Edges, interpolation, templates

Nuno Vasconcelos

ECE Department, UCSD

(with thanks to David Forsyth)

Edge detection

- edge detection has many applications in image processing
- an edge detector implements the following steps:
  - compute gradient magnitude

$$\left\|\nabla f(x_0, y_0)\right\|^2 = \left(\frac{\partial f}{\partial x}(x_0, y_0)\right)^2 + \left(\frac{\partial f}{\partial y}(x_0, y_0)\right)^2$$

  - thin and follow edge points
    - find locations of maximum gradient magnitude
    - follow these maxima to form contours
    - discard points that are not maxima
    - declare maxima as edges

Derivatives

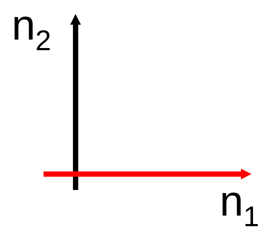
- to compute the derivatives

$$\left(f_x(x,y), f_y(x,y)\right) = \left(\frac{\partial f}{\partial x}(x_0, y_0), \frac{\partial f}{\partial y}(x_0, y_0)\right)$$

we rely on a sequence of

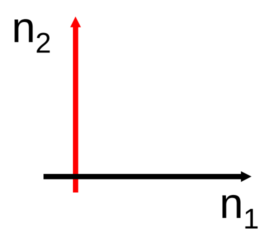
- smoothing with a Gaussian (to eliminate noise)
- convolution with difference filter
- f_x:

$$\begin{matrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{matrix} \qquad\qquad \begin{matrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{matrix}$$



- f_y:

$$\begin{matrix} 0 & -1 & 0 \\ 0 & 1 & 0 \end{matrix} \qquad\qquad \begin{matrix} -1 & -1 & -1 \\ 1 & 1 & 1 \end{matrix}$$



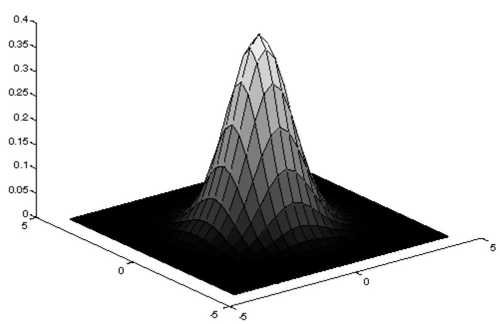
Derivatives

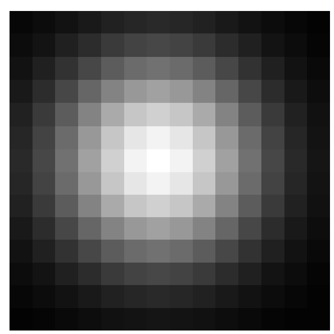
- accomplished in a single step
  - by convolving image with two derivative of a Gaussian (DoG) filters

$$h_x(n_1,n_2) = g(n_1+1,n_2) - g(n_1,n_2)$$
$$h_y(n_1,n_2) = g(n_1,n_2+1) - g(n_1,n_2)$$

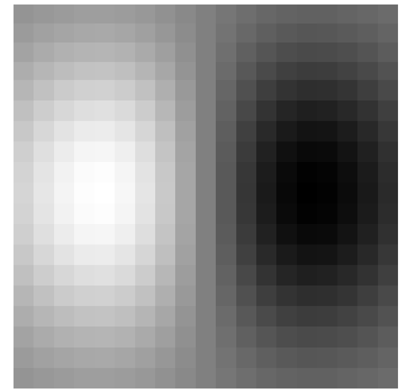
  - where

$$g(n_1,n_2) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{n_1^{\ 2}+n_2^{\ 2}}{2\sigma^2}\right)$$

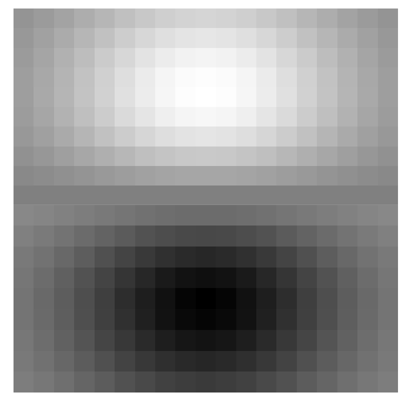




DoG along n_1



DoG along n_2



The Canny edge detector

- original image (Lena)

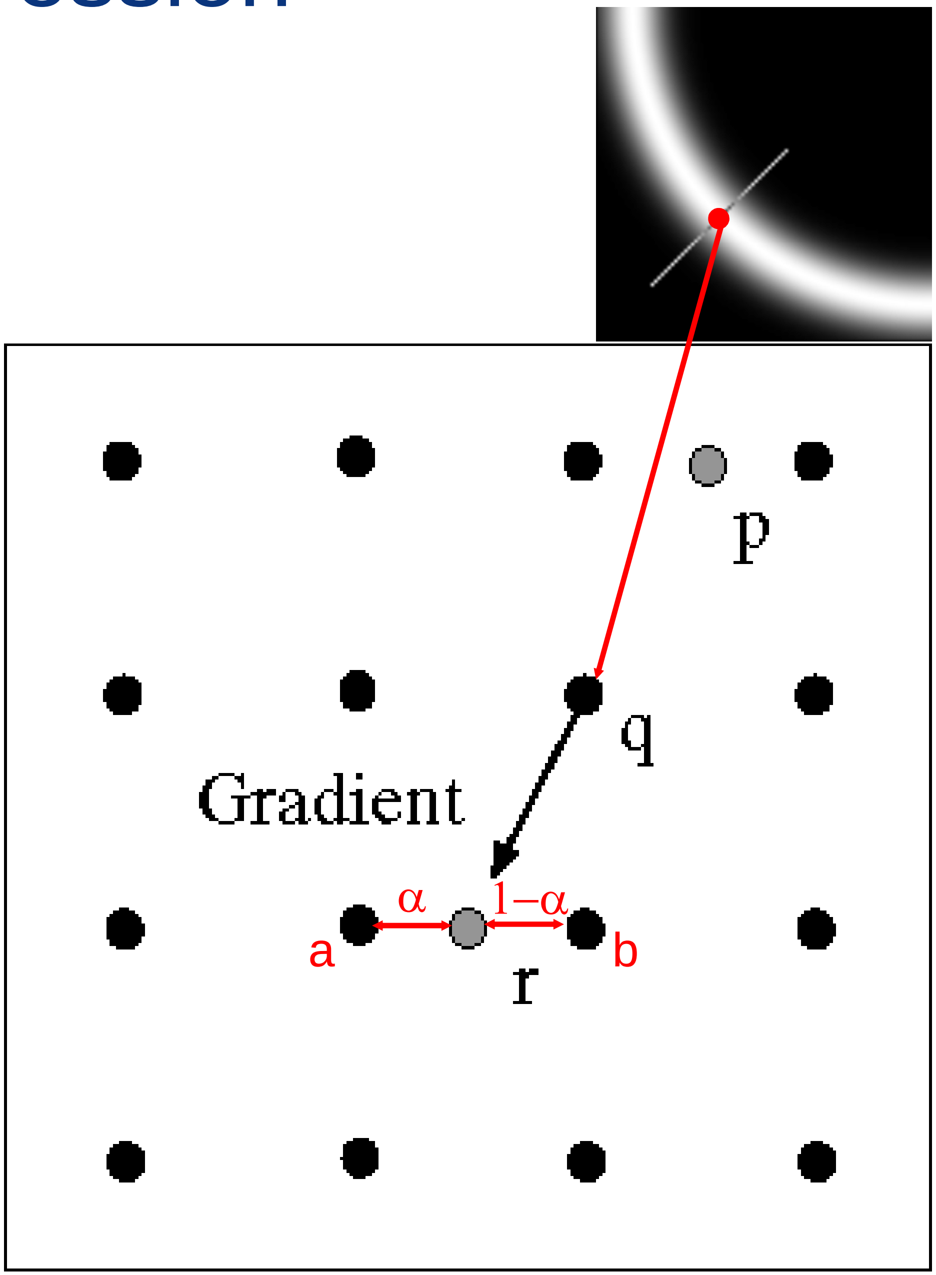
The Canny edge detector

- norm of the gradient

Non-maximum suppression

- is there a maximum at q?
- yes, if value at q is larger than those at both p and r
- p and r are the pixels in the direction of the gradient that are 1 pixel apart from q
- typically they do not fall in the pixel grid
- we need to interpolate, e.g.

$$r = \alpha\, b + (1 - \alpha)\, a$$

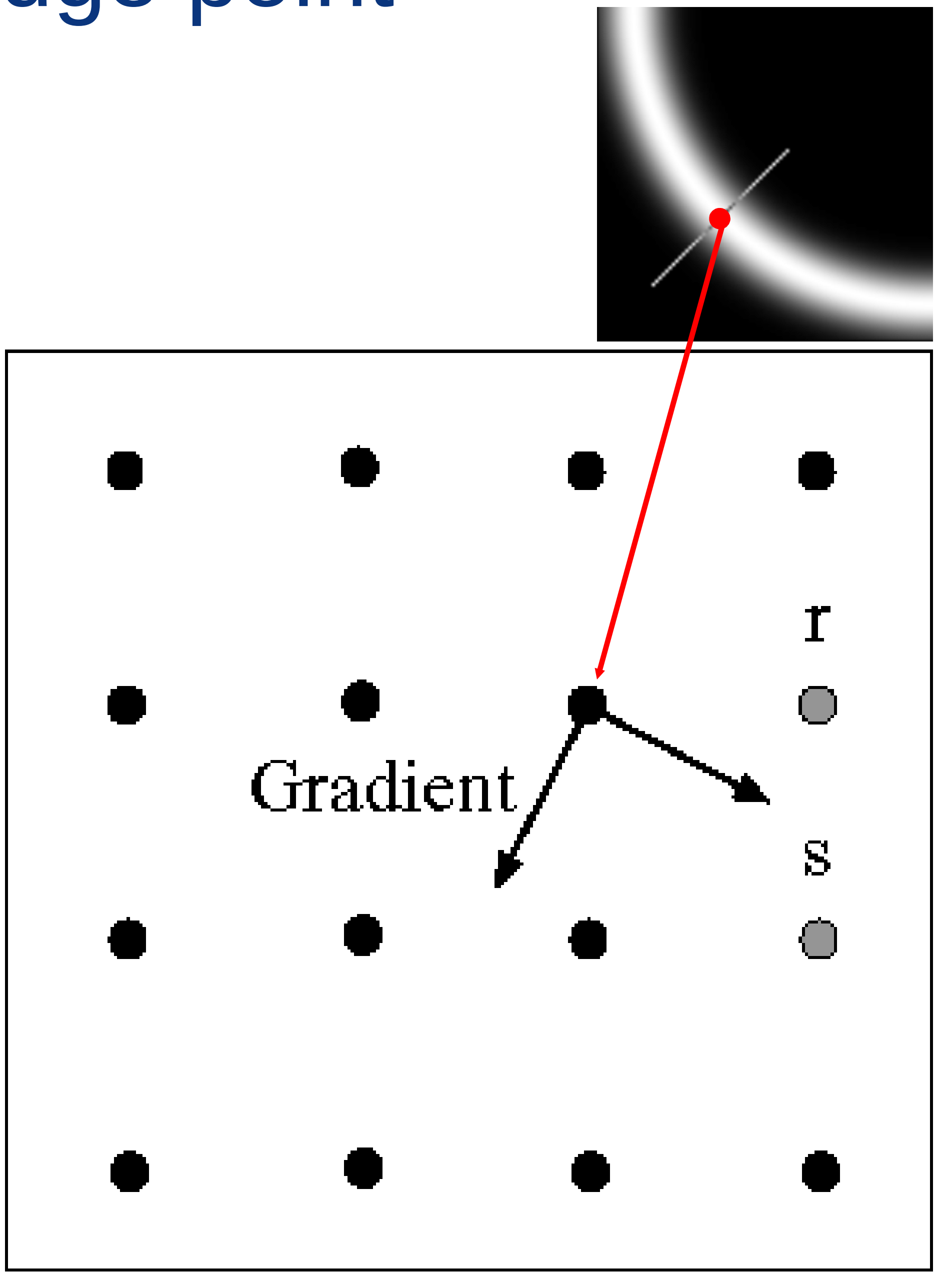


Predicting the next edge point

- assume the marked point is an edge point
- we construct the tangent to the edge curve (which is normal to the gradient at that point)

$$t(x,y) = \left(-f_y(x,y), f_x(x,y)\right)^T$$

- use this to predict the next points (here either r or s).



Cleaning up

- even when gradient is ~ zero, there are maxima due to noise
- check that maximum value of gradient value is large enough (threshold)
- once we are following an edge we must avoid gaps due to similarity with background
- use **hysteresis**
    - use a high threshold to start edge curves and a low threshold to continue them.

