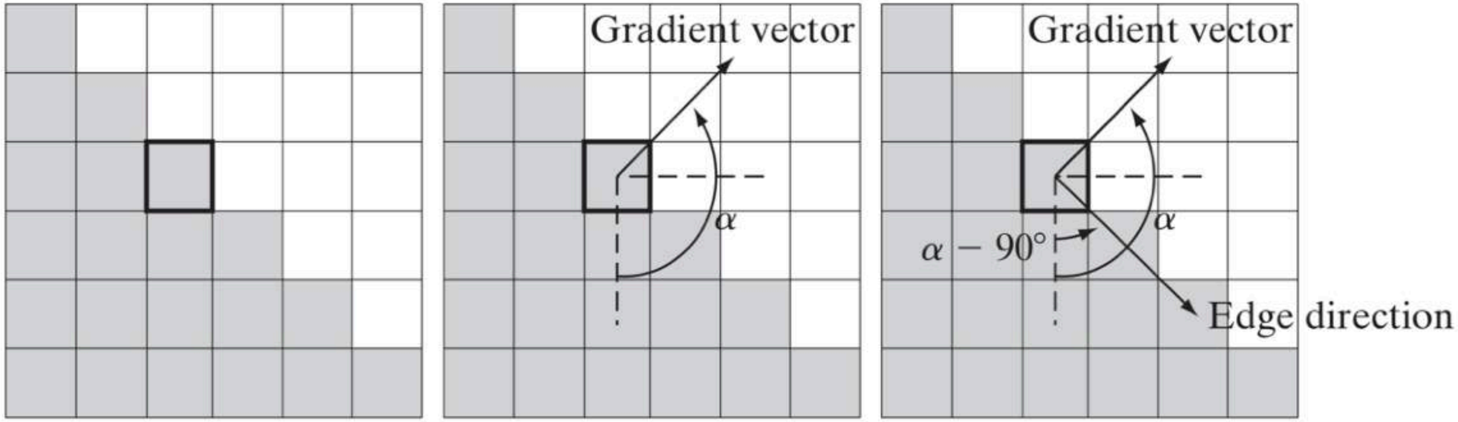
$$\alpha(u,v) = \tan^{-1}\left[\frac{\nabla_v I(u,v)}{\nabla_u I(u,v)}\right]$$

Basic edge detection (2/2)

- Example



- **Direction of an edge** at an arbitrary point (x,y) is **orthogonal** to the **direction of the gradient** vector at the point, i.e. α(x,y)



First-order edge detectors (1/4)

Prewitt gradient operator

- We know that finite differences are *very sensitive to noise*
- More robust estimates by **averaging in the neighborhood**

$$\nabla_u I(u,v) = \frac{1}{2}\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$$

$$\nabla_v I(u,v) = \frac{1}{2}\begin{bmatrix}-1 & 0 & 1\end{bmatrix}^T$$

➡

$$\nabla_u I(u,v) = \frac{1}{6}\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix}$$

$$\nabla_v I(u,v) = \frac{1}{6}\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix}^T$$

Sobel gradient operator

- Similar to Prewitt, but averaging is *stronger in central pixel*

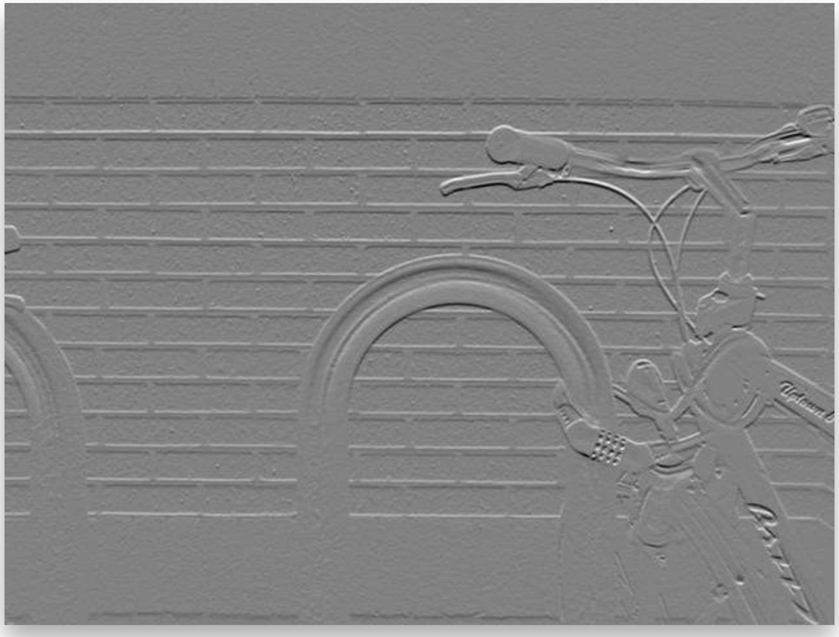
$$\nabla_u I(u,v) = \frac{1}{8}\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}$$

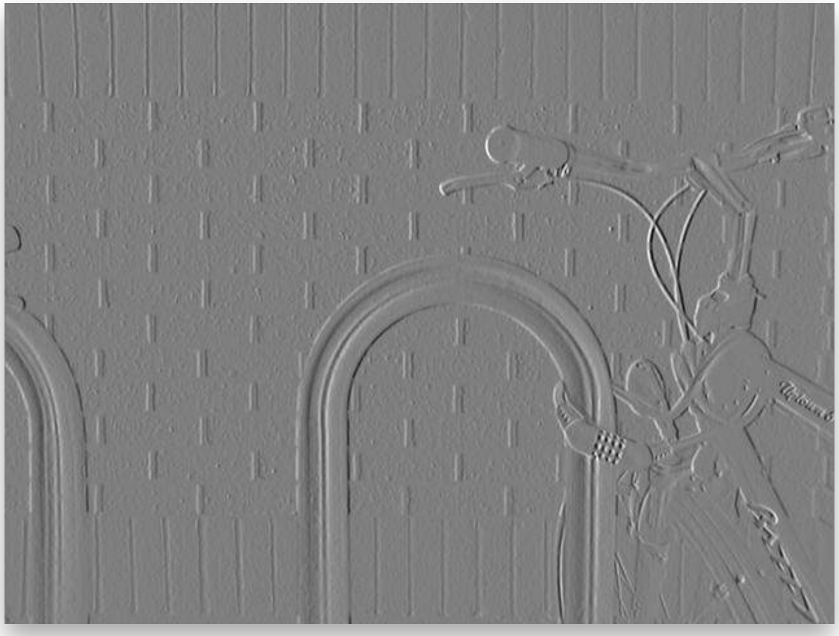
$$\nabla_v I(u,v) = \frac{1}{8}\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}^T$$

First-order edge detectors (2/4)

- Which is $\nabla_u I(u,v)$?
  And $\nabla_v I(u,v)$?

$\nabla_u I(u,v)$
or
$\nabla_v I(u,v)$

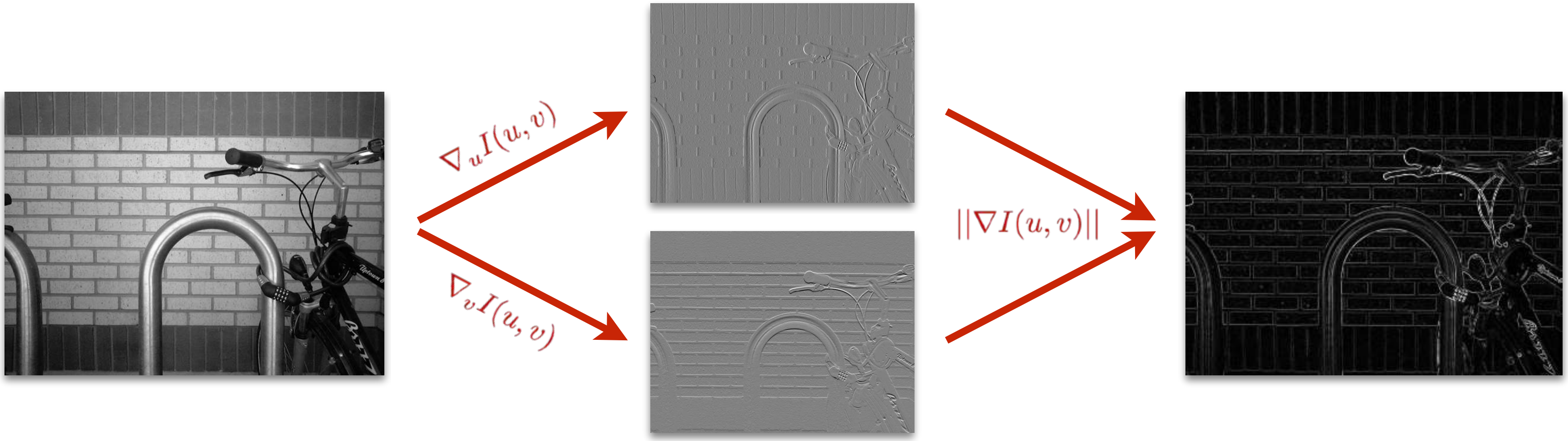




$\nabla_u I(u,v)$
or
$\nabla_v I(u,v)$

First-order edge detectors (3/4)

- Note 1: **magnitude** is used to *highlight edge pixels*



- Note 2: both filters implicitly **perform smoothing**
  - Prewitt

$$H = \frac{1}{6}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \overbrace{\frac{1}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}^{\text{averaging}} \cdot \underbrace{\frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}}_{\text{derivative}}$$

  - Sobel

$$H = \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \overbrace{\frac{1}{4}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}^{\text{averaging}} \cdot \underbrace{\frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}}_{\text{derivative}}$$

First-order edge detectors (4/4)

Comparison

- Very **similar performances**
- May vary depending on *specific applications*
- In practice, try both and choose

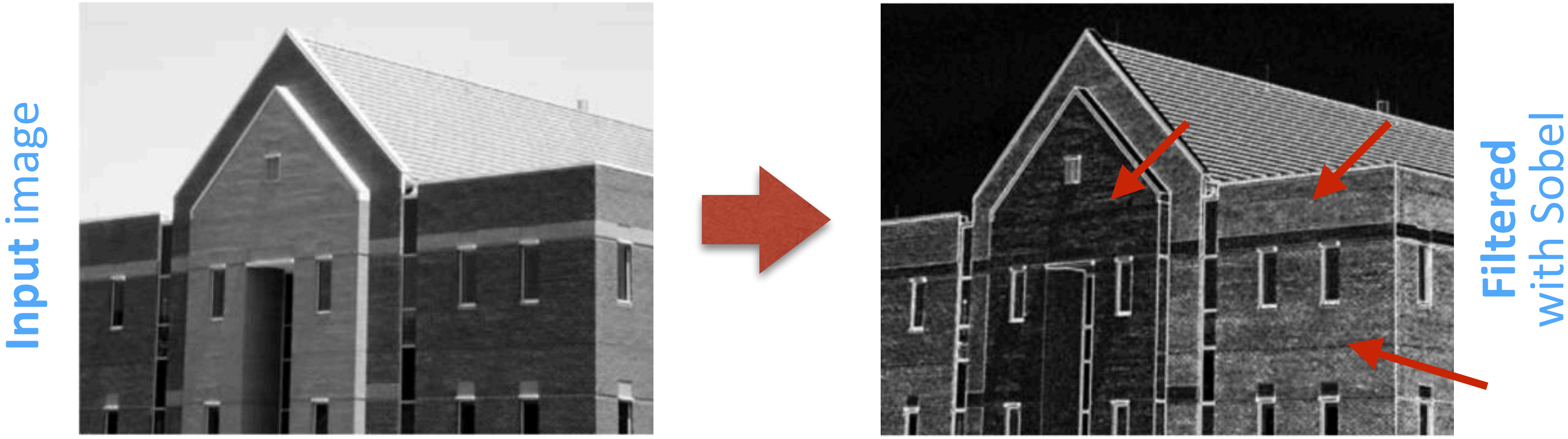
Prewitt

Sobel

Image details and edge detection (1/2)

- Sometimes the **level of fine detail** is undesirable in edge detection because it **tends to act as noise**
  - *Fine details are enhanced* by derivative computations
  - Complicates *detection of the principal edges* in an image
  - Which are the **true edges**?

