CS 484/684

Computational Vision

Image Pre-Processing

(the elements of filtering)

& Low-level Features

(low-dimensional, e.g. intensity, color, edges, corners, SIFT, ...)

raw input features (intensity, color): *output of sensor*

filtered features (edges, corners, SIFT, ...): *output of "**hand-designed**" low-level filters*

Later in the course:

compositions of **learnable filters**

(deep neural networks)

↓

high-level features

(high-dimensional, good for semantics)

Acknowledgements:

Steven Seitz, Aleosha Efros, David Forsyth, Gonzalez & Woods

Image Processing Basics

- Point Processing

  Extra Reading: Szeliski, Sec 3.1

  intensities, colors

  - gamma correction
  - window-center correction
  - histogram equalization

- Filtering (linear & non-linear neighborhood processing)

  Extra Reading: Szeliski, Sec 3.2-3.3

  contrast edges

  texture

  templates, patches

  - **convolution, gradient**
  - mean, Gaussian, and median filters
  - normalized cross-correlation (NCC)
  - etc…: Fourier, Gabor, wavelets (Szeliski, Sec 3.4-3.5)

- Higher-order gradient-based features

  Harris corners, MOPS, SIFT, etc.

  Extra Reading: Szeliski, Sec. 4.1

Summary of image transformations

- **image processing** or image transformation operation typically defines a new image g in terms of an existing image f.

Preview Examples:

Summary of image transformations

- **image processing** or image transformation operation typically defines a new image g in terms of an existing image f.

Preview Examples:

- **Geometric (domain) transformation**: $$g(x,y) = f(t_x(x,y), t_y(x,y))$$
  - What kinds of operations can this transformation $t = (t_x, t_y)$ perform?

Summary of image transformations

- **image processing** or image transformation operation typically defines a new image g in terms of an existing image f.

Preview Examples:

– **Geometric (domain) transformation**: $$g(x,y) = f(t_x(x,y), t_y(x,y))$$

- What kinds of operations can this transformation $t = (t_x, t_y)$ perform?

– **Range transformation**: $$g(x,y) = t(f(x,y))$$

- What kinds of operations can this transformation t perform?

Summary of image transformations

- **image processing** or image transformation operation typically defines a new image g in terms of an existing image f.

Preview Examples:

Topic 4

- **Geometric (domain) transformation**: $g(x,y) = f(t_x(x,y), t_y(x,y))$
  - What kinds of operations can this transformation $t = (t_x, t_y)$ perform?
- **Range transformation**: $g(x,y) = t(f(x,y))$
  - What kinds of operations can this transformation t perform?

point processing

- **Filtering** also generates new images from an existing image

$$g(x,y) = \int_{\substack{|u|<\varepsilon \\ |v|<\varepsilon}} h(u,v) \cdot f(x-u, y-v) \cdot du \cdot dv$$

  - more on filtering later

neighborhood processing

NOTE: neural networks use such operations (e.g. **activations**, **convolutions**) in each layer (e.g. CNNs in Topics 10-12)

Point Processing

$$g(x,y) = t(f(x,y))$$

$$t : R \rightarrow R$$

image range image range

for each original image intensity value I function $t(\cdot)$ returns a transformed intensity value $t(I)$.

$$\tilde{I} = t(I)$$

NOTE: we will often use notation I_p instead of $f(x,y)$ to denote intensity at pixel $p=(x,y)$

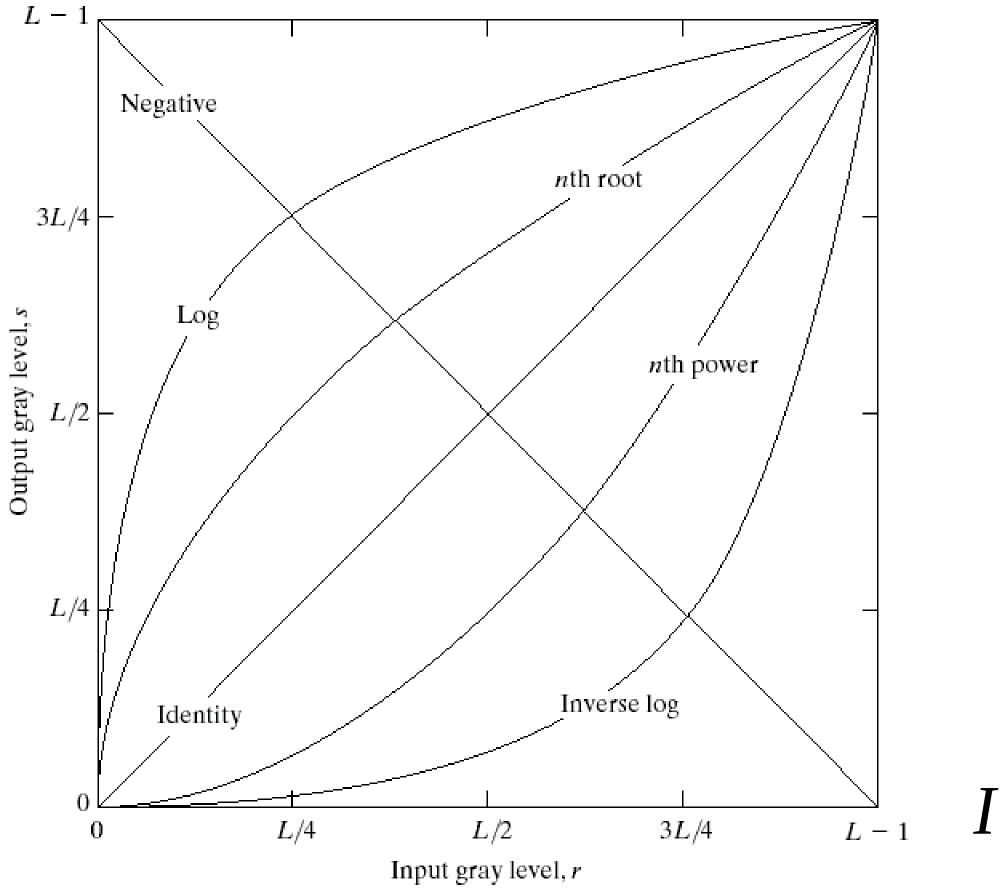
- **Important:** every pixel is for itself
  - spatial information is ignored!
- What can point processing do?

(we will focus on grey scale images, see Szeliski 3.1 for examples of point processing for color images)

Examples of gray-scale transforms *t*

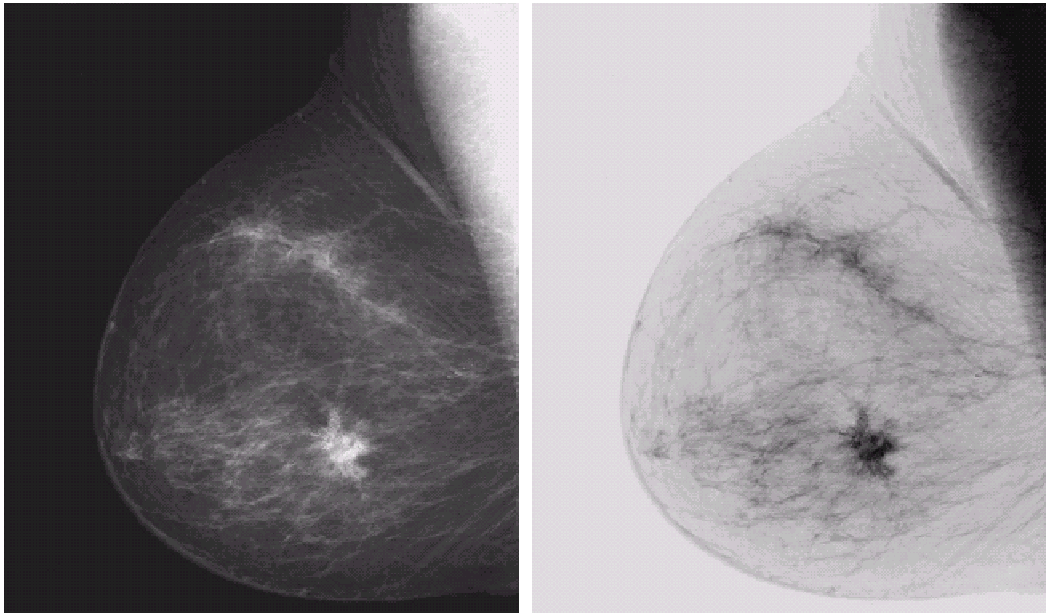
$$\tilde{I} = t(I)$$

FIGURE 3.3 Some basic gray-level transformation functions used for image enhancement.



Point Processing:

Negative



a b

FIGURE 3.4 (a) Original digital mammogram. (b) Negative image obtained using the negative transformation in Eq. (3.2-1). (Courtesy of G.E. Medical Systems.)

I_p or $f(x,y)$ $\qquad$ I'_p or $g(x,y)$

$$t(I) = 255 - I$$

$$g(x,y) = t(f(x,y)) = 255 - f(x,y)$$

Point Processing:

Power-law transformations t

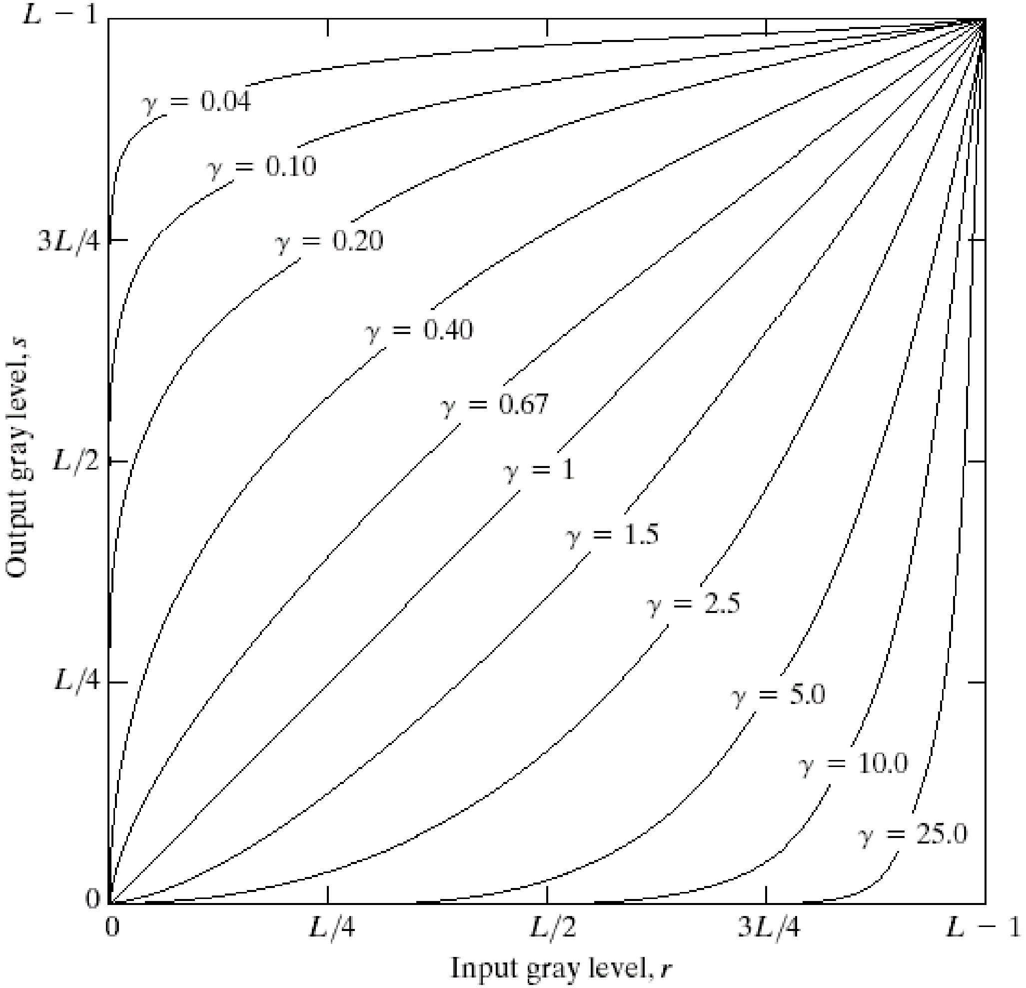


FIGURE 3.6 Plots of the equation $s = cr^{\gamma}$ for various values of γ ($c = 1$ in all cases).

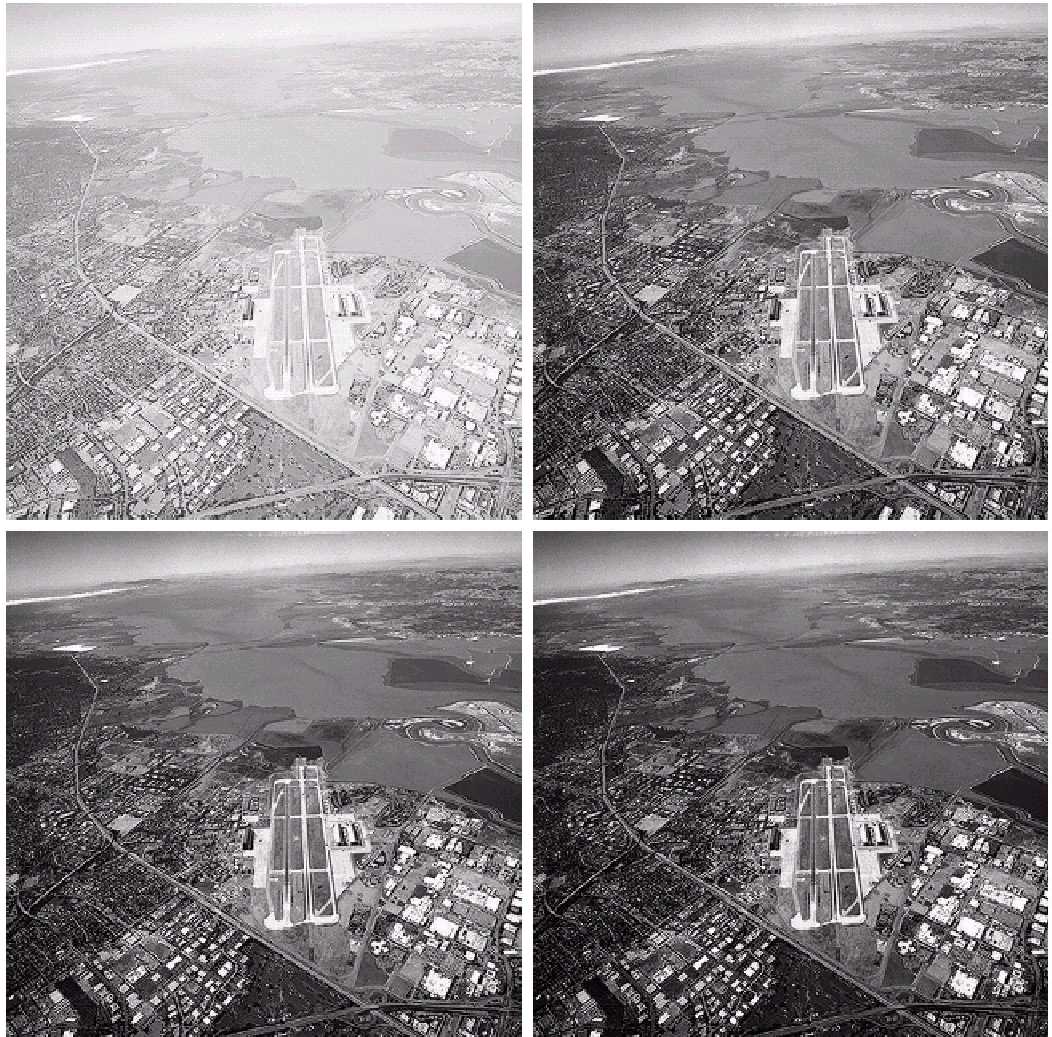
$$t(I) = I^{\gamma}$$

Point Processing:

Enhancing Image via Gamma Correction

a b
c d

FIGURE 3.9
(a) Aerial image. (b)–(d) Results of applying the transformation in Eq. (3.2-3) with $c = 1$ and $\gamma = 3.0, 4.0,$ and 5.0, respectively. (Original image for this example courtesy of NASA.)



Point Processing:

Understanding Image Histograms

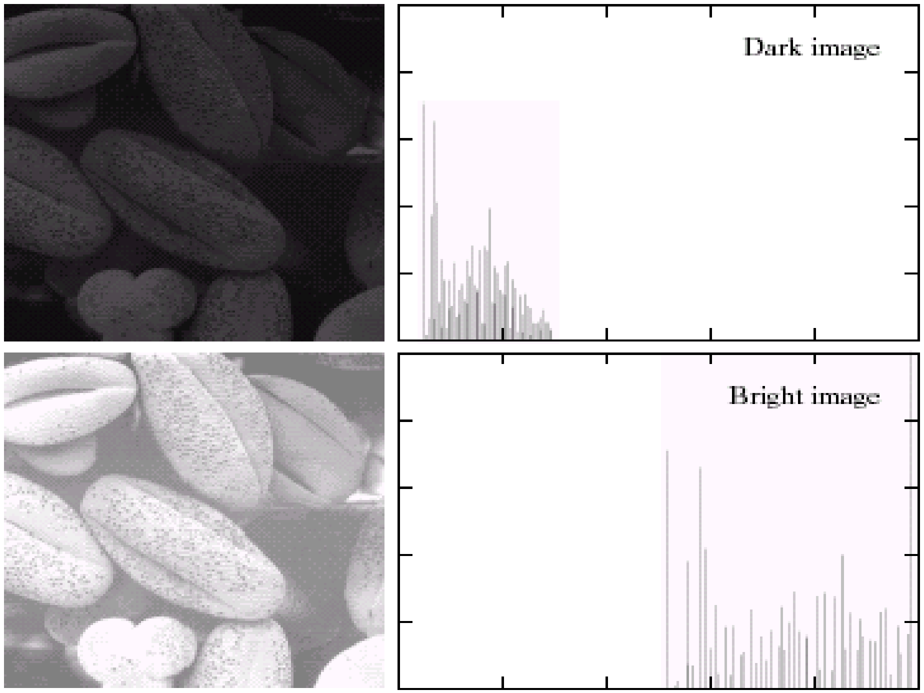


Image Brightness

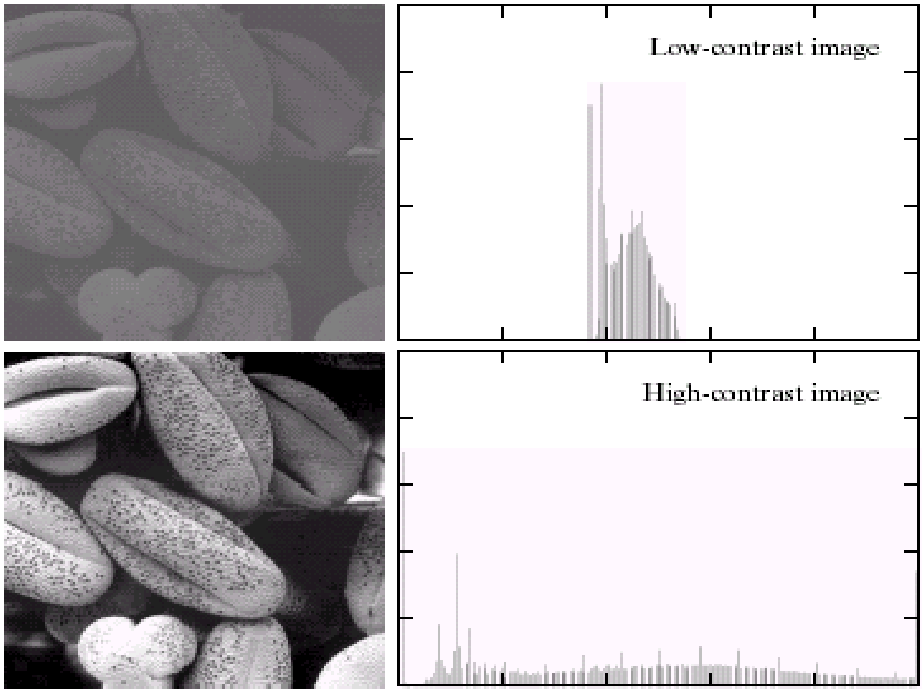


Image Contrast

probability of intensity i : $p(i) = \frac{n_i}{n}$

n_i ---number of pixels with intensity i

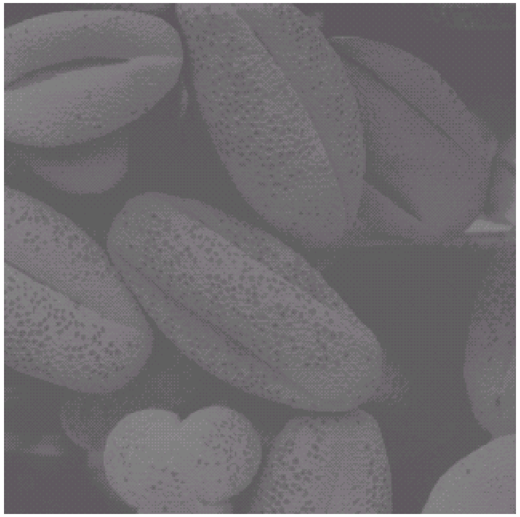
n ---total number of pixels in the image

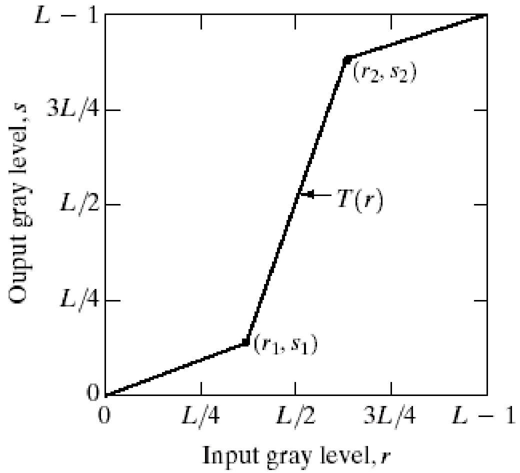
Point Processing:

Contrast Stretching

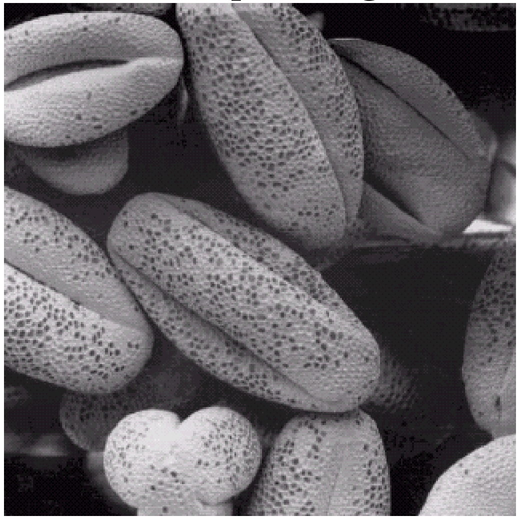
Note the difference between
contrast and **dynamic range**
(max I - min I) (min #bits needed)

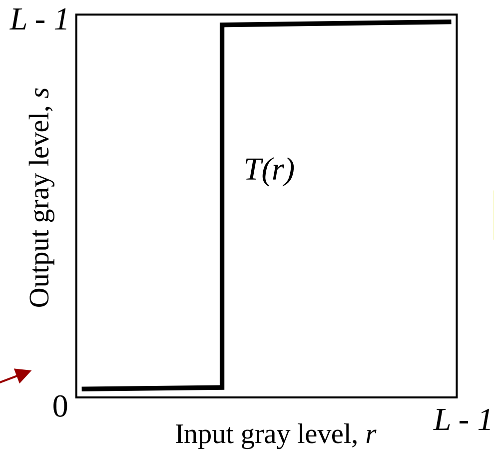
Original image





Output image





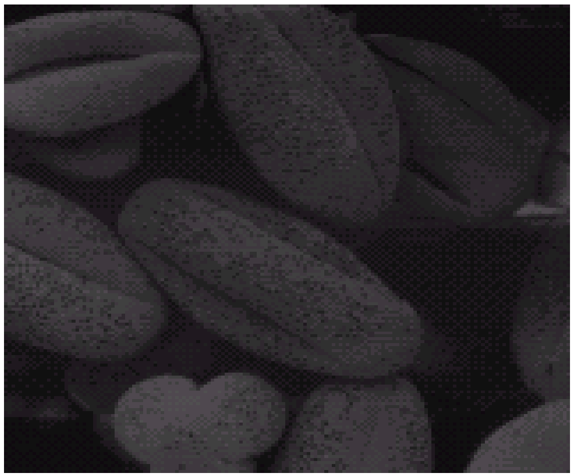
a.k.a. intensity *thresholding*

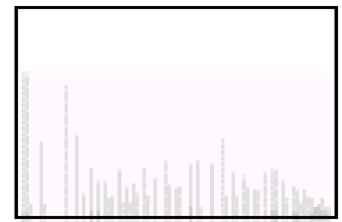
Point Processing:

Contrast Stretching

Original images | Histogram corrected images

1)





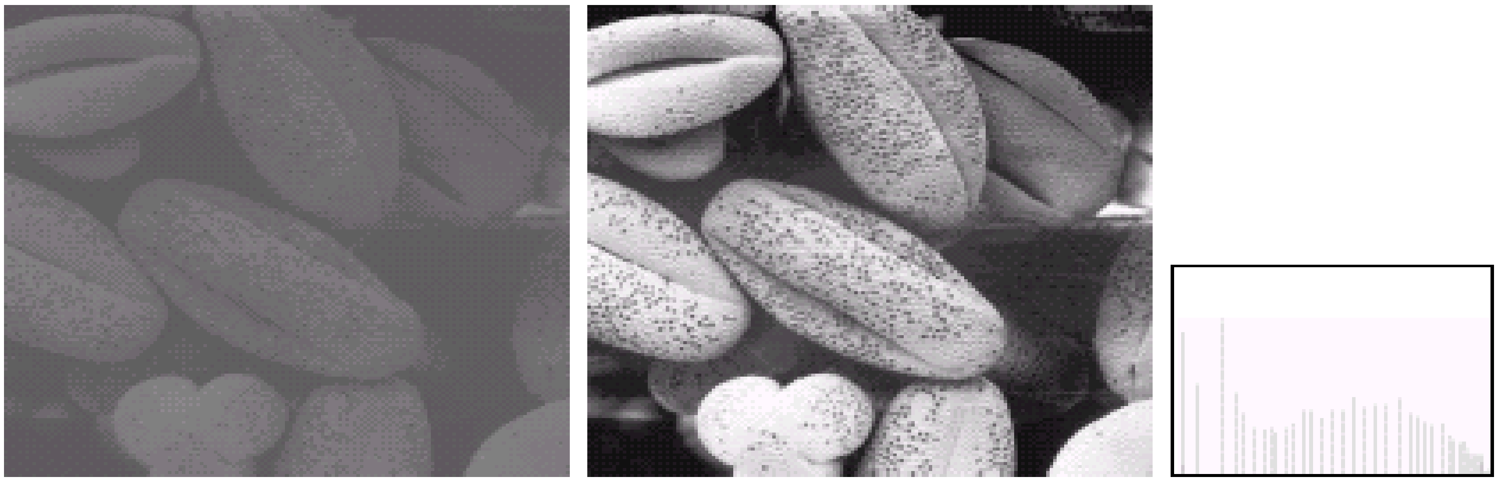
2)