The Canny edge detector

- original image (Lena)

The Canny edge detector

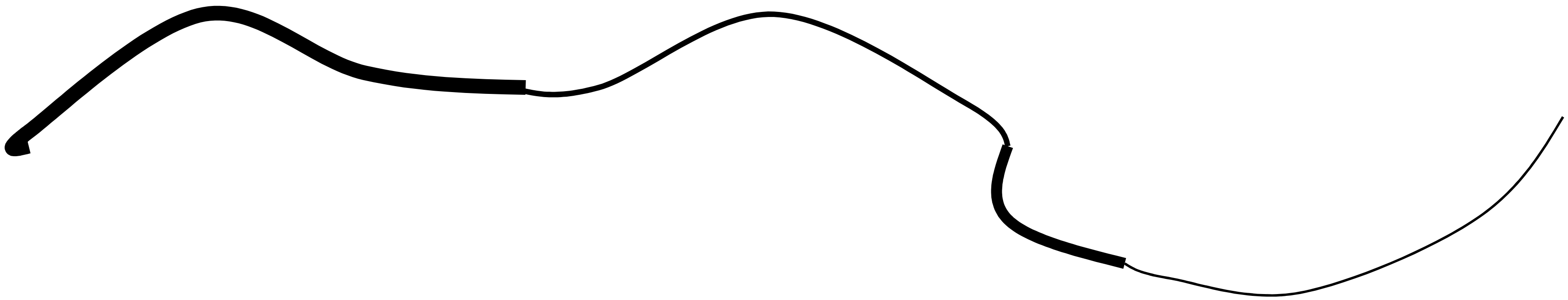
- norm of the gradient

The Canny edge detector

- thinning
- (non-maximum suppression)

Hysteresis

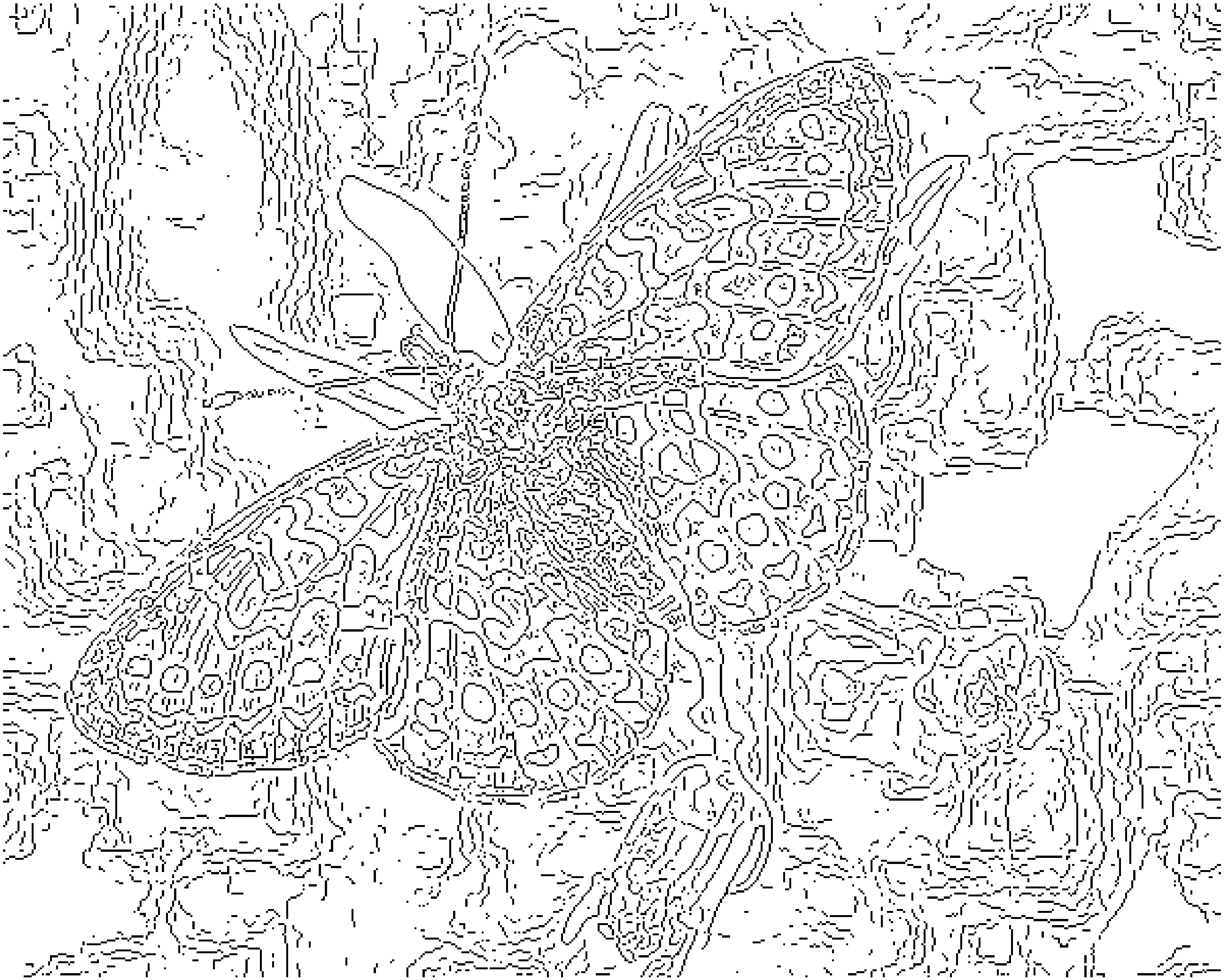
- suppose this is a curve that we are following
  - thickness represents the magnitude of the gradient



  - we require a large magnitude to start, i.e. above a threshold T_1
  - once we start we keep going even if the magnitude falls below the threshold
  - we only declare the contour as done if it falls below a second threshold T_2, where $T_2 < T_1$
  - once again, the optimal values of these thresholds are image dependent

Parameter tuning

- in summary, the combination of
  - smoothed derivatives,
  - detection of maxima of gradient magnitude,
  - edge following
- is the essence of most modern edge detectors
- the classical is the "Canny edge detector" which implements all this steps
- as we have seen there are a number of parameters
  - smoothing scale
  - two hysteresis thresholds
- in practice these can have significant effect on the quality of the resulting edge maps
- unfortunately there are no universally good values



fine scale
high
threshold

too many
false
edges

coarse scale, high threshold

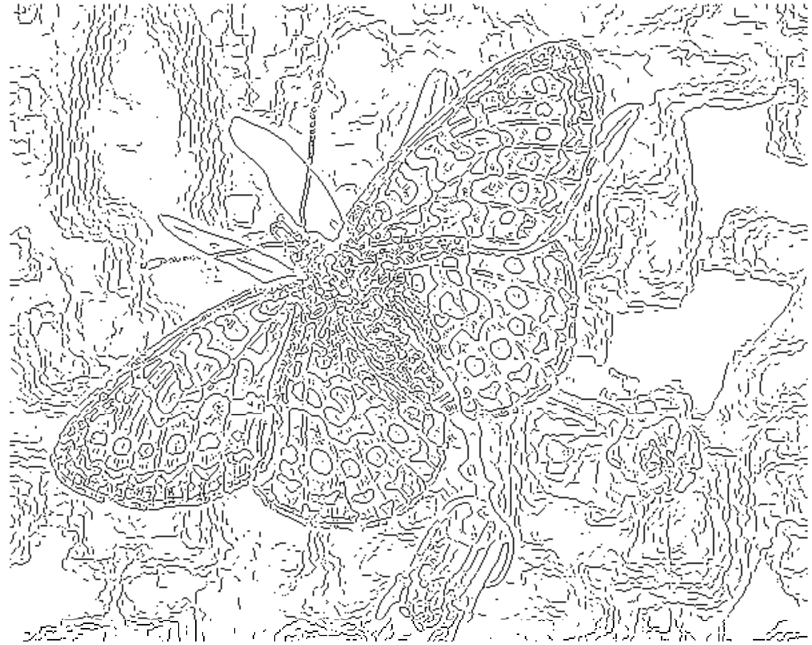
we loose edge points

coarse
scale
low
threshold

we still
have gaps
but once
again
false
edges

The Canny edge detector

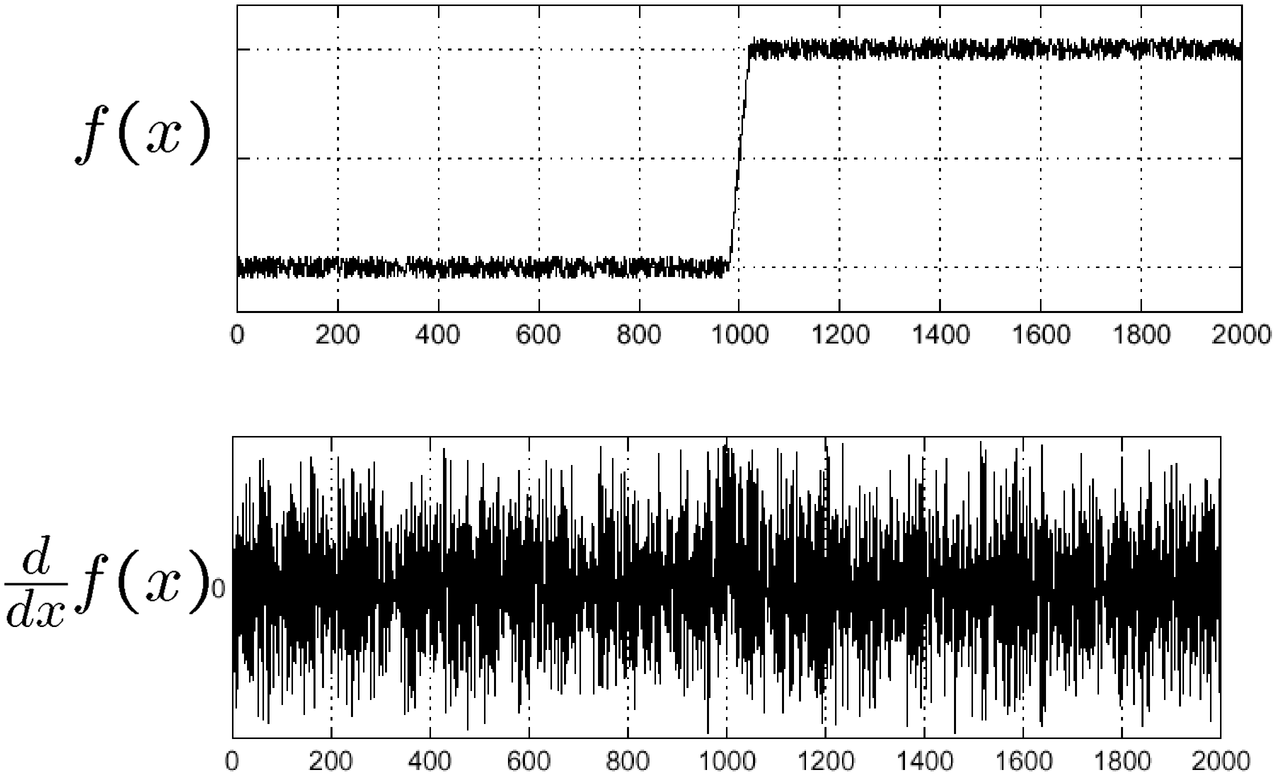
- there are many implementations available
  - matlab has one
  - there is freely available C code on the web
  - there are various applets that allow you to play with the parameters
  - an example is
  - http://www.cs.washington.edu/research/imagedatabase/demo/edge/
  - make sure you experiment and get a feel for how the parameters influence the edge detection results
  - the Canny edge detectors is the closest that you will find to a standard solution to a vision problem

problem: various parameters, for all values we tried result was not perfect


Effects of noise

- Is there an alternative?
  - recall we followed this path to overcome the noise problem



- are there other alternatives?