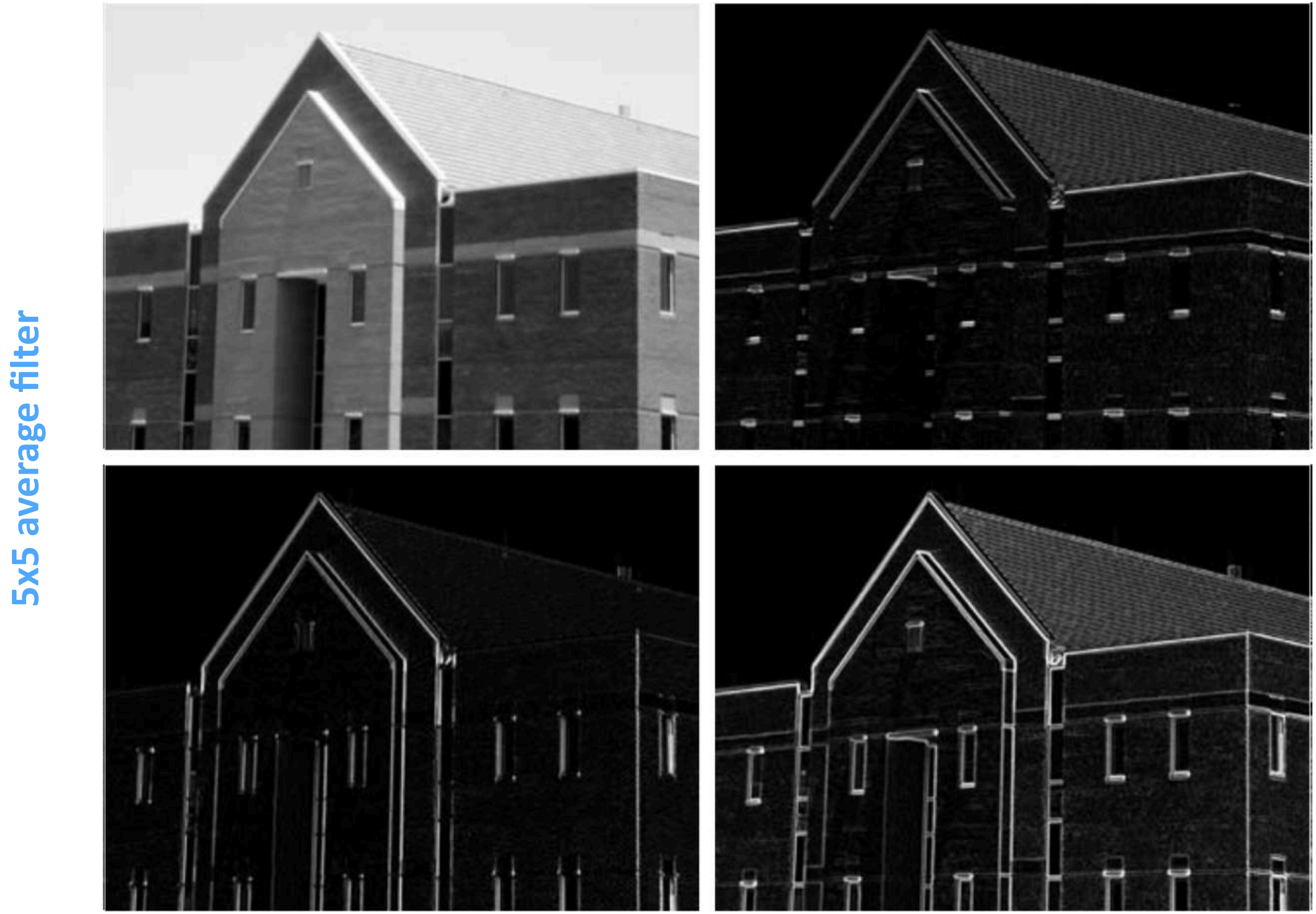
- Example



  - Contribution to image detail by the **wall bricks** is significant ➜ this in **undesired**
- **Idea**: **smooth the image** prior to edge detection

Image details and edge detection (2/2)

- Comparison



Second-order edge detectors (1/8)

Based on the **2nd derivatives**

- **NB**: *more sensitive to noise* than 1st derivatives ➜ need to *smooth the image*

The **Marr-Hildreth** algorithm

$$\nabla^2 G(x, y) = \frac{\partial^2 G(x, y)}{\partial x^2} + \frac{\partial^2 G(x, y)}{\partial y^2}$$

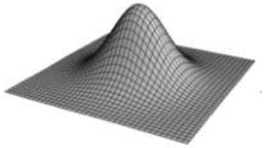
- Combines the *Laplacian operator*

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

1	1	1
1	-8	1
1	1	1

with *Gaussian filtering*

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



- Also called **Laplacian of Gaussian** (*LoG*) filter
- The parameter σ is called "*space constant*"

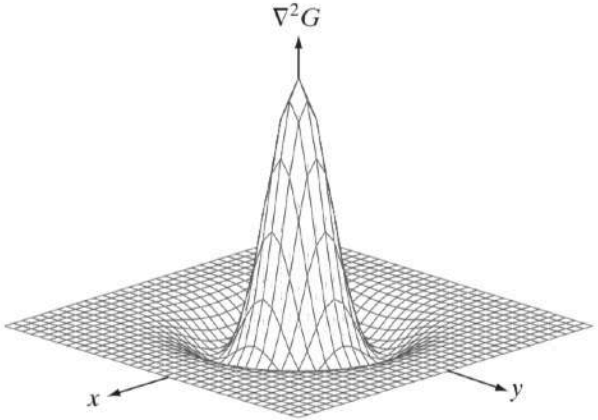
Second-order edge detectors (2/8)

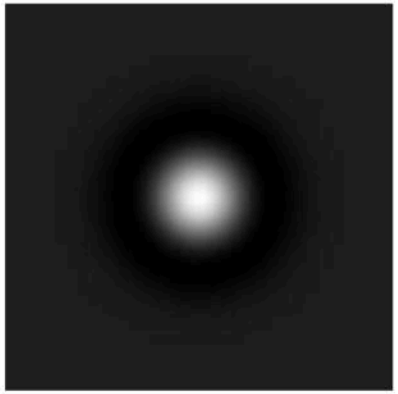
Commonly called “**Mexican hat**“ filter

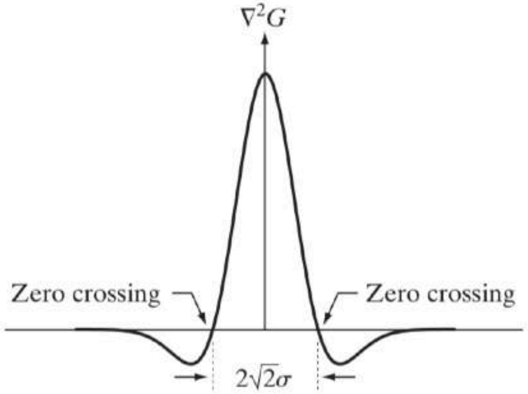
- From the previous equation, we can *derive the following expression*:

$$\nabla^2 G(x, y) = \left[\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- Which *looks like* this:







0	0	−1	0	0
0	−1	−2	−1	0
−1	−2	16	−2	−1
0	−1	−2	−1	0
0	0	−1	0	0

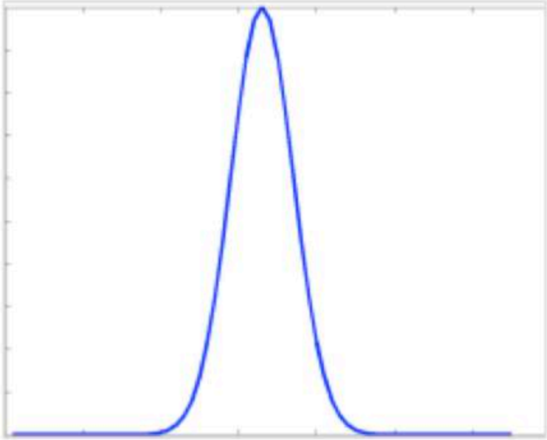
Actually, this is the **negative of the LoG**

NB: for *detecting edges*, it achieves the same effect

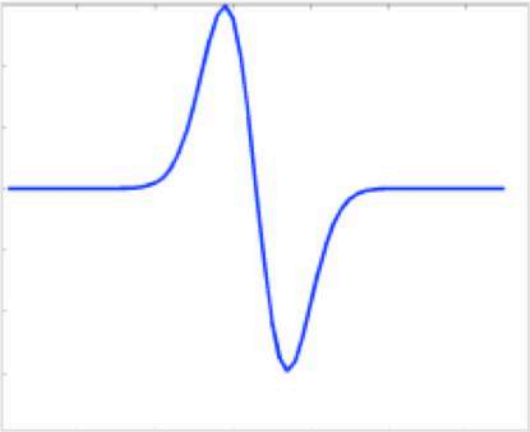
Second-order edge detectors

[images from R. Collins - CSE486]

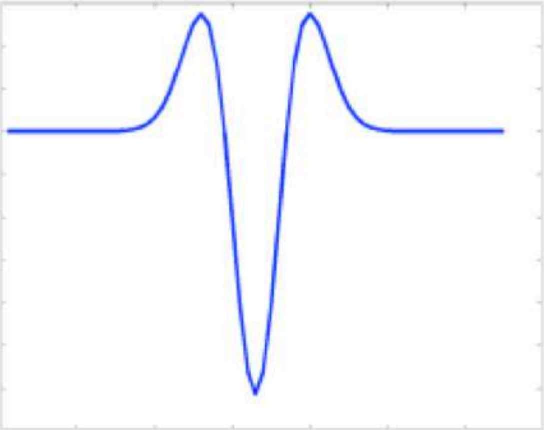
$$g(x) = e^{-\frac{x^2}{2\sigma^2}}$$

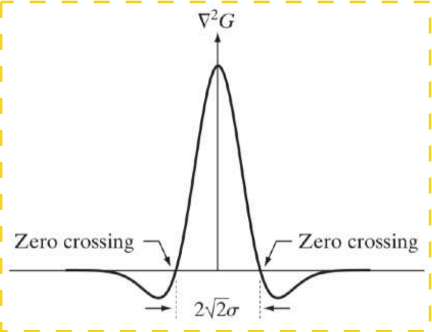


$$g'(x) = -\frac{1}{2\sigma^2} 2xe^{-\frac{x^2}{2\sigma^2}} = -\frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$



$$g''(x) = \left(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}\right) e^{-\frac{x^2}{2\sigma^2}}$$



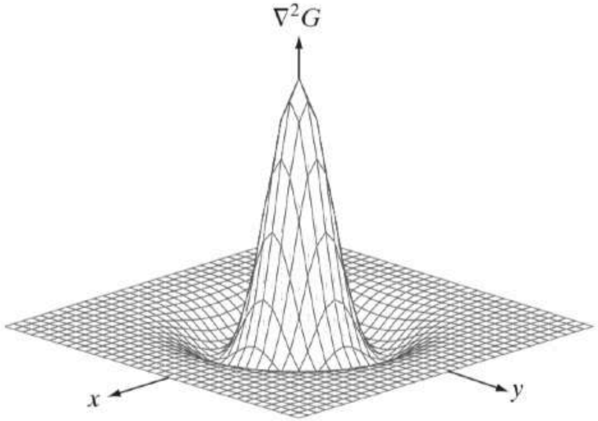


Second-order edge detectors (4/8)