Point Processing:

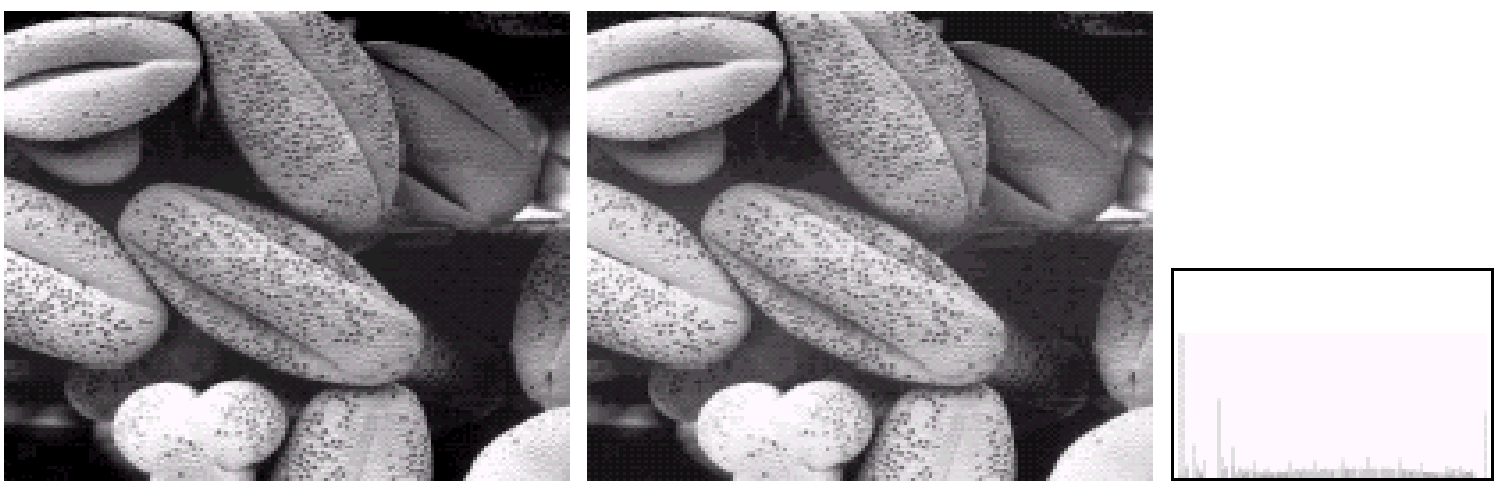
Contrast Stretching

Original images | Histogram corrected images

3)



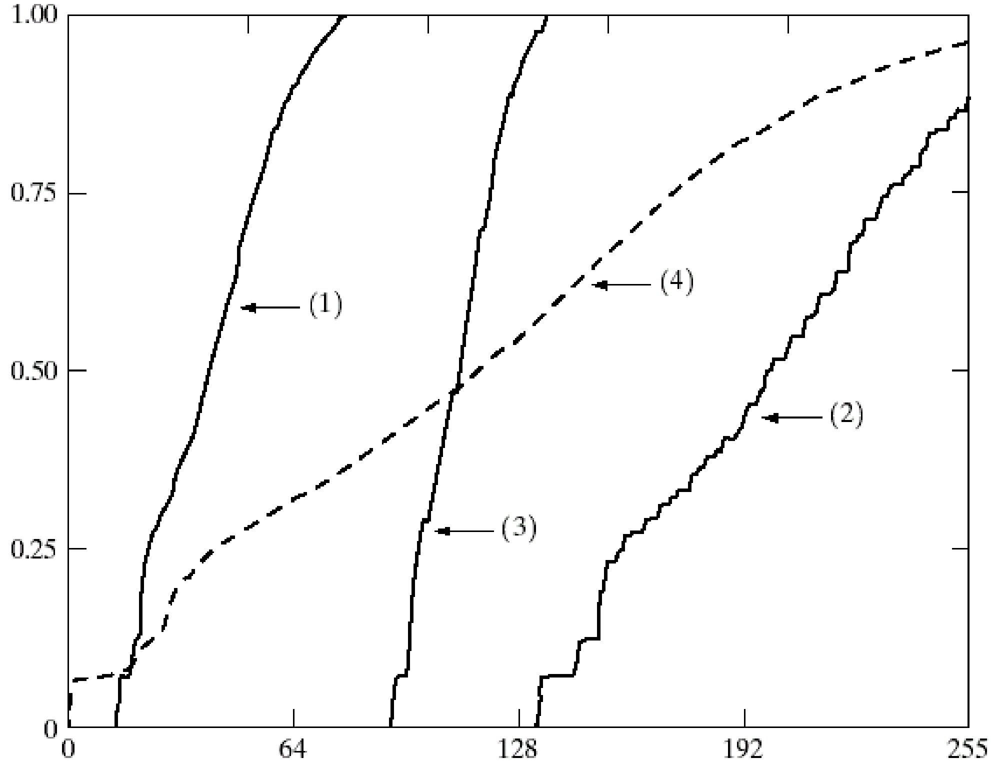
4)



One way to automatically select transformation t :

Histogram Equalization

FIGURE 3.18 Transformation functions (1) through (4) were obtained from the histograms of the images in Fig.3.17(a), using Eq. (3.3-8).



$$t(i) = \sum_{j=0}^{i} p(j) = \sum_{j=0}^{i} \frac{n_j}{n}$$

= cumulative distribution of image intensities

…see Gonzalez and Woods, Sec3.3.1, for more details

Histogram Equalization

$$t(i) = \sum_{j=0}^{i} p(j) = \sum_{j=0}^{i} \frac{n_j}{n}$$

= cumulative distribution of image intensities

Q: Why does that work?

Answer in probability theory:

I – random variable with *probability* density $p(i)$ over i in $[0,1]$

If $t(i)$ is a *cumulative* distribution function for I then

$I'=t(I)$ – is a random variable with *uniform* density over its range [0,1]

That is, transform image I' will have a uniformly-spread histogram (good contrast)

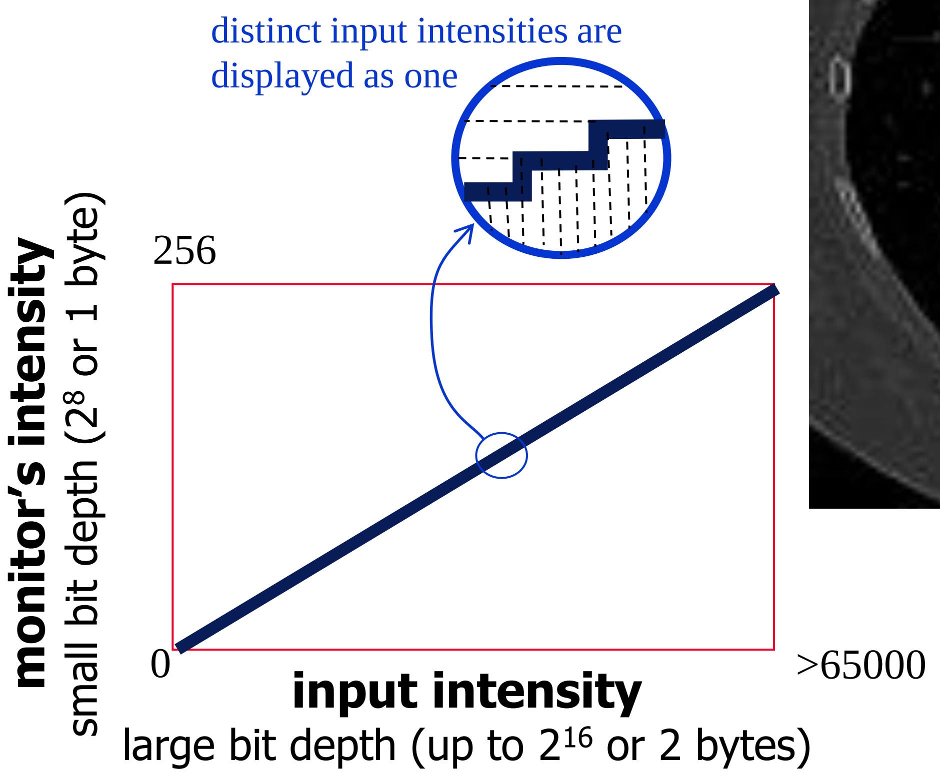
Point Processing:

Window-Center adjustment

input image range — output image range

$$t : R \to \tilde{R}$$

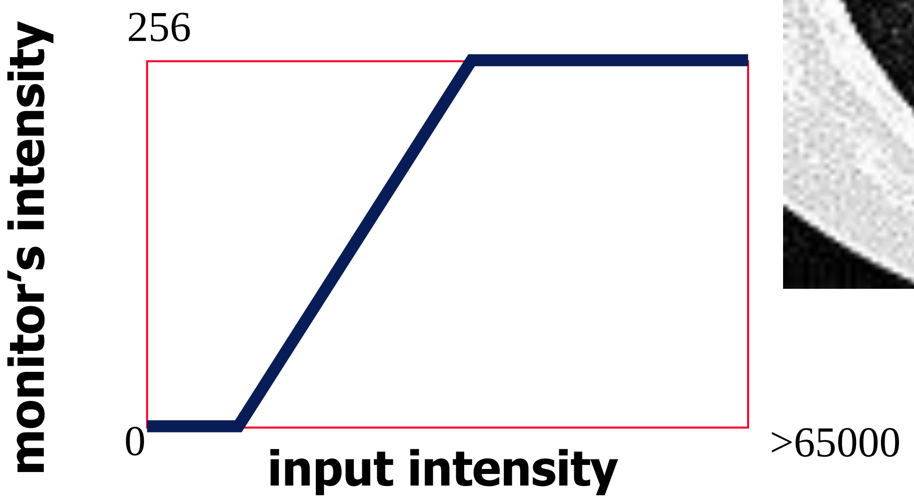
Displaying high dynamic range image (e.g. CT or MR) over low dynamic range monitor



A lot of information is lost!

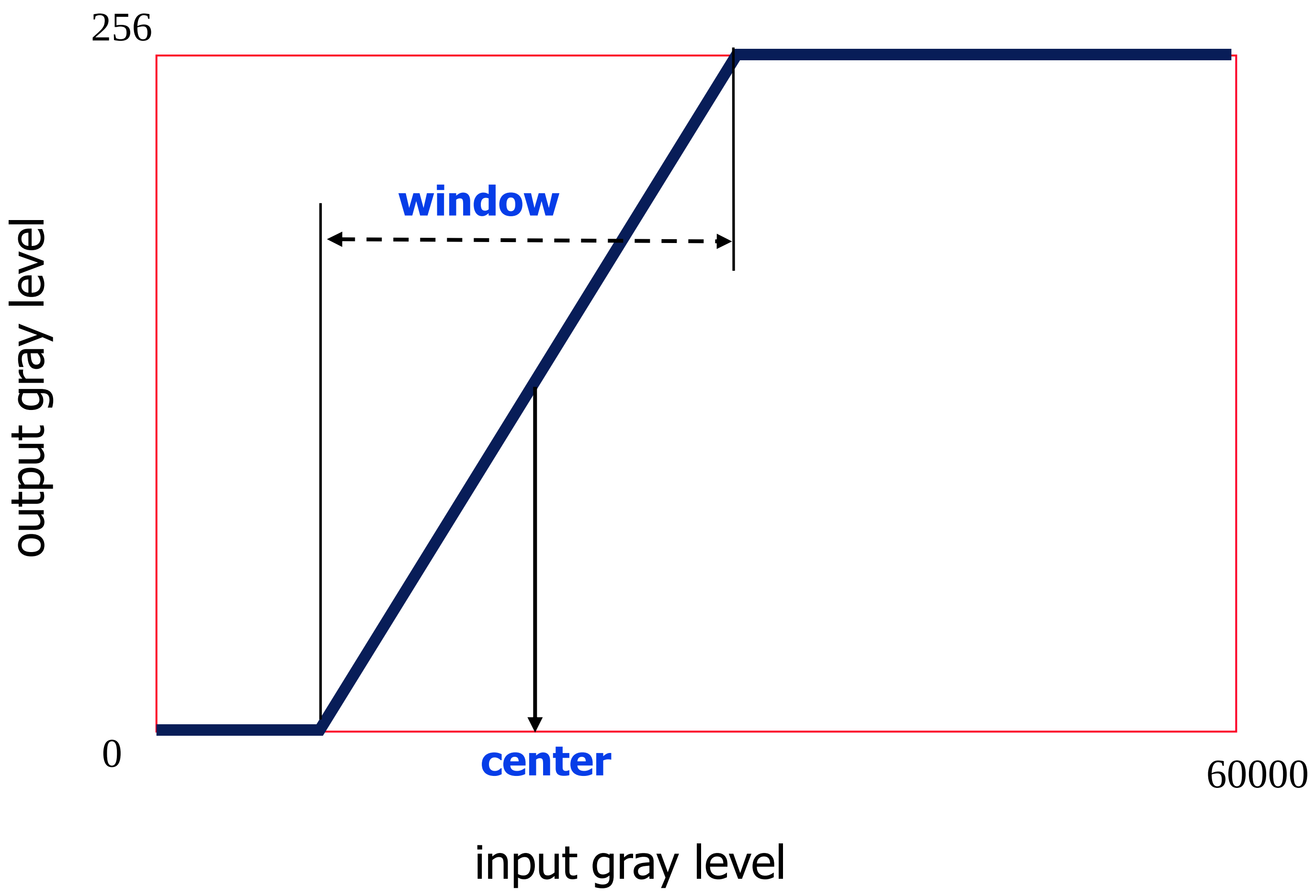
Point Processing:

Window-Center adjustment



Point Processing:

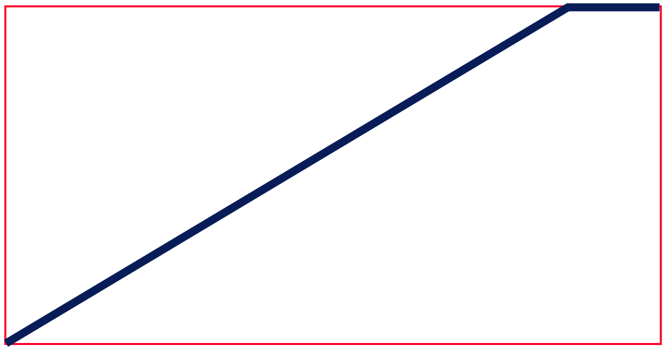
Window-Center adjustment



Point Processing:

Window-Center adjustment

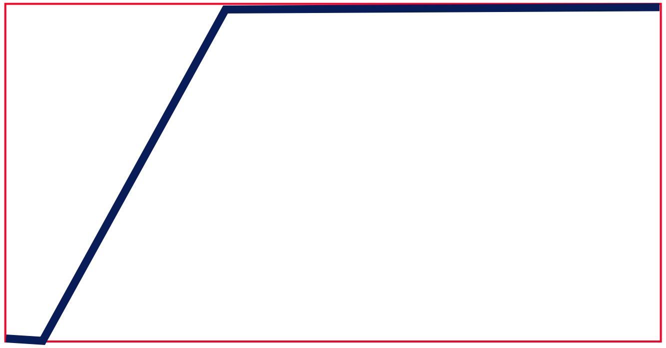
Window = 4000
Center = 500



Point Processing:

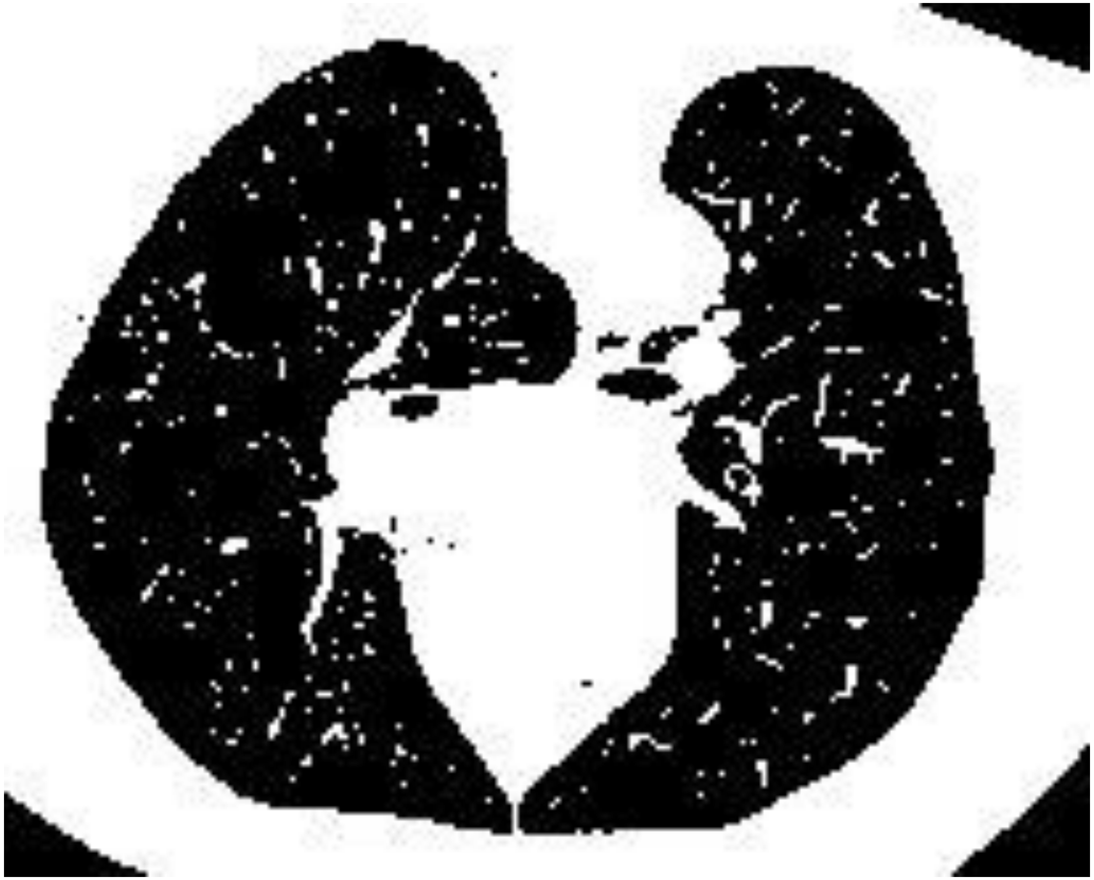
Window-Center adjustment

Window = 800
Center = 500



Window-Center adjustment

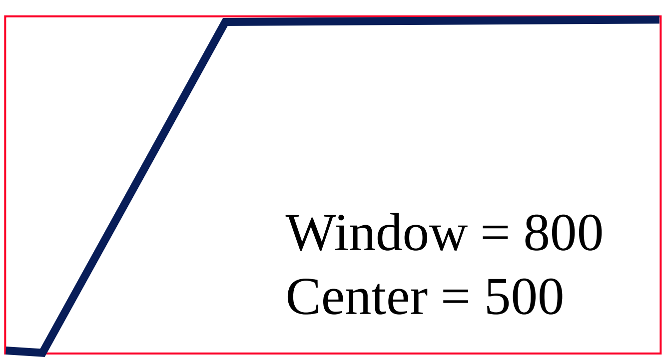
Window = 0
Center = 500

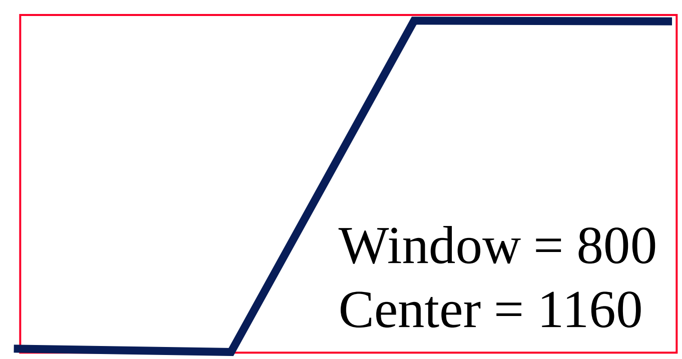


If *window=0* then we get binary image *thresholding*

Point Processing:

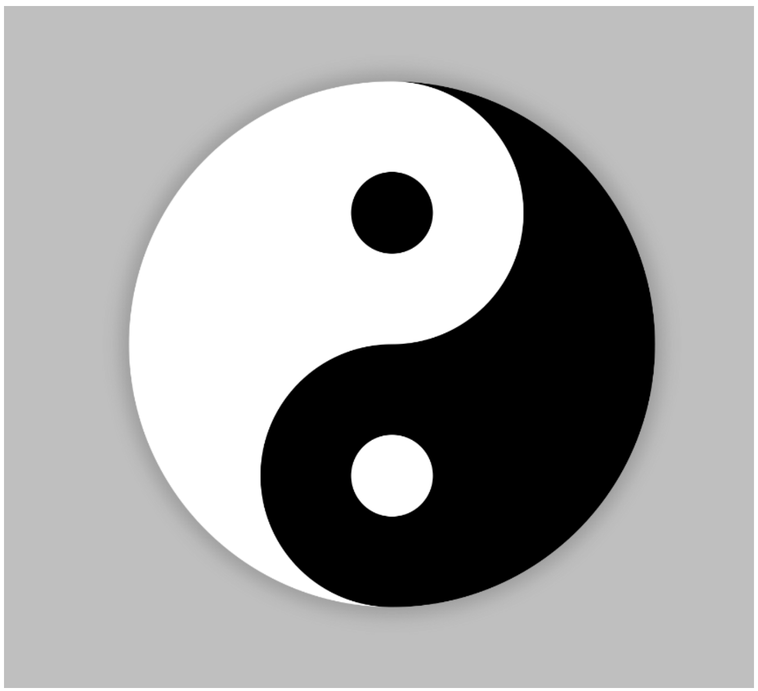
Window-Center adjustment



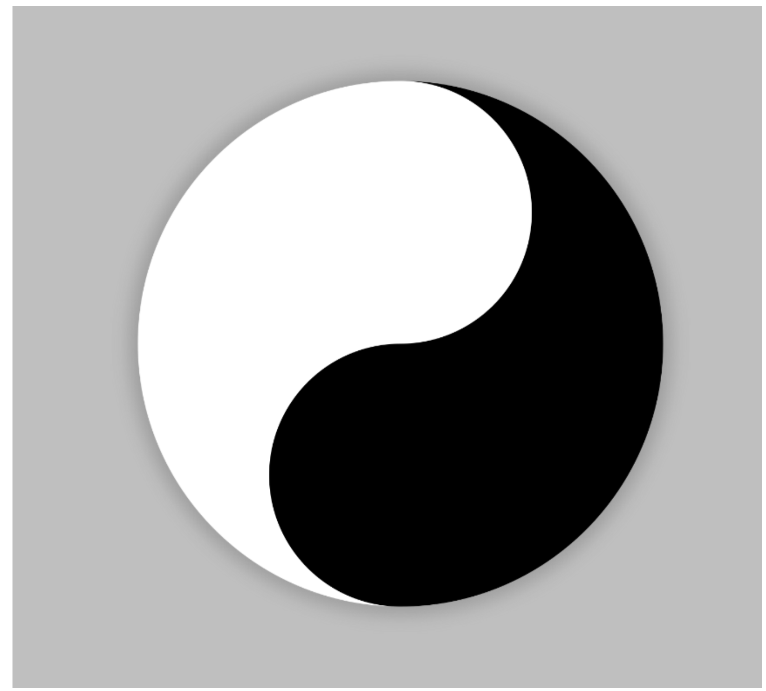


Q. Is This an Example of Point Processing?

$f(x, y)$

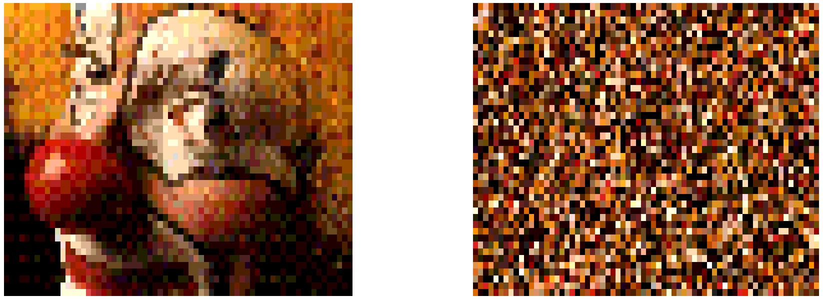


$g(x, y) = t(f(x, y))$



Neighborhood Processing (or filtering)

- Q: What happens if I reshuffle all pixels within the image?



- A: It's histogram won't change.
  No point processing will be affected…

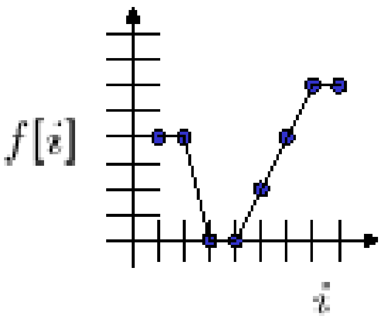
- Images contain a lot of "spatial information"

Readings: Szeliski, Sec 3.2-3.3

Neighborhood Processing (filtering)

Linear image transforms

Let's start with 1D image (a signal): f[i]



$$f[i] = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

A very general and useful class of transforms are the **linear transforms** of f, defined by a matrix M

$$\begin{bmatrix} * & * & \cdots & * \\ * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \end{bmatrix} \begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix} = \begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix}$$

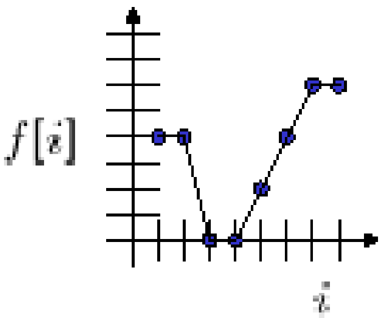
$$M[i,j] \qquad f[i] \qquad g[i]$$

$$g[i] = \sum_{j=1} M[i,j] f[j]$$

Neighborhood Processing (filtering)

Linear image transforms

Let's start with 1D image (a signal): f[i]



$$f[i] = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

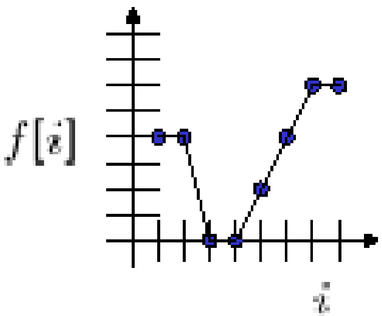
matrix M

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad f[i] \longrightarrow$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad f[i] \longrightarrow$$

Linear image transforms

Let’s start with 1D image (a signal): f[i]



$$f[i] = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

matrix M

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad f[i] \longrightarrow$$

$$\frac{1}{2}\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad f[i] \longrightarrow$$

Neighborhood Processing (filtering)

Linear shift-invariant filters

NOTE: in image analysis, ML, statistics, and signal processing the common term *kernel* stands for some "window" function. Not to be confused with "null space" in linear algebra.

matrix M

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ a & b & c & 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & c & 0 & 0 & 0 & 0 \\ 0 & 0 & a & b & c & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & c & 0 & 0 \\ 0 & 0 & 0 & 0 & a & b & c & 0 \\ 0 & 0 & 0 & 0 & 0 & a & b & c \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix}$$

This pattern is very common
- same entries in each row
- all non-zero entries near the diagonal

$$g = M \cdot f$$

It is known as a **linear shift-invariant filter** and is represented by a so-called (1D) **kernel** or **mask** *h*:

$$h[i] = [a \;\; b \;\; c]$$

and can be written (for kernel of size *2k+1*) as:

$$g[i] = \sum_{u=-k}^{k} h[u] \cdot f[i+u]$$

The above allows negative filter indices.