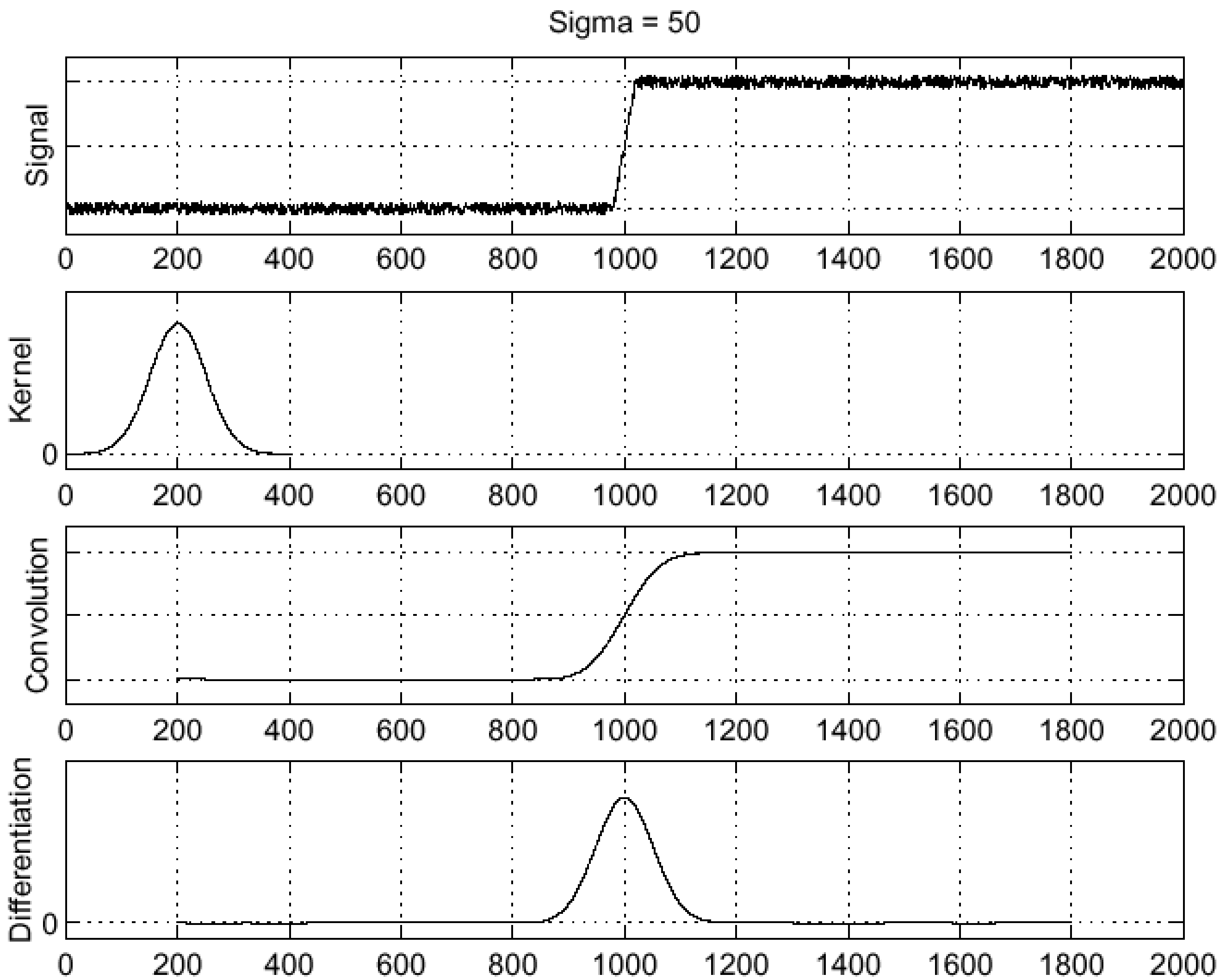
Solution: smooth first

f

h

$h \star f$

$\frac{\partial}{\partial x}(h \star f)$



- this is what we get with 1st order derivatives

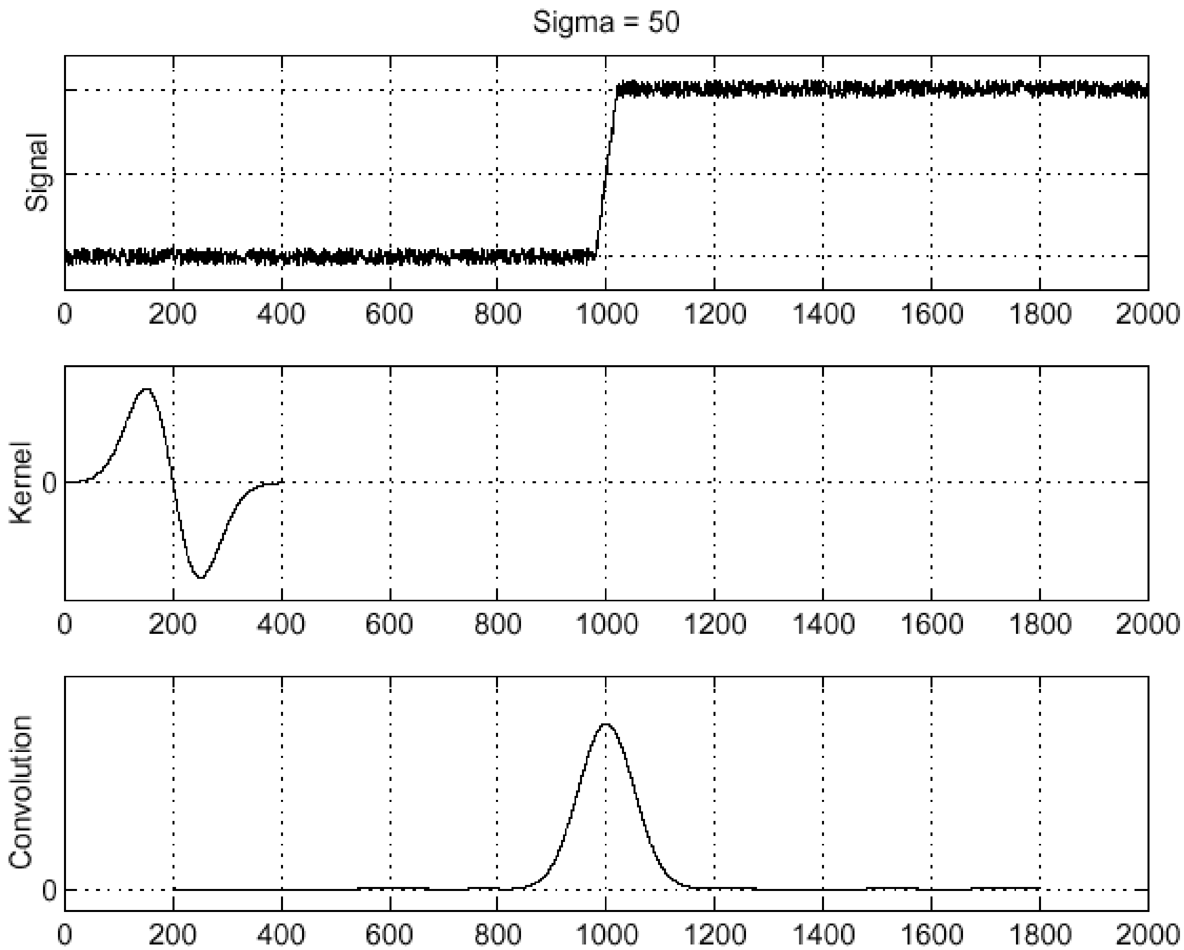
Derivative theorem of convolution

$$\frac{\partial}{\partial x}(h \star f) = (\frac{\partial}{\partial x}h) \star f$$

f

$\frac{\partial}{\partial x}h$

$(\frac{\partial}{\partial x}h) \star f$



- can we extend this idea?

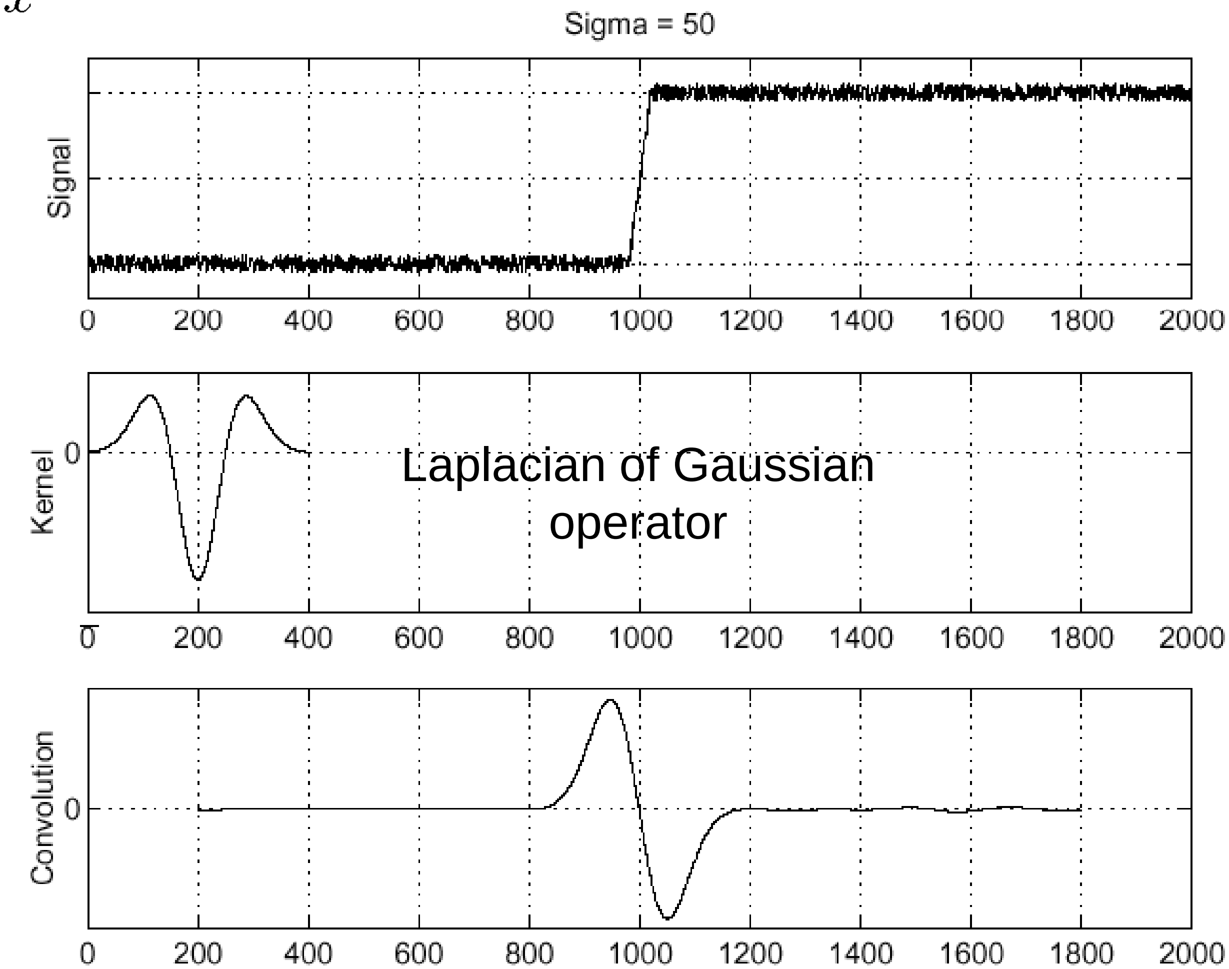
Laplacian of Gaussian

- Consider $\frac{\partial^2}{\partial x^2}(h \star f)$

$$f$$

$$\frac{\partial^2}{\partial x^2}h$$

$$(\frac{\partial^2}{\partial x^2}h) \star f$$



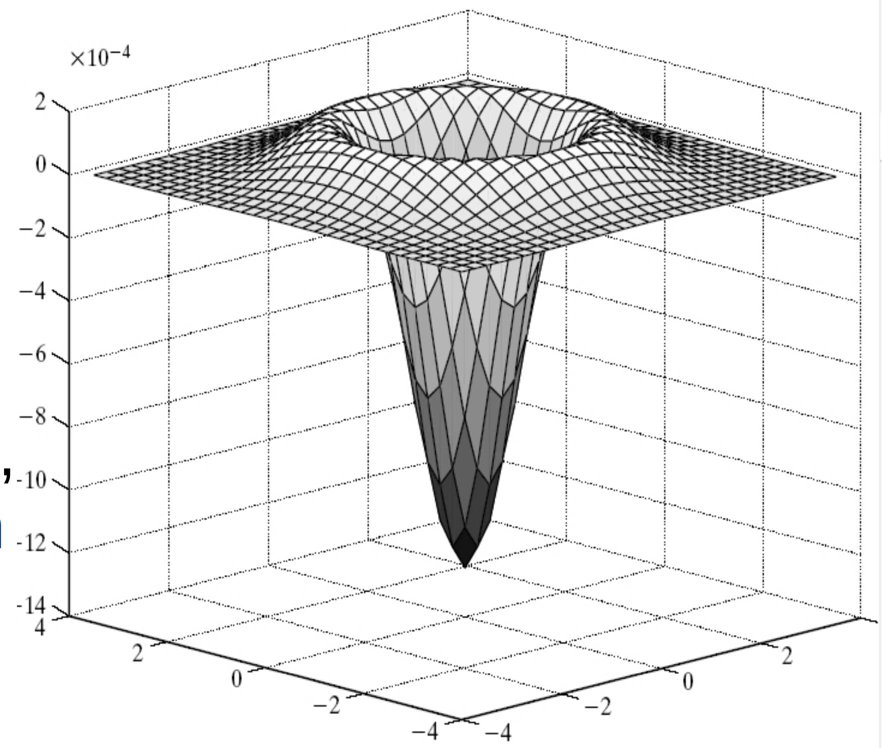
- where is the edge?
- zero-crossings of bottom graph

The Laplacian of Gaussian

- another way to detect max of first derivative is to look for a zero second derivative
- 2D analogy is the Laplacian