- Edge-detector **algorithm**
  1) Filter the input image *f(x,y)* by convolving with **LoG filter**

$$g(x, y) = [\nabla^2 G(x, y)] \star f(x, y)$$

  2) **Find the zero crossings** of the resulting image *g(x,y)* to determine the locations of edges in *f(x,y)*

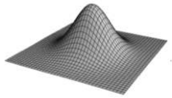


- **NB**: the operations involved are **linear**
  i.e. *Gaussian filtering* and *Laplacian*

$$[\nabla^2 G(x, y)] \star f(x, y) = \nabla^2 [G(x, y) \star f(x, y)]$$

- *Equivalent* algorithm
  1) Filter the input image *f(x,y)* with **Gaussian lowpass filter**
  2) **Compute the Laplacian** of the resulting image
  3) **Find the zero crossings**

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



$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

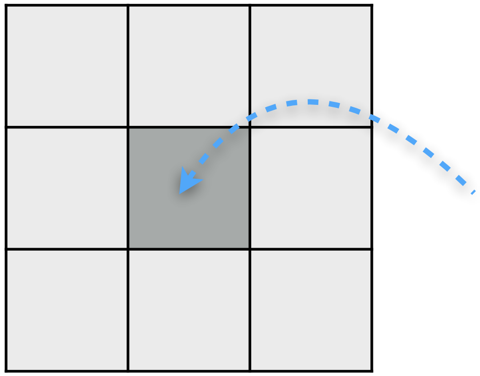
1	1	1
1	-8	1
1	1	1

Second-order edge detectors

How to **find the zero crossings**?

- Attempting to find the coordinates (x,y), such that $g(x,y) = 0$ is **impractical**
  (e.g. because of noise and computational inaccuracies)

- **In practice**, a *3x3 neighborhood* is used

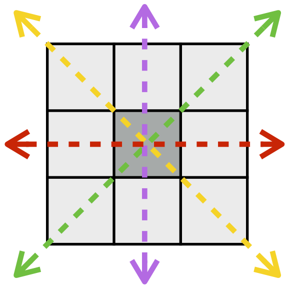


Is there a zero-crossing at the **central pixel**?

- **<u>IDEA:</u>** there's a *zero crossing* in the central pixel ⟺ the **signs** of **at least two** of its **opposing neighboring pixels** are different

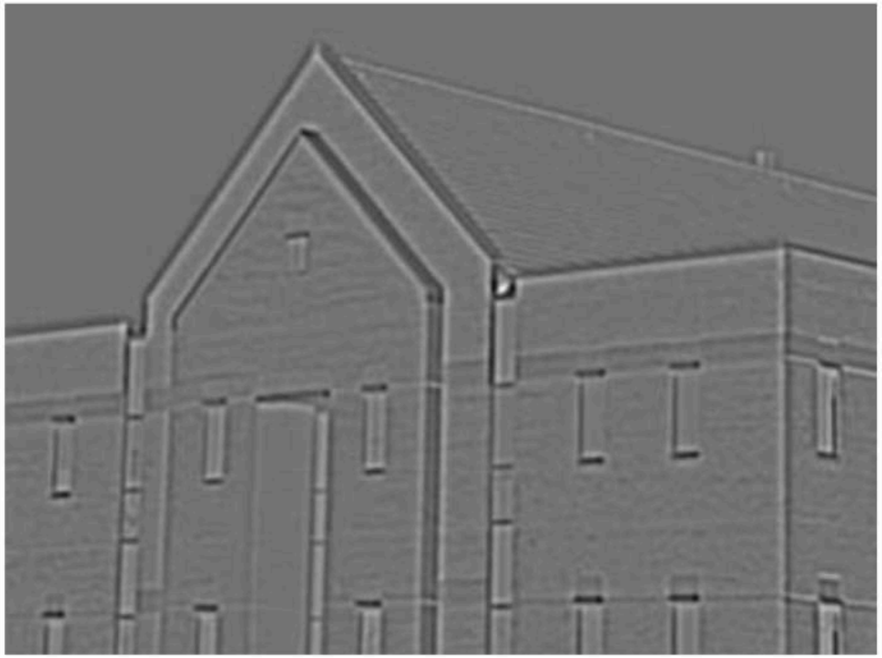
- There are **four cases to test**
  - left/right, up/down, and the two diagonals

- A **threshold** is also typically used
  - Test the *absolute value* of their numerical difference



Second-order edge detectors

input image



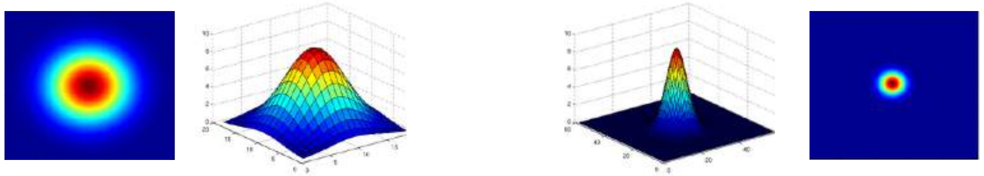
LoG filtered

zero crossings with **threshold = 0**

zero crossings with **threshold = 4% of max**

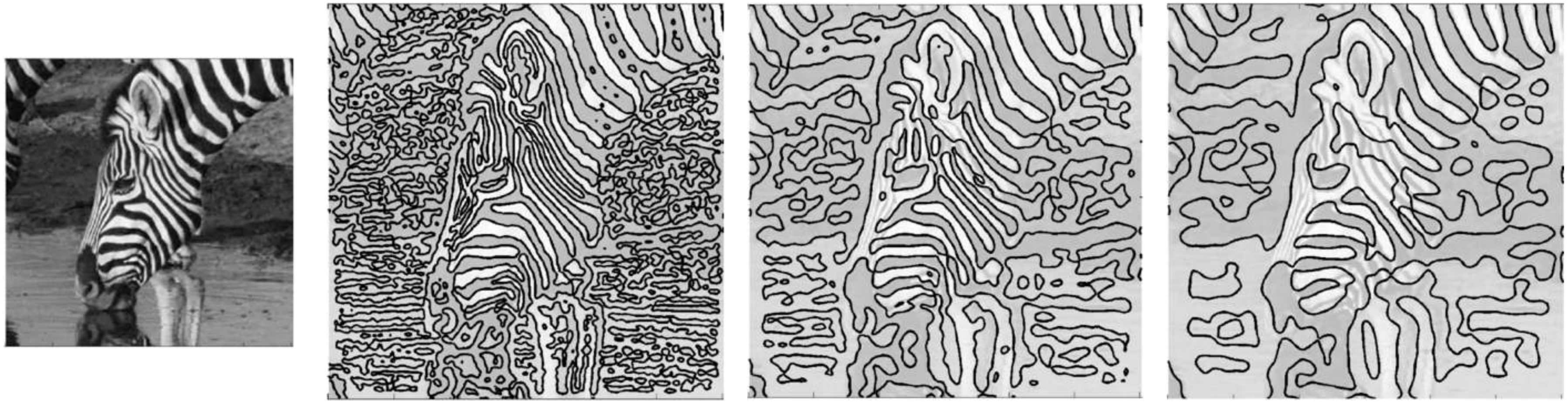
Second-order edge detectors

- Note on **parameter σ**, i.e. standard deviation
  - Controls the *width of the Gaussian blur*



  - Blur reduces the intensity of structures at *scales smaller than* σ

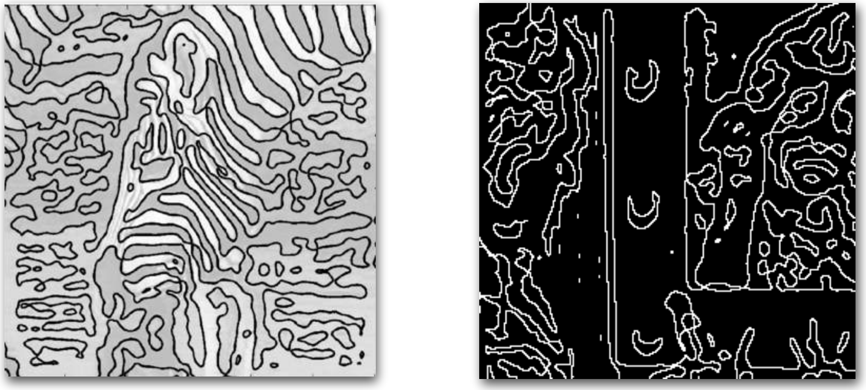
- Can be tuned to **extract edges at any desired scale**
  - *Large operators* can be used to detect blurry edges
  - *Small operators* to detect sharply focused fine detail



zero-crossings with σ=2 zero-crossings with σ=4 zero-crossings with σ=8

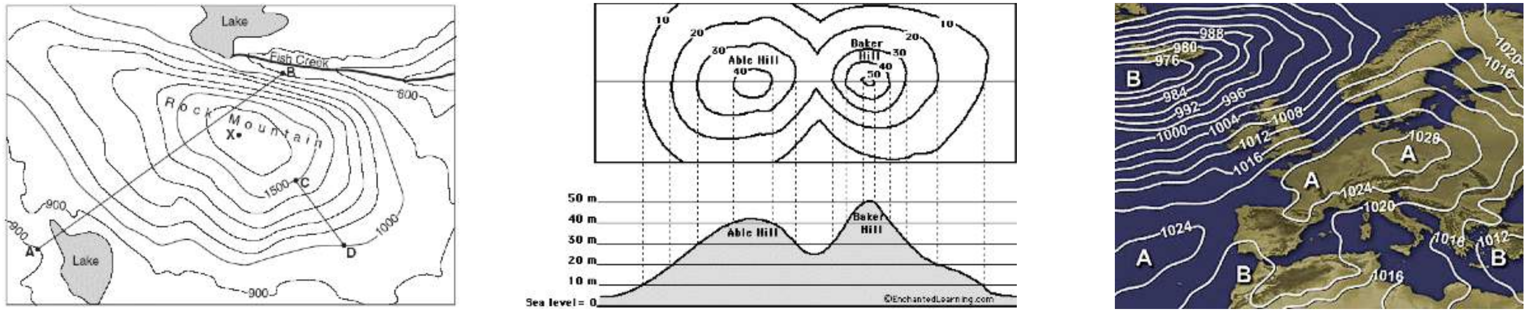
Second-order edge detectors (8/8)

- **Q**: why do zero-crossings **form closed contours**?
  (NB: in case of *threshold = 0*)



- A little digression…
  - The **level set** of a real-valued function f is:

$$L_c(f) = \{(x_1, \cdots, x_n) \mid f(x_1, \cdots, x_n) = c\}$$



NB: they form closed contours

- **A**: in our case, the "heigh map" is a **LoG filtered image**
  - It's a surface with both *positive and negative "elevations"*…
  - …and *zero-crossings* are contours with *elevation = 0*!