Neighborhood Processing (filtering)

Linear shift-invariant filters

Linearity of H: H (f+g) = Hf + Hg

Shift-invariance of H: H (Sf) = S (Hf) for shift operator, *e.g* S = $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Any **linear shift-invariant** operator H can be expressed in a form like

$$h[i] = [a \;\; b \;\; c \;\; ...]$$

$$g[i] = \sum_{u=-k}^{k} h[u] \cdot f[i+u]$$

Neighborhood Processing (filtering)

2D linear transforms

Similar linear neighborhood-processing operations on 2D images can also be expressed via matrix multiplication after concatenating all image rows into one long vector (in a "*raster-scan*" order):

$$\hat{f}[i] \;=\; f[\lfloor i/m \rfloor, i\%m]$$

$$\underbrace{\begin{bmatrix} * & * & \cdots & * \\ * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \end{bmatrix}}_{M[i,j]} \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix}}_{\hat{f}[i]} = \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix}}_{\hat{g}[i]}$$

However, matrix M will have many zeros and kernel-based representation is significantly simpler…

Neighborhood Processing (filtering)

2D filtering

2D image $f[i,j]$ can be filtered by **2D kernel** $h[u,v]$ to produce output image $g[i,j]$:

$$g[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v] \cdot f[i+u, j+v]$$

This is called a **cross-correlation** operation and written:

$$g = h \circ f$$

h is called "**kernel**" or "**mask**" or "**filter**" which representing a given "window function"

Neighborhood Processing (filtering)

2D filtering

Closely related **convolution** operation is defined slightly differently

$$g[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v] \cdot f[i-u, j-v]$$

It is written as:

$$g = h * f$$

$$= \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[-u,-v] \cdot f[i+u, j+v]$$

Convolution is cross-correlation where the filter is flipped both horizontally and vertically before being applied to the image:

If $h[u,v] = h[-u,-v]$ then convolution is not different from cross-correlation

Convolution has additional "technical" properties: commutativity, associativity. Also, "nice" properties wrt Fourier analysis.

(see Szeliski Sec 3.2, Gonzalez and Woods Sec. 4.6.4)

Cross-correlation is a statistically motivated operation computing similarity between a pattern defined by kernel *h* (seen as an image too) and patches at different locations inside image *f* (more later)

Neighborhood Processing (filtering)

convolution = linearity + shift-invariance

$$g[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v] \cdot f[i-u, j-v]$$

$$g = h * f \qquad = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[-u,-v] \cdot f[i+u, j+v]$$

Note: any linear shift-invariant operation is a **convolution** (or **cross-correlation**)

NOTE: since the two operations are equivalent after trivial kernel "flipping", in practice, they are often used indiscriminately. For example, CNNs implementations often use cross correlations.

kernels + convolution in Image Processing

https://en.wikipedia.org/wiki/Kernel_(image_processing)

Examples to be discussed now:

- **denoising** (mean filtering, Gaussian kernel)
- **edge detection** (differentiation, gradient & Laplace kernels)
- **sharpening** (unsharp mask, LoG & DoG kernels)
- **pattern matching** (template matching, NCC)

Also in this topic:

- non-linear filtering (median filtering, Harris Corners, ...)

- feature localization (detection) – non-maximum suppression

  VS.

  feature descriptors (matching) – MOPS, SIFT, ...

2D filtering for

Noise Reduction

Original

Salt and pepper noise

Common types of noise:

- **Salt and pepper noise**: random occurrences of black and white pixels
- **Impulse noise:** random occurrences of white pixels
- **Gaussian noise**: variations in intensity drawn from a Gaussian normal distribution

Impulse noise

Gaussian noise

Mean filtering

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$f[x, y]$

		10							

$g[x, y]$

Mean filtering

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$f[x,y]$

				80					
		10							

$g[x,y]$

Mean filtering

side effect of mean filtering: **blurring**

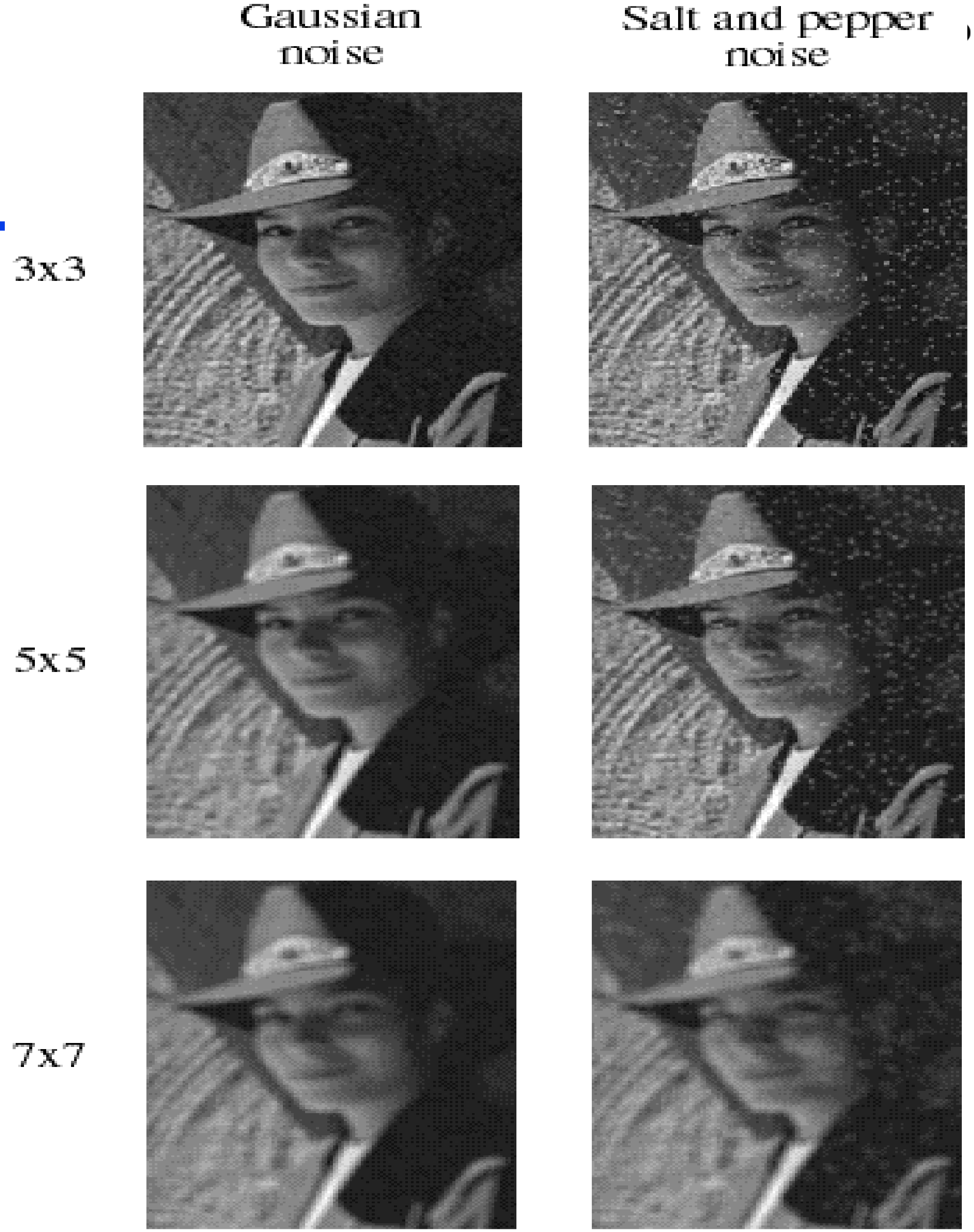
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$f[x, y]$

	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

$g[x, y]$

Effect of mean filters



Mean kernel

- What's the kernel for a 3x3 mean filter?

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$$\frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

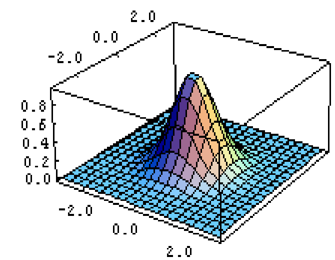
Gaussian Filtering

- A Gaussian kernel gives less weight to pixels further from the center of the window

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$$\frac{1}{16} \cdot$$

1	2	1
2	4	2
1	2	1



discrete approximation of
a Gaussian (density) function

$$h(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{\sigma^2}}$$

NOTE: ***Gaussian*** distribution is a synonym for ***Normal*** distribution!

Gaussian Filtering

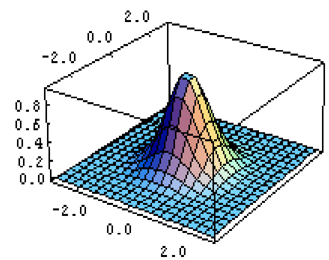
- A Gaussian kernel gives less weight to pixels further from the center of the window

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$\frac{1}{16} \cdot$

1	2	1
2	4	2
1	2	1

$\mathbf{G}_\sigma$

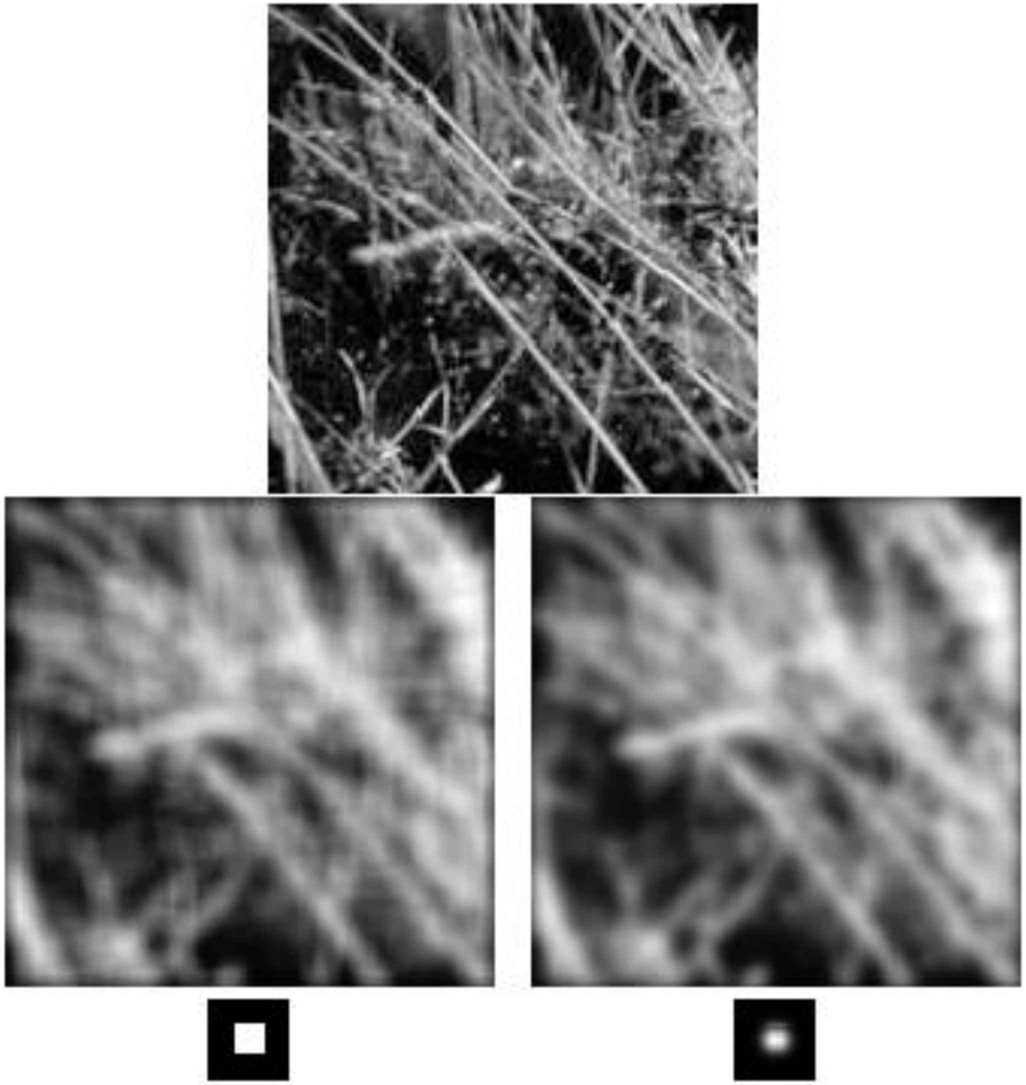


discrete approximation of
a Gaussian (density) function

$$h(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{\sigma^2}}$$

We denote such Gaussian kernels by ***G*** or ***G***$_\sigma$

Mean vs. Gaussian filtering



no rotational invariance

Median filters

- A **Median Filter** operates over a window by selecting the median intensity in the window.

- What advantage does a median filter have over a mean filter?

- Is a median filter a kind of convolution?

  - No, median filter is non-linear (homework exercise)

Comparison: **salt and pepper** noise

Comparison: **Gaussian** noise

Mean | Gaussian | Median

3x3

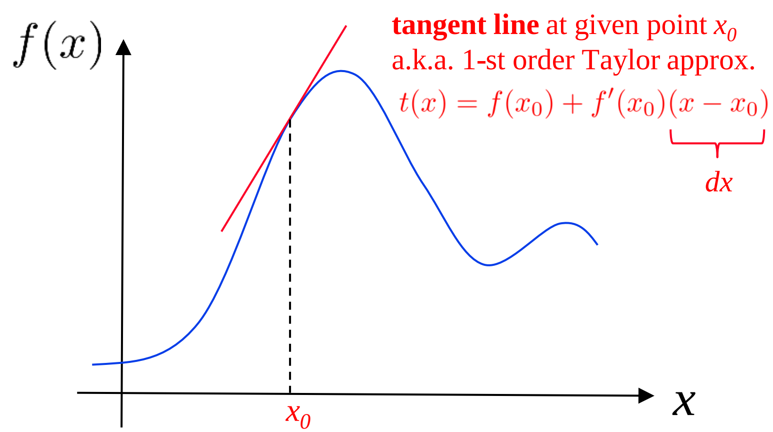
5x5

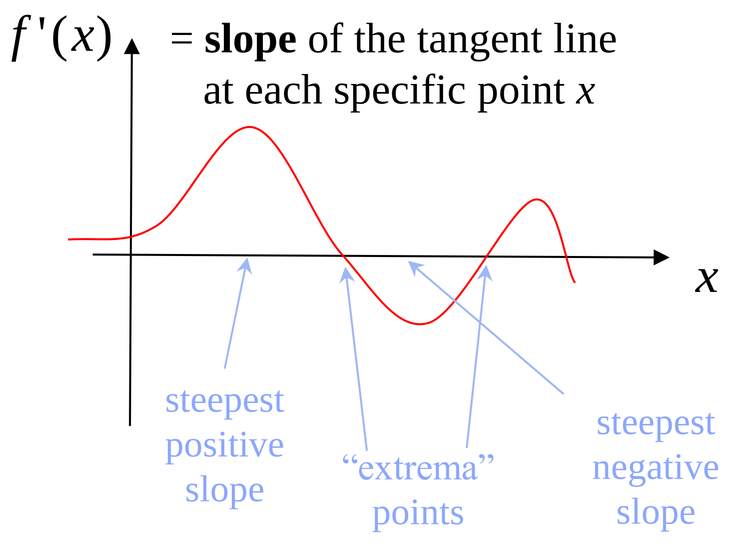
7x7

Differentiation and convolution

- Recall for $f(x)$

$$f'(x) = \lim_{\varepsilon \to 0} \left(\frac{f(x+\varepsilon) - f(x)}{\varepsilon} \right)$$

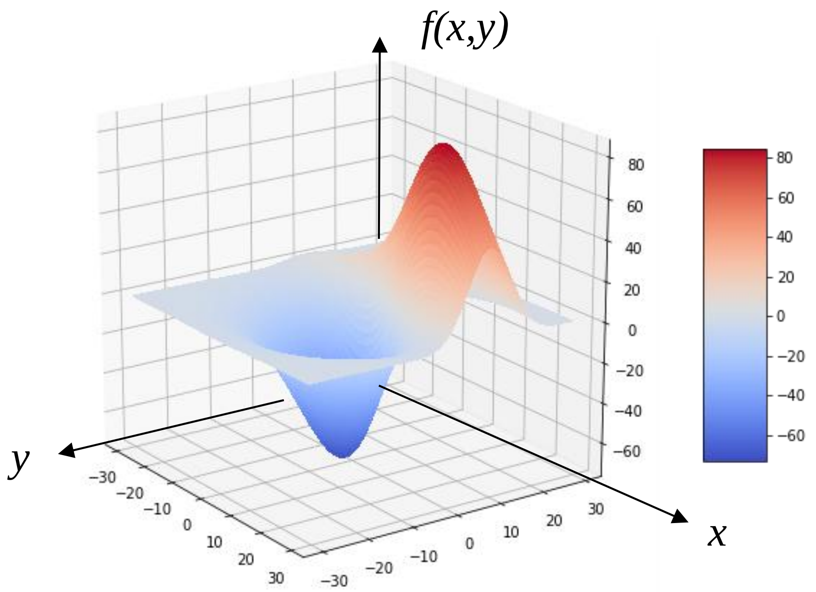




- Useful for analyzing *f(x)*
- How to extend differentiation to multivariate functions like $f(x, y)$ or $f(x, y, z)$?

Differentiation and convolution

$f(x, y)$



3D plot of *f*

What is "**slope**" of *f(x,y)* at a given point (*x*,*y*)?

Some intuition first:

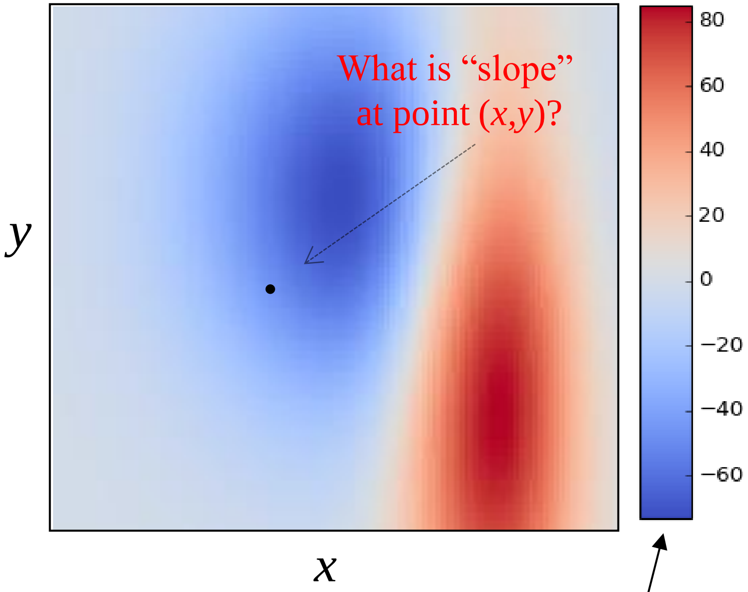
- For functions *f(x,y)* think about the **slope** of a tangent plane for its 3D plot at point (x,y).
- Such a slope could be characterized by direction and magnitude - attributes of a **vector** (?)

Differentiation and convolution

$$f(x, y)$$

"heat-map" visualization of f

domain of $f(x,y)$ in R^2



range of $f(x,y)$

Differentiation and convolution

- For $f(x,y)$ use fixed directions

(e.g. "partial" derivatives)

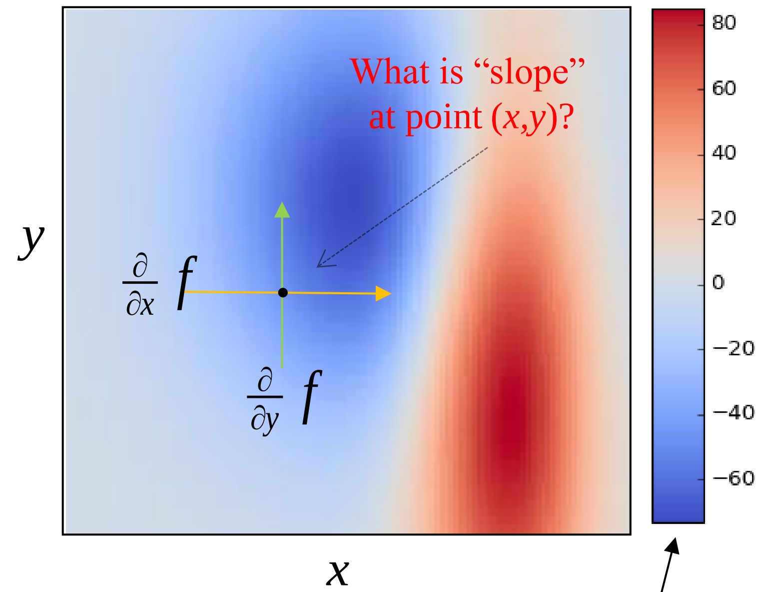
$$\frac{\partial}{\partial x} f = \lim_{\varepsilon \to 0}\left(\frac{f(x+\varepsilon, y) - f(x,y)}{\varepsilon}\right)$$

slope in *x*-direction

$$\frac{\partial}{\partial y} f = \lim_{\varepsilon \to 0}\left(\frac{f(x, y+\varepsilon) - f(x,y)}{\varepsilon}\right)$$

slope in *y*-direction

domain of *f(x,y)*



range of *f(x,y)*

NOTE: we compute partial derivatives at specific points (*x*,*y*) so, formally one can write

$$\frac{\partial}{\partial x} f(x,y) \quad \text{or} \quad \frac{\partial}{\partial y} f(x,y)$$

Another common notation

$$f_x'(x,y) \qquad f_y'(x,y)$$

Differentiation and convolution

- For $f(x, y)$ use fixed directions

(e.g. "partial" derivatives)

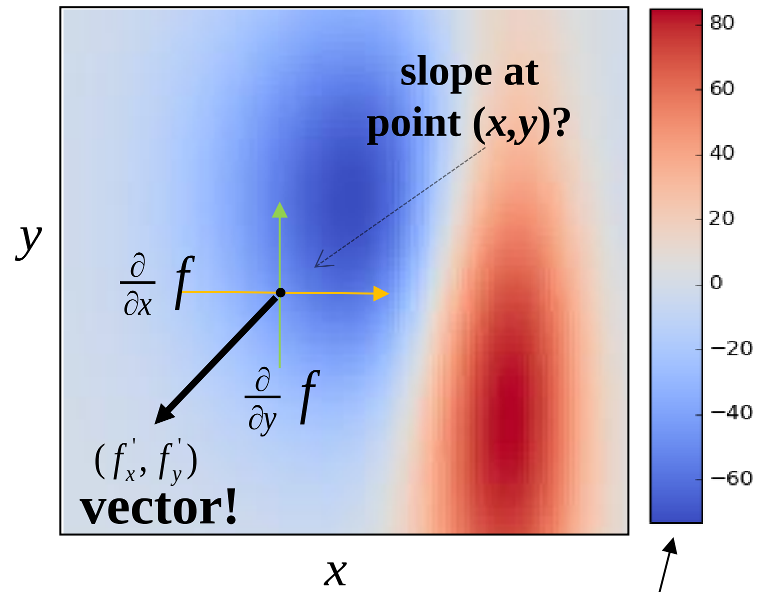
$$\frac{\partial}{\partial x} f = \lim_{\varepsilon \to 0}\left(\frac{f(x+\varepsilon, y) - f(x, y)}{\varepsilon}\right)$$

slope in *x*-direction

$$\frac{\partial}{\partial y} f = \lim_{\varepsilon \to 0}\left(\frac{f(x, y+\varepsilon) - f(x, y)}{\varepsilon}\right)$$

slope in *y*-direction

domain of *f(x,y)*



gradient ∇f

range of *f(x,y)*

Gradients for function *f* (*x*,*y*)

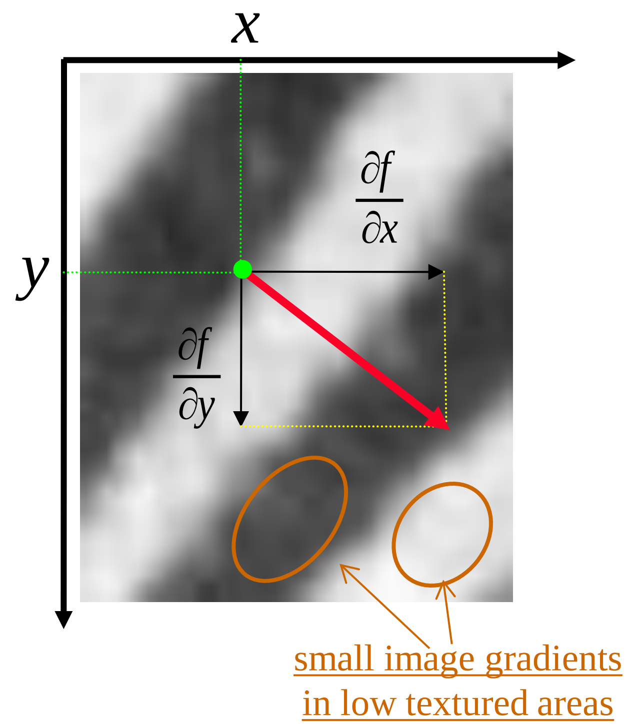
(a.k.a. *intensity gradients*, if *f* represents image intensities)

- For a function of two (or more) variables $f(x,y)$

Gradient at point (x,y)

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

two (or more) dimensional vector



- Gradient's absolute value $|\nabla f| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$ describes the slope's "steepness"
  - large at contrast edges, small in inform color regions

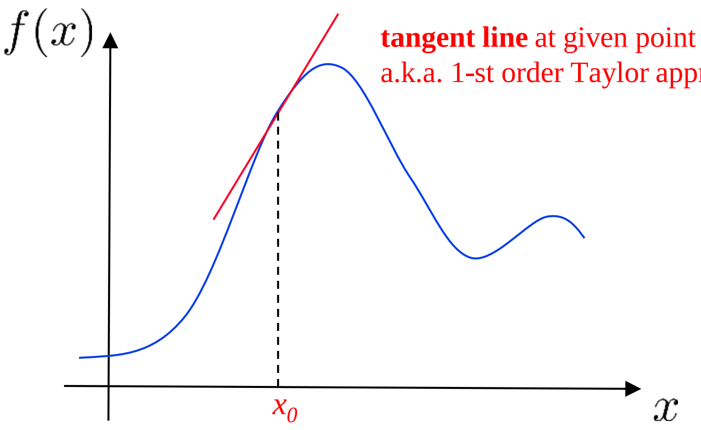
- Gradient's direction corresponds to the **steepest ascend** direction of the "slope"
  - gradient is orthogonal to image object boundaries