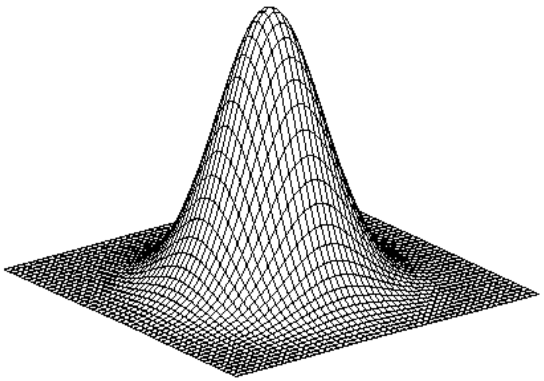
$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y)$$

- with second-order derivatives, noise is even greater concern
- smoothing
  - smooth with Gaussian, apply Laplacian
  - this is the same as filtering with a Laplacian of Gaussian filter



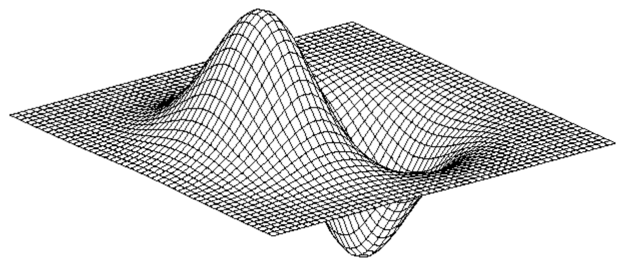
$\nabla^2 G_\sigma(x,y)$

2D edge detection filters



Gaussian

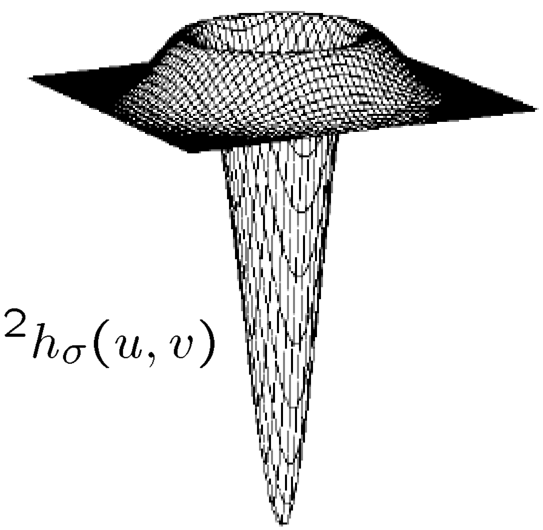
$$h_\sigma(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}}$$



derivative of Gaussian

$$\frac{\partial}{\partial x} h_\sigma(u,v)$$

Laplacian of Gaussian



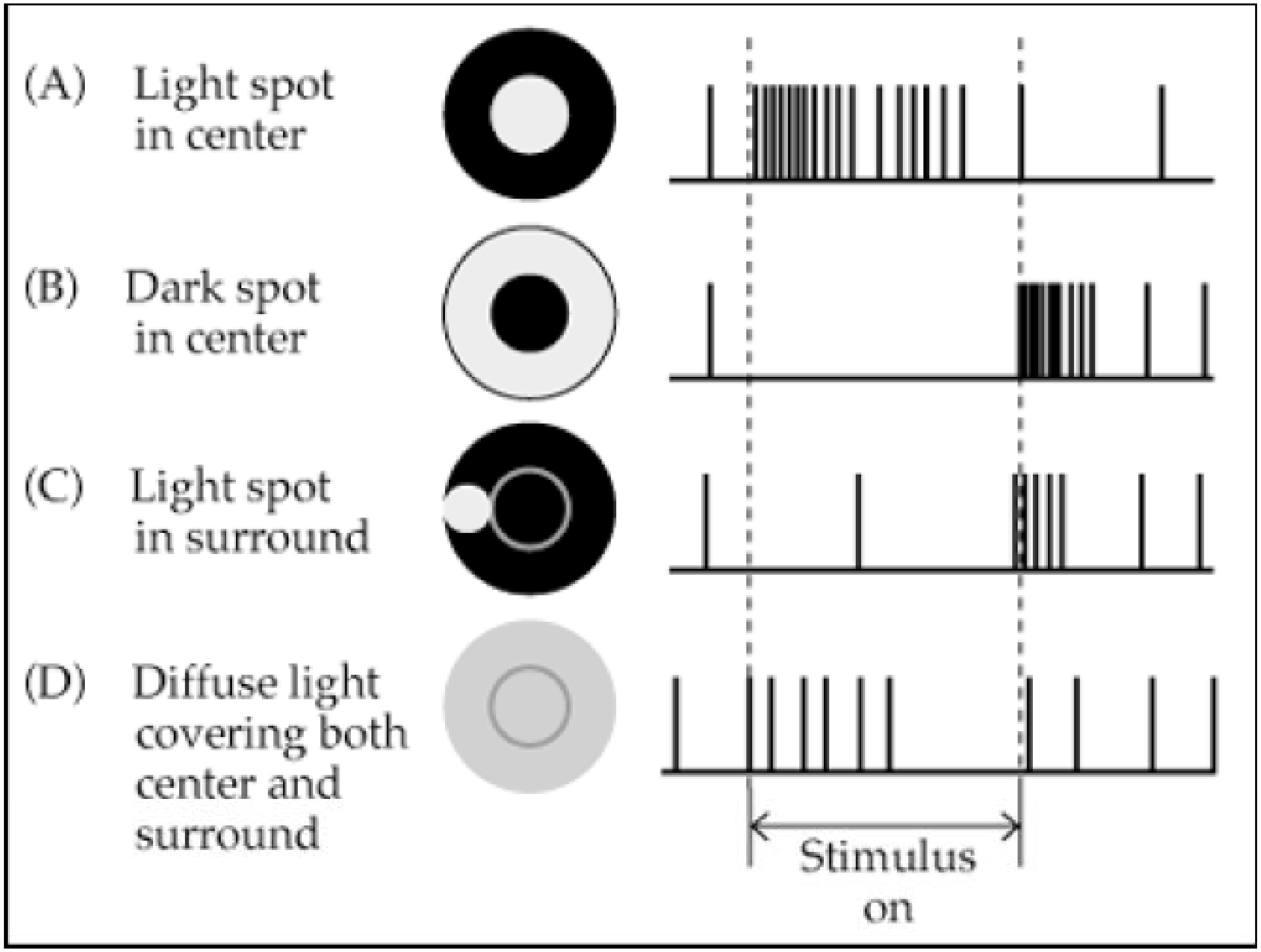
$$\nabla^2 h_\sigma(u,v)$$

- ∇^2 is the **Laplacian** operator:

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

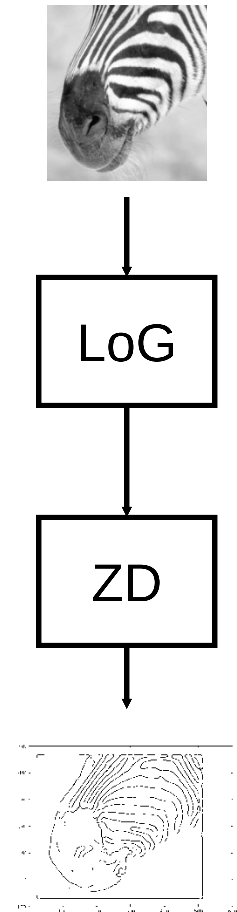
The Laplacian of Gaussian

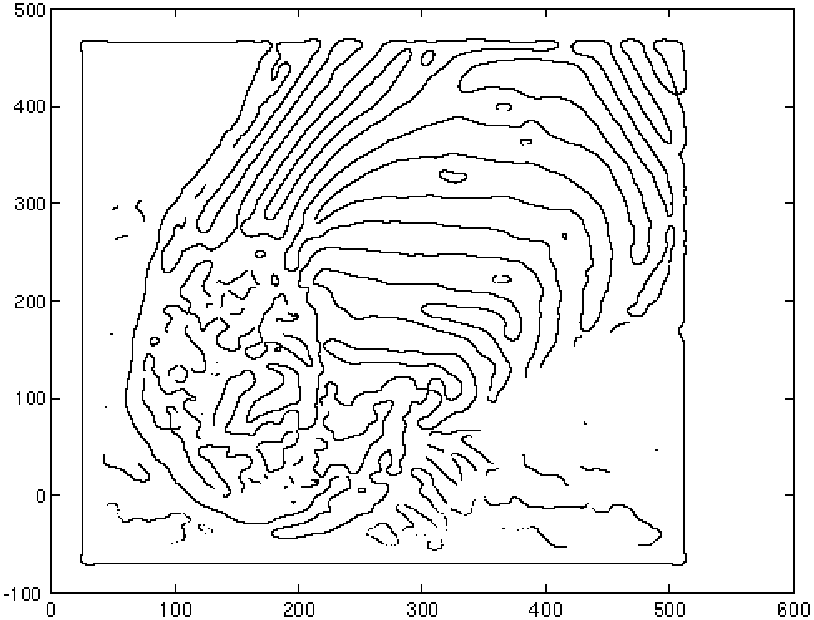
- this is very close to what the early stages of the brain seem to be doing
- recordings of retinal ganglion cells
- called "center-surround" cells
- two types:
  - on-center
  - off-center



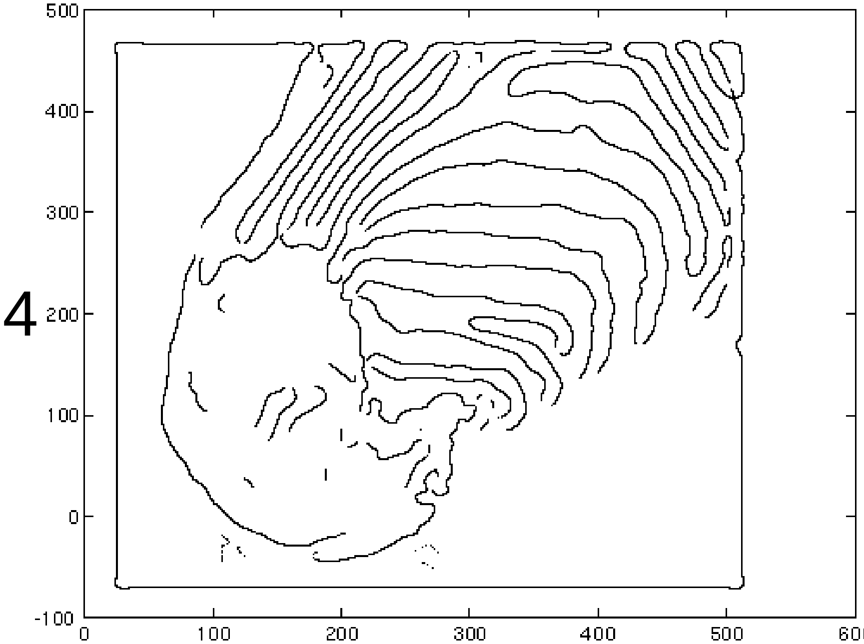
Edge detection strategy

- filter with Laplacian of Gaussian
- detect zero crossings
- mark the zero points where:
  - there is a sufficiently large derivative,
  - and enough contrast
- once again we have parameters
  - scale of Gaussian smoothing
  - thresholds
- once again no set of universal parameters
- does not seem to be better than the strategy of looking for maxima of gradient magnitude.