More advanced examples

Canny edge detector

- Developed by **John F. Canny** in 1986
  - More **complex** than any edge detectors discussed thus far
  - **Performance** is much better than any other
- Based on **four basic objectives**
  1) **All edges should be found**
  2) There should be **no spurious responses**
  3) The edges located must be **as close as possible** to the true edges
  4) The detector should return only **one point for each true edge point**
- Essence of Canny's work: **express these criteria mathematically**
  - **Difficult** (or impossible) to find a *closed-form solution*
  - Canny attempted to **find an optimal solution**
    (NB: using numerical optimization with 1D step edges corrupted by additive white Gaussian noise)

Canny edge detector (2/9)

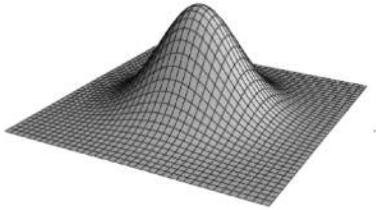
Algorithm

1) **Smooth** input image with a Gaussian filter
2) Compute the **gradient** (*magnitude* and *angle*) of the image
3) **Thin edges** by applying **non-maxima suppression** to the gradient magnitude
4) Detect edges by using **double thresholding**

Step 1: Gaussian smoothing

$f(x, y)$

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

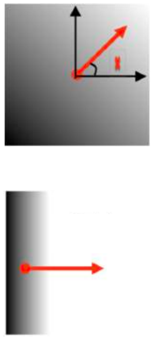


$f_s(x, y) = G(x, y) \star f(x, y)$

Canny edge detector (3/9)

Step 2: compute the gradient

$$\nabla I(u,v) = \begin{bmatrix} \nabla_u I(u,v) \\ \nabla_v I(u,v) \end{bmatrix}$$



- **Recall**: points in the *direction of most rapid increase* in intensity
- **Magnitude** image

$$||\nabla I(u,v)|| = \sqrt{(\nabla_u I(u,v))^2 + (\nabla_v I(u,v))^2}$$

- **Direction/angle** image

$$\alpha(u,v) = \tan^{-1}\left[\frac{\nabla_v I(u,v)}{\nabla_u I(u,v)}\right]$$

alternative notation

$$M(x,y) = \sqrt{g_x^2 + g_y^2}$$

$$g_x = \partial f_s / \partial x \qquad g_y = \partial f_s / \partial y$$

$$\alpha(x,y) = \tan^{-1}\left[\frac{g_y}{g_x}\right]$$

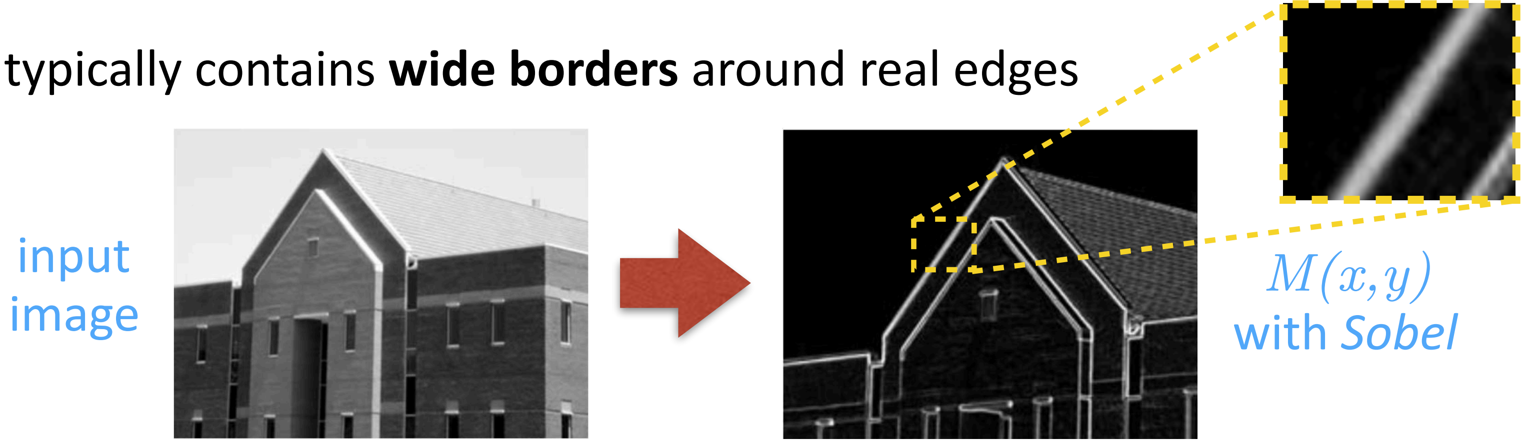
Notes

- **Any basic edge detectors** previously seen can be used, e.g. *Prewitt*, *Sobel*…
- $M(x,y)$ and $\alpha(x,y)$ have **the same size** as the input image

Canny edge detector (4/9)

Step 3: thin edges with **non-maxima suppression**

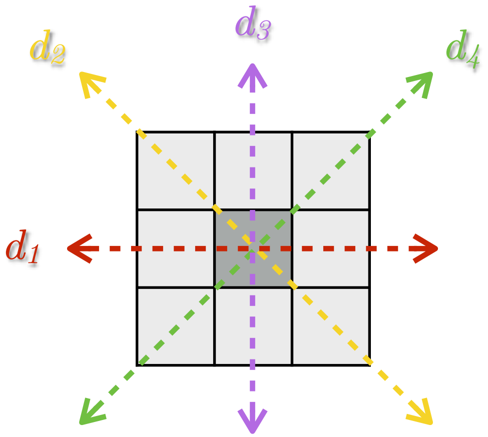
- $M(x,y)$ typically contains **wide borders** around real edges



- Thus, next step is to **thin those edges**

Based on a **discretization of gradient directions**

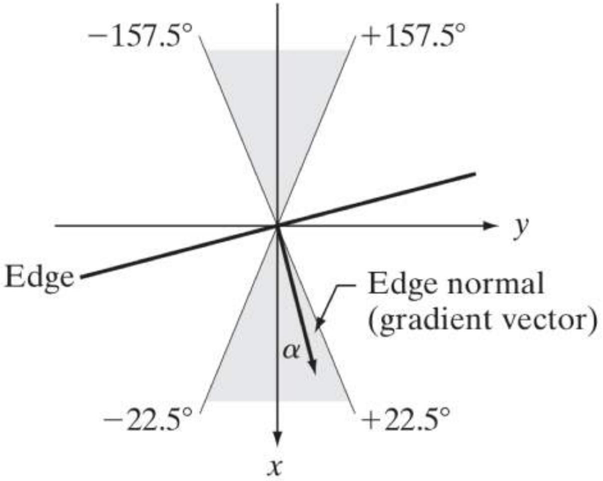
- *Example*: 3x3 neighborhood
- Let d_1, d_2, d_3, and d_4 denote the four basic edge directions available in such a neighborhood
- **NB**: can be extended to bigger sizes

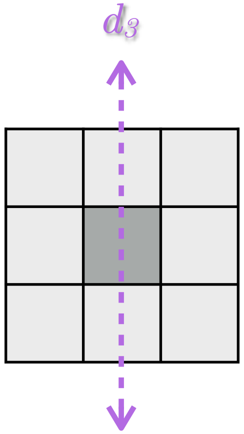


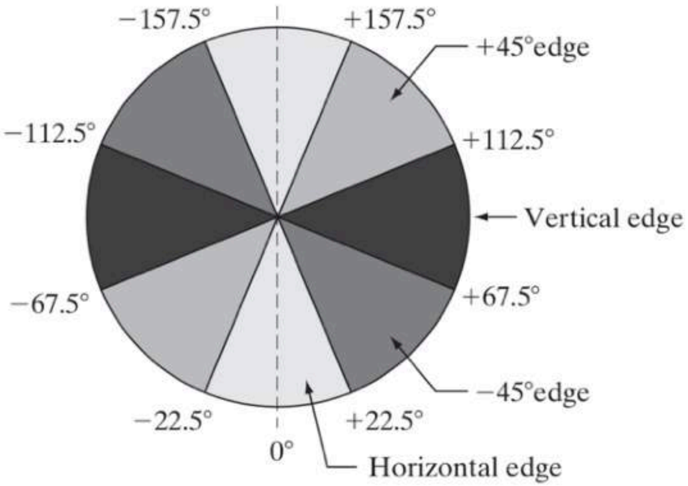
Canny edge detector (5/9)

IDEA

- Find direction d_k that is closest to $\alpha(x,y)$
- Remember that gradient is *perpendicular to edge*





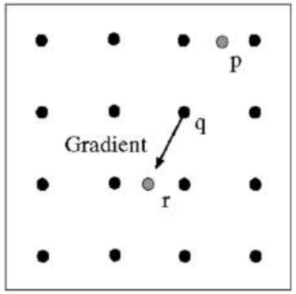


- **If** $M(x,y)$ **larger than two neighbors** <u>along direction d_k</u>

  let $g_N(x,y) = M(x,y)$ *(i.e. keep the edge pixel)*

  otherwise

  let $g_N(x,y) = 0$ *(i.e. suppression of a non-maxima)*

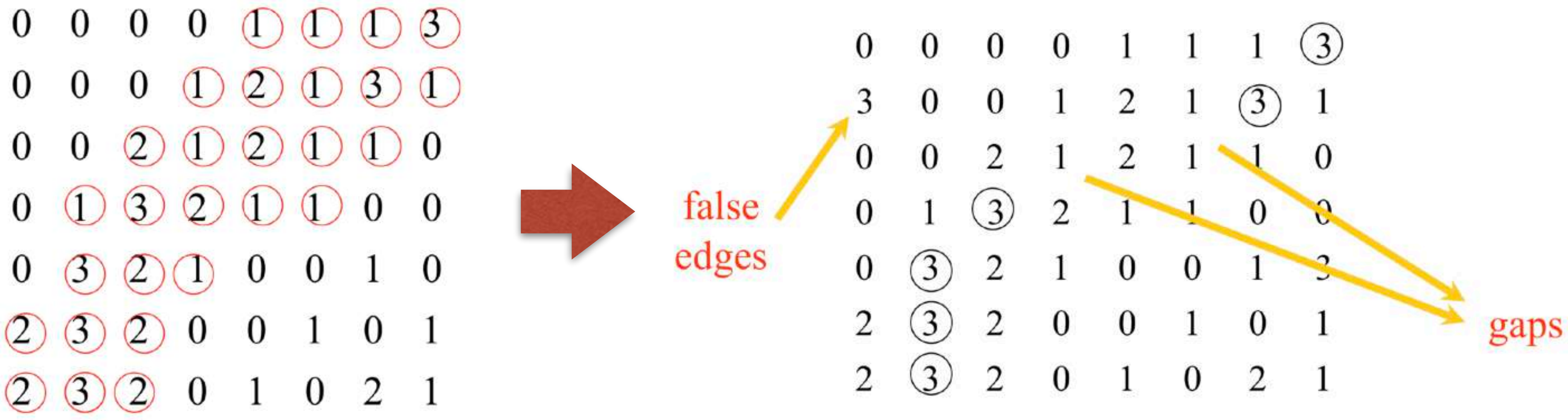


NB: if we don't discretize directions, *interpolation* is needed

Canny edge detector

[images from Euripides G.M. Petrakis]