Gradients and linear approximations
(tangent hyperplanes)

Functions of a single variable

$$f : R^1 \to R^1$$

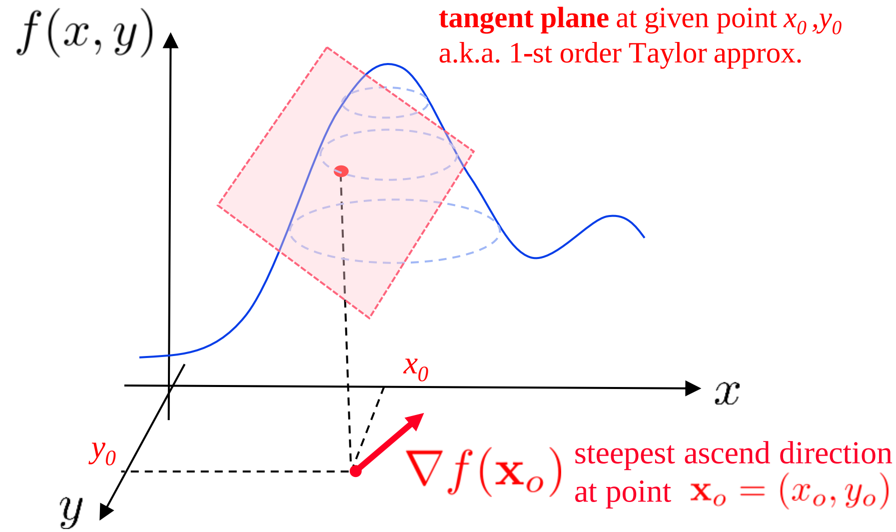
$$t(x) = f(x_0) + f'(x_0)\underbrace{(x - x_0)}_{dx}$$



Functions of two variables

$$f : R^2 \to R^1$$

$$t(x,y) = f(x_o,y_o) + \frac{\partial f}{\partial x}(x_o,y_o)\underbrace{(x - x_0)}_{dx} + \frac{\partial f}{\partial y}(x_o,y_o)\underbrace{(y - y_o)}_{dy}$$



$$t(\mathbf{x}) \equiv f(\mathbf{x}_o) + \nabla f(\mathbf{x}_o)^\top d\mathbf{x}$$

General formula for linear (1st order) Taylor approximation for function $f(\mathbf{x})$ of *n* variables $\mathbf{x} \in R^n$

Comment: gradient ∇f is independent of specific coordinate system for domain of f

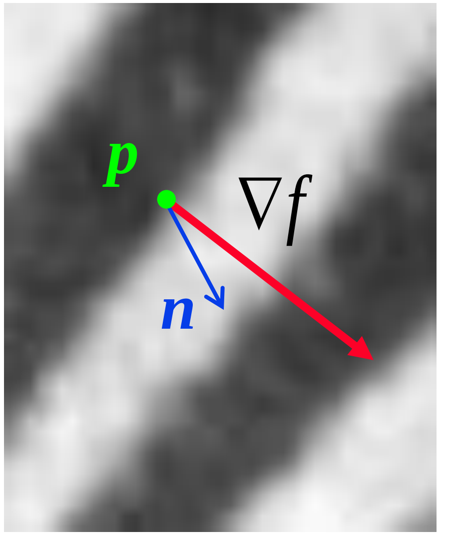
- In general, *gradient* of function $f(p)$ at point $p \in R^2$ can be defined as a vector ∇f s.t. for any unit vector $\vec{n}$

Gradient at point *p*

$$\nabla f^\top \bar{n} = \frac{\partial f}{\partial \bar{n}} \approx \frac{f(p+\varepsilon \cdot \vec{n}) - f(p)}{\varepsilon}$$

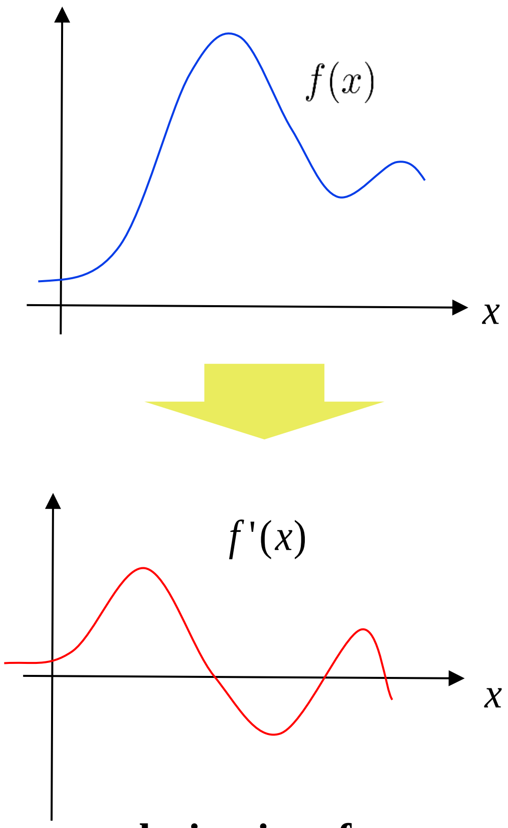
dot product

directional derivative of function ***f*** along direction ***n***



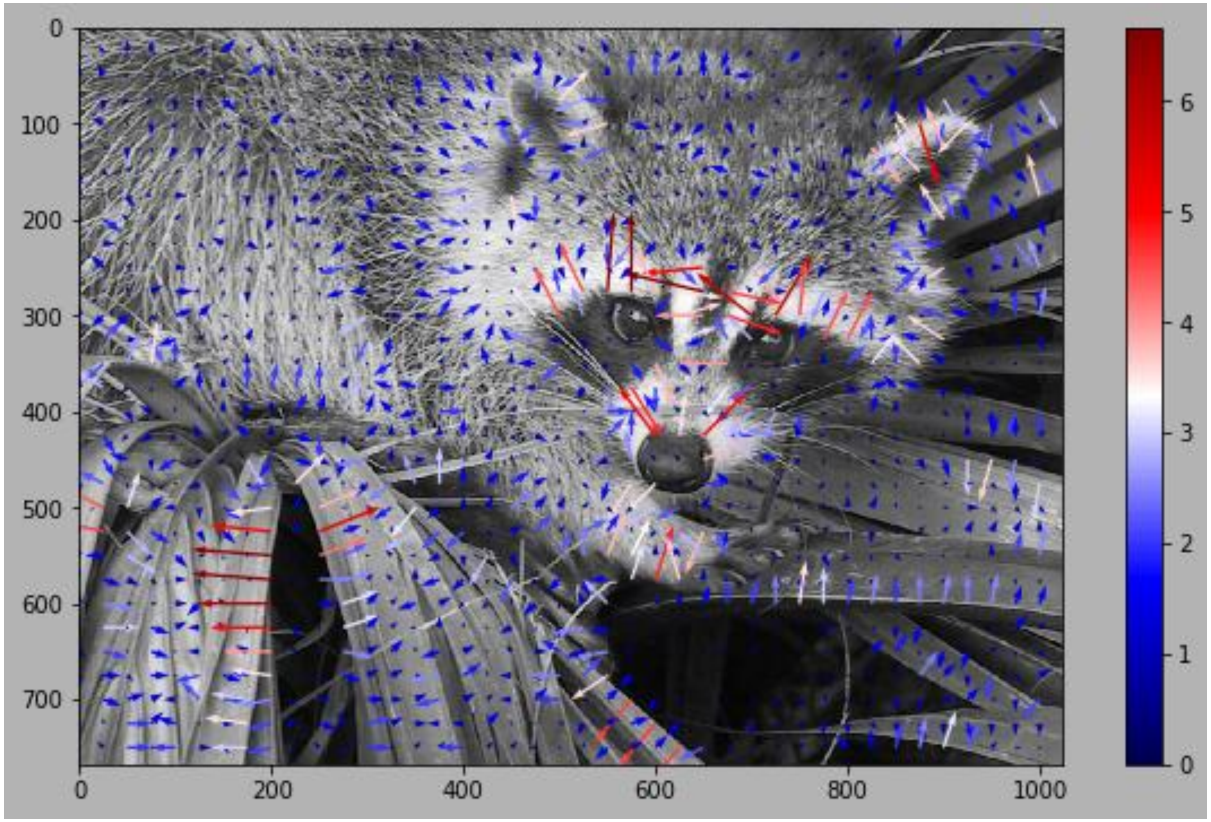
- pure vector algebra, specific coordinate system is irrelevant
- works for functions of two, three, or any larger number of variables
- partial derivatives $(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ represent gradient ∇f for any given orthonormal XY basis for 2D domain of function $f : \mathcal{R}^2 \rightarrow \mathcal{R}$

Image gradients as a "vector field"



derivatives for function of one variable $f(x)$

from Jupiter notebook "Convolution.ipynb"



gradients (derivatives) for a function of two variables, image $f(x,y)$

Computing gradients for closed-form functions
(iClicker moment)

$$f(x, y, z) = yx^2$$

A: $\nabla f = \begin{bmatrix} 2yx \\ x^2 \\ 0 \end{bmatrix}$

B: $\nabla f = \begin{bmatrix} 2yx \\ x^2 \end{bmatrix}$

C: none of the above

D: both A and B

UNIVERSITY OF WATERLOO

Estimating gradients for numerically defined functions, e.g. images

Differentiation and convolution

- Estimating partial derivatives for numerically-defined $f(x, y)$ e.g. images

$$\frac{\partial}{\partial x} f = \lim_{\varepsilon \to 0} \left(\frac{f(x+\varepsilon, y) - f(x, y)}{\varepsilon} \right)$$

$$\frac{\partial}{\partial y} f = \lim_{\varepsilon \to 0} \left(\frac{f(x, y+\varepsilon) - f(x, y)}{\varepsilon} \right)$$

- Both are linear and shift-invariant, so "must be" the result of a convolution.

Estimating gradients for numerically defined functions, e.g. images

Differentiation and convolution

partial derivative with respect to *x*

$$\frac{\partial}{\partial x} f = \lim_{\varepsilon \to 0}\left(\frac{f(x+\varepsilon, y) - f(x,y)}{\varepsilon} \right)$$

At given point (x_i, y_i)
one can approximate this as

$$\frac{\partial}{\partial x} f \approx \frac{f(x_{i+1}, y_i) - f(x_{i-1}, y_i)}{2 \cdot \Delta x}$$

$$= \nabla_x * f$$

convolution
with kernel

∇_x

$\frac{1}{2\Delta x} \cdot$

0	0	0
1	0	-1
0	0	0

Estimating gradients for numerically defined functions, e.g. images

Differentiation and convolution

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

At given point (x_i, y_i) one can approximate this as

$$\frac{\partial}{\partial x} f \approx \frac{f(x_{i+1}, y_i) - f(x_{i-1}, y_i)}{2 \cdot \Delta x}$$

$$= \nabla_x * f$$

convolution with kernel ∇_x

$\frac{1}{2\Delta x} \cdot$

0	0	0
1	0	-1
0	0	0

1/2*(90-0) = 45

Estimating gradients for numerically defined functions, e.g. images

Differentiation and convolution

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

At given point (x_i, y_i) one can approximate this as

$$\frac{\partial}{\partial x} f \approx \frac{f(x_{i+1}, y_i) - f(x_{i-1}, y_i)}{2 \cdot \Delta x}$$

$$= \nabla_x * f$$

convolution with kernel ∇_x

$$\frac{1}{2\Delta x} \cdot$$

0	0	0
1	0	-1
0	0	0

1/2*(0-90) = -45

Estimating gradients for numerically defined functions, e.g. images

Differentiation and convolution

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	60	90	90	90	60	0	0
0	0	0	40	60	60	60	40	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

At given point (x_i, y_i) one can approximate this as

$$\frac{\partial}{\partial x} f \approx \frac{f(x_{i+1}, y_i) - f(x_{i-1}, y_i)}{2 \cdot \Delta x}$$

$$= \nabla_x * f$$

convolution with kernel ∇_x

$\frac{1}{2\Delta x} \cdot$

0	0	0
1	0	-1
0	0	0

1/2*(60-60) = 0

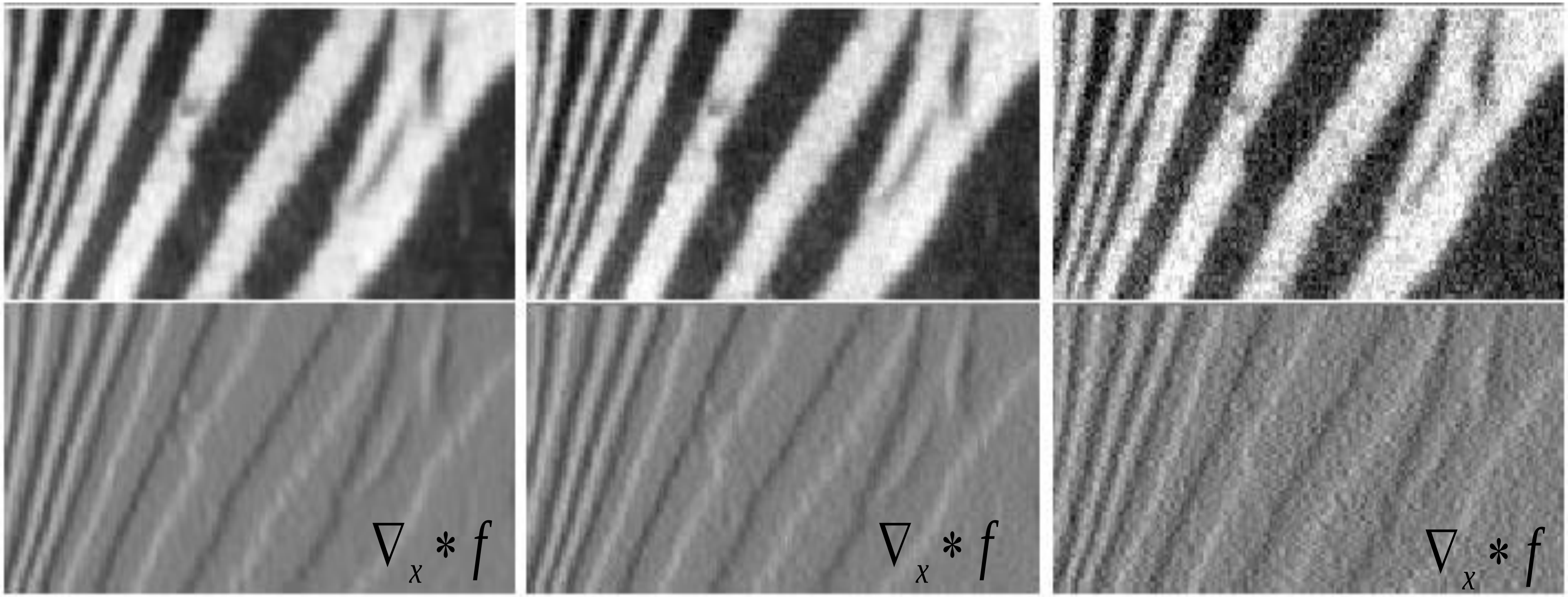
Finite differences

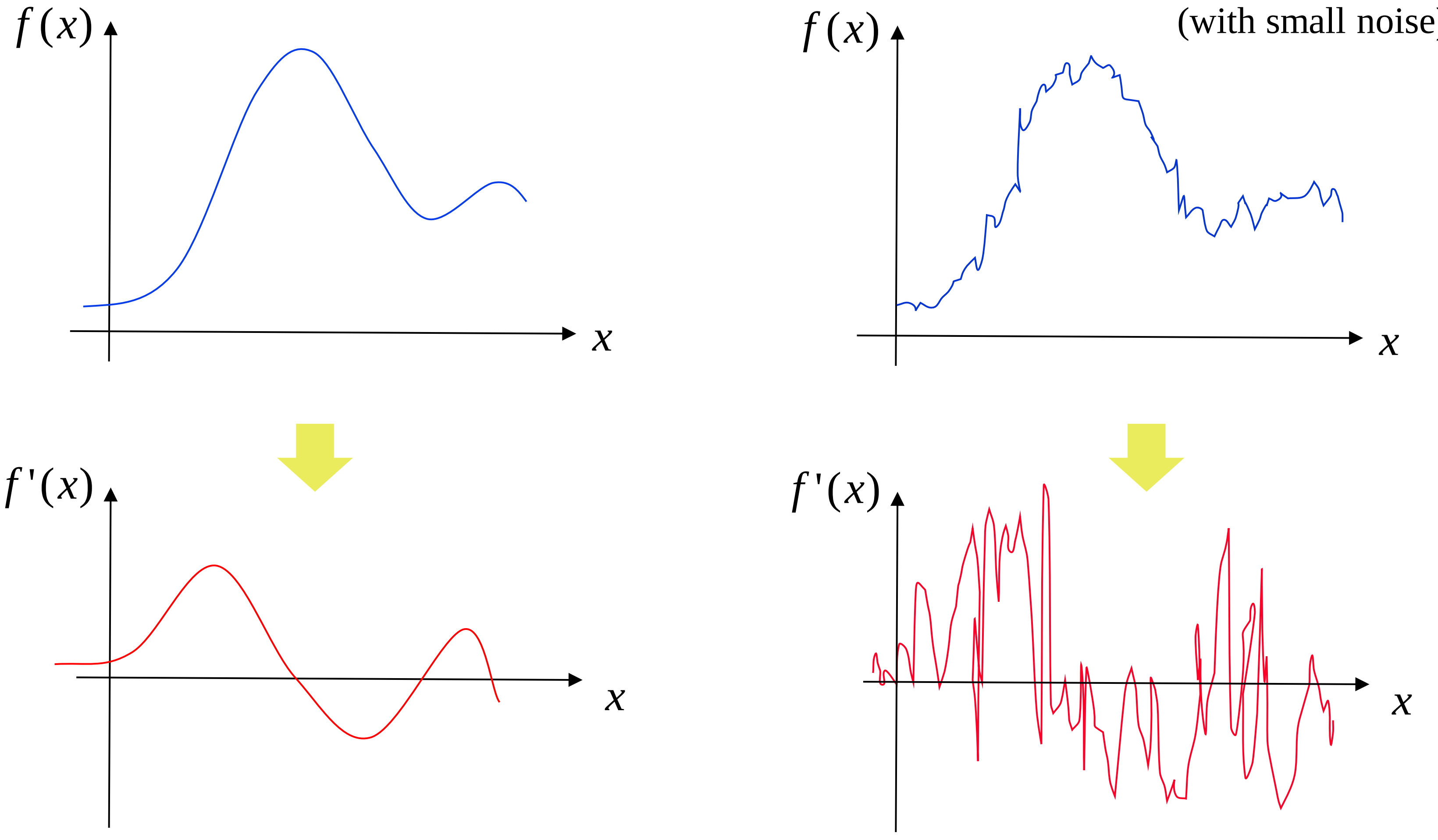
$$\nabla_x * f$$

Finite differences responding to noise

increasing noise ->

(this is zero mean additive Gaussian noise)





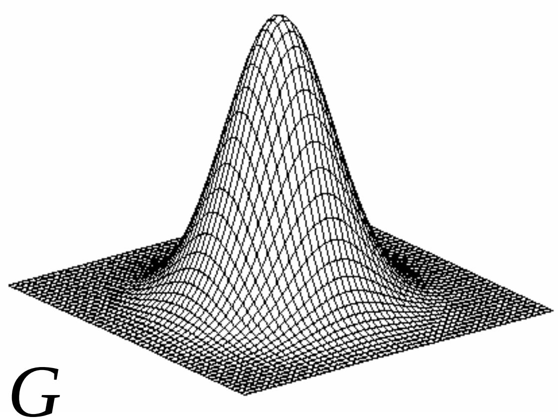
f (x)
x
f '(x)
x
f (x)
(with small noise)
x
f '(x)
x

Finite differences and noise

- Finite difference filters respond strongly to noise
  - obvious reason: image noise results in pixels that look very different from their neighbours
- Generally, the larger the noise the stronger the response
- What is to be done?
  - intuitively, most pixels in images look quite a lot like their neighbors
  - this is partially true even at edges: along the edge they are similar (but not across the edge)
  - suggests that smoothing the image should help, by forcing pixels different to their neighbors (=noise pixels?) to look more like neighbors

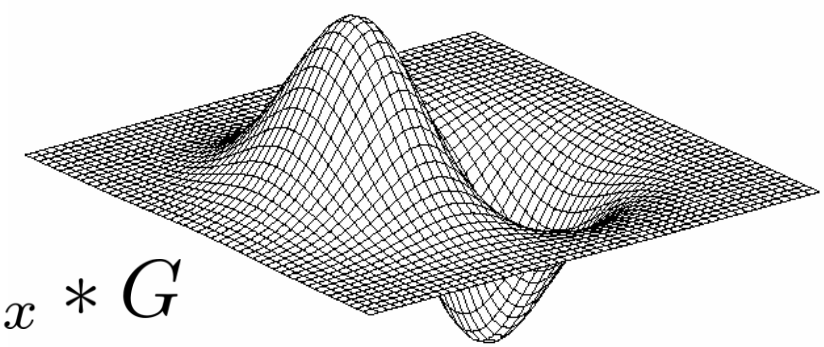
Smoothing and Differentiation

- Issue: noise
  - smooth before differentiation
  - two convolutions: $\nabla_x * (G * f)$
  - actually, we can use a derivative of Gaussian filter
    - differentiation is convolution, and convolution is associative $\nabla_x * (G * f) = (\nabla_x * G) * f$



G

(2D gaussian)



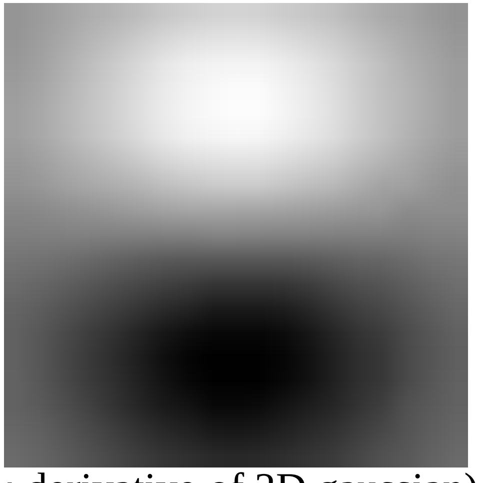
$\nabla_x * G$

(*x*-derivative of 2D gaussian)

Smoothing and Differentiation

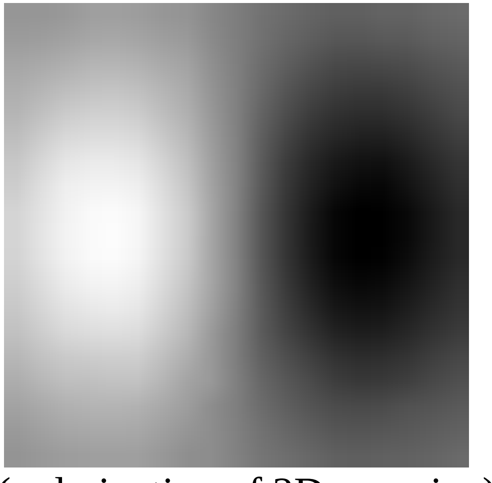
- Issue: noise
  - smooth before differentiation
  - two convolutions: $\nabla_x * (G * f)$
  - actually, we can use a derivative of Gaussian filter
    - differentiation is convolution, and convolution is associative $\nabla_x * (G * f) = (\nabla_x * G) * f$

$\nabla_y * G$



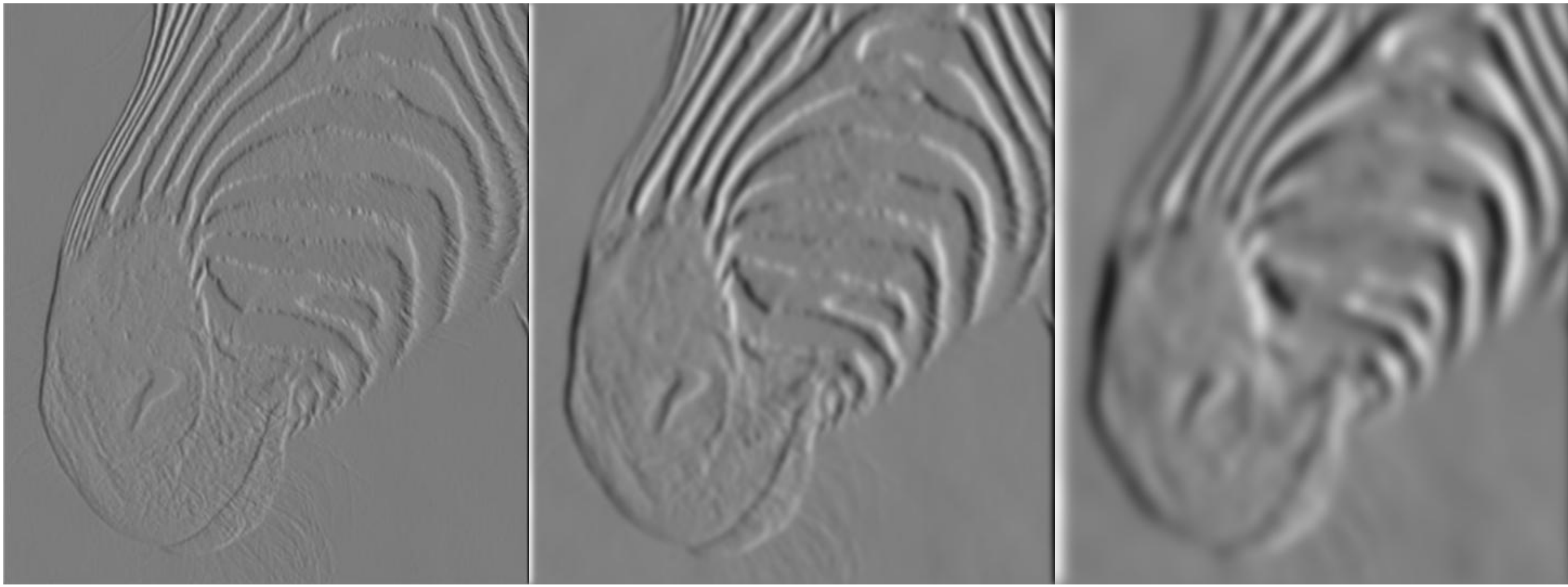
(*y*-derivative of 2D gaussian)

$\nabla_x * G$



(*x*-derivative of 2D gaussian)

$$(\nabla_x * G) * f$$



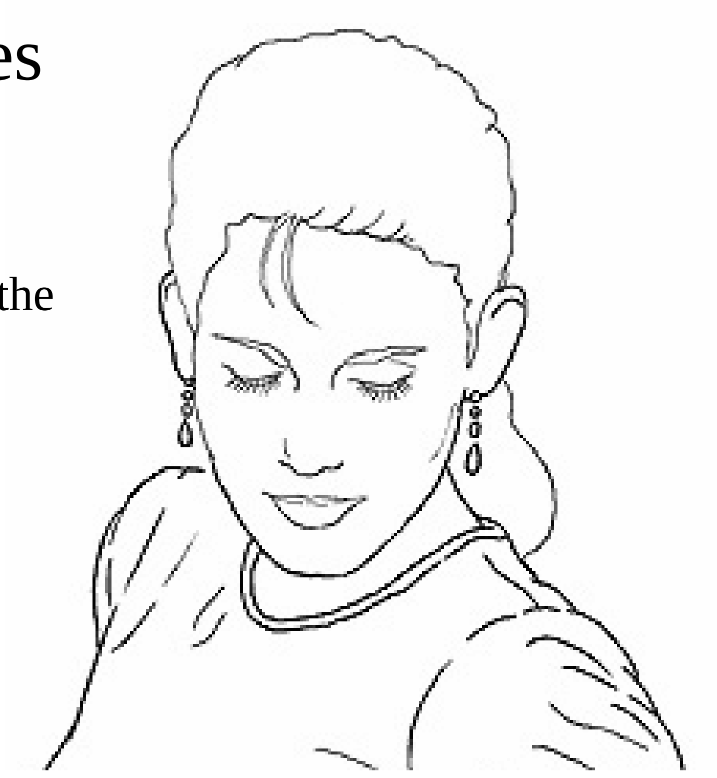
1 pixel 3 pixels 7 pixels

The scale of the smoothing filter (e.g. "bandwidth" σ of a Gaussian kernel) affects derivative estimates, and also the semantics of the edges recovered.