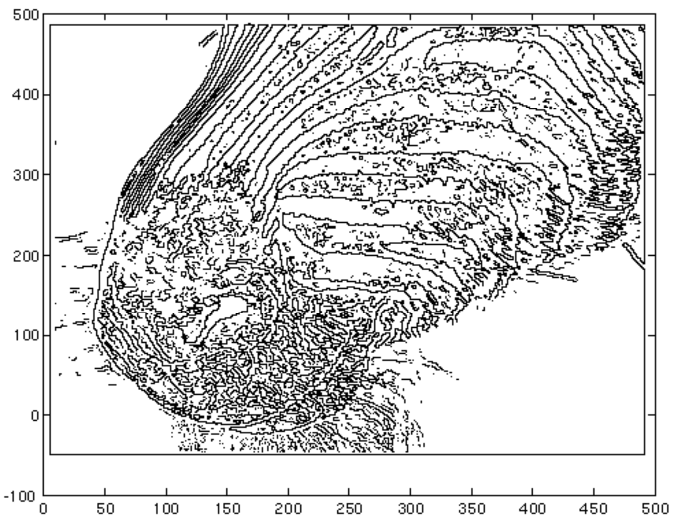
sigma=4



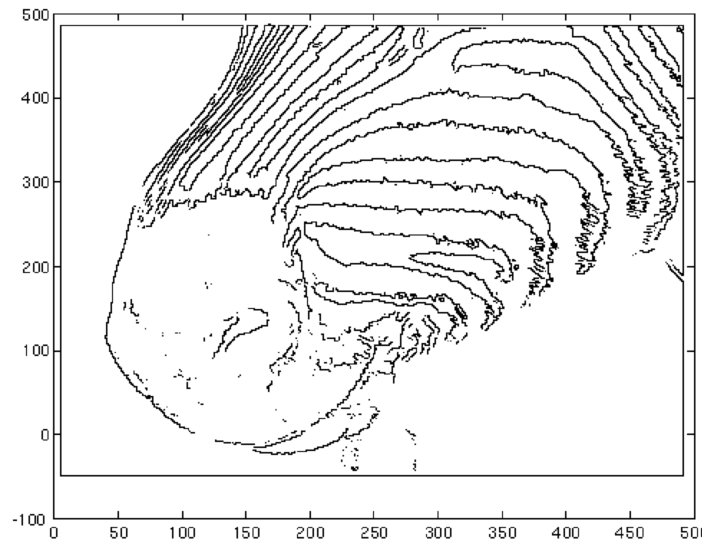
contrast=1

LOG zero crossings

contrast=4

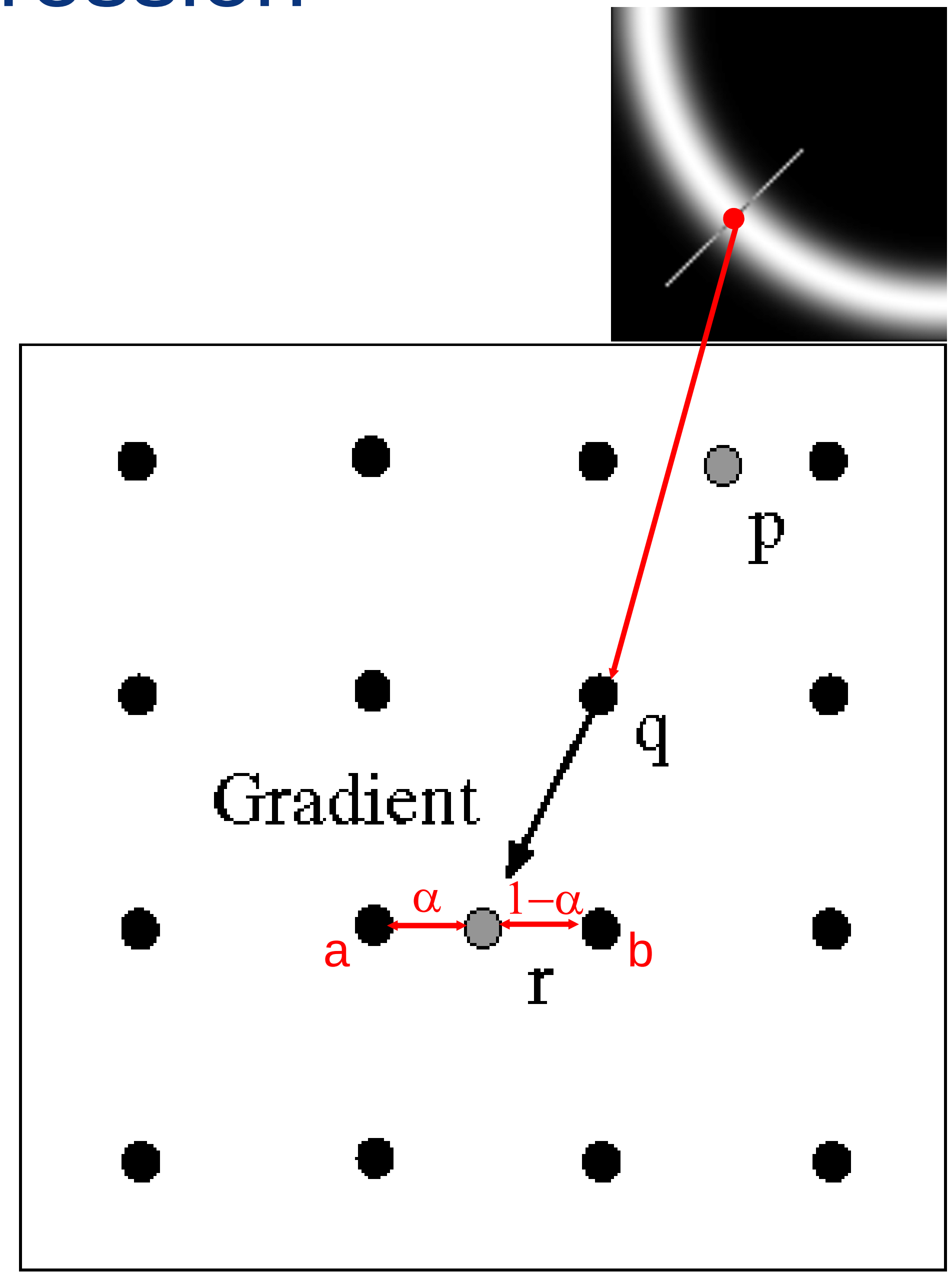


sigma=2



Non-maximum suppression

- we have seen that to find if q is a maximum
- we need to know what is the image value at r
- but this does not fall on the pixel grid
- this is called interpolation
- it is a very frequent operation in image processing



Interpolation

- the most obvious application is to improve the resolution

image

super-resolved

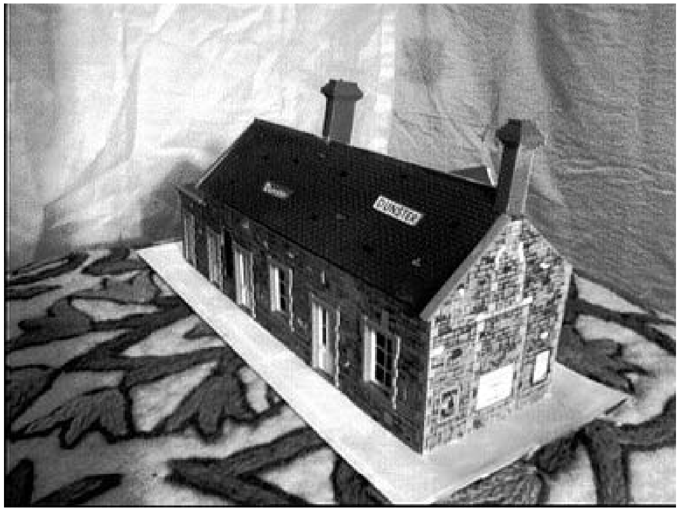
- note the increased detail, e.g. the reduced artifacts on the lines

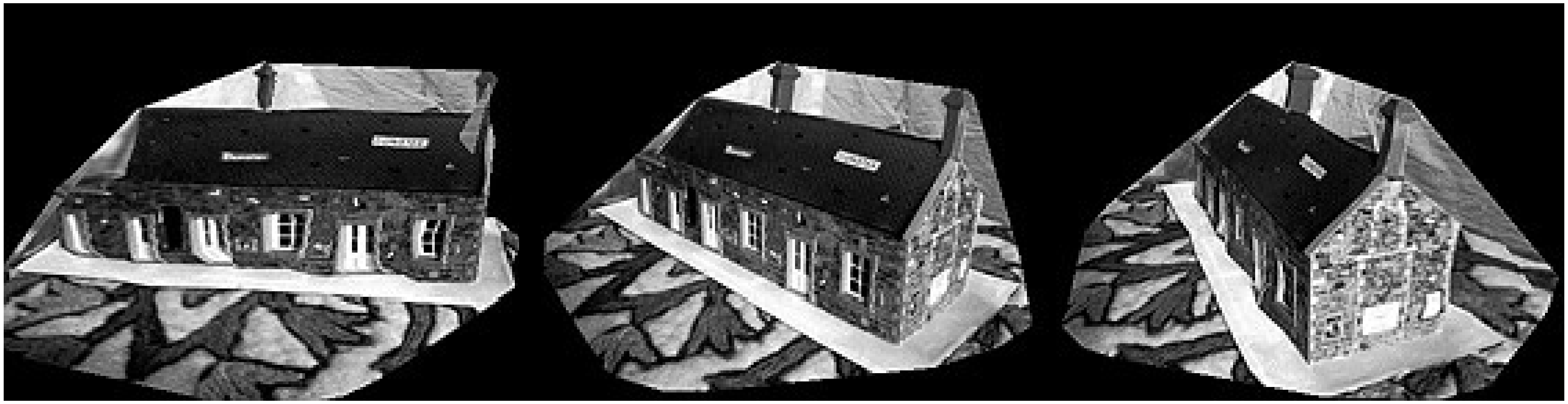
Interpolation

- but there are many others
- e.g. the restoration of degraded movies

Interpolation

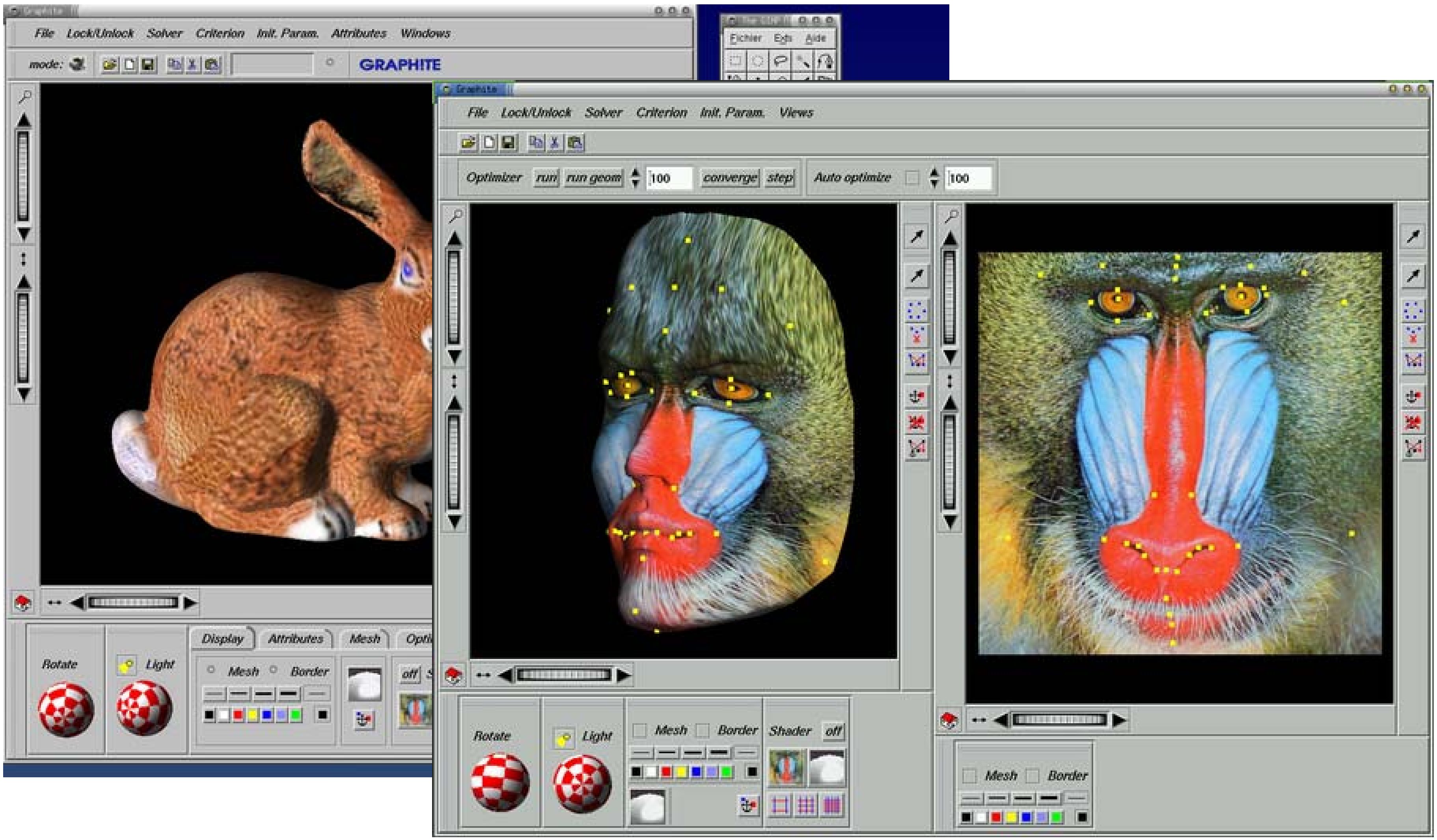
- image synthesis





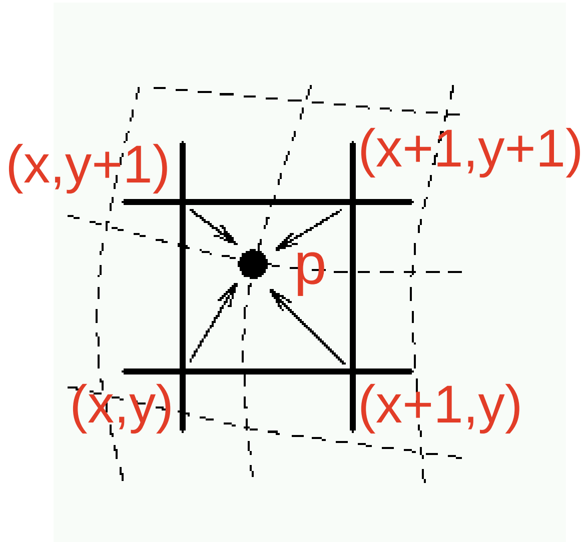
Interpolation

- texture mapping



Interpolation

- how does one do this?
- the simplest method is nearest-neighbor interpolation
- we simply replicate the image intensity (or color) of the closest pixel
- e.g. in this case, because the desired location p is closest to (x,y+1)
- we make