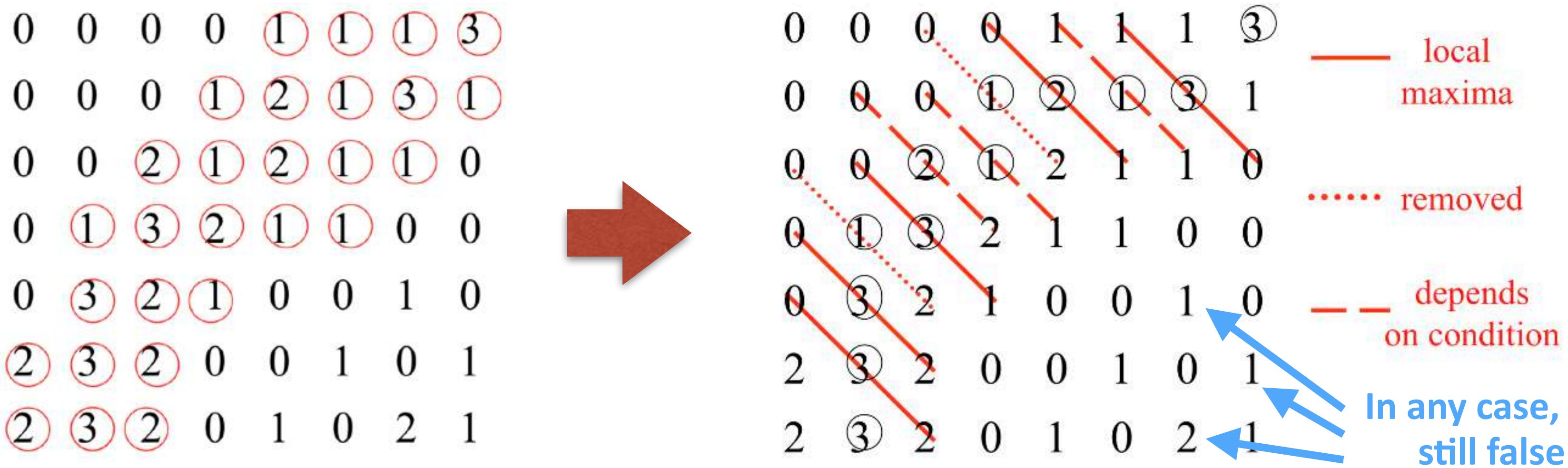
- **Keeping large values** of gradient



0	0	0	0	①	①	①	③
0	0	0	①	②	①	③	①
0	0	②	①	②	①	①	0
0	①	③	②	①	①	0	0
0	③	②	①	0	0	1	0
②	③	②	0	0	1	0	1
②	③	②	0	1	0	2	1

0	0	0	0	1	1	1	③
3	0	0	1	2	1	③	1
0	0	2	1	2	1	1	0
0	1	③	2	1	1	0	0
0	③	2	1	0	0	1	3
2	③	2	0	0	1	0	1
2	③	2	0	1	0	2	1

- Using **non-maxima suppression**



0	0	0	0	①	①	①	③
0	0	0	①	②	①	③	①
0	0	②	①	②	①	①	0
0	①	③	②	①	①	0	0
0	③	②	①	0	0	1	0
②	③	②	0	0	1	0	1
②	③	②	0	1	0	2	1

0	0	0	0	1	1	1	③
0	0	0	①	②	①	③	1
0	0	②	①	2	1	1	0
0	①	③	2	1	1	0	0
0	③	2	1	0	0	1	0
2	③	2	0	0	1	0	1
2	③	2	0	1	0	2	1

In any case, there are still false edges

Canny edge detector

Step 4: threshold $g_N(x,y)$ to **reduce false edge pixels**

Single threshold

- If threshold *too low*, there will still be some false edges (**false positives**)
- If threshold *too high*, actual valid edge pixels will be eliminated (**false negatives**)

Hysteresis threshold

- Define a *low threshold*, $\mathrm{T_L}$, and a *high threshold*, $\mathrm{T_H}$
- Create two *additional images*

$$g_{NL}(x, y) = g_N(x, y) \geq T_L$$

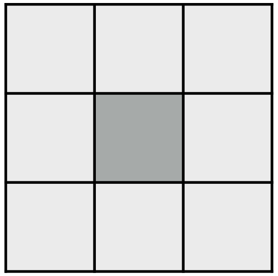
“**weak**” edge pixels

$$g_{NH}(x, y) = g_N(x, y) \geq T_H$$

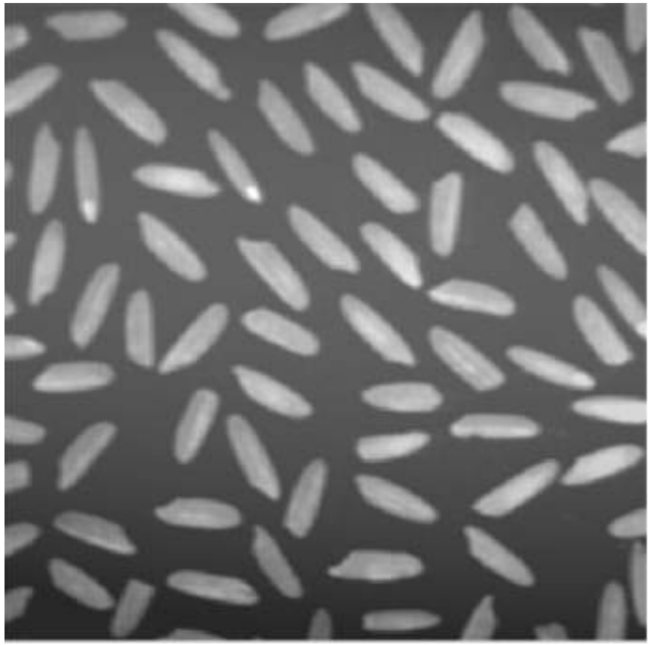
“**strong**” edge pixels

- **NB**: $g_{NH}(x,y)$ has fewer nonzero pixels than $g_{NL}(x,y)$
  All nonzero pixels in $g_{NH}(x,y)$ are contained in $g_{NL}(x,y)$

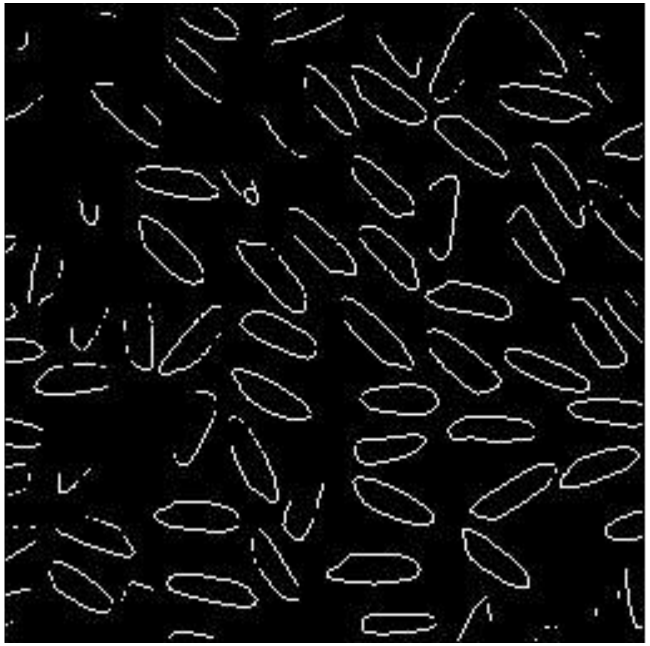
- Edge-tracking algorithm
  - If a value lies above $\mathrm{T_H}$ ➜ immediately **accepted**
  - If a value lies below $\mathrm{T_L}$ ➜ immediately **rejected**
  - If $\mathrm{T_L} \leq$ value $\leq \mathrm{T_H}$, it’s accepted if *connected to strong pixels*
  - **NB**: use a 8-connectivity mask



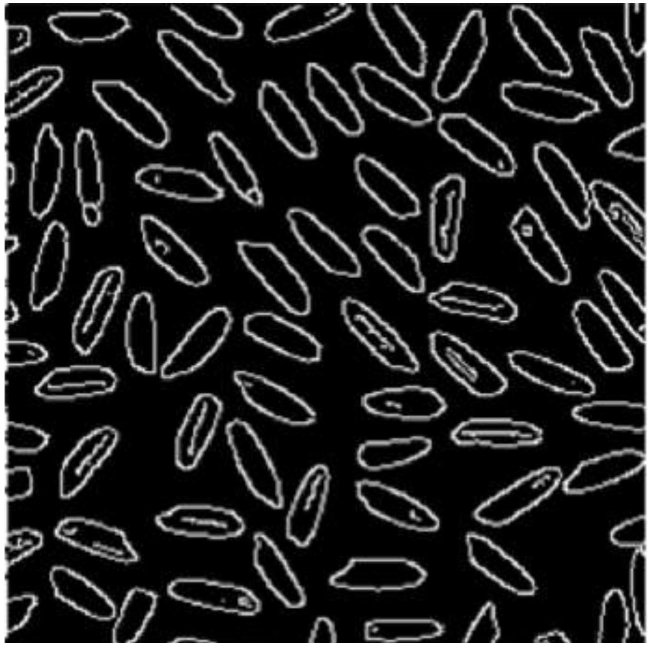
Canny edge detector



input image



Sobel

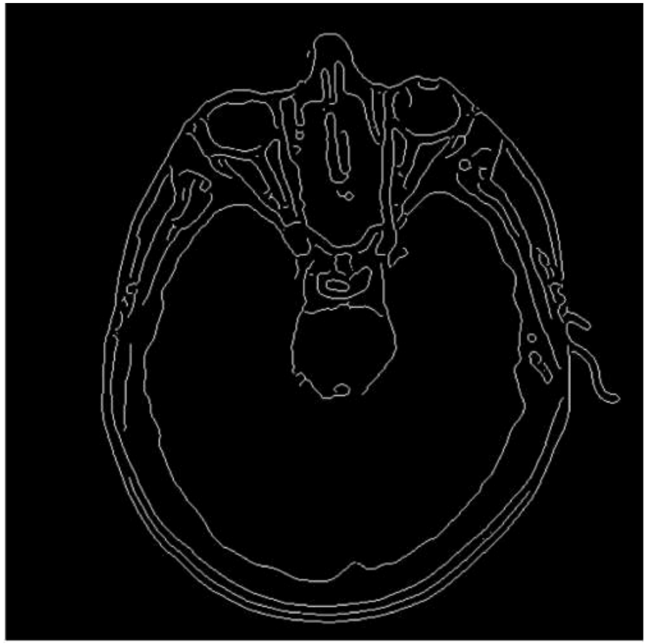


Canny

Canny edge detector

input image

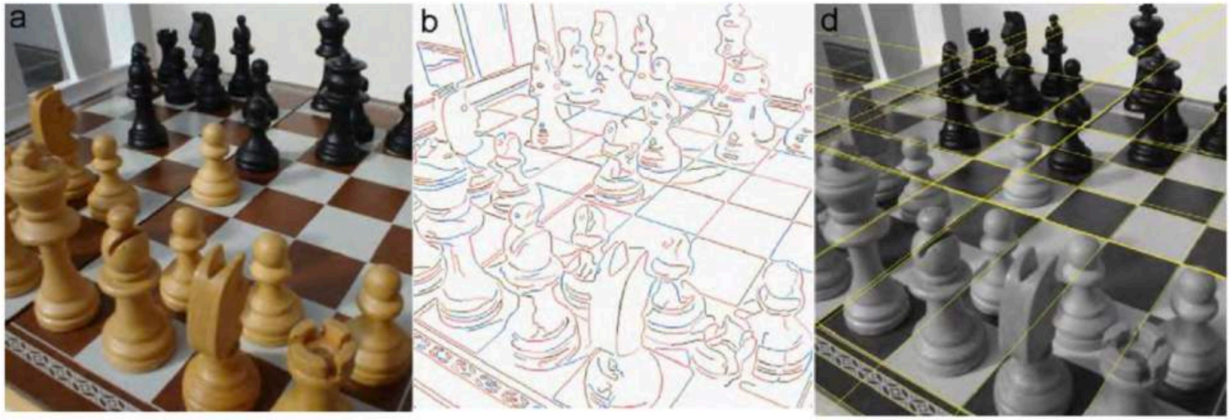
LoG



Canny

Hough transform (1/6)

- So far we've **identified only "edge pixels"**, not really edges



  - How do we realize that such pixels **lie on specific geometrical objects**? e.g. *lines*