Image Gradients and Edges

Goal: Identify sudden (large) changes (discontinuities) in an image

– Intuitively, most semantic and shape information from the image can be encoded in the edges
– More compact than pixels



- **Ideal:** artist's line drawing

(note that artist is using object-level knowledge)

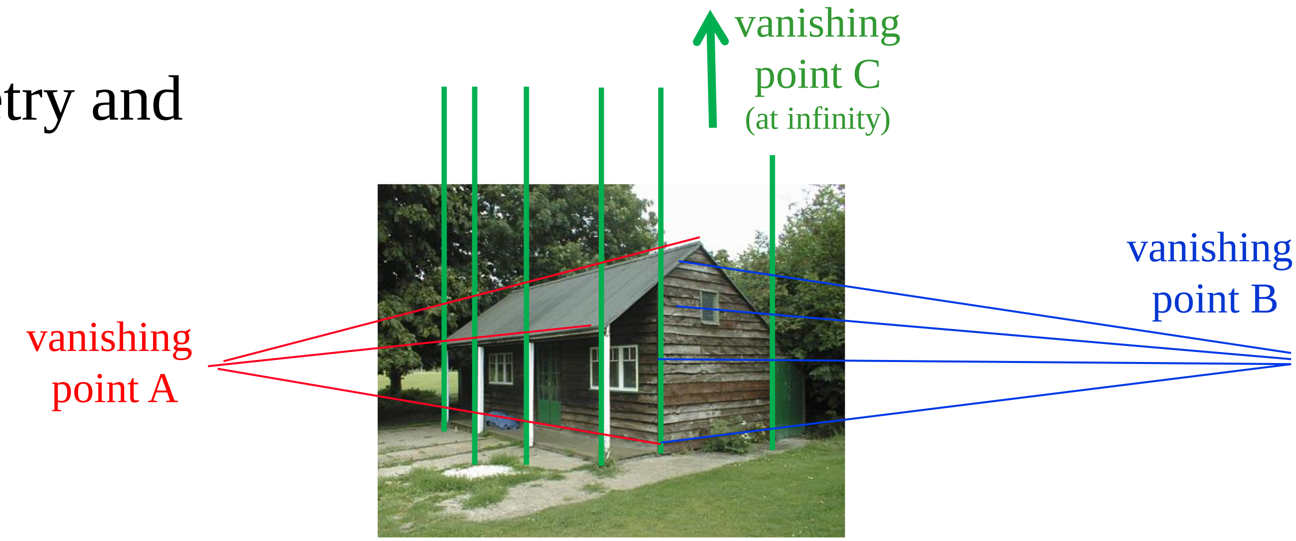
Source: D. Lowe

Image Gradients and Edges

Why do we care about edges?

- Extract information, recognize objects

- Recover geometry and viewpoint



Source: J. Hayes

Image Gradients and Edges

- Typical application where image gradients are used is *image edge* detection
  - find points with large image gradients

"Lena's image"

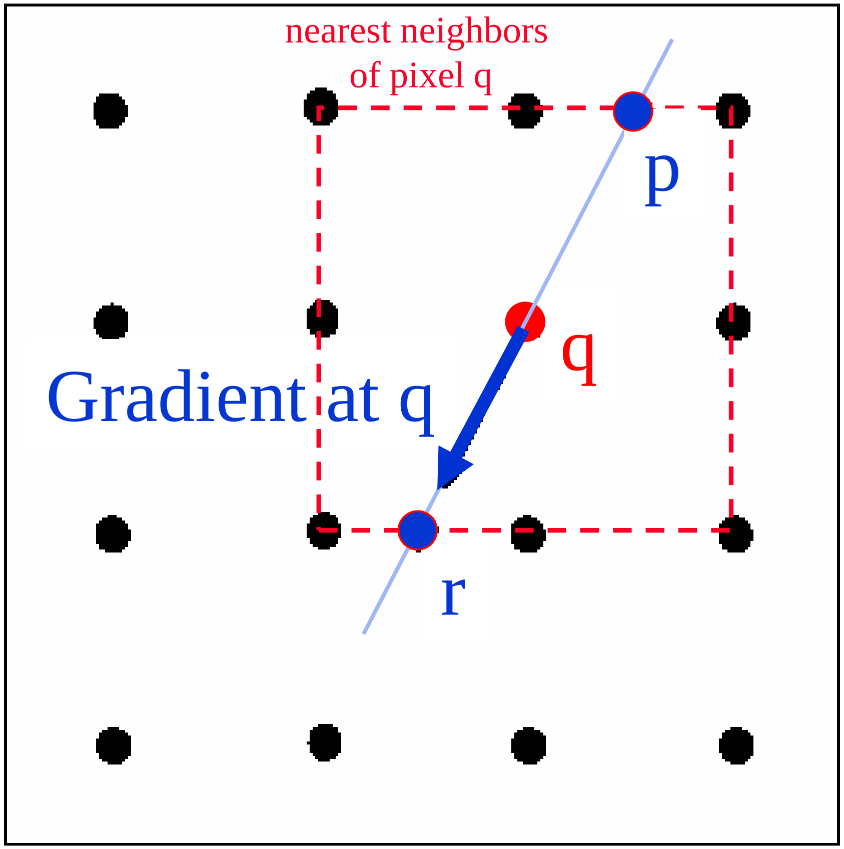
gradient magnitudes

$$||\nabla f||$$

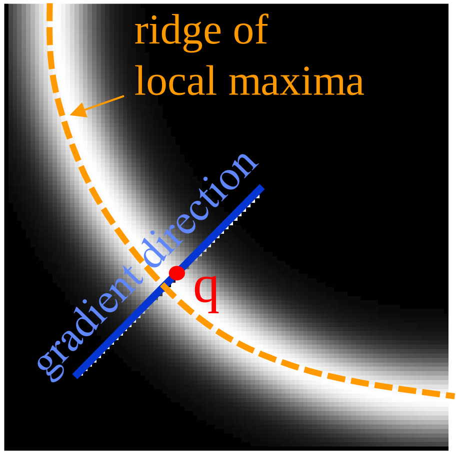
thresholded
gradient magnitudes

Image Gradients and Edges

Edge *thinning* via *non-maximum suppression*



At any given point q we have a local maximum if gradient magnitude $\|\nabla f\|$ at q is larger than those at both p and r
(may need to interpolate to estimate gradients at p,r)

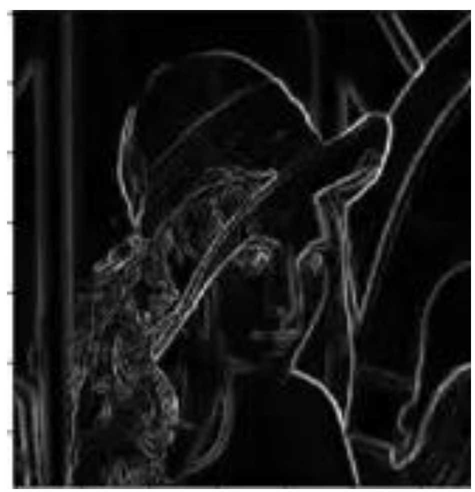


gradient magnitudes

Image Gradients and Edges

- Typical application where image gradients are used is *image edge detection*
  - find points with large image gradients

"Lena's image"



gradient magnitudes

$$\|\nabla f\|$$

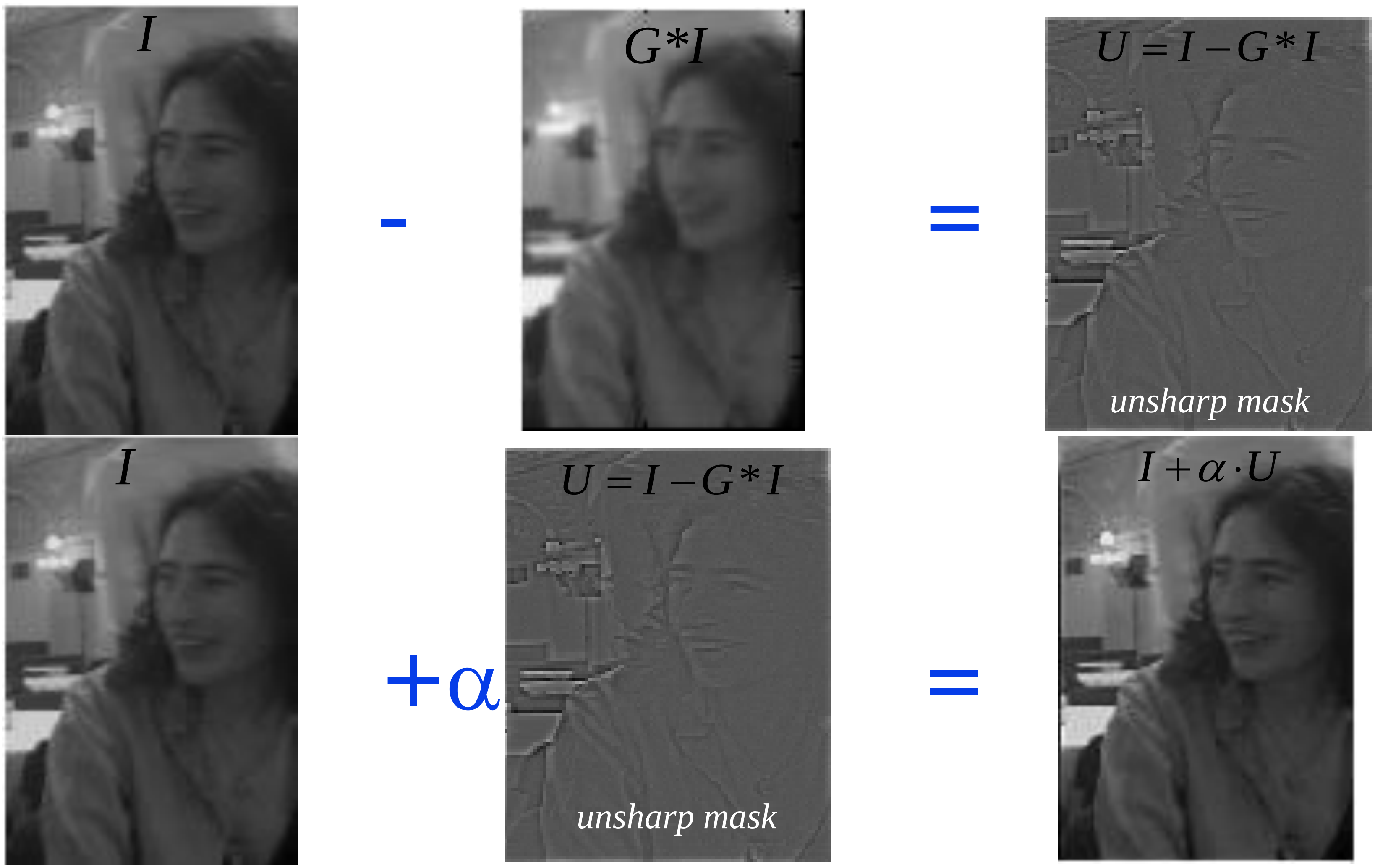
"edge features"

Canny edge detector

(non-maxima suppression + adaptive thresholding)

Unsharp masking

- What does blurring take away?

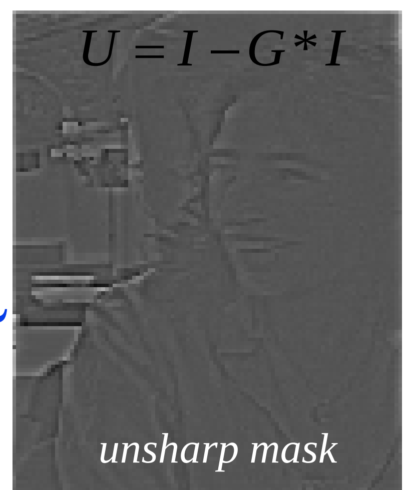


Unsharp masking

$$(1+\alpha)I - \alpha \cdot G * I \quad \approx \quad (1+\alpha)G_{\varepsilon} * I - \alpha \cdot G_{\sigma} * I$$

$$\varepsilon < 1$$

$+\alpha$



$=$

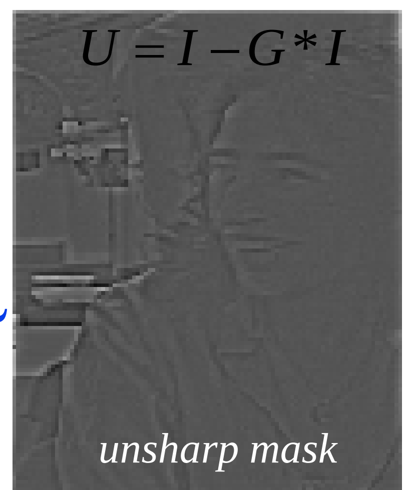
Unsharp masking

unsharp masking can be seen as
a convolution with
difference of Gaussians (DoG) kernel

$$(1+\alpha)I - \alpha \cdot G * I \quad \approx \quad [(1+\alpha)G_{\varepsilon} - \alpha \cdot G_{\sigma}] * I$$

$$\varepsilon < 1$$

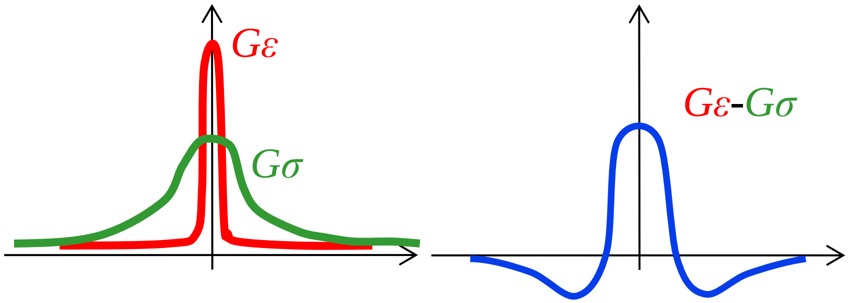
$+\alpha$



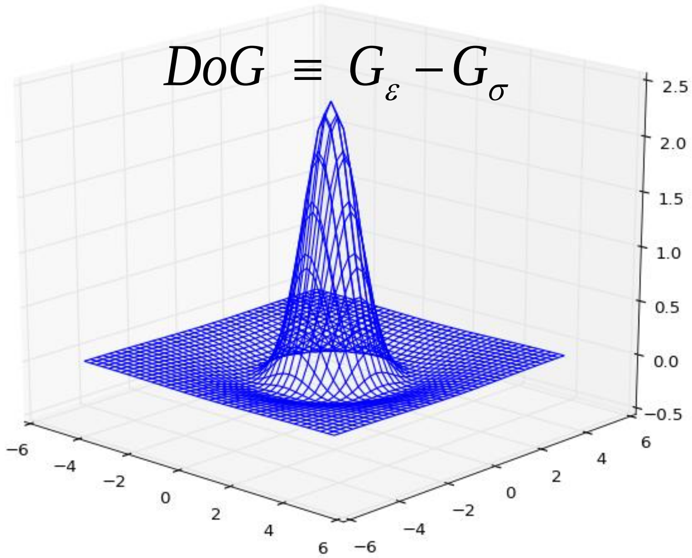
$=$

Unsharp masking

unsharp masking can be seen as
a convolution with
difference of Gaussians (DoG) kernel



$$DoG \equiv G_{\varepsilon} - G_{\sigma}$$

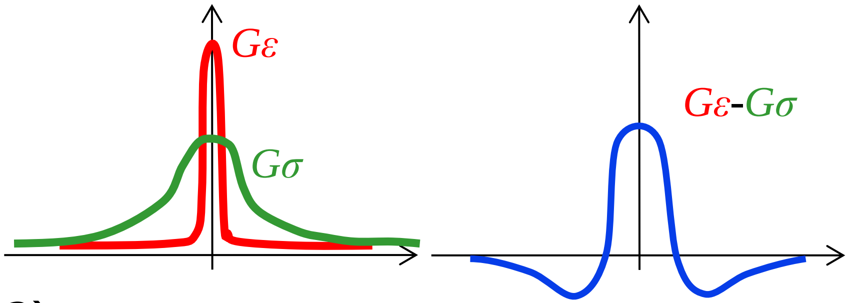


Unsharp masking

Python:

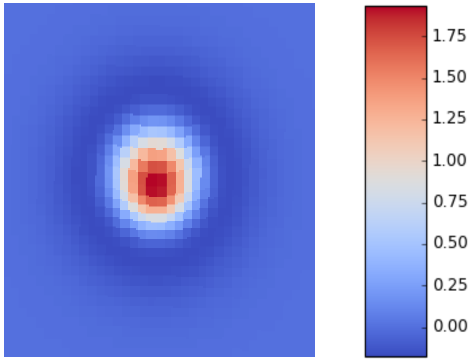
```
im=image.imread("xxxxx.jpg")
# assume "im" is gray scale

blurred = ndimage.gaussian_filter(im, sigma=3)
unsharp = im - 1.0*blurred
sharp = im + 10.0*unsharp
```

unsharp masking can be seen as a convolution with difference of Gaussians (DoG) kernel



$$DoG \equiv G_{\varepsilon} - G_{\sigma}$$

One can obtain the same effect using an explicit convolution with the DoG kernel



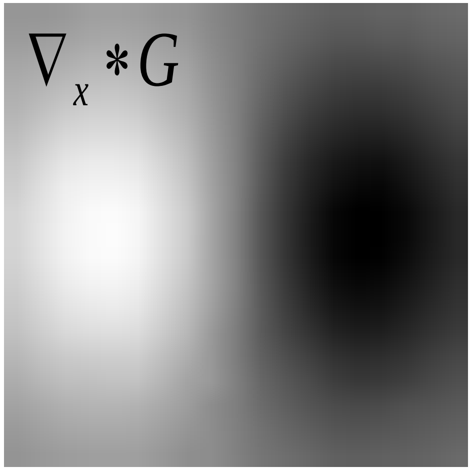
$$DoG \equiv G_{\varepsilon} - G_{\sigma}$$

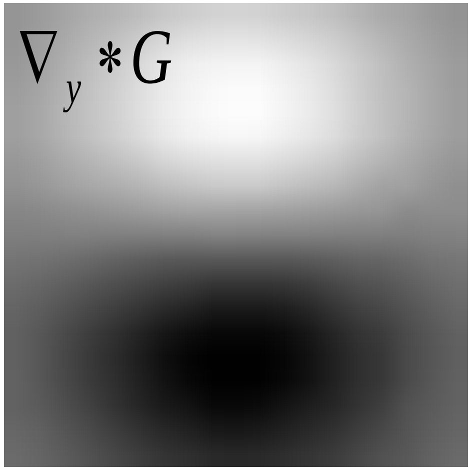
Reading: Forsyth & Ponce ch.7.5

Filters and Templates

- Applying a filter at some point can be seen as taking a dot-product between the image and some vector
- Filtering the image is a set of dot products

- Insight
  - filters may look like the effects they are intended to find
  - filters find effects they look like





Normalized Cross-Correlation (NCC)

- filtering as a **dot product**
- now **measure the angle:**

  i.e. divide filter output by the norms of kernel and image patch

cross-correlation of h and f around $t=(x,y)$

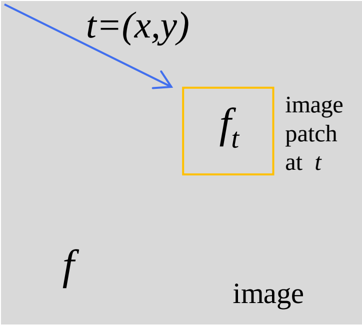
$$\frac{\sum_{u=-k}^{k}\sum_{v=-k}^{k} h[u,v]\cdot f[x+u,y+v]}{|h|\cdot|f_t|}$$

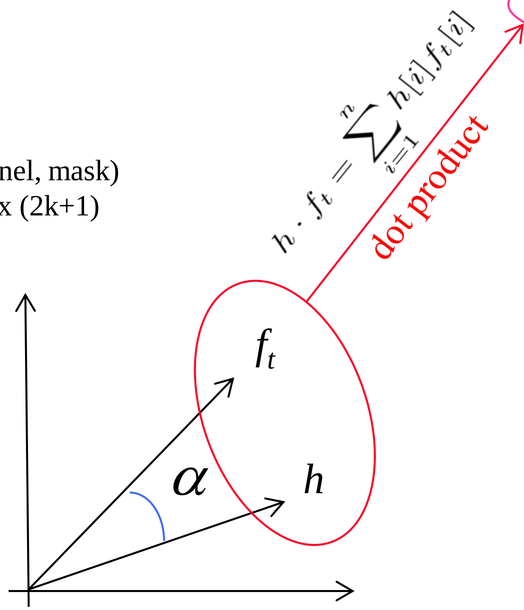
division makes this a non-linear operation

$$g[t] = \frac{h\cdot f_t}{|h|\cdot|f_t|} = \cos(\alpha)$$

vector lengths $|z| = \sqrt{\sum_{i=1}^{n} z_i^2}$

h template (filter, kernel, mask) of size $n = (2k+1) \times (2k+1)$





raster-scan h and f_t as vectors in R^n

Normalized Cross-Correlation (NCC)

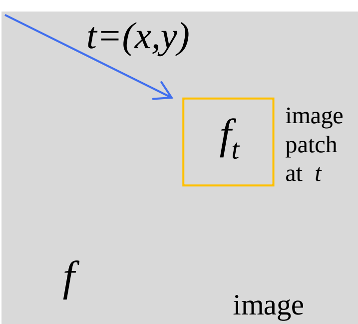
- filtering as a **dot product**
- now **measure the angle:**

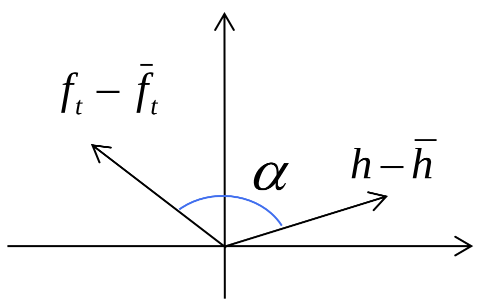
  i.e. divide filter output by the norms of kernel and image patch

- Tricks:
  - subtract *template average* $\bar{h}$
  - subtract *patch average* $\bar{f}_t$ (subtract the image mean in the neighborhood of *t*)

- gives zero output for constant regions, reducing response to irrelevant background
- invariance to (additive) intensity bias

h template (filter, kernel, mask) of size n = (2k+1) x (2k+1)





such vectors do not have to be in the "positive" quadrant

NCC

$$g[t] = \frac{(h-\bar{h})\cdot(f_t - \bar{f}_t)}{|h-\bar{h}|\cdot|f_t - \bar{f}_t|}$$

Normalized Cross-Correlation (NCC)

- filtering as a **dot product**
- now **measure the angle:**

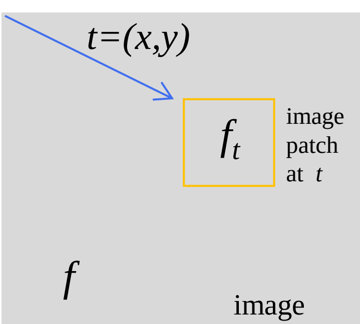
  i.e. divide filter output by the norms of kernel and image patch

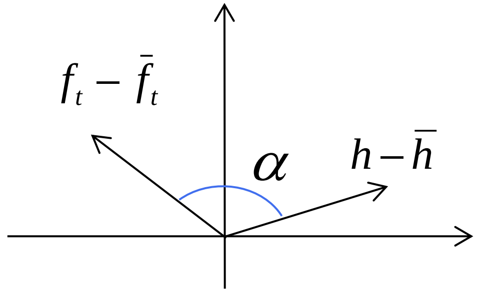
Tricks:

- subtract *template average* $\bar{h}$
- subtract *patch average* $\bar{f}_t$ (subtract the image mean in the neighborhood of *t*)

- gives zero output for constant regions, reducing response to irrelevant background
- invariance to (additive) intensity bias

h template (filter, kernel, mask) of size n = (2k+1) x (2k+1)





such vectors do not have to be in the "positive" quadrant

equivalently using statistical term σ (standard diviation)

$$g[t] = \frac{(h-\bar{h})\cdot(f_t-\bar{f}_t)}{n\cdot\sigma_h\cdot\sigma_{f_t}} = cov(h,f_t)$$

NCC

Remember: st.div. $\sigma_z \equiv \sqrt{\frac{1}{n}\sum_{i=1}^{n}(z_i-\bar{z})^2} = \sqrt{\frac{1}{n}}\cdot|z-\bar{z}|$

Normalized Cross-Correlation (NCC)

- filtering as a **dot product**
- now **measure the angle:**

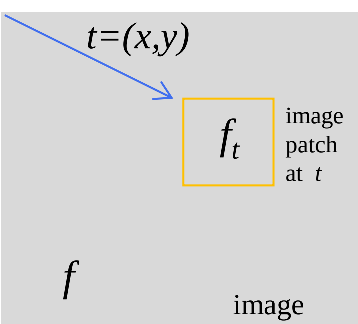
  i.e. divide filter output by the norms of kernel and image patch

□ Tricks:

- subtract *template average* $\bar{h}$
- subtract *patch average* $\bar{f}_t$ (subtract the image mean in the neighborhood of *t*)

- gives zero output for constant regions, reducing response to irrelevant background
- invariance to (additive) intensity bias

h template (filter, kernel, mask) of size n = (2k+1) x (2k+1)



standard in statistics *correlation coefficient*

ρ ⟷ between h and f_t

equivalently using statistical term cov (covariance)