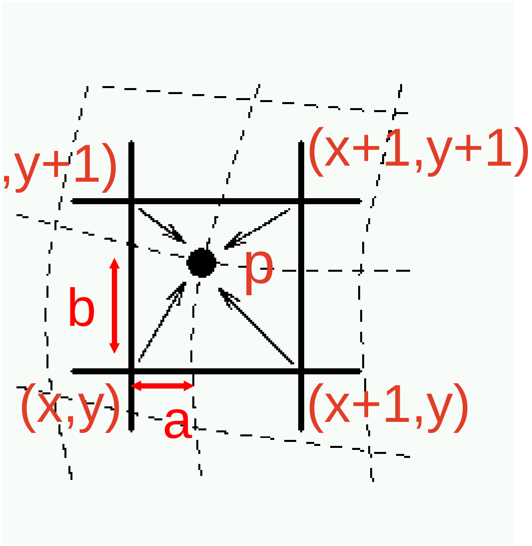
$$\boxed{I(p) = I(x, y+1)}$$

- this is not very good because it generates artifacts
  - one location replicated from one pixel
  - an infinitesimally close neighbor replicated from another

Interpolation

- much better is bilinear interpolation
- assume image varies linearly, weight each pixel according to their distance to p
- let $a = p_x - x$, $b = p_y - y$ and make

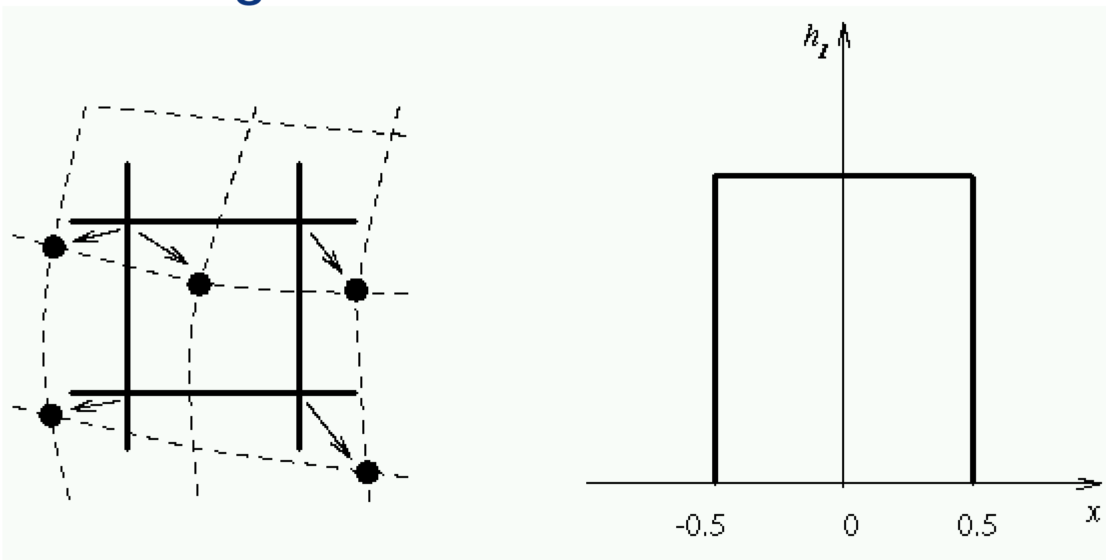
$$\boxed{\begin{aligned} I(p) = {} & (1-a)\times b\times I(x,y+1) \\ & +(1-a)\times(1-b)\times I(x,y) \\ & +a\times(1-b)\times I(x+1,y) \\ & +a\times b\times I(x+1,y+1) \end{aligned}}$$



- works much better than nearest neighbor

Interpolation

- note that these can be implemented with filtering
- for nearest neighbors

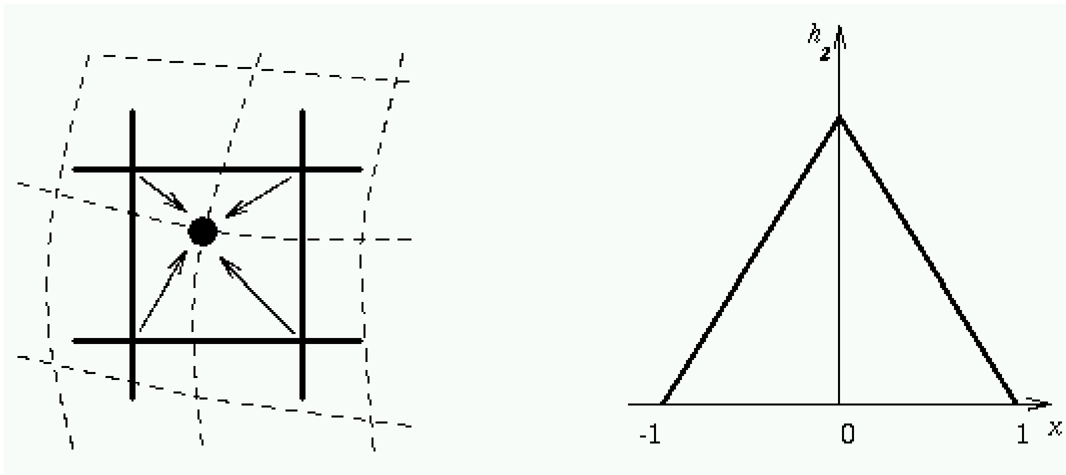


$$\boxed{h_1^1(t) = \begin{cases} 1, & \text{if } t \in [-0.5, 0.5] \\ 0, & \text{otherwise} \end{cases}}$$

$$\boxed{h_1(x,y) = h_1^1(x) h_1^1(y).}$$

Interpolation

- for bilinear interpolation



$$h_2^1(t) = h_1^1 * h_1^1(t) = \begin{cases} 1-t, & \text{if } t \in [0,1] \\ t+1, & \text{if } t \in [-1,0] \\ 0, & \text{otherwise} \end{cases}$$

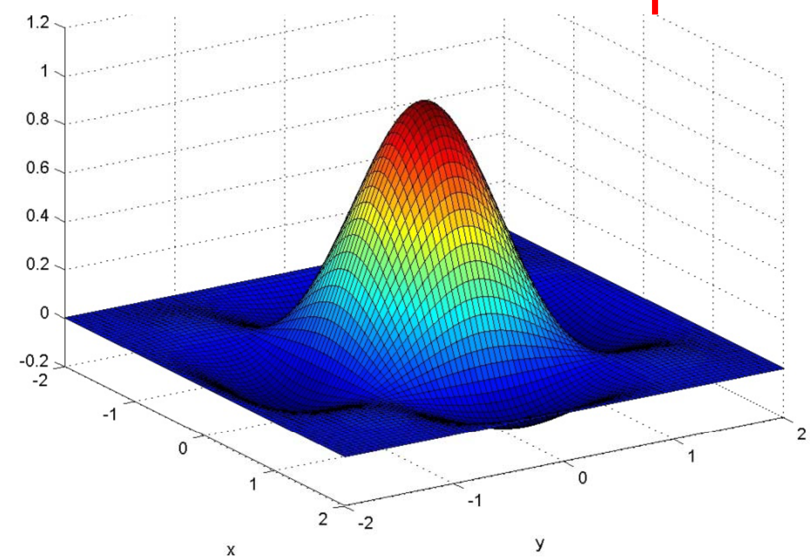
$$h_2(x,y) = h_2^1(x)h_2^1(y)$$

Interpolation

- and there are obviously many other filters
- the best method is frequently bi-cubic interpolation

$$h_3^1(t) = \begin{cases} 1 - 2|t|^2 + |t|^3, & \text{if } |t| < 1 \\ 4 - 8|t| + 5|t|^2 - |t|^3, & \text{if } 1 \leq |t| < 2 \\ 0, & \end{cases}$$

$$h_3(x, y) = h_3^1(x) h_3^1(y).$$



Interpolation

- how do the three methods compare?
- image interpolated with nearest neighbor

Interpolation

- how do the three methods compare?
- image interpolated with bilinear method

Interpolation

- how do the three methods compare?
- image interpolated with bi-cubic method

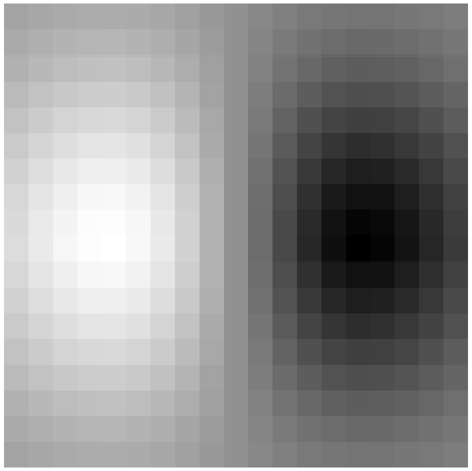
Interpolation

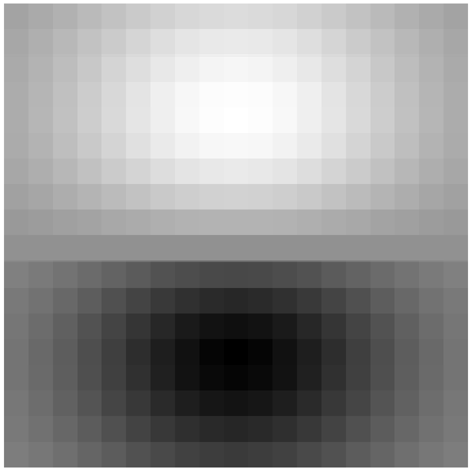
- so, what method should I use?
  - the higher order the filter, the more computation required
  - the gains are diminishing after some point
  - bilinear usually justified over nearest neighbor
  - bi-cubic sometimes worth it, but judge on a case by case basis
  - higher order than cubic is usually not worth it
- to play with this:
  - the matlab interp2 function implements all the methods
  - plus a spline-based method that we will not get into
  - very good applet at

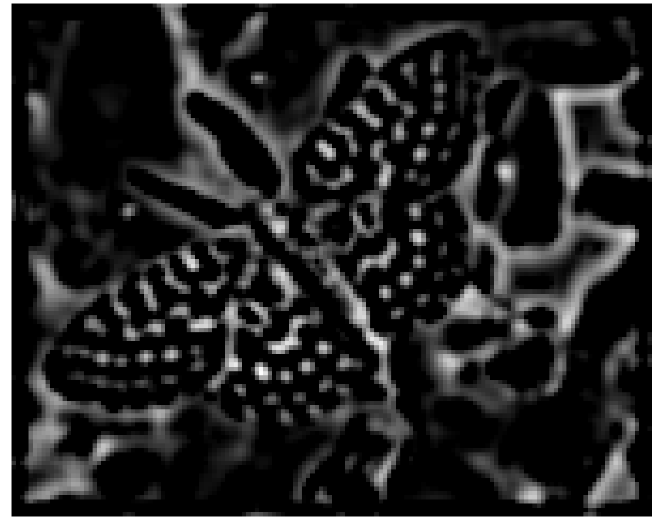
    http://www.s2.chalmers.se/research/image/Java/NewApplets/Interpolation/index.htm

Filters as 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 look like the effects they are intended to find
  - filters find effects they look like







Positive responses

Additional material

The z transform

- once again, it is a straightforward extension of 1D
- **Definition:** the z-transform of the sequence $x[n_1,n_2]$ is

$$\boxed{X(z_1,z_2)=\sum_{n_1}\sum_{n_2}x[n_1,n_2]z_1^{-n_1}z_2^{-n_2}}$$

- the region of the (z_1,z_2) plane where this sum is finite is called the Region of Convergence (ROC)
- it turns out that:
  - in 2D the ROC is much more complicated than in 1D
  - while in 1D the ROC is bounded by poles (0D subspace of the 2D complex plane)
  - in 2D is bounded by pole surfaces (2D subspaces of the 4D space of two complex variables)

The z-transform

- computation is also much harder:
  - as you might remember from 1D
  - most useful tool in computing z-transforms is polynomial factorization
  - z-transform is a ratio of two polynomials

$$\boxed{Y(z) = \frac{N(z)}{D(z)}}$$

  - we factor in to a sum of low order terms, e.g.

$$\boxed{Y(z) = \sum_i \frac{1}{1 - a_i z^{-1}}}$$

  - and then invert each of the terms to get y[n]

z-transform

- in 2D we only have one of two situations
- 1) the sequence is separable, in which case everything reduces to the 1D case

$$\boxed{\begin{aligned} x[n_1,n_2] = x_1[n_1]x_2[n_2] \leftrightarrow X(z_1,z_2) &= X_1(z_1)X_2(z_2) \\ ROC: |z_1| \in ROC \text{ of } X_1(z_1) &\text{ and} \\ |z_2| \in ROC \text{ of } X_2(z_2) & \end{aligned}}$$

  the proof is identical to that of the DSFT

- 2) the signal is not separable
  - here our polynomials are of the form $z_1^m z_2^n$ and, in general, it is not know how to factor them
  - we can solve only if sequence is simple enough that we can do it by inspection (from the definition of the z-transform)