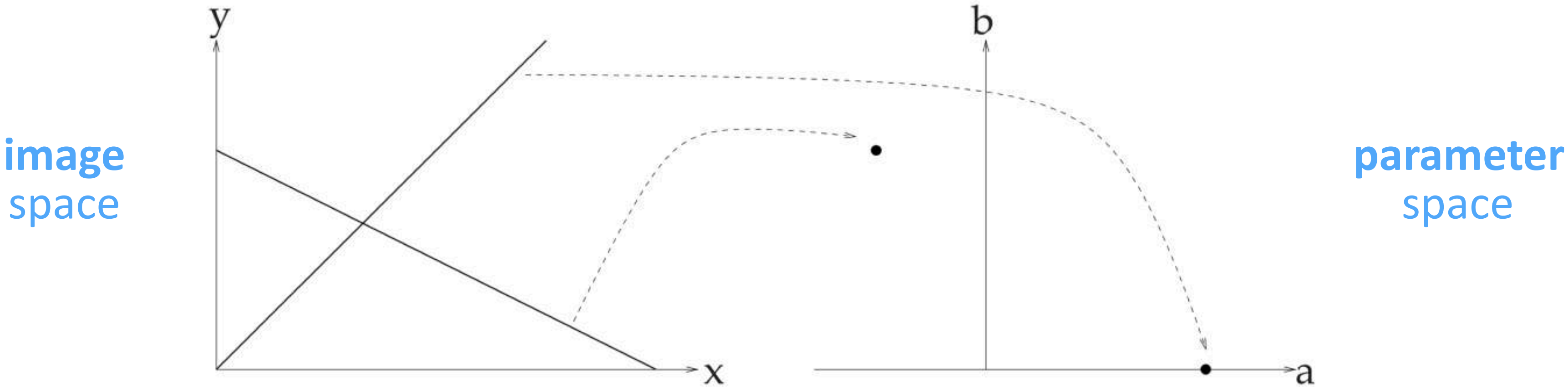
- Goal: **find objects** within a **certain shape**
  - Shapes can be described in **parametric form**, e.g. lines $y=ax+b$
  - Extract **characteristic features** of such objects from the image
  - Identify shapes by using a **voting scheme**

- In this course, we'll focus on the **Hough transform for lines**
  - The algorithm **can be extended** for the identification of **arbitrary shapes**

Hough transform (2/6)

Basic idea

- Transform image from (x,y) space into a **parameter space suited for describing lines**
- Line $y=ax+b$ has **two params**: by varying a and b, we can model any line in the image
- Each line in the image is therefore **represented by a dot** in (a,b) space

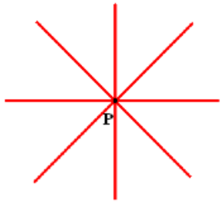


(x,y) space ➜ (a,b) space

- Create an empty image H (called **accumulator**)
- **For each pixel** that may be part of a line
  - Determine which pairs (a,b) create a line $y=ax+b$ that passes through that pixel
  - Increase $H(a,b)$ for those pairs (a,b)

Hough transform (3/6)

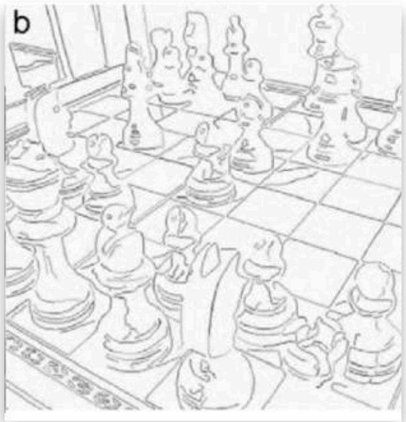
This algorithm is **not very practical**



- There are **infinite lines** passing through a pixel
- A more practical algorithm examines only a **discrete number of pairs** (a,b)

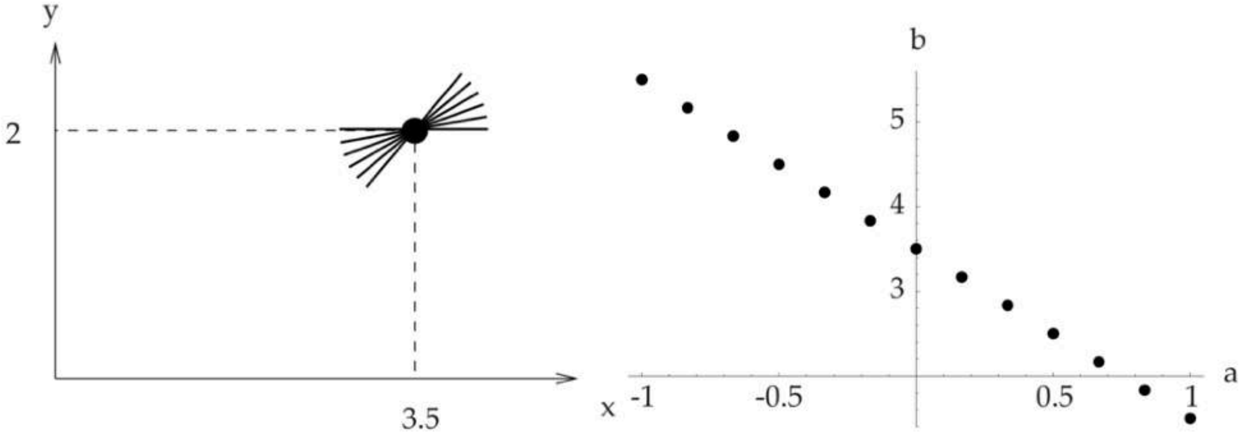
Hough transform algorithm

- Select a set of **discrete values for** a **and** b
  e.g. $a \in \{0, 0.1, \ldots, 100\}, \quad b \in \{0, 1, \ldots 1000\}$
- Create the empty image H
- Run an **edge detector** on the input image
- For each "edge pixel" p
  - For each discrete value a_i
    - Compute corresponding b that forms a line $y = a_i x + b$ that passes through p
    - Round b to the nearest discrete value b_j
    - $H(a_i, b_j) = H(a_i, b_j) + 1$



Hough transform (4/6)

- **Example**: some possible lines passing through a pixel p

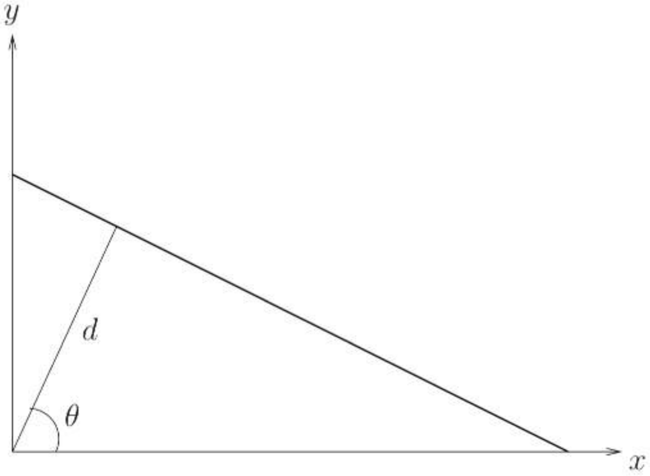


- In practice, *(a,b) parametrization* is **not very efficient**
  - Equidistant discretization of a corresponds to lines with **not equidistant angles**
    - e.g. $a \in \{1, 2, 3, 4, 5, 6\}$ ➜ lines have approximate angles of {45, 63, 72, 76, 79, 81} degrees
  - For vertical lines, a is infinite

- **Better parametrization**

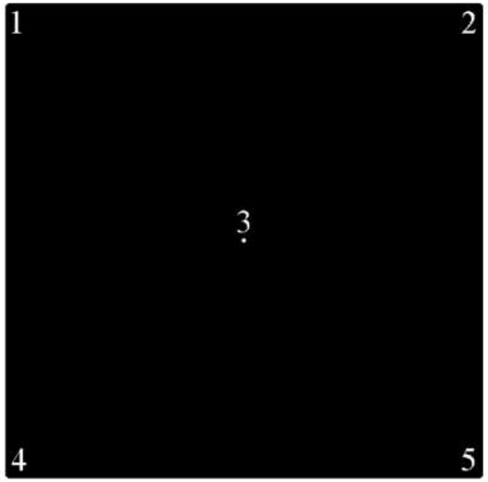
$$x \cos\theta + y \sin\theta = d$$

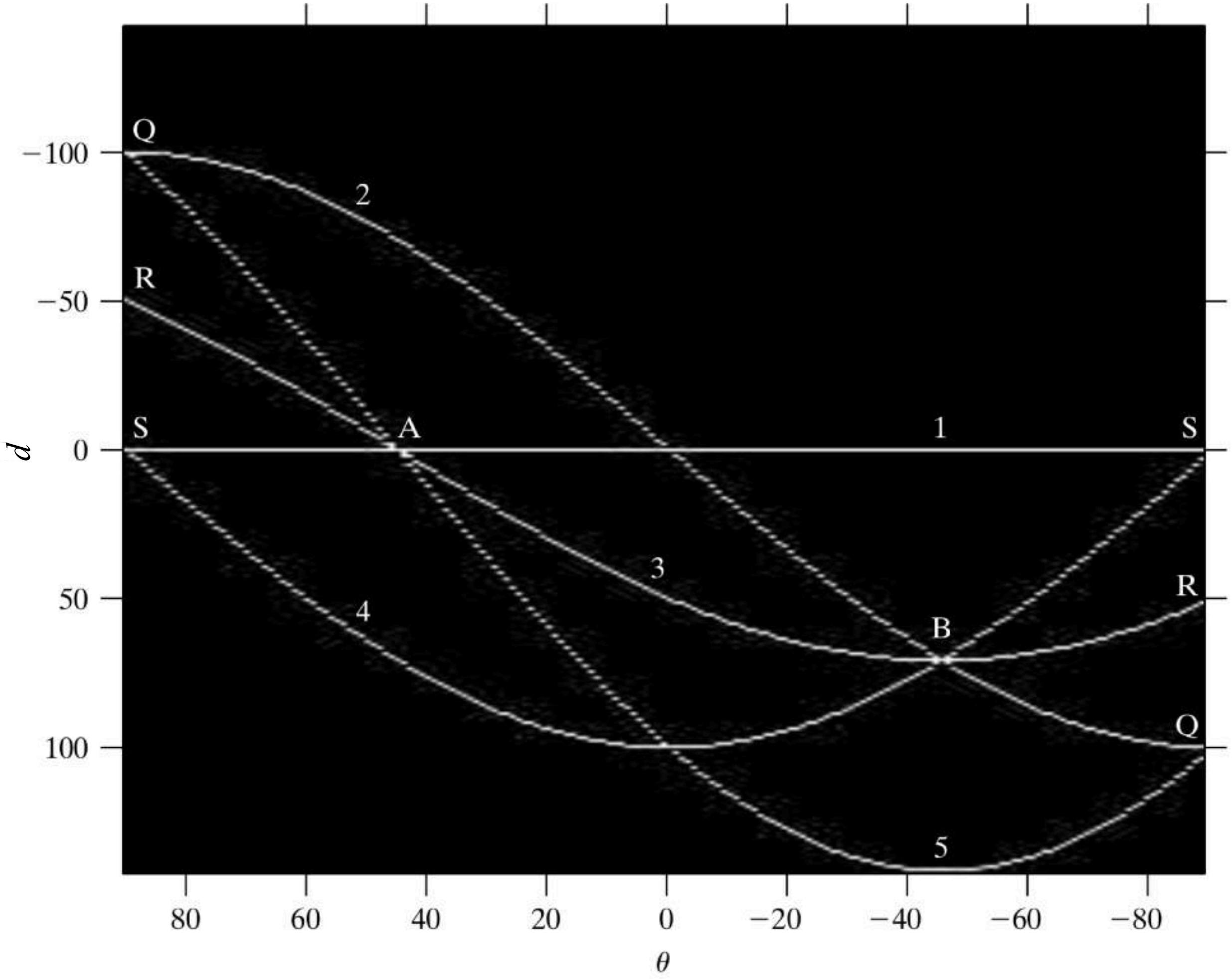
  - d : distance from origin
  - θ : angle from the positive x-axis