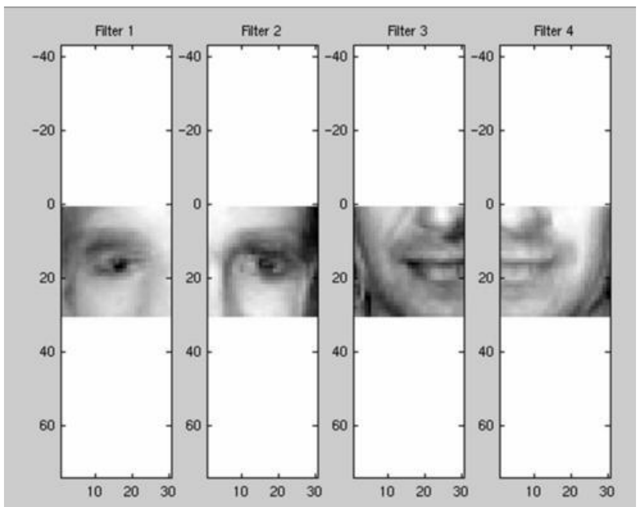
$$g[t] = \frac{\mathrm{cov}(h, f_t)}{\sigma_h \cdot \sigma_{f_t}}$$

NCC

$$\mathrm{cov}(a,b) \equiv E(a-\bar{a})(b-\bar{b}) = \frac{1}{n}\sum_{i=1}^{n}(a_i-\bar{a})(b_i-\bar{b}) = \frac{(a-\bar{a})\cdot(b-\bar{b})}{n}$$

Normalized Cross-Correlation (NCC)

h_A h_B h_C h_D



templates

NCC for h and f

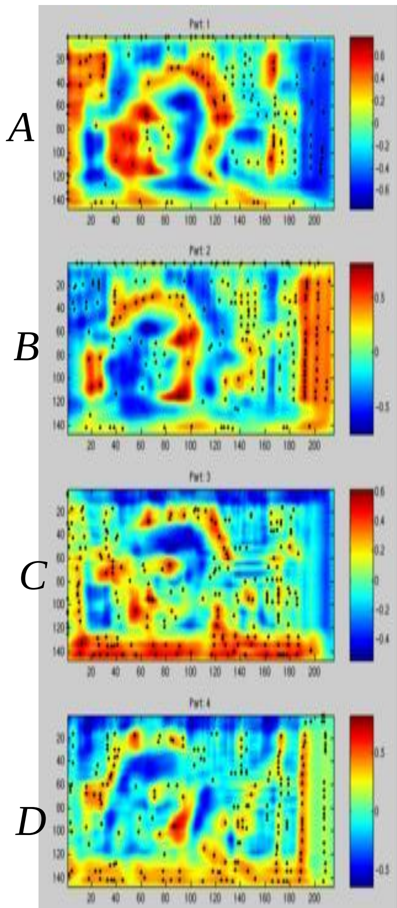
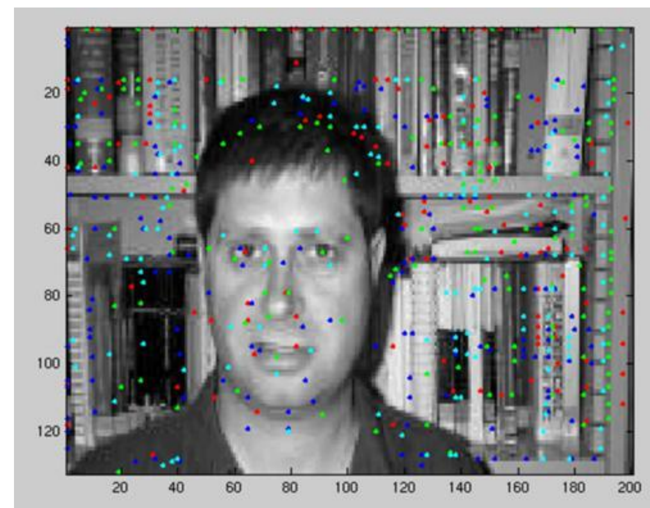


image f

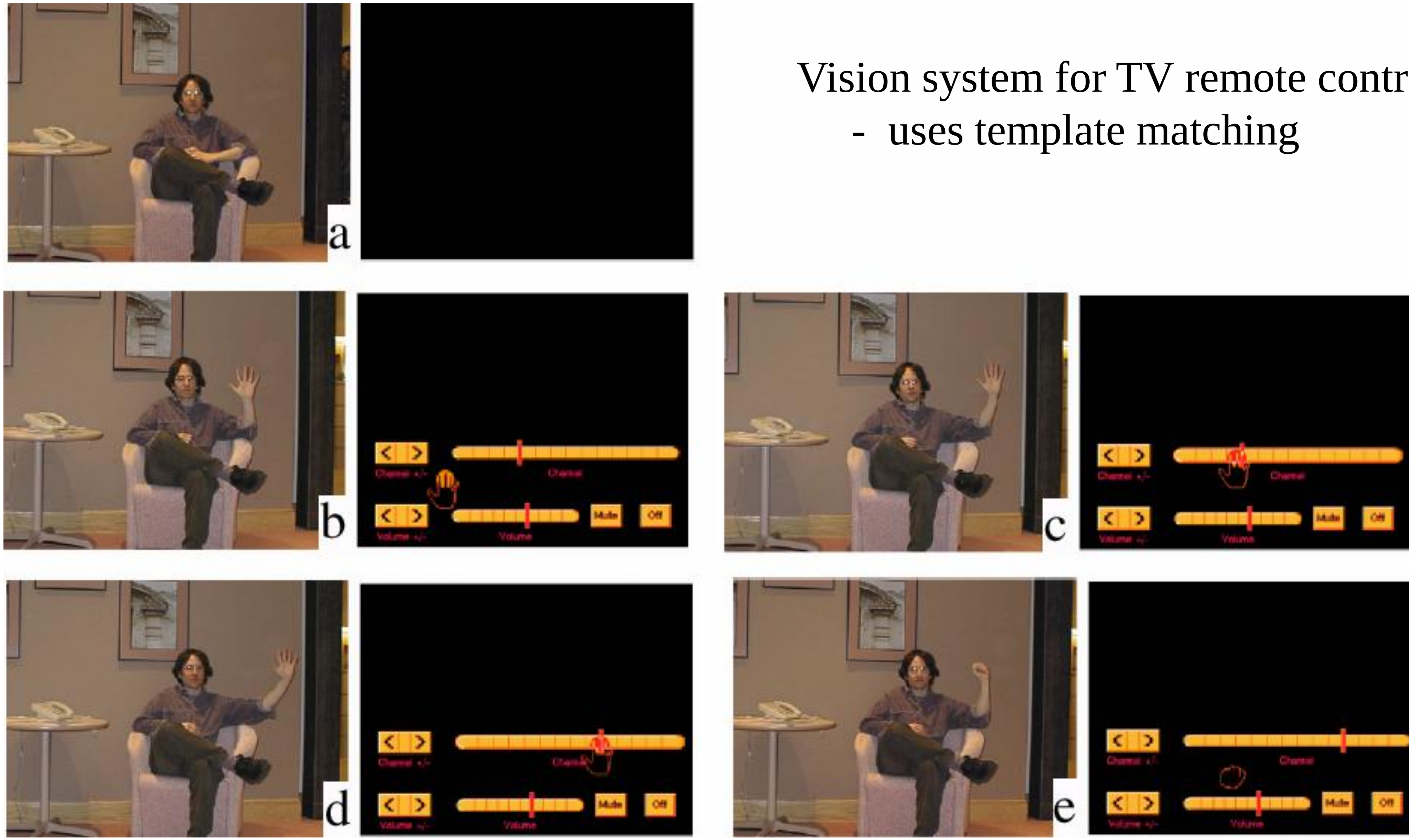


points mark local maxima of NCC
for each template

points of interest or **feature points**

(detected via non-maxima suppression of NCCs)

Normalized Cross-Correlation (NCC)



Vision system for TV remote control

- uses template matching

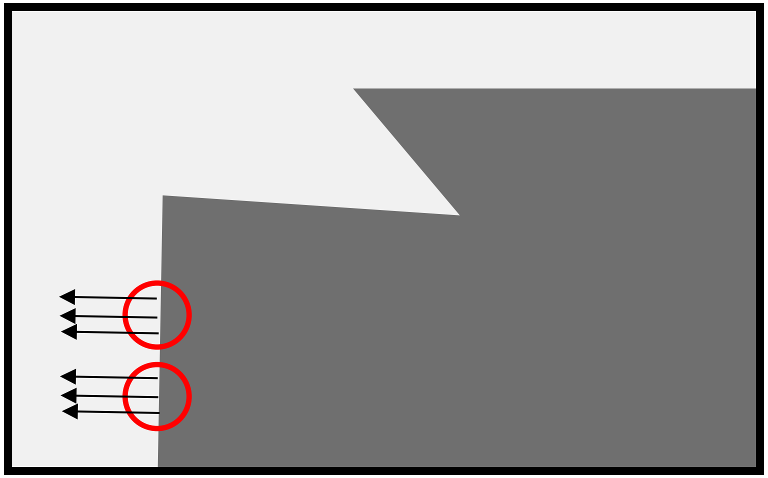
Figure from “Computer Vision for Interactive Computer Graphics,” W.Freeman et al, IEEE Computer Graphics and Applications, 1998 copyright 1998, IEEE

Other feature points…
(Szeliski sec 4.1.1)

Many applications require
generic "discriminant" feature points with
identifiable appearance and location
(so that they can be matched across multiple images)

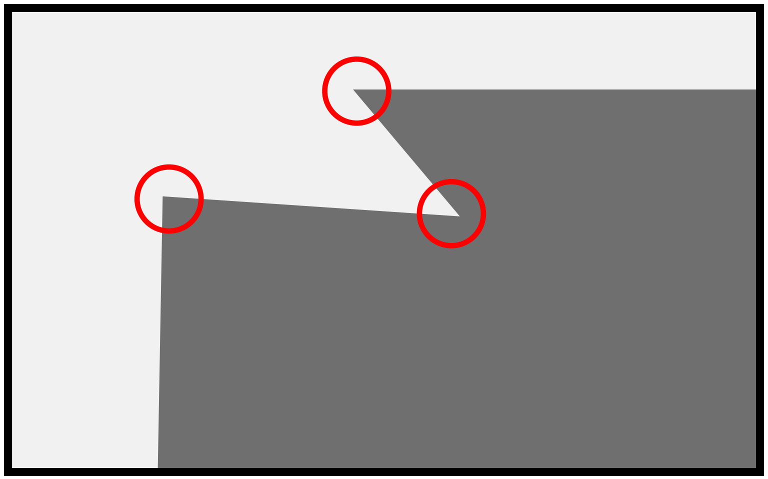
- Image alignment/registration
- 3D reconstruction
- Motion tracking
- Object recognition
- Indexing and database retrieval
- Robot navigation
- … other

How discriminant are "intensity edges"

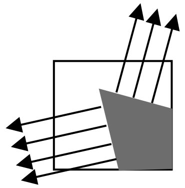


Patches at different near-by locations along the edge
look identical and cannot be discriminated (uniquely identified and localized).

Harris corners



Intuition:
find patches where
strong gradients
vary in orientation



corner features are extracted by analyzing image **"auto-correlation"** matrices $\begin{bmatrix} (f'_x)^2 & f'_x f'_y \\ f'_x f'_y & (f'_y)^2 \end{bmatrix} = \nabla f \cdot \nabla f^T$

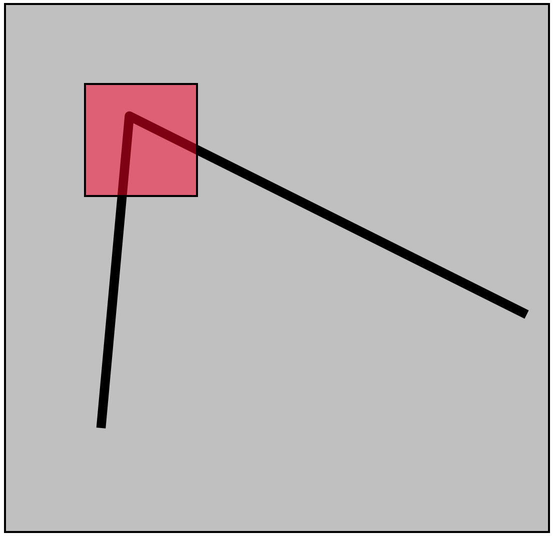
(see next slides, also Selizski – Sec 4.1.1)

in contrast, basic **edge features** use only magnitude of image **gradients** $\|\nabla f\|^2 = (f'_x)^2 + (f'_y)^2 = \nabla f^T \cdot \nabla f$

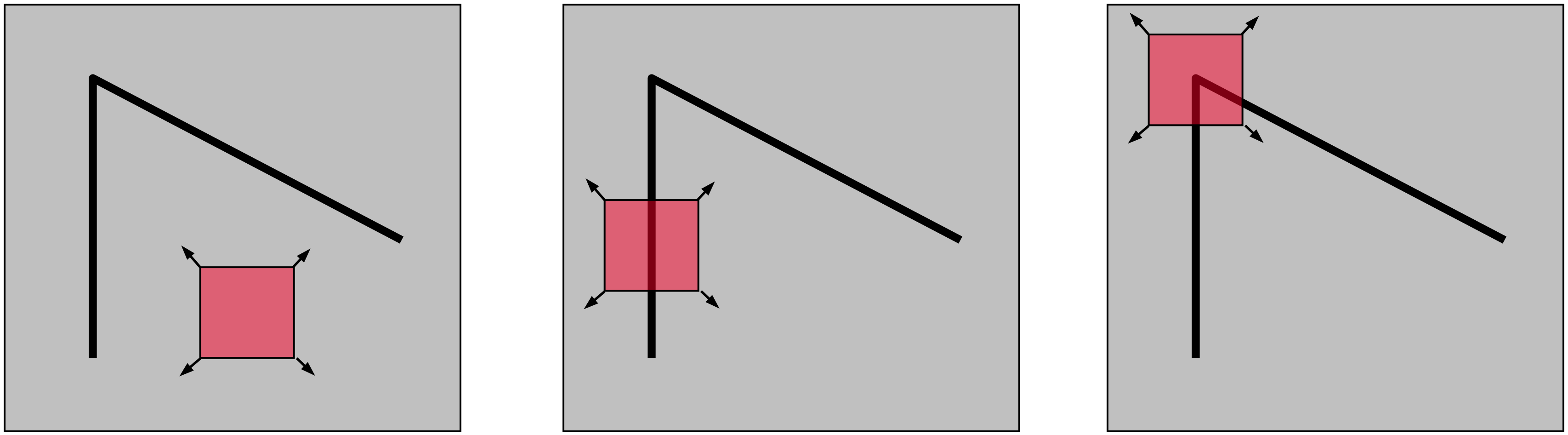
no orientation information

The Basic Idea

- We should easily recognize the point by looking through a small window
- Shifting a window in *any direction* should give *a large change* in observed intensities



Harris Detector: Basic Idea



"flat" region:
no change in all directions

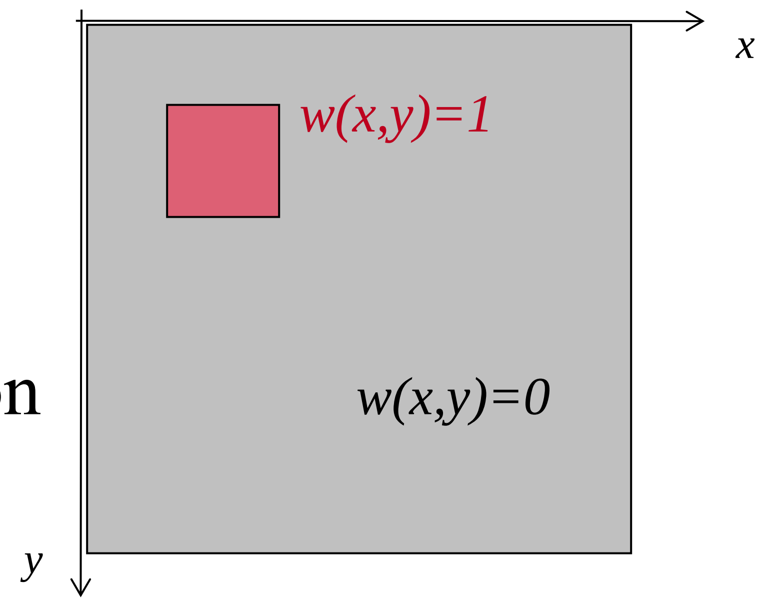
"edge":
change across the edge direction
no change along the edge direction

"corner":
significant change in all directions

Harris Detector: Mathematics

For any given image patch or window w we should measure how it changes when shifted by $ds = \begin{bmatrix} u \\ v \end{bmatrix}$

Notation: a patch can be defined by its indicator or "support" function $w(x,y)$ over image pixels



Harris Detector: Mathematics

patch w change measure for shift $ds = \begin{bmatrix} u \\ v \end{bmatrix}$:

weighted sum of squared differences

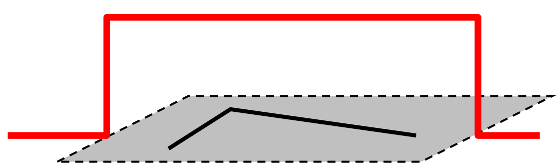
$$E_w(u,v) := \sum_{x,y} w(x,y) \cdot [I(x+u, y+v) - I(x,y)]^2$$

Window function

Shifted intensity

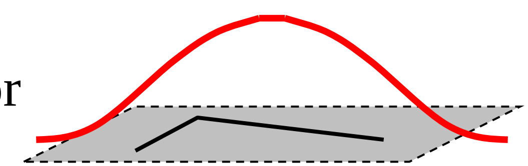
Intensity

NOTE:
window support functions $w(x,y) =$



1 in window, 0 outside

or



Gaussian (weighted) support

Harris Detector: Mathematics

Change of intensity for the shift $ds = \begin{bmatrix} u \\ v \end{bmatrix}$ assuming image gradient $\nabla I \equiv \begin{bmatrix} I_x \\ I_y \end{bmatrix}$

$$I(x+u, y+v) - I(x,y) \approx I_x \cdot u + I_y \cdot v = ds^T \cdot \nabla I$$

difference/change in I at (x,y) for shift (u,v) = ds

(remember gradient definition on earlier slides!!!!)

this is 1st order Taylor expansion (see slide 55)

$$[\, I(x+u, y+v) - I(x,y)\,]^2 \approx ds^T \cdot \nabla I \cdot \nabla I^T \cdot ds$$

$$E_w(u,v) = \sum_{x,y} w(x,y) \cdot [I(x+u, y+v) - I(x,y)]^2$$

$$\approx ds^T \cdot \left(\sum_{x,y} w(x,y) \cdot \nabla I \cdot \nabla I^T \right) \cdot ds = ds^T \cdot M_w \cdot ds$$

M_w

Harris Detector: Mathematics

Change of intensity for the shift $ds = \begin{bmatrix} u \\ v \end{bmatrix}$ assuming image gradient $\nabla I \equiv \begin{bmatrix} I_x \\ I_y \end{bmatrix}$

$$E_w(u,v) \cong [u \ \ v] \cdot M_w \cdot \begin{bmatrix} u \\ v \end{bmatrix} = ds^T \cdot M_w \cdot ds = E_w(ds)$$

where M_w is a 2×2 matrix computed from image derivatives inside patch w

matrix M is also called
Harris matrix or ***structure tensor***

$$\ldots \cdot \left(\sum_{x,y} w(x,y) \cdot \underbrace{\nabla I \cdot \nabla I^T}_{\begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}} \right) \cdot \ldots$$

← M_w

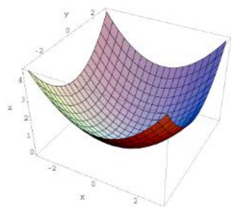
This tells you how to compute M_w at any window w (t.e. any image patch)

Harris Detector: Mathematics

Change of intensity for the shift $ds = \begin{bmatrix} u \\ v \end{bmatrix}$ assuming image gradient $\nabla I \equiv \begin{bmatrix} I_x \\ I_y \end{bmatrix}$

$$E_w(u,v) \cong [u \;\; v] \cdot M_w \cdot \begin{bmatrix} u \\ v \end{bmatrix} = ds^T \cdot M_w \cdot ds$$

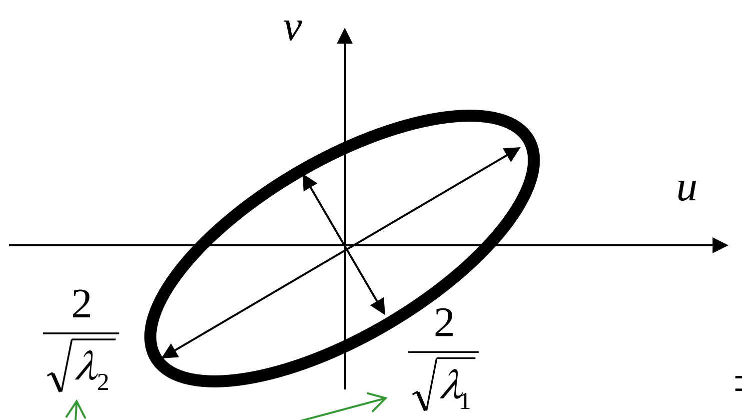
paraboloid



M is a *positive semi-definite* (p.s.d.) matrix (**Exercise**: show that $ds^T \cdot M \cdot ds \geq 0$ for any ds)

M can be analyzed via ***isolines***, *e.g.* $ds^T \cdot M_w \cdot ds = 1$ (**ellipsoid**)

see next slide



two eigen values of matrix M_w

Points on this ellipsoid are shifts $ds=[u,v]^T$ that have the same value of function $E(u,v)=1$.

$\Rightarrow$ This isoline visually illustrates how function E depends on shifts $ds=[u,v]^T$ in different directions.

iClicker moment

Quadratic forms and their matrix-based expressions

$$x^2 + 2xy + 8y^2 \equiv \mathbf{p}^\top M\, \mathbf{p} \qquad \text{where} \quad \mathbf{p} := \begin{bmatrix} x \\ y \end{bmatrix}$$

(all terms are of order 2)

$$M = \mathbf{?}$$

A: $\begin{bmatrix} 1 & 0 \\ 8 & 2 \end{bmatrix}$

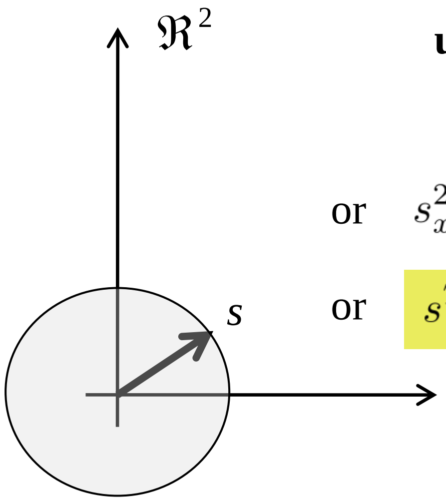
C: $\begin{bmatrix} 1 & 0 \\ 2 & 8 \end{bmatrix}$

B: $\begin{bmatrix} 1 & 2 \\ 8 & 1 \end{bmatrix}$

D: $\begin{bmatrix} 1 & 1 \\ 1 & 8 \end{bmatrix}$

technical note from linear algebra:

Ellipsoids

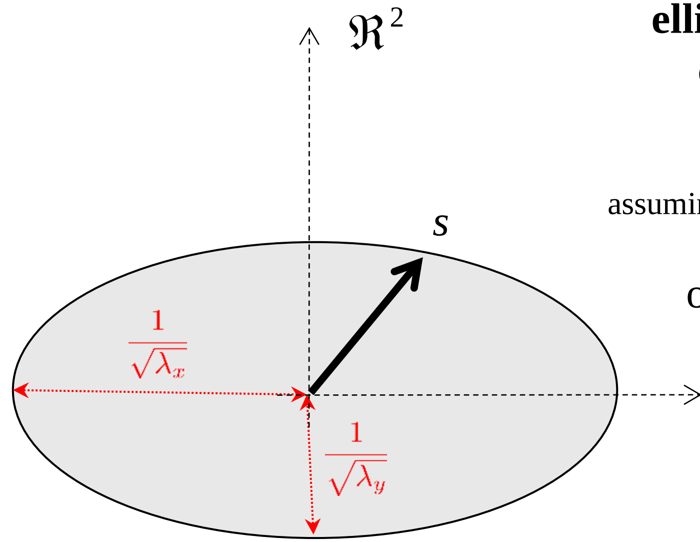


unit circle

$$\|s\| = 1$$

or $s_x^2 + s_y^2 = 1$

or $s^T \cdot s = 1$



ellipsoid aligned with coordinate axes

$$\lambda_x s_x^2 + \lambda_y s_y^2 = 1$$

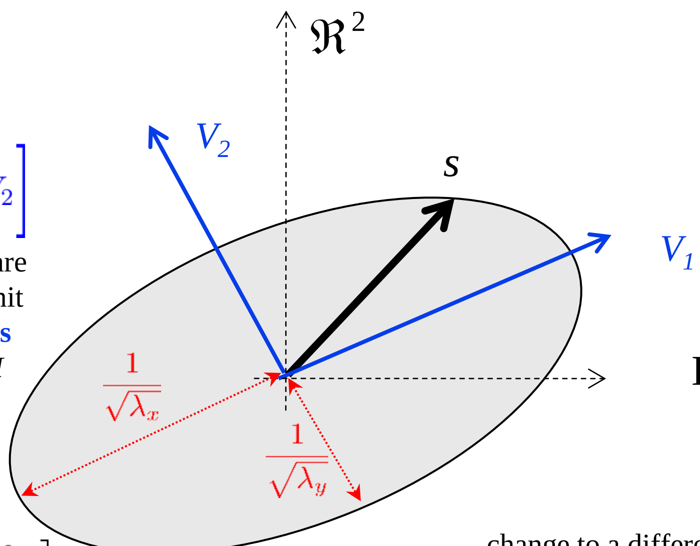
assuming non-negative λ_x and λ_y

or $s^T \cdot \Lambda \cdot s = 1$

for $\Lambda = \begin{bmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{bmatrix}$