Example

- consider the sequence

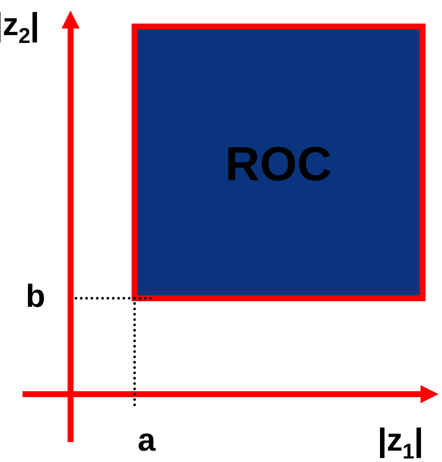
$$x[n_1, n_2] = a^{n_1} b^{n_2} u[n_1, n_2]$$

- the z-transform is

$$X(z_1, z_2) = \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} \left(az_1^{-1}\right)^{n_1} \left(az_2^{-1}\right)^{n_2}$$

$$= \sum_{n_1=0}^{\infty} \left(az_1^{-1}\right)^{n_1} \sum_{n_2=0}^{\infty} \left(az_2^{-1}\right)^{n_2}$$

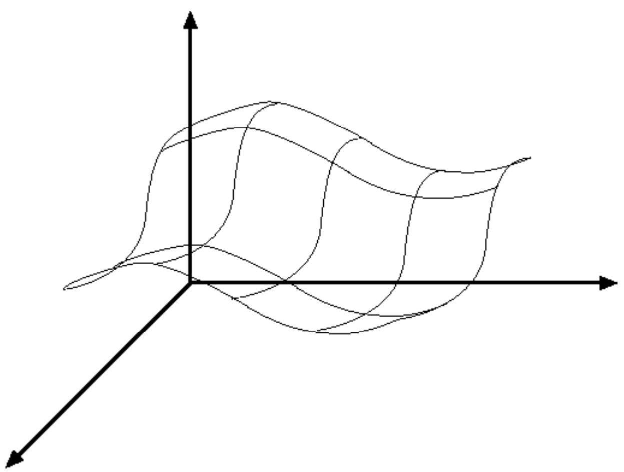
$$= \frac{1}{1 - az_1^{-1}} \frac{1}{1 - bz_2^{-1}}, \quad |z_1| > a, |z_2| > b$$

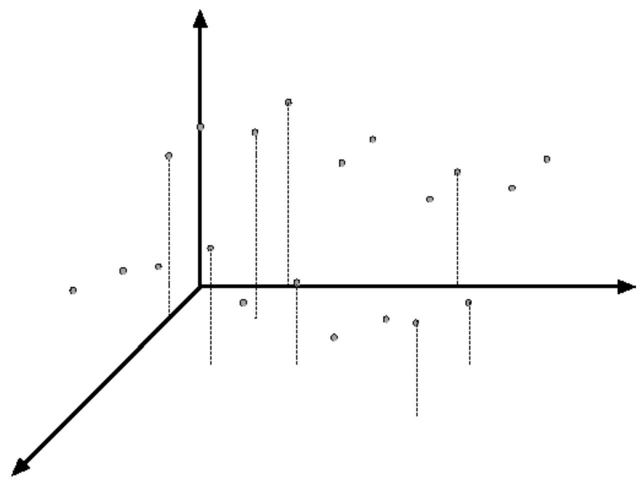


Sampling in 2D

- consider an analog signal $x_c(t_1,t_2)$ and let its analog Fourier transform be $X_c(\Omega_1,\Omega_2)$
  - we use capital Ω to emphasize that this is analog frequency
- sample with period (T_1,T_2) to obtain a discrete-space signal

$$\boxed{x[n_1,n_2] = x_c(t_1,t_2)\Big|_{t_1=n_1T_1;\,t_2=n_2T_2}}$$





Sampling in 2D

- relationship between the Discrete-Space FT of $x[n_1,n_2]$ and the FT of $x_c(t_1,t_2)$ is simple extension of 1D result

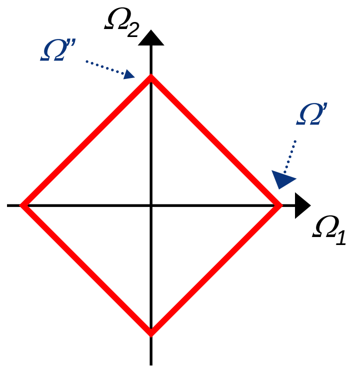
$$\boxed{X(\omega_1,\omega_2)=\frac{1}{T_1T_2}\sum_{r_1=-\infty}^{\infty}\sum_{r_2=-\infty}^{\infty}X_c\left(\frac{\omega_1-2\pi r_1}{T_1},\frac{\omega_1-2\pi r_1}{T_1}\right)}$$

DSFT of $x[n_1,n_2]$
"discrete spectrum"

FT of $x_c(\omega_1,\omega_2)$
"analog spectrum"

- Discrete Space spectrum is sum of replicas of analog spectrum
  - in the "base replica" the analog frequency Ω_1 (Ω_2) is mapped into the digital frequency $\Omega_1 T_1$ ($\Omega_2 T_2$)
  - discrete spectrum has periodicity $(2\pi,2\pi)$

For example



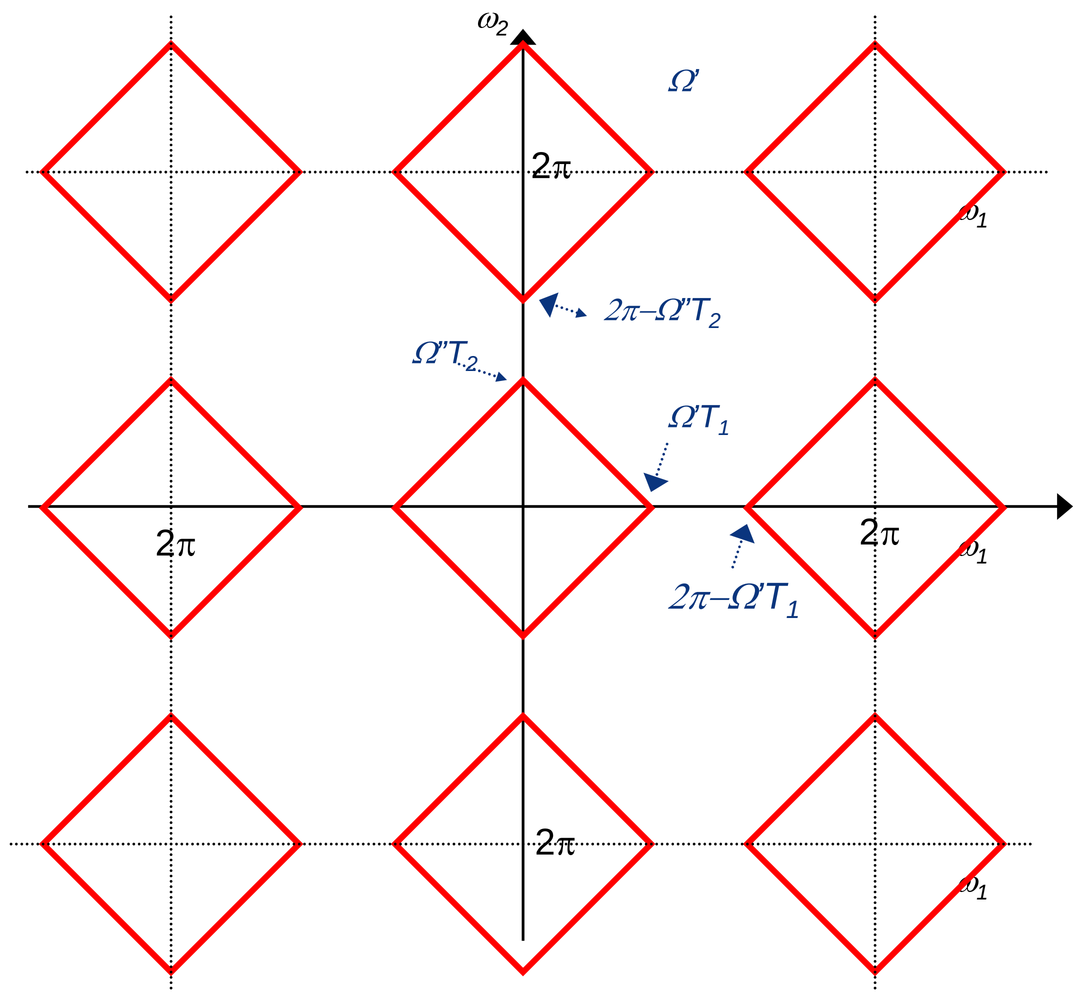
$$\Omega' \rightarrow \alpha = \Omega' T_1$$

$$\Omega'' \rightarrow \beta = \Omega'' T_2$$

- no aliasing if

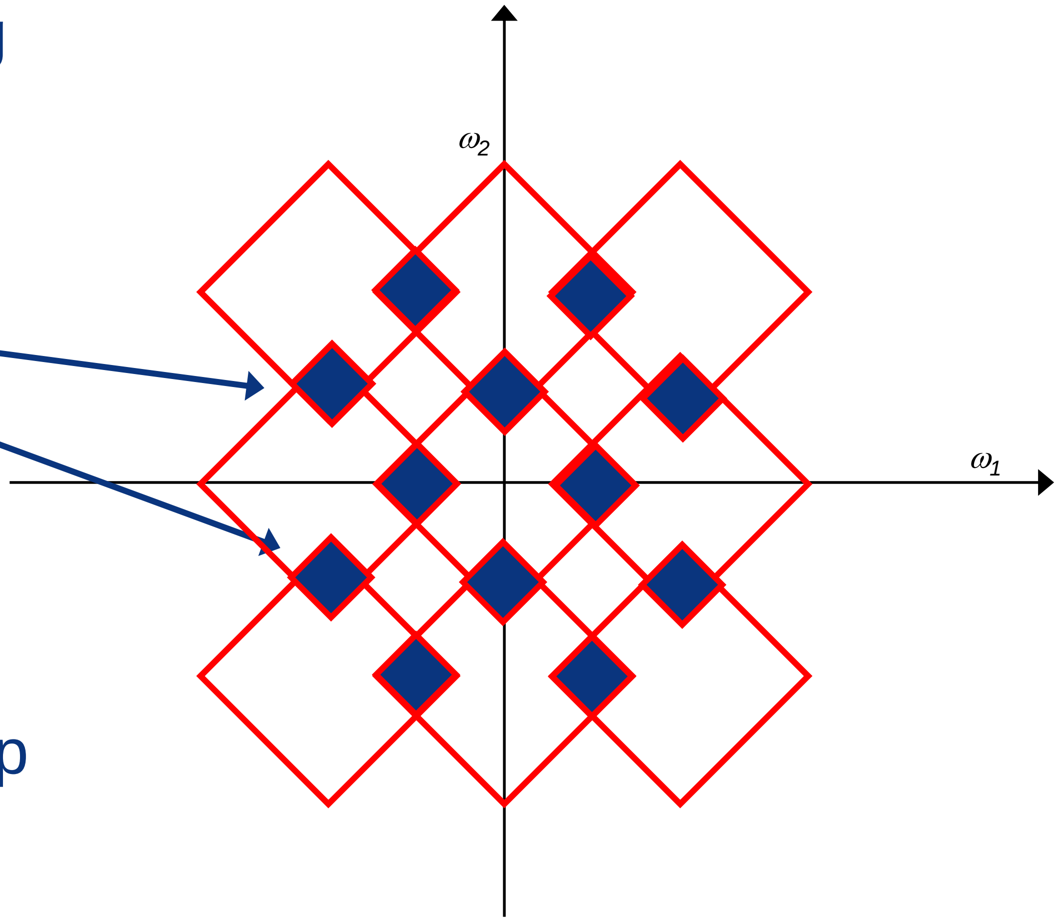
$$\begin{cases} \Omega' T_1 \le 2\pi - \Omega' T_1 \\ \Omega'' T_2 \le 2\pi - \Omega' T_2 \end{cases} \Leftrightarrow$$

$$\Leftrightarrow \boxed{\begin{cases} T_1 \le \pi / \Omega' \\ T_2 \le \pi / \Omega'' \end{cases}}$$



Aliasing

- the frequency $(\Omega'/\pi, \Omega''/\pi)$ is the critical sampling frequency
- below it we have aliasing
- this is just like the 1D case, but now there are more possibilities for overlap