Hough transform (5/6)

- **Detecting lines** in the original image = **finding maxima in H**
  - All **pixels lying on a line** have identical values d and θ (or a and b)
  - $H(d,\theta)$ **has local maxima** at points corresponding to lines in original image

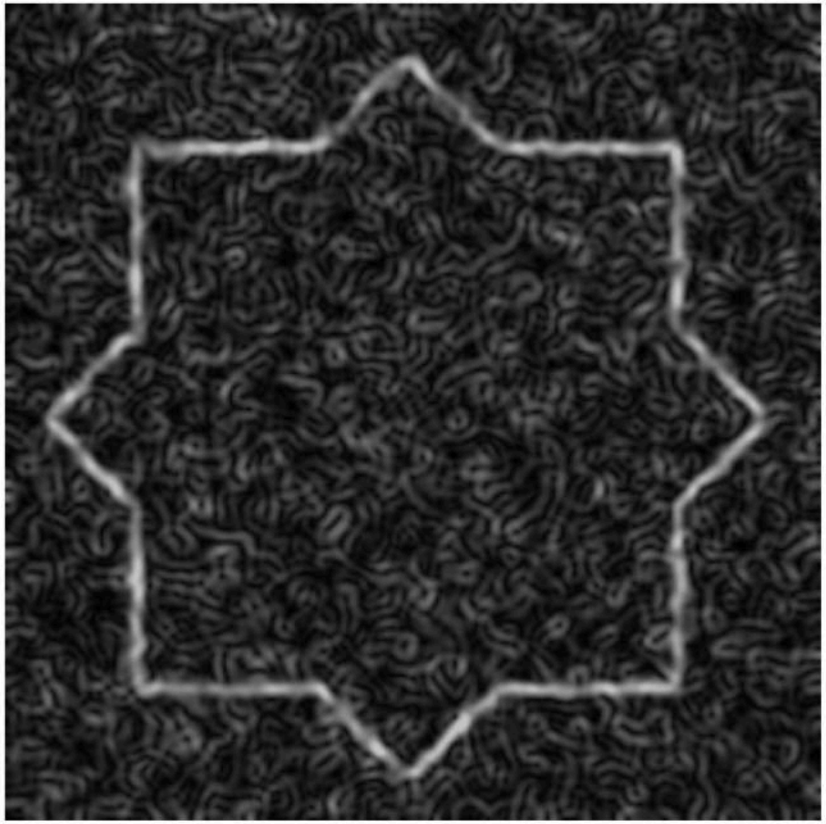




Hough transform

- It is a **very robust technique**

input image



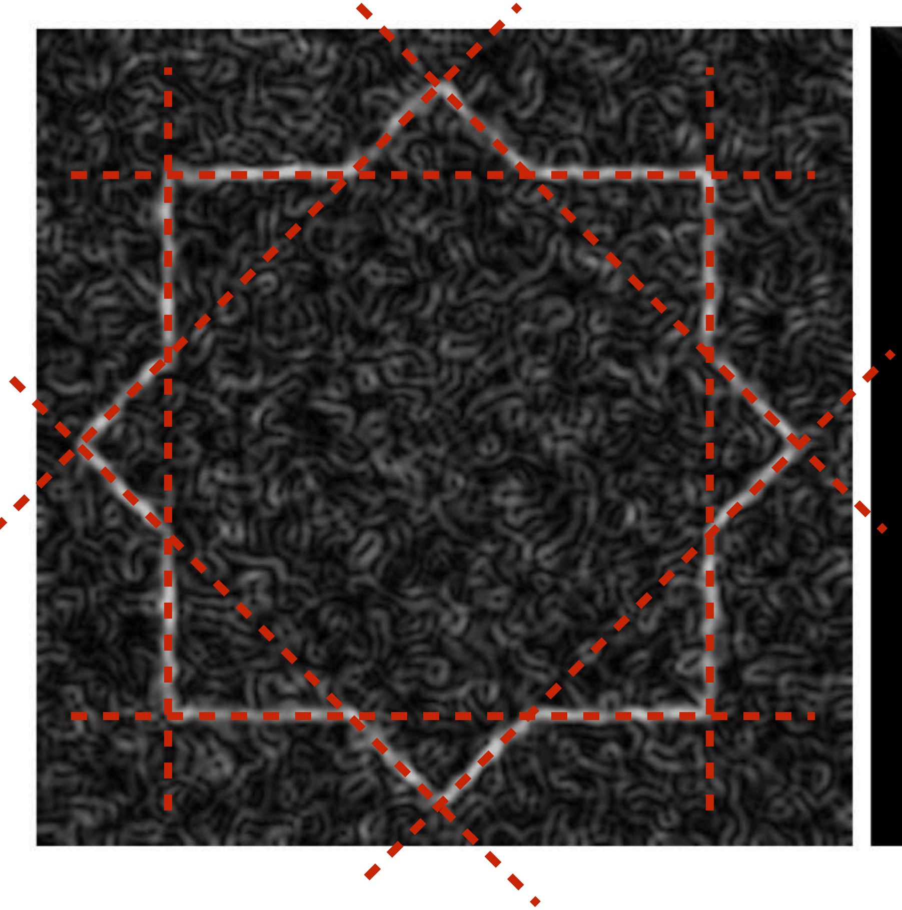
Hough transform H

?

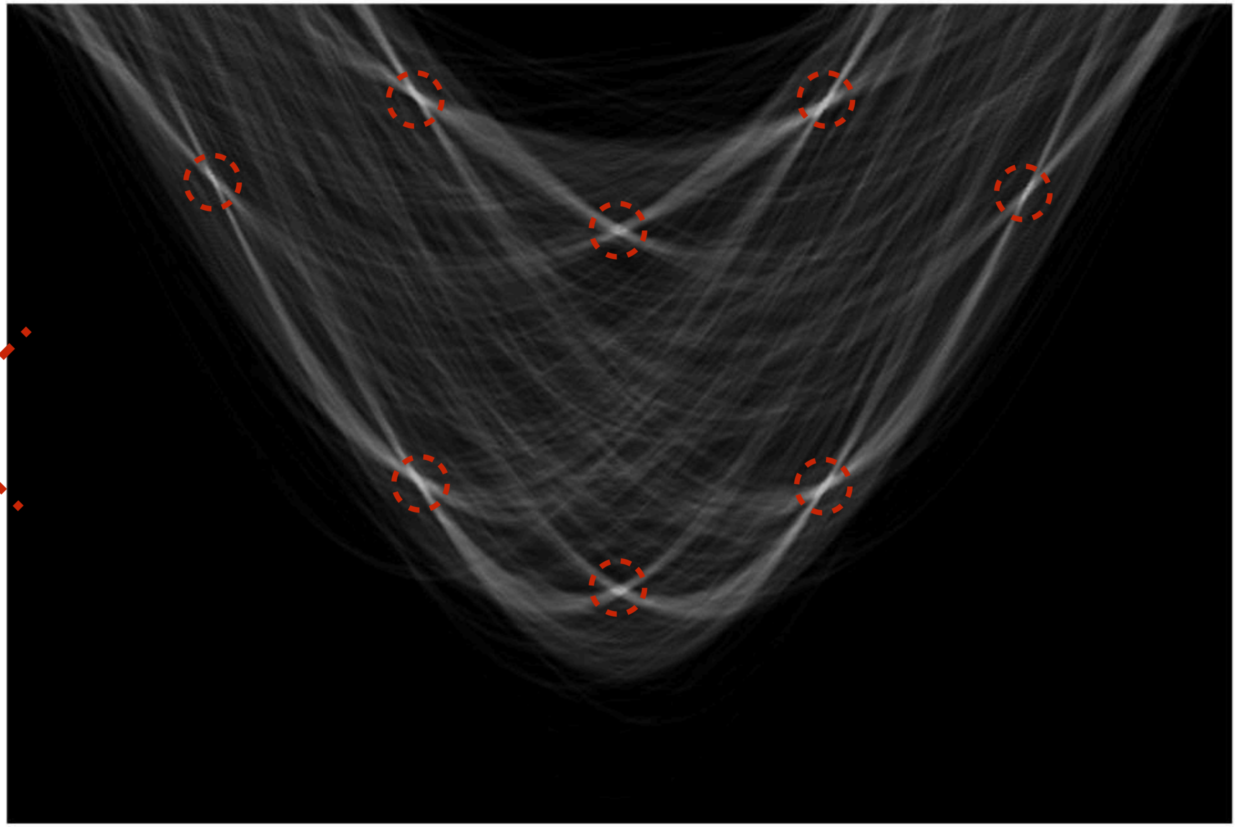
Hough transform

- It is a **very robust technique**

input image



Hough transform H

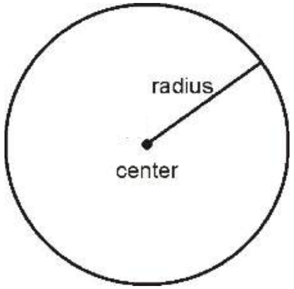


- Rather **insensitive to noise, object occlusion, missing line parts …**

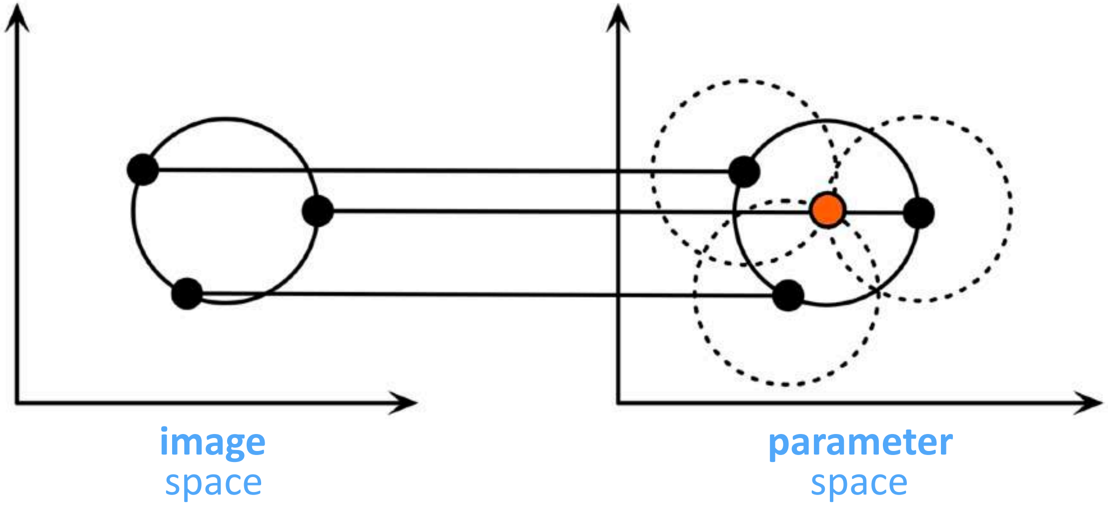
Extension of the Hough transform (1/3)