$V = \left[V_1 \middle| V_2 \right]$
2x2

two columns are orthogonal unit **eigenvectors** of matrix M



eigenvalues of matrix M

$\Lambda = \begin{bmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{bmatrix}$

ellipsoid with general principal axes

$s^T \cdot M \cdot s = 1$ for any p.s.d. matrix M

Explanation: $M = V \Lambda V^T$ (*eigendecomposition* for any p.s.d. M)

$$s^T \cdot M \cdot s = 1 \quad \Rightarrow \quad s^T \cdot V \cdot \Lambda \cdot V^T \cdot s = 1$$

$$\Rightarrow \quad (V^T s)^T \cdot \Lambda \cdot (V^T s) = 1$$

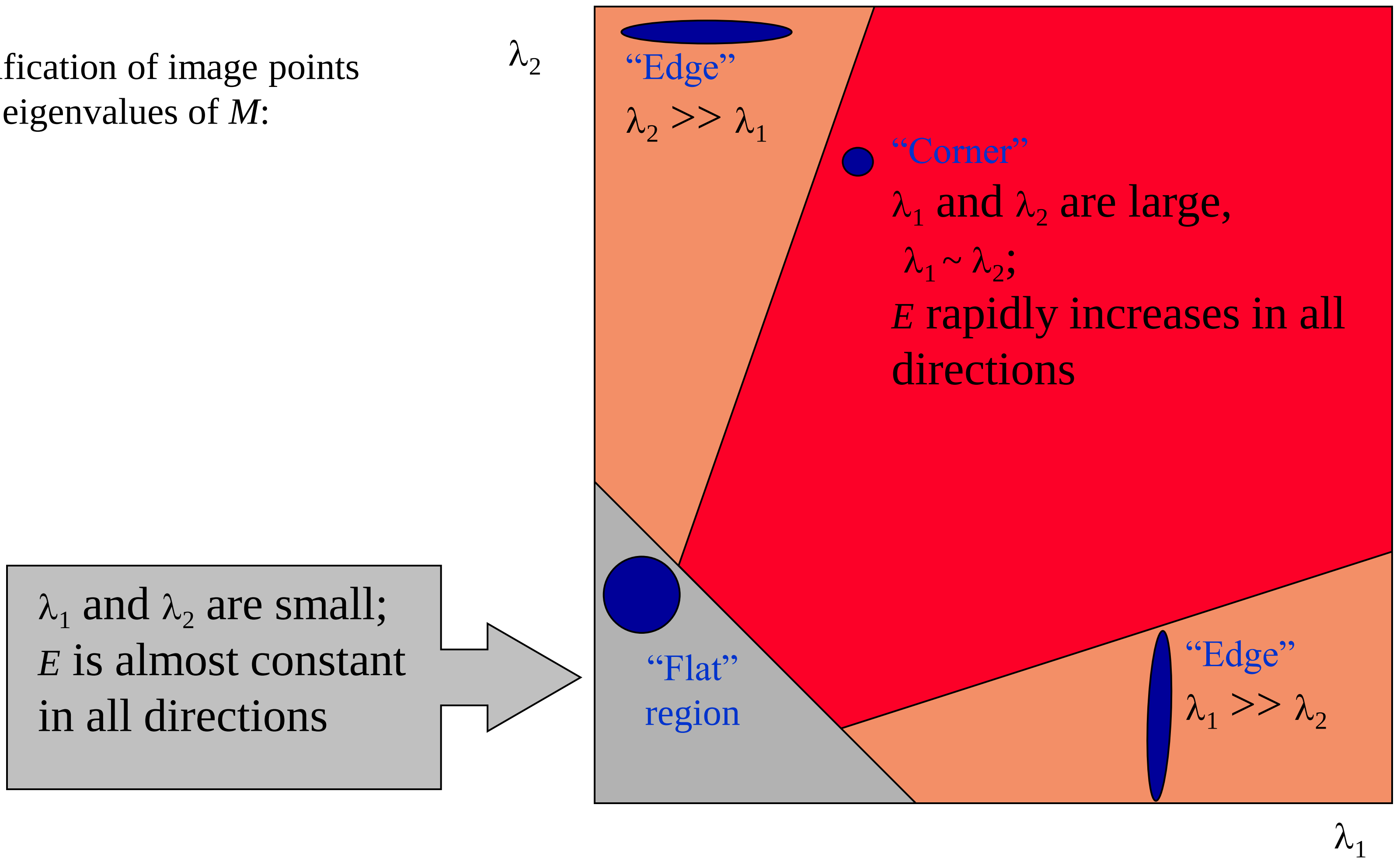
change to a different orthogonal coordinate basis $t = V^T \cdot s$ (more in next topic)

rotation

$$\Rightarrow \quad t^T \cdot \Lambda \cdot t = 1$$

Harris Detector: Mathematics

Classification of image points using eigenvalues of *M*:

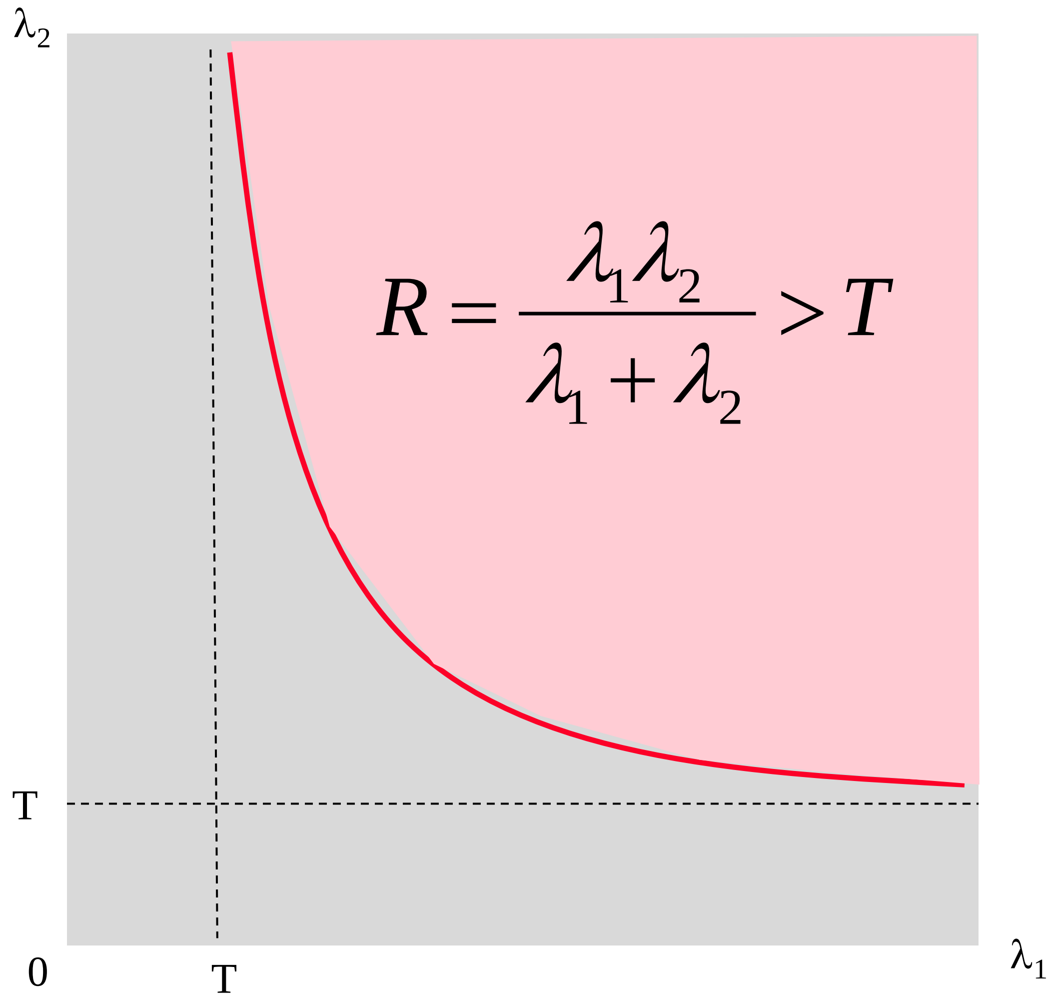


Harris Detector: Mathematics

One common measure of corner response:

$$R = \frac{\det M}{\text{Trace}\, M}$$

$$\det M = \lambda_1 \lambda_2$$
$$\text{trace}\, M = \lambda_1 + \lambda_2$$



$$R = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} > T$$

Q: computational complexity for computing R (corner response) at all image pixels?

(assume window of size nxm and image of size NxM)

Harris Detector

- The Algorithm:

  - Find points with large corner response function R

    $$R > \text{threshold}$$

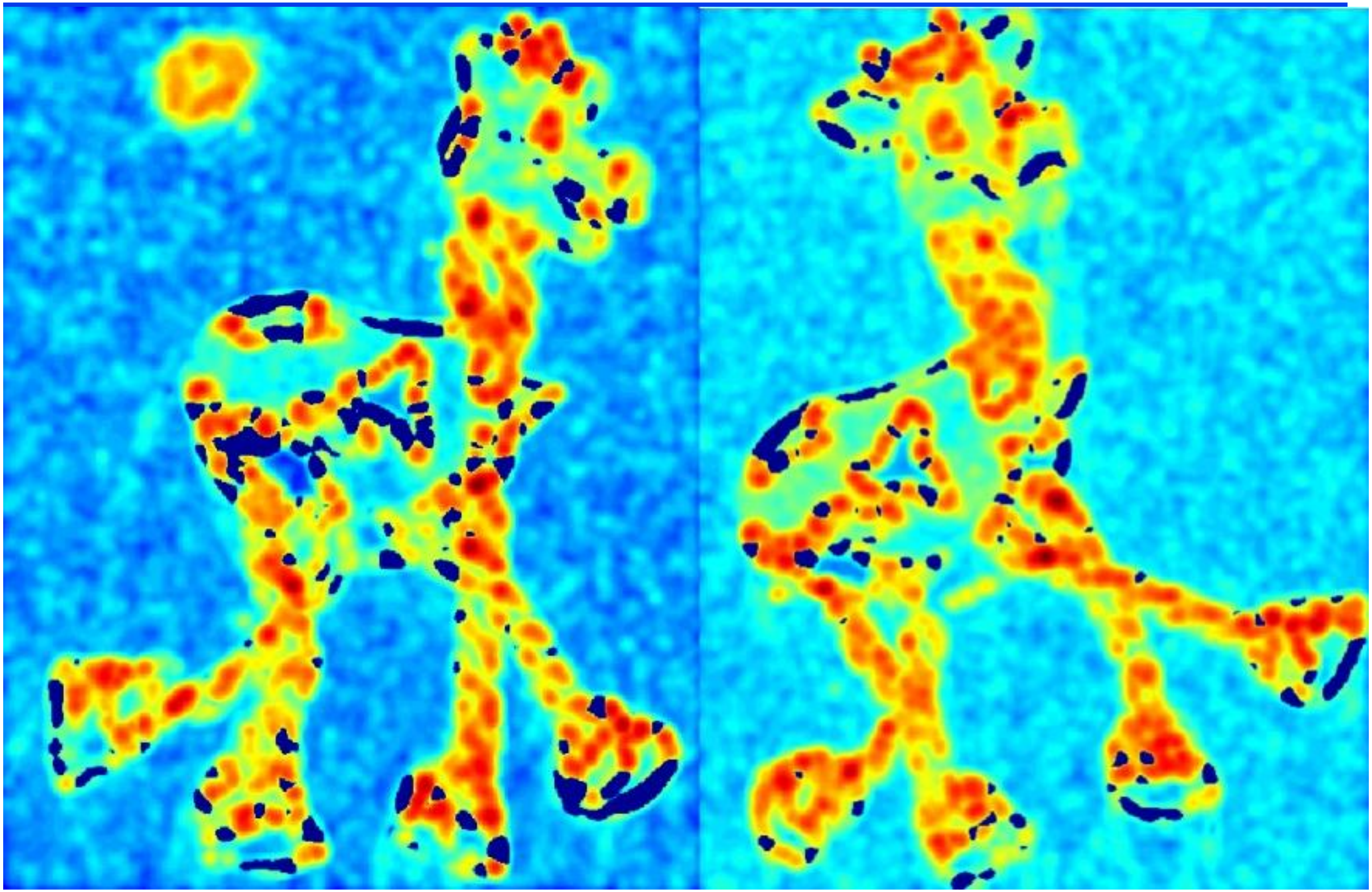
  - Take the points of local maxima of R

Harris Detector: Workflow

Features are often needed to register different views of the same object

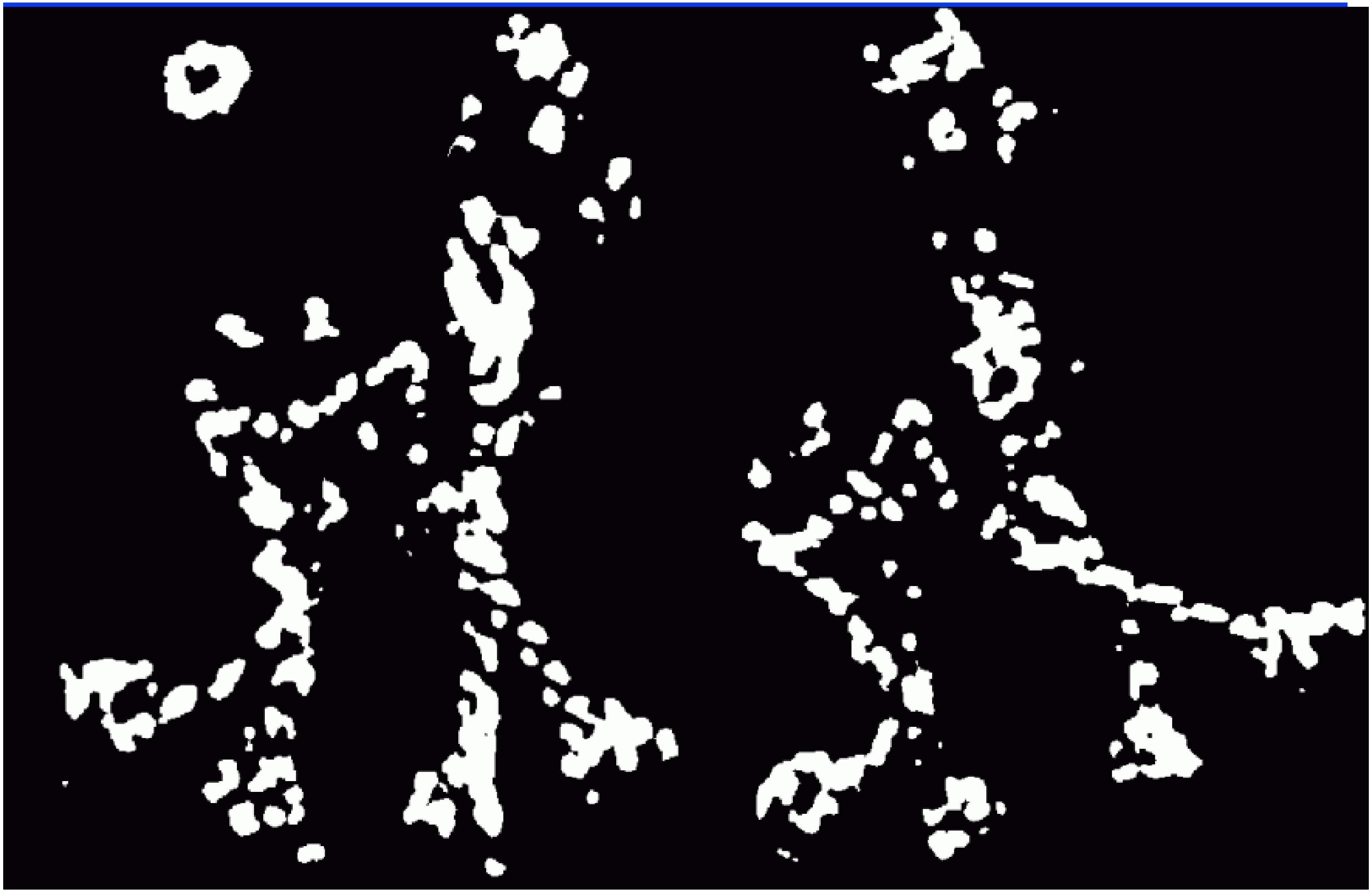
Harris Detector: Workflow

Compute corner response R



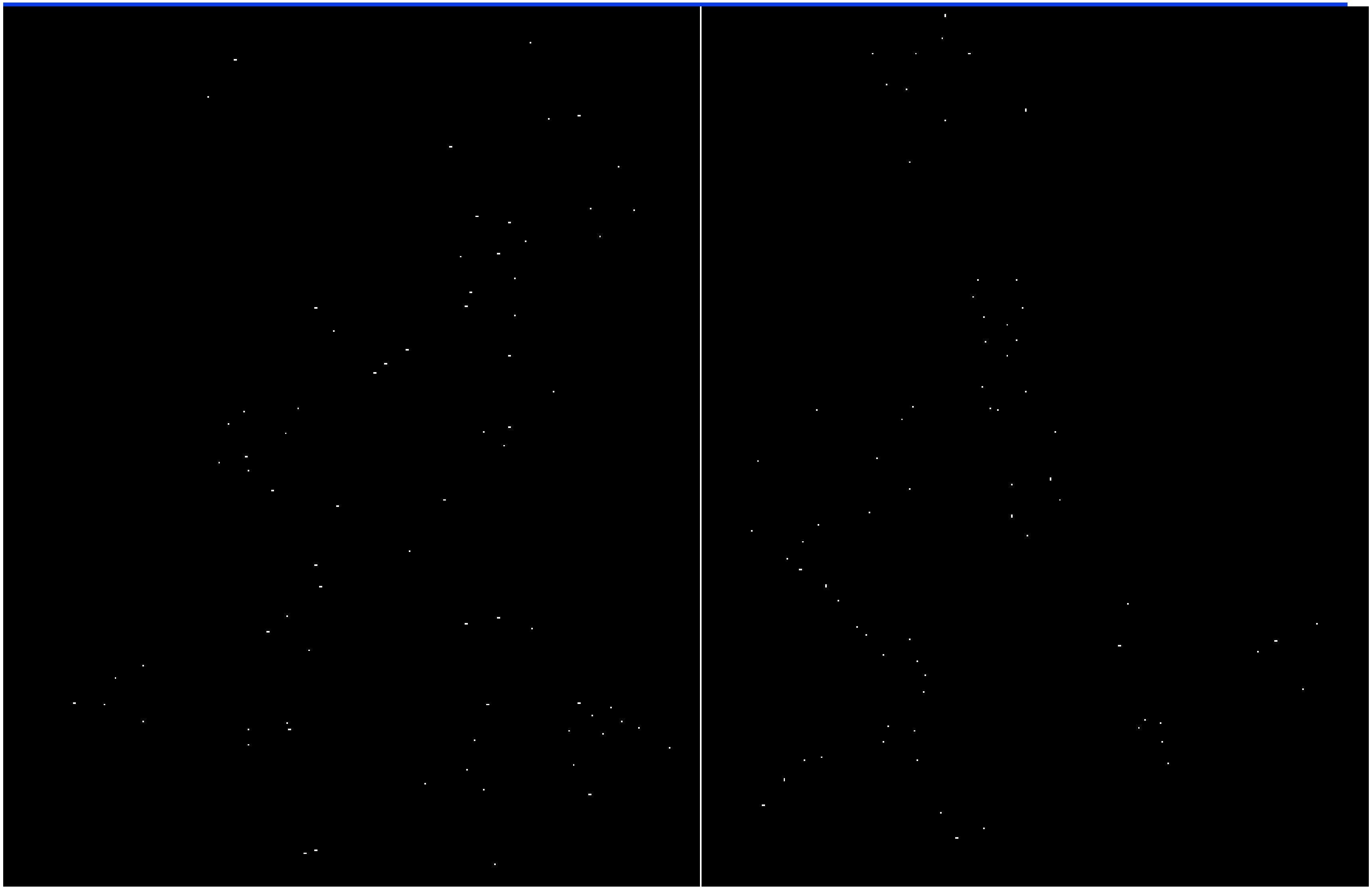
Harris Detector: Workflow

Find points with large corner response: $R > \text{threshold}$



Harris Detector: Workflow

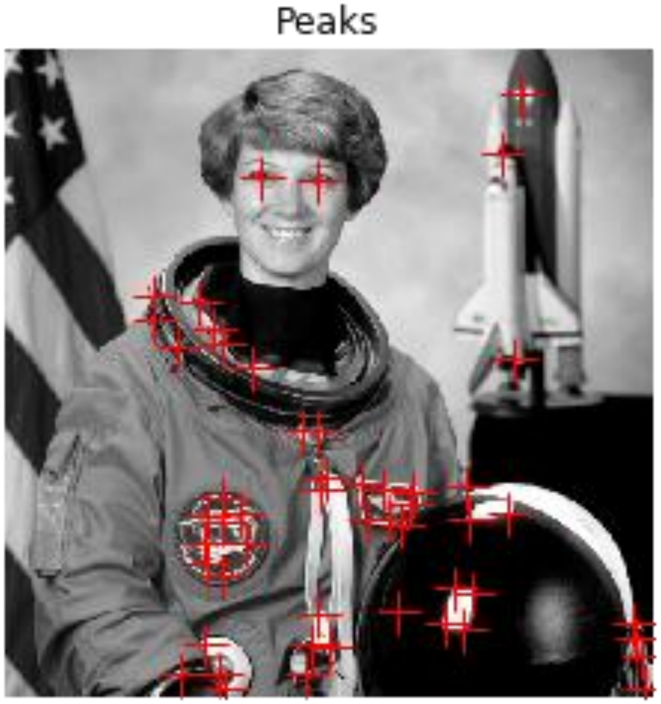
Take only the points of local maxima of R



Harris Detector: Workflow

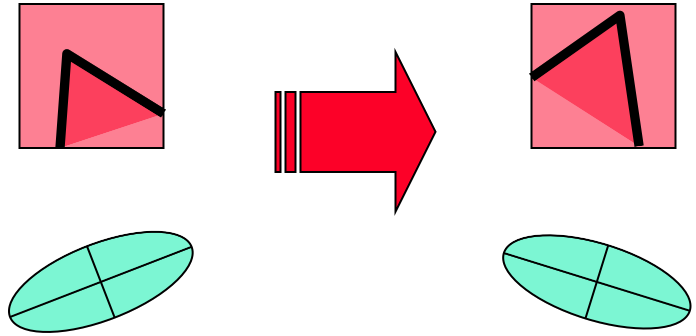
Example of corner features (python)

- from jupiter notebook “FeaturePoints.ipynb”



Harris Detector: Some Properties

- Rotation invariance



Ellipse rotates but its shape (i.e. eigenvalues) remains the same

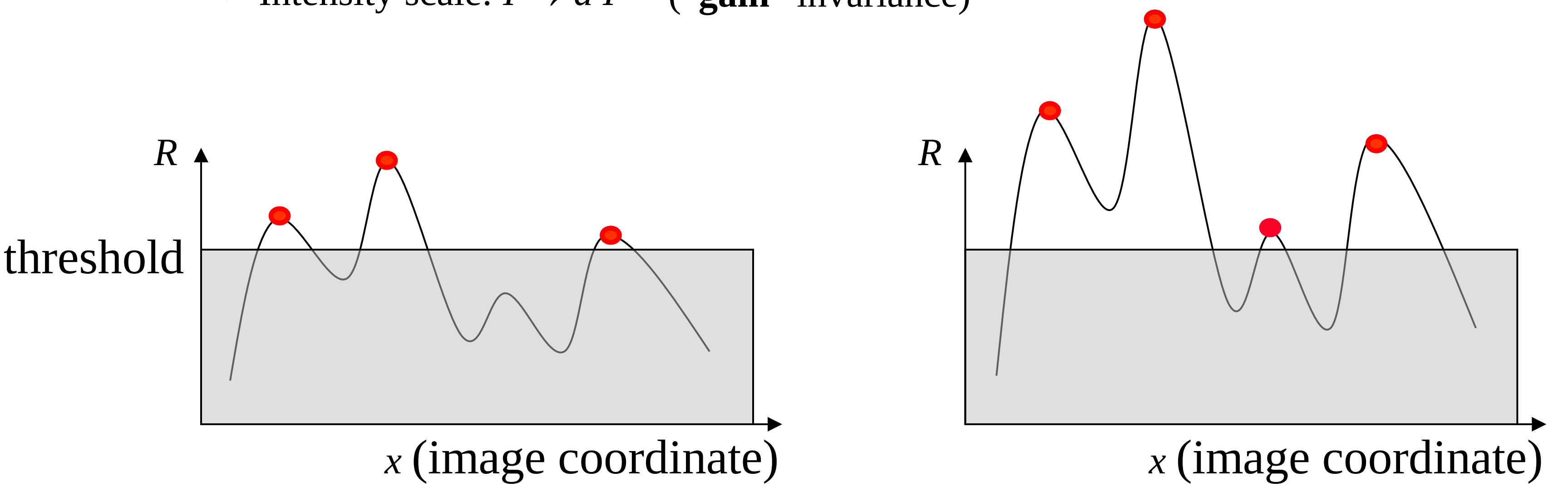
Corner response R is invariant to image rotation

Harris Detector: Some Properties

- Partial invariance to *affine* intensity change

✓ Only derivatives are used => invariance to intensity shift $I \rightarrow I + b$ ("**bias**" invariance)

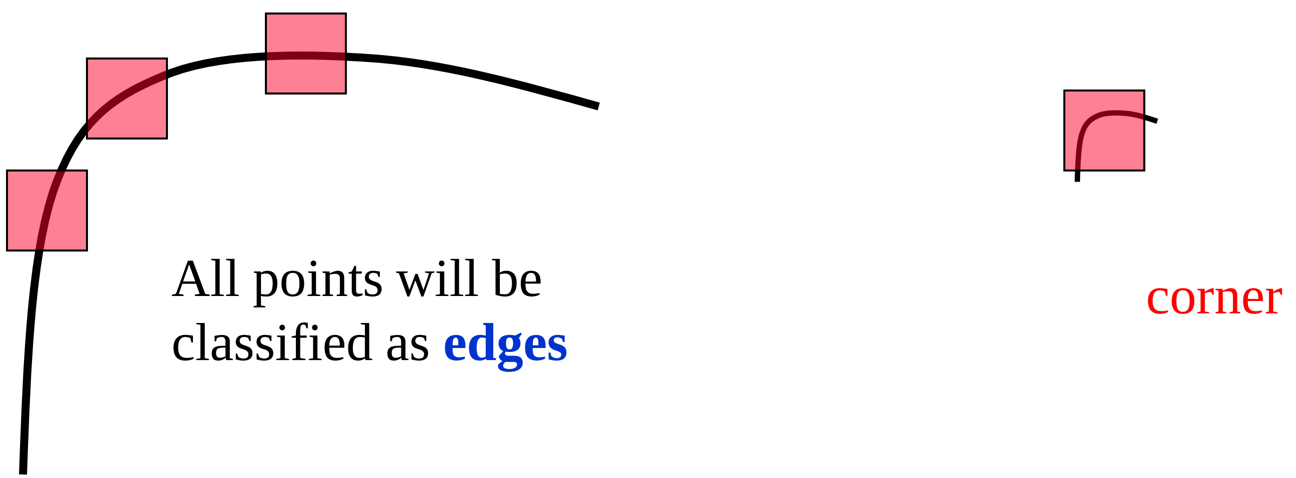
✓ Intensity scale: $I \rightarrow a\, I$ ("**gain**" invariance)



features locations stay the same,
but some may appear or disappear depending on gain a

Harris Detector: Some Properties

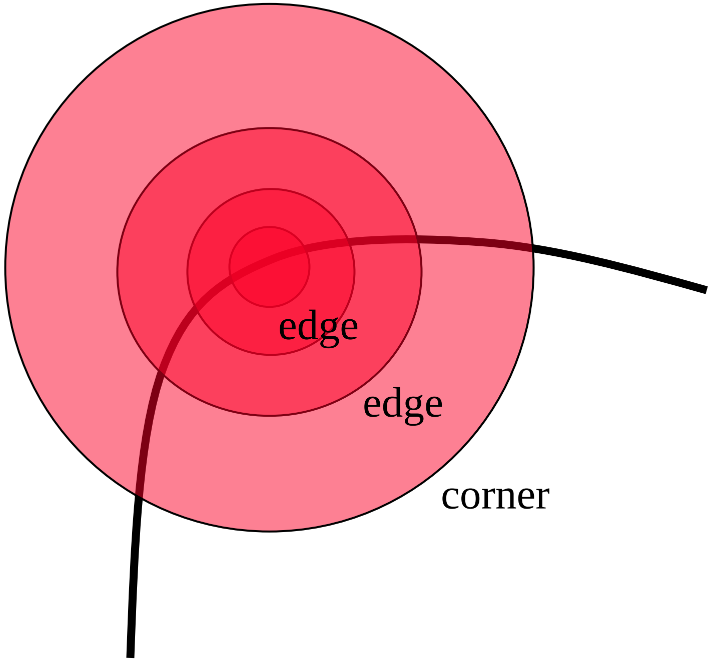
- non-invariant to *image scale*!