Reconstruction

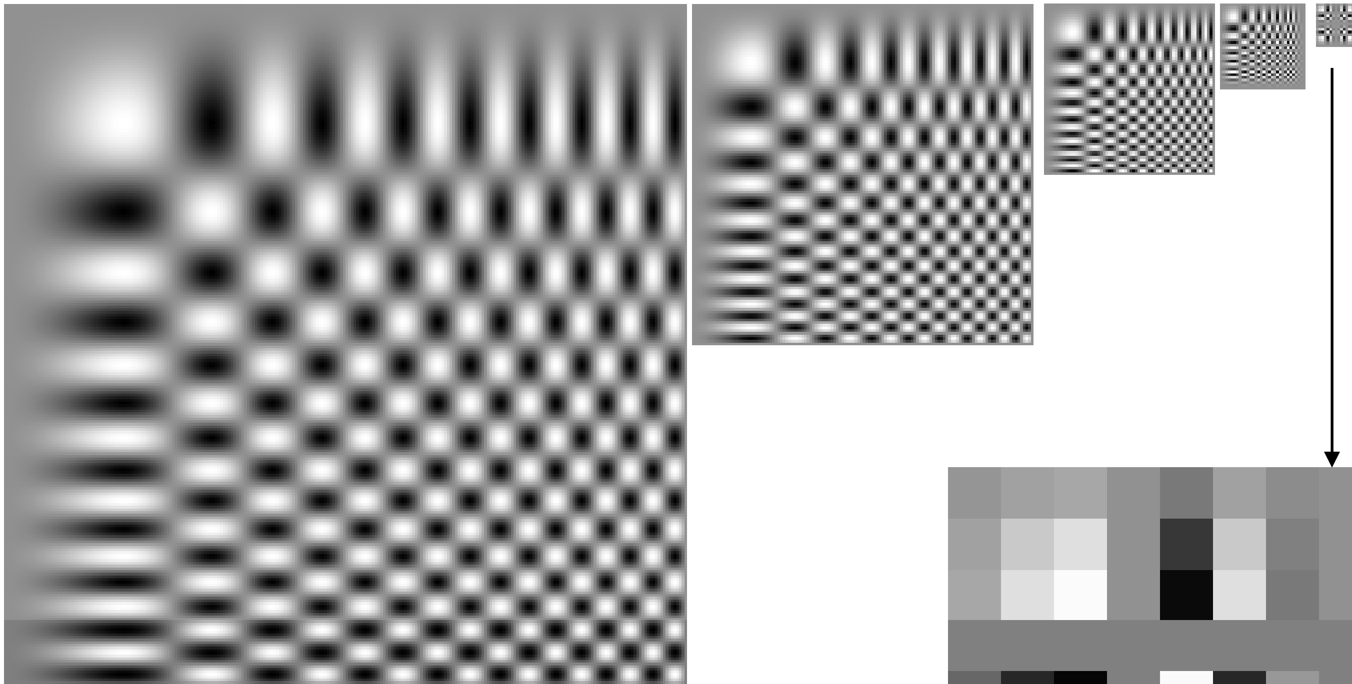
- if there is no aliasing we can recover the signal in a way similar to the 1D case

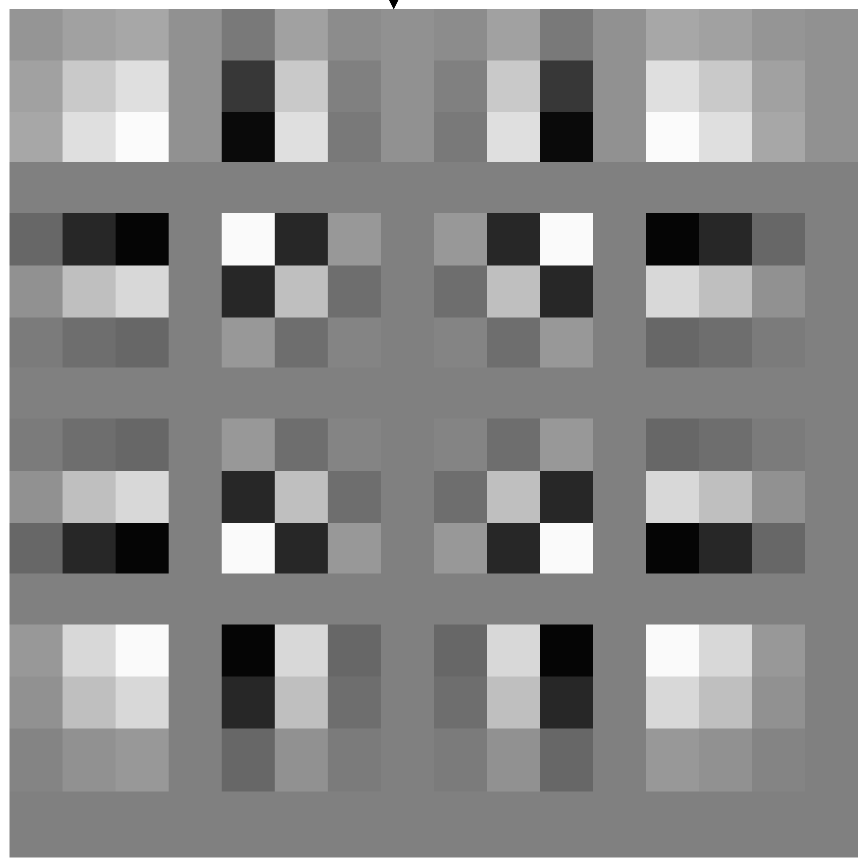
$$y_c(t_1,t_2) = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} x[n_1,n_2] \frac{\sin \frac{\pi}{T_1}(t_1 - n_1 T_1)}{\frac{\pi}{T_1}(t_1 - n_1 T_1)} \frac{\sin \frac{\pi}{T_2}(t_2 - n_2 T_2)}{\frac{\pi}{T_2}(t_2 - n_2 T_2)}$$

- note: in 2D there are many more possibilities than in 1D
  - e.g. the sampling grid does not have to be rectangular, e.g. hexagonal sampling when $T_2 = T_1/sqrt(3)$ and

$$x[n_1,n_2] = \begin{cases} x_c(t_1,t_2)\big|_{t_1=n_1T_1;t_2=n_2T_2} & n_1, n_2 \text{ both even or odd} \\ 0 & \text{otherwise} \end{cases}$$

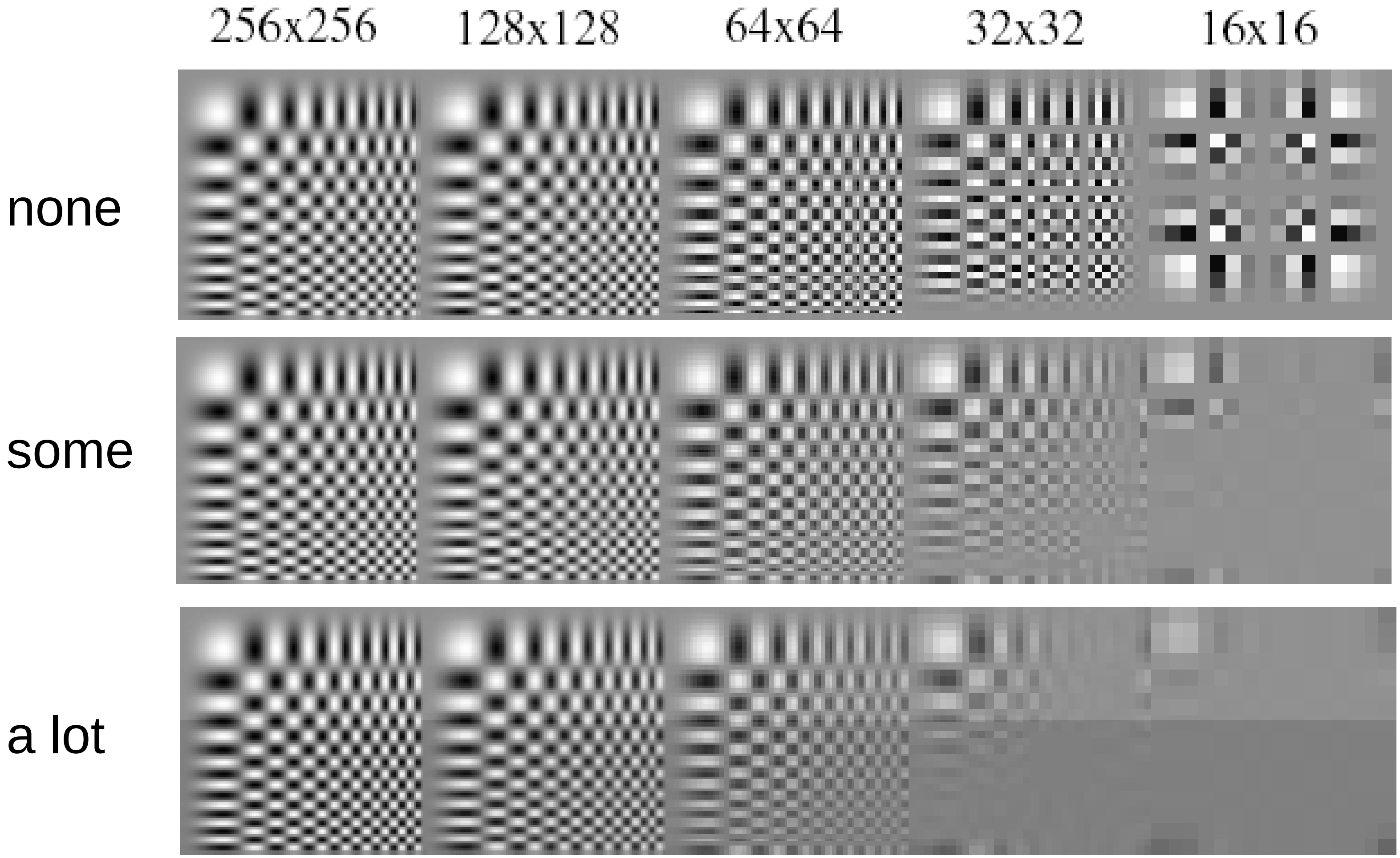
  - in practice, however, one usually adopts the rectangular grid





- a sequence of images obtained by down-sampling without any filtering
- aliasing: the low-frequency parts are replicated throughout the low-res image

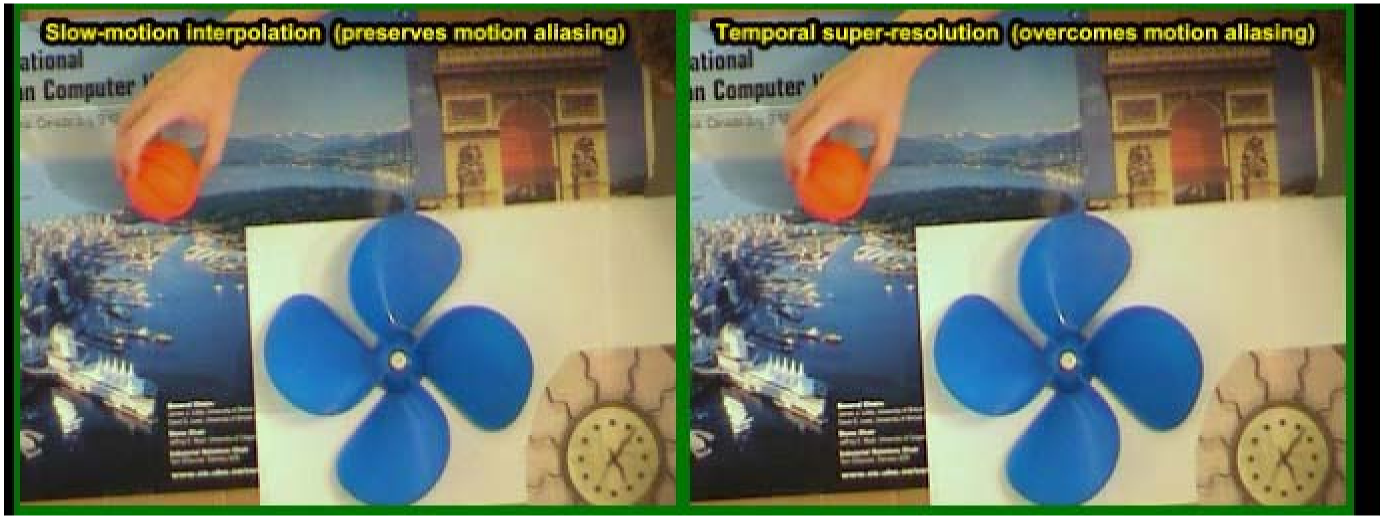
The role of smoothing



- too little leads to aliasing
- too much leads to loss of information

Aliasing in video

- video frames are the result of temporal sampling
  - fast moving objects are above the critical frequency
  - above a certain speed they are aliased and appear to move backwards
  - this was common in old western movies and become known as the "wagon wheel" effect
  - here is an example: super-resolution increases the frame rate and eliminates aliasing



from
"Space-Time Resolution in Video" by E. Shechtman, Y. Caspi and M. Irani

(PAMI 2005).

Any questions?