- **Q**: how can we **detect circles** in an image?
- **A**: the algorithm is basically **the same** as for detecting lines, except now we have **three parameters** (i.e. accumulator is 3D)
  - Circle described completely by **center** (a,b) and the **radius** R

$$x = a + R\cos(\theta)$$
$$y = b + R\sin(\theta)$$

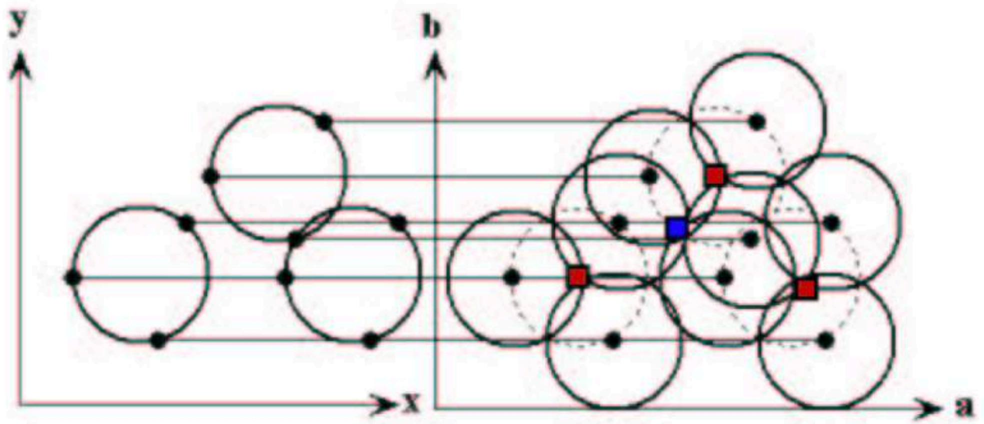


- **Case 1**: search for **known radius** R
  - 2D search space
  - For each edge pixel (x,y), define a **circle in the parameter space** centered at (x,y) with radius R
  - The **intersection of all such circles** in the parameter space would correspond to the *center point of the original circle*



Extension of the Hough transform (2/3)

- NB: **multiple circles with same R** can be recognized



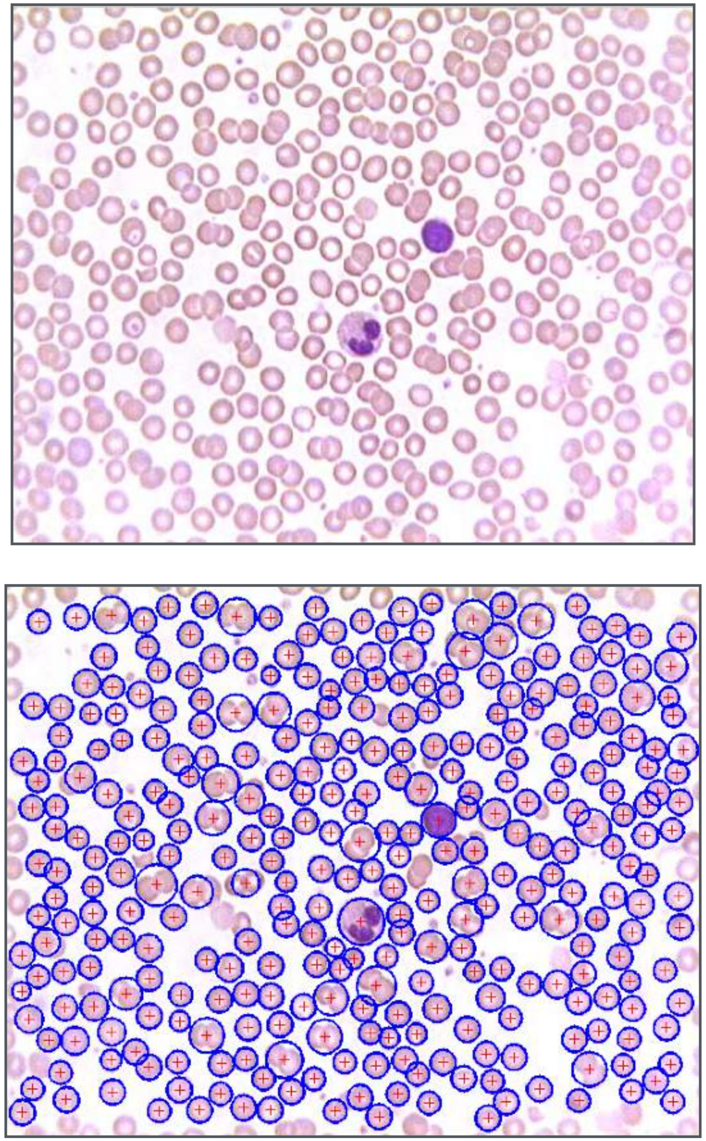
  - Overlap of circles can cause **spurious centers** to be found (*blue cell*)

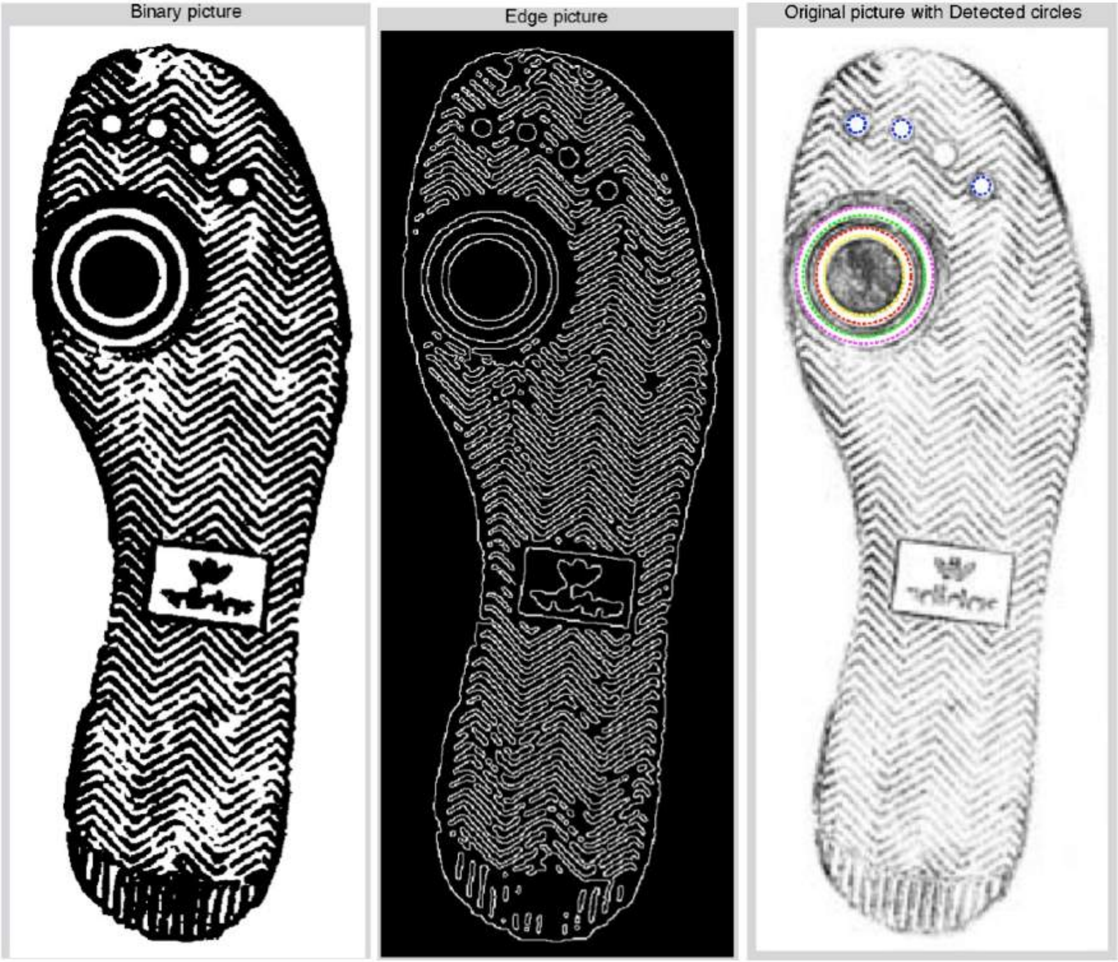
- **Case 2**: search for **unknown radius**
  - **Iterate** through all possible radii (discretized)
  - For each radius, use the **previous technique**
  - Find the maxima in the **3D accumulator**

- Similar variations of the Hough transform allows us to detect **all kinds of parametrized shapes**

Extension of the Hough transform (3/3)

- Examples



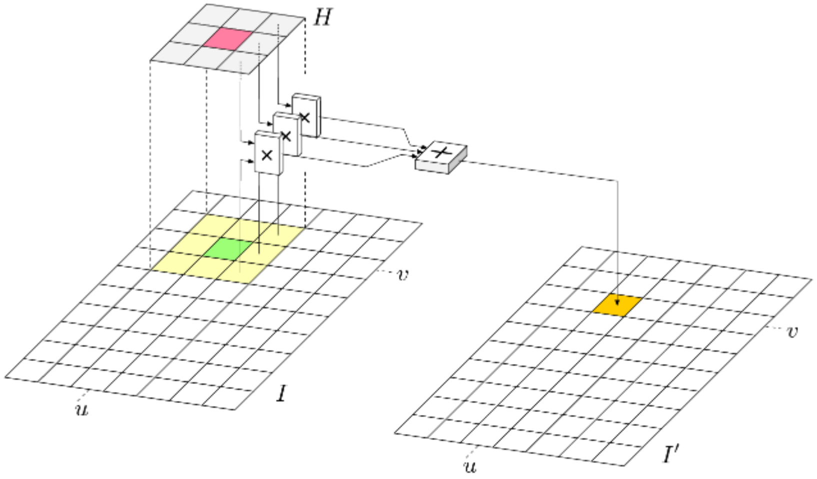


Template matching (1/3)

- Template matching is one of the most fundamental means of **object detection** within an image

- **IDEA**: **find a given template/patch** in an image
  - A *template/patch* is a small image with certain features
  - Search can be done using *correlation*

$$I'(u,v) = \sum_{(i,j)\in\mathcal{N}} I(u+i, v+j)\cdot H(i,j)$$



template image H

source image I

Template matching (2/3)

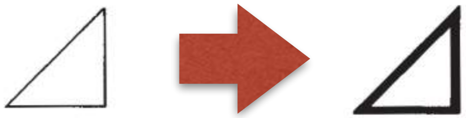
It is a very *basic* and *straightforward* approach

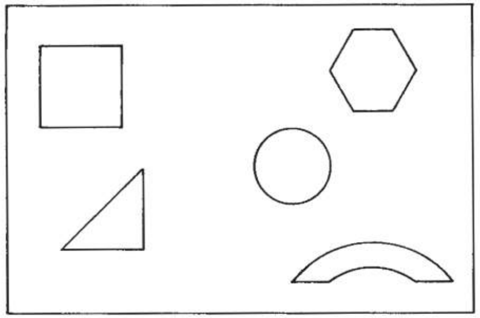
- We find the most correlating area
- **Limitations**
  - No *scale* invariant
  - No *rotation* invariant

- This basic method may be good enough for **basic applications**
  e.g. non scale and rotation changing input

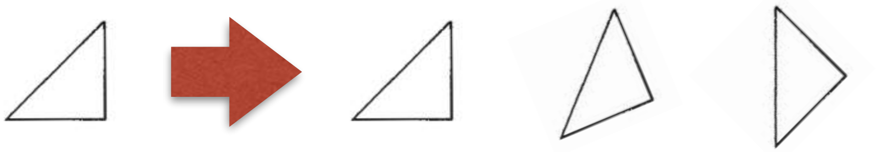
Simple tricks that work in some applications

- **Scale**: thicken the edges





- **Rotation**: try a discrete number of rotations



Template matching (3/3)

- More advanced approaches exist
  - **Pattern recognition**: branch of *machine learning* that focuses on the recognition of patterns and regularities in data