Two images of the same object taken at different scales (e.g. zoom settings)

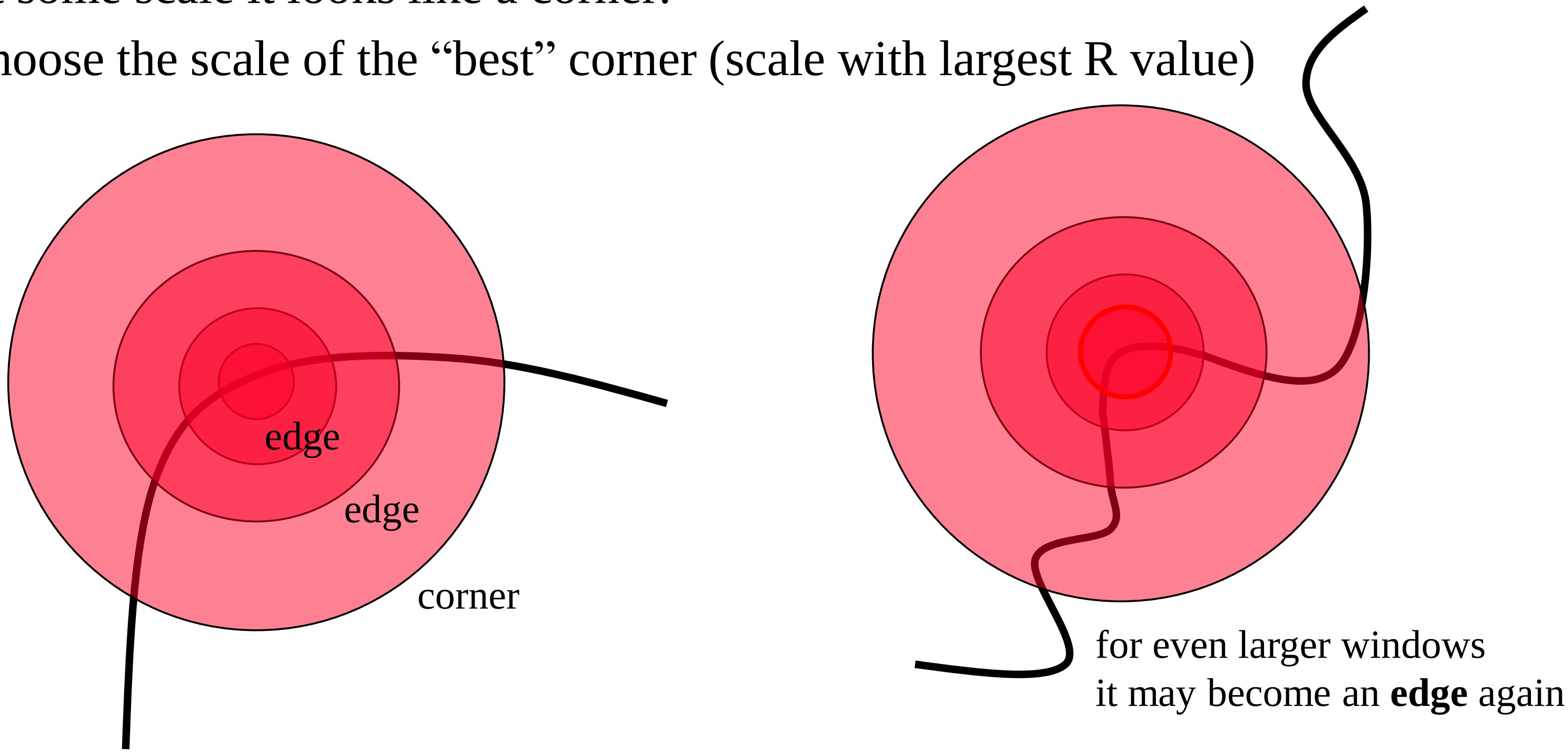
Scale Invariant Detection

- Consider windows (circles) of different sizes (scales) around a point
- At some scale it looks like a corner.



Scale Invariant Detection

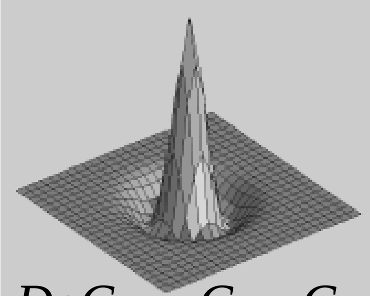
- Consider windows (circles) of different sizes (scales) around a point
- At some scale it looks like a corner.
- Choose the scale of the "best" corner (scale with largest R value)



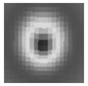
Can use *Gaussian pyramid* for efficient optimal scale selection (see 2 slides later)

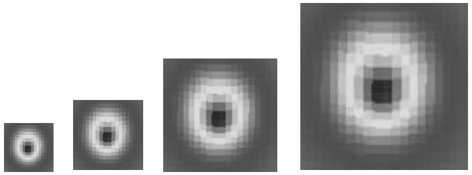
Blob-like discriminant feature points

- *DoG* (or a similar *LoG*) kernels
  are used to detect blob-like features



$$DoG \equiv G_{\varepsilon} - G_{\sigma}$$

Feature locations: extrema points for convolution with 

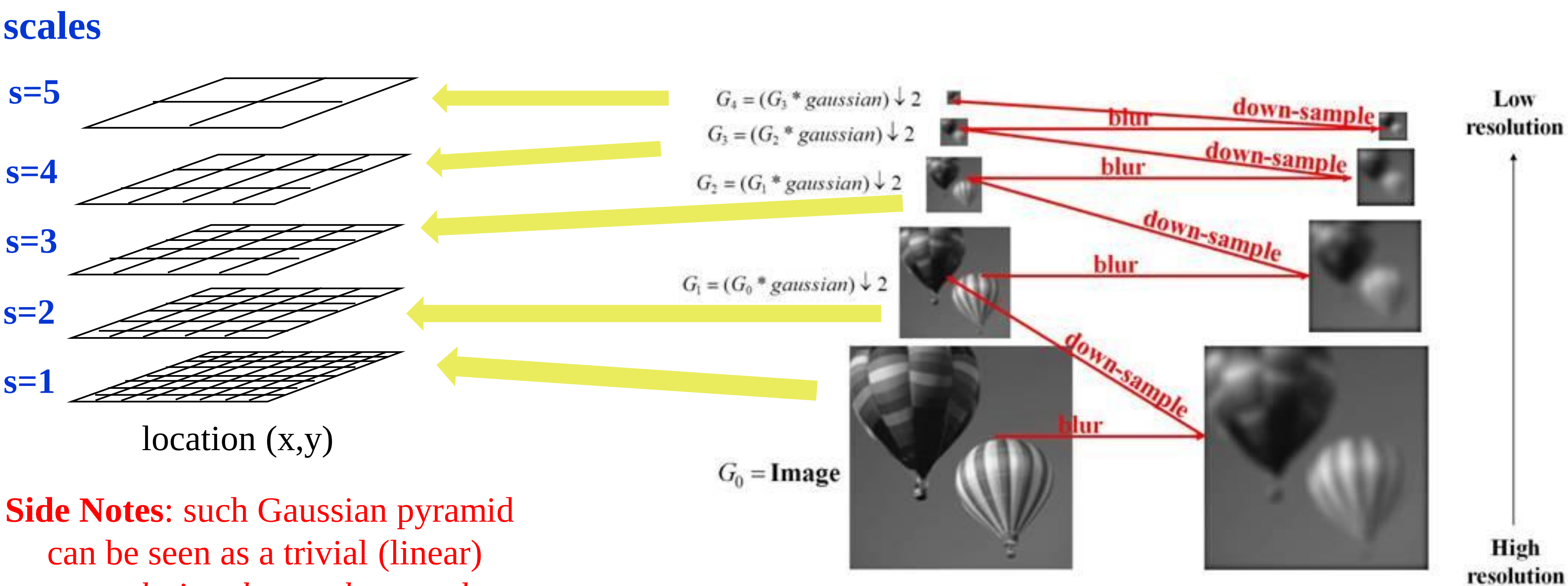
Feature scale is still not known:  ?

How to find the **right scale**?

Instead of rescaling the kernel, **rescaling the image** is more efficient…

Gaussian pyramid

- *Gaussian pyramid* helps to find "optimal scale" for features



Side Notes: such Gaussian pyramid can be seen as a trivial (linear) *convolutional neural network*

- similar multiresolution pyramid also appears in the "*encoder*" part of common segmentation CNNs

Gaussian pyramid

- *Gaussian pyramid* helps to find "optimal scale" for features

e.g. consider **DoG features**

scales

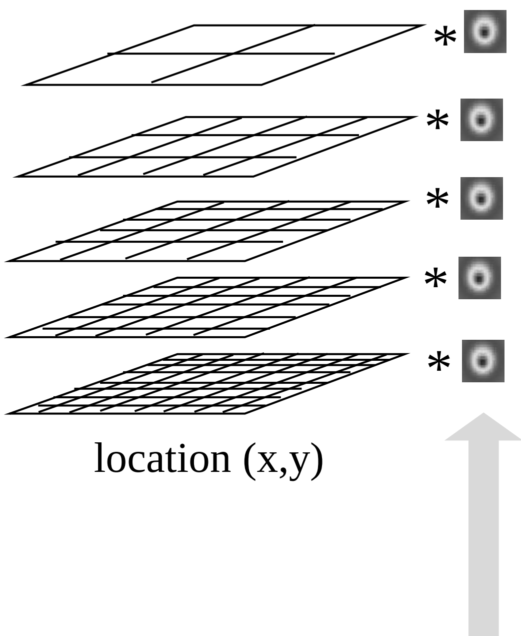
s=5

s=4

s=3

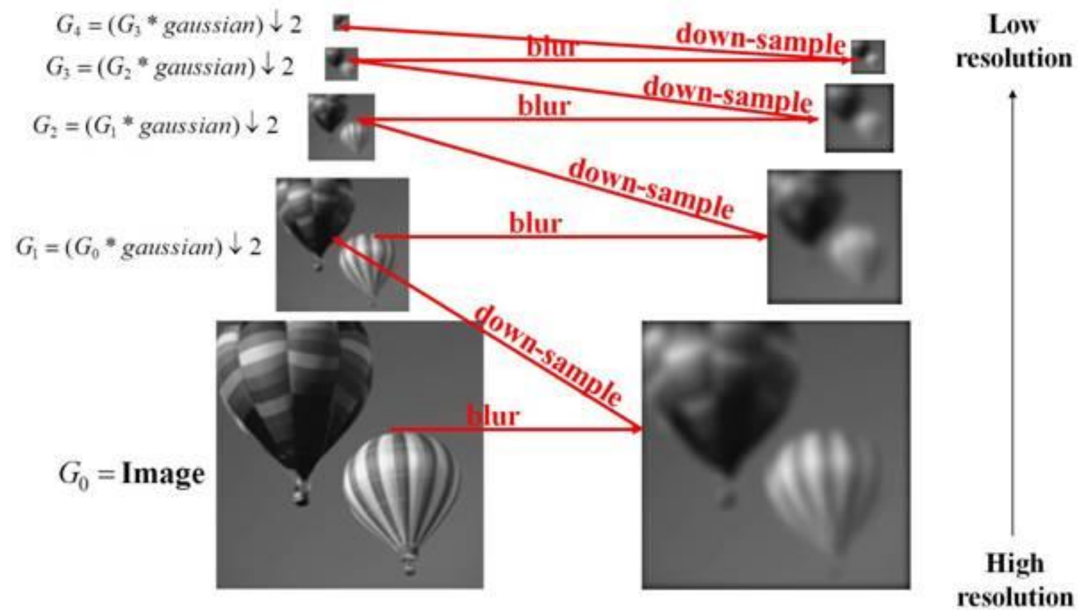
s=2

s=1



location (x,y)

compute feature response
(*e.g.* convolve or NCC w. kernel)
with image **at each scale**

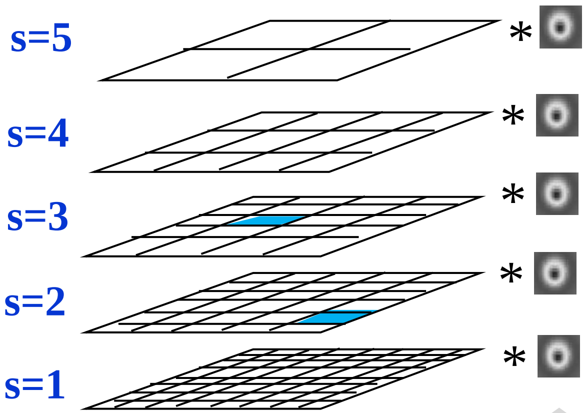


Gaussian pyramid

- *Gaussian pyramid* helps to find "optimal scale" for features

e.g. consider **DoG features**

scales

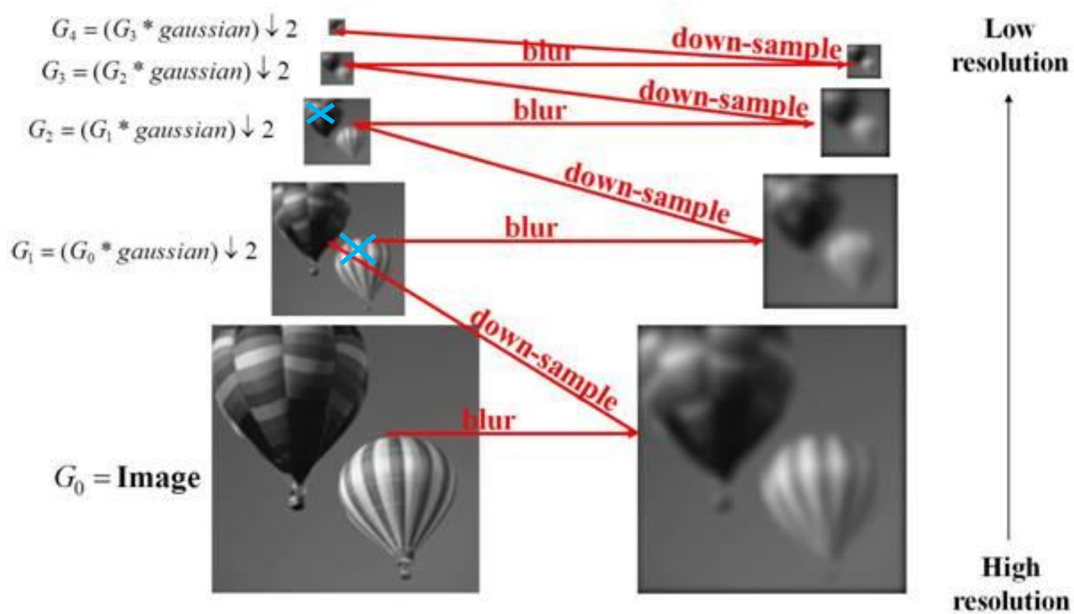


location (x,y)

find local maxima response in volume

(x,y,s)

compute feature response
(*e.g.* convolve or NCC w. kernel)
with image **at each scale**



Example (python)

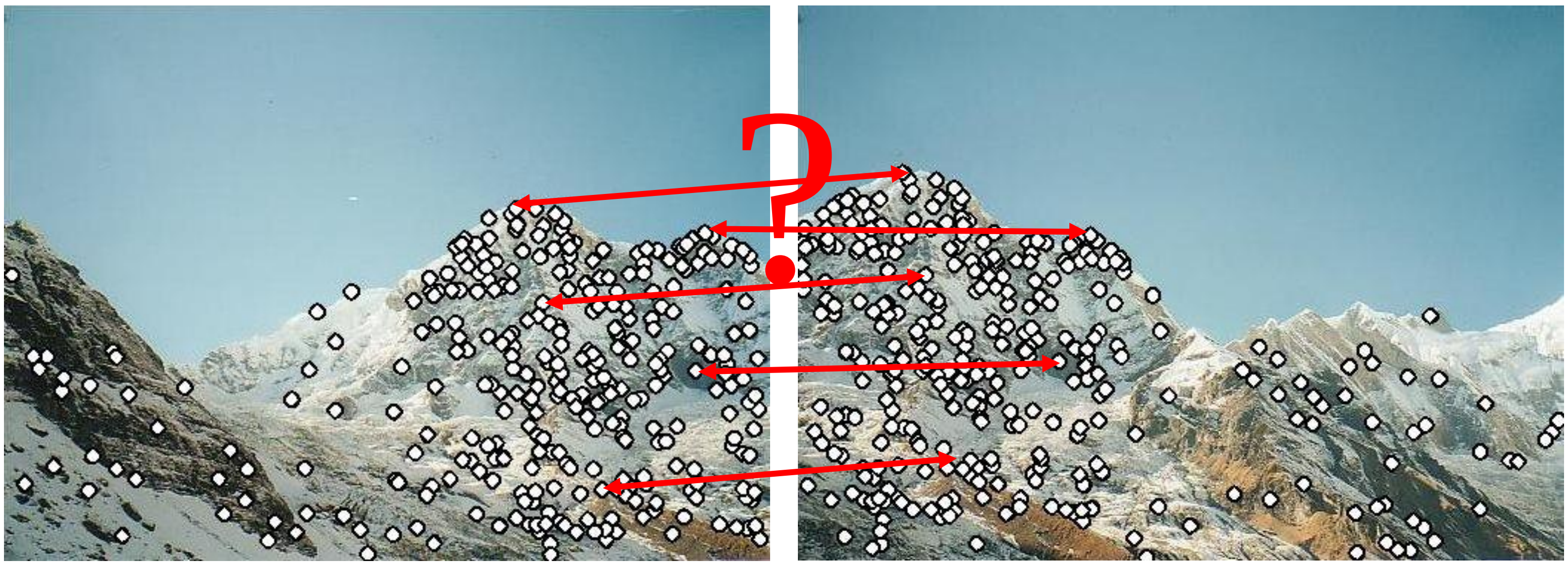
- from jupiter notebook “FeaturePoints.ipynb”

Dog

circle center -> feature location
circle radius -> feature scale

Features: location + **descriptor**

- Now we know how to detect (locate) interest points or features
- Next question: **How to match them?**



Besides location each feature point should have its signature or **descriptor**

Point descriptor should be:	**invariant**	(stable to illumination and view point changes)
	distinctive	(discriminant)

Common generic feature points

- MOPS, Hog, SIFT, …

Features are characterized by	**location** and	**descriptor**
color	any pixel	RGB vector
edge	local extrema of $\|\nabla f\|$	∇f
MOPS	corners	normalized intensity patch
HOG SIFT	DOG or LOG extrema points or other interest points	gradient orientation histograms

more below

highly discriminative

(see Szeliski, Sec. 4.1.2)

Multi-Scale Oriented **Patches** (MOPS)

Summary of main ideas:

- Patch location and orientation
  - Multi-scale Harris corners
  - Orientation from blurred gradient => invariant to rotation

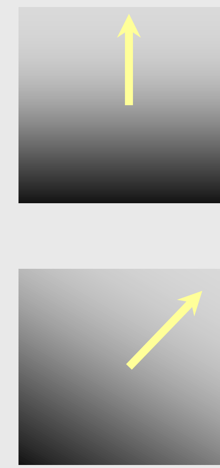
- Descriptor vector
  - Sampling of intensities in a local 8x8 patch
  - Bias/gain normalization => invariance to affine intensity changes

[Brown, Szeliski, Winder, CVPR'2005]

MOPS: patch location and orientation

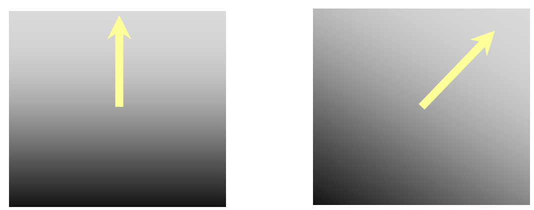
- Location and Scale – Harris corner
- Orientation - blurred gradient
- **Rotation Invariant Frame**
  - Scale-space position (x, y, s) + orientation (θ)

blurred gradient orientation



Descriptors Invariant to Image Rotation

find "dominant" direction of image gradient in the neighborhood (e.g. blurred-image gradient) to set patch orientation (θ)

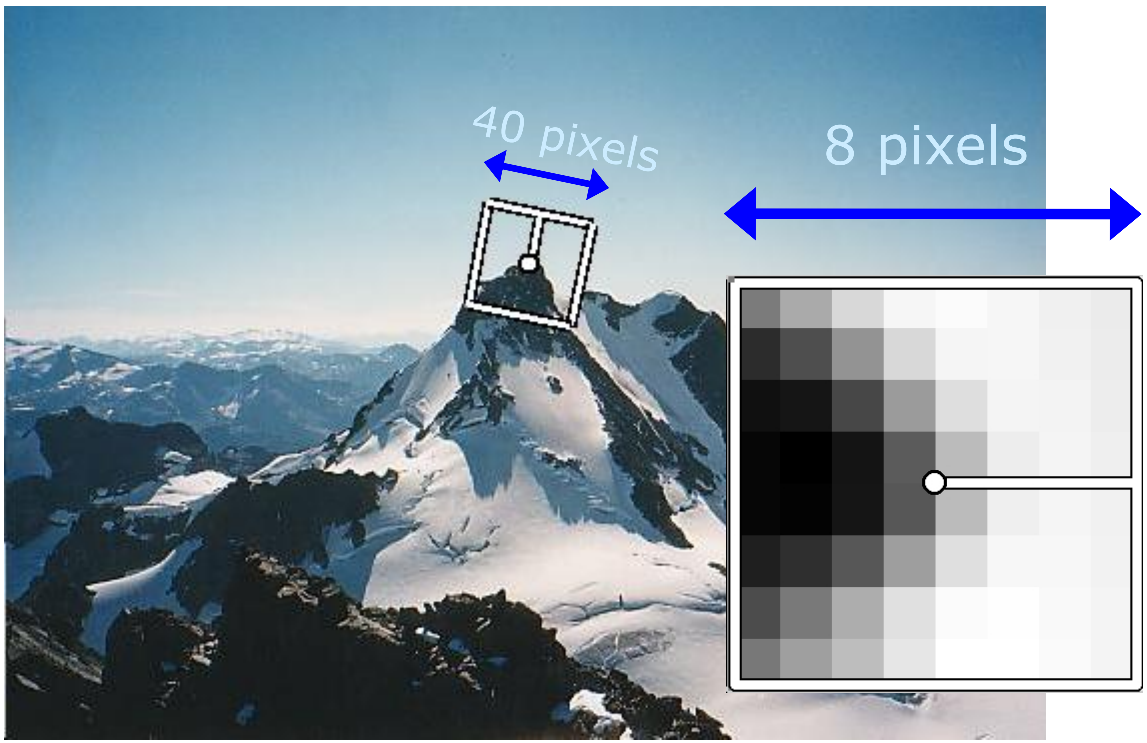


- Set patch/descriptor orientation based on such direction
  => **invariance to camera/image rotation**

MOPS: descriptor vector

- 8x8 <u>oriented</u> patch
  - sampled at 5 x scale
- Bias/gain normalization: $I' = (I - \mu)/\sigma$

invariance to image intensity <u>bias</u> & <u>gain</u>



Detections at multiple scales

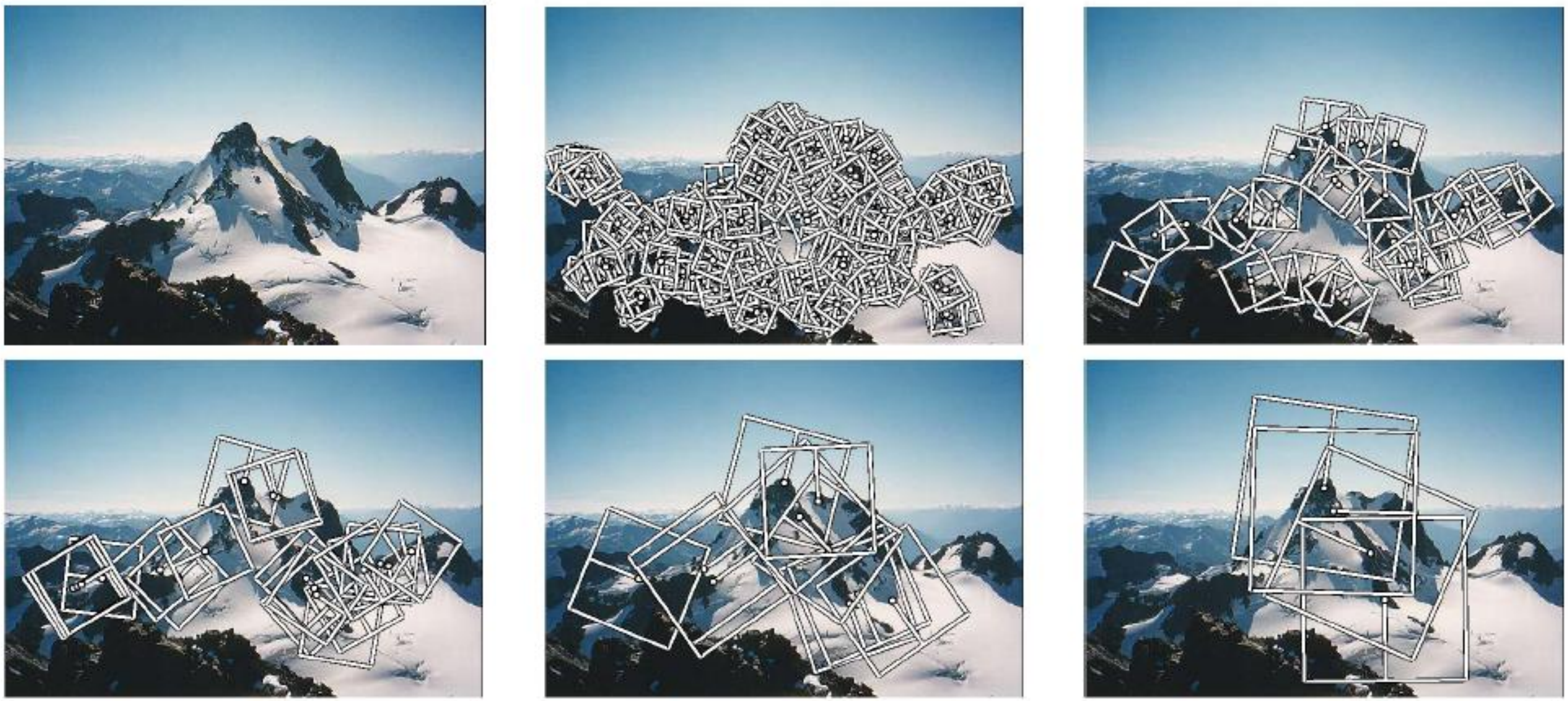


Figure 1. Multi-scale Oriented Patches (MOPS) extracted at five pyramid levels from one of the Matier images. The boxes show the feature orientation and the region from which the descriptor vector is sampled.