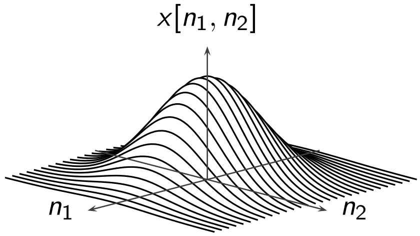
# COM-202: Signal Processing

## Chapter 9: Introduction to Image Processing

## two-dimensional signal processing

## Two-dimensional discrete-space signals

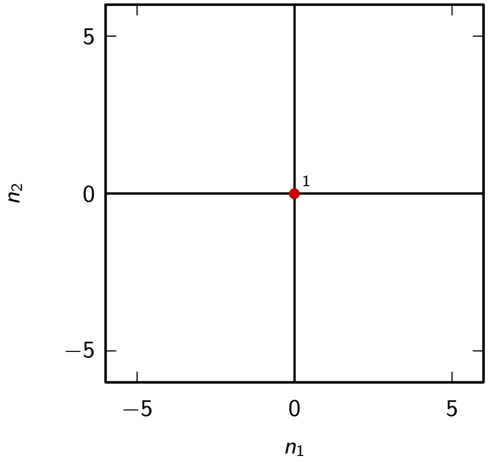
- notation: $x[n_1, n_2]$, $n_1, n_2 \in \mathbb{Z}$
- indexes $n_1, n_2$ locate a point on a grid
- grid is usually regularly spaced



## 2D signals: support representation

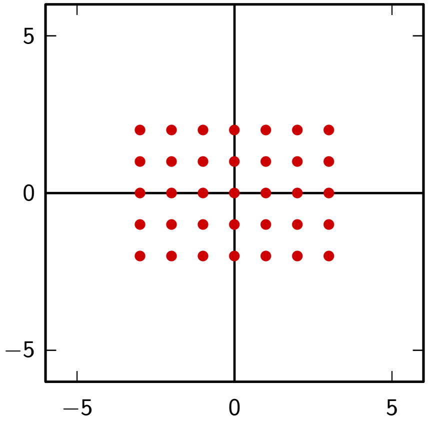
- show coordinates of nonzero values
- values may be written alongside position
- example: the 2D discrete-space impulse:

$$\delta[n_1, n_2] = \begin{cases} 1 & \text{if } n_1 = n_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$



## The 2D discrete-space rect

$$\operatorname{rect}\left(\frac{n_1}{2N_1}, \frac{n_2}{2N_2}\right) = \begin{cases} 1 & \text{if } |n_1| < N_1 \\ & \text{and } |n_2| < N_2 \\ & \\ 0 & \text{otherwise} \end{cases}$$



## A new concept: separability

A separable 2D signal can be decomposed as the product of two 1D signals:

$$x[n_1, n_2] = x_1[n_1]x_2[n_2]$$

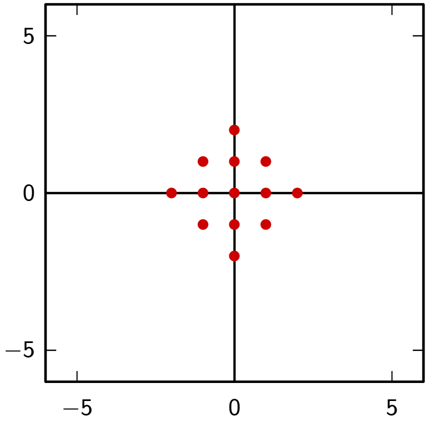
## Examples of separable signals

$$\delta[n_1, n_2] = \delta[n_1]\delta[n_2]$$

$$\mathrm{rect}\left(\frac{n_1}{2N_1}, \frac{n_2}{2N_2}\right) = \mathrm{rect}\left(\frac{n_1}{2N_1}\right)\mathrm{rect}\left(\frac{n_2}{2N_2}\right).$$

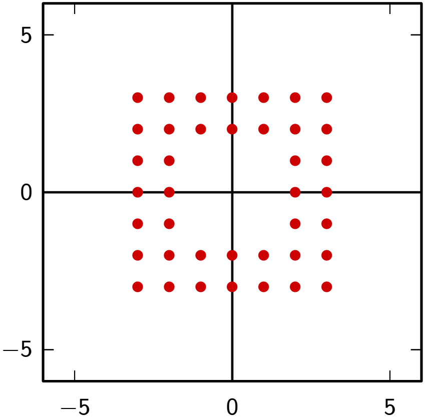
## Nonseparable signal

$$x[n_1, n_2] = \begin{cases} 1 & \text{if } |n_1| + |n_2| < N \\ 0 & \text{otherwise} \end{cases}$$



## Nonseparable signal

$$x[n_1, n_2] = \text{rect}\left(\frac{n_1}{2N_1}, \frac{n_2}{2N_2}\right) - \text{rect}\left(\frac{n_1}{2M_1}, \frac{n_2}{2M_2}\right)$$



## Two-dimensional filters

$$y[n_1, n_2] = \mathcal{H}\{x[n_1, n_2]\}$$

- linearity: $\mathcal{H}\{ax[n_1, n_2] + bw[n_1, n_2]\} = a\mathcal{H}\{x[n_1, n_2]\} + b\mathcal{H}\{w[n_1, n_2]\}$
- space invariance: $\mathcal{H}\{x[n_1 - d_1, n_2 - d_2]\} = y[n_1 - d_1, n_2 - d_2]$
- impulse response $h[n_1, n_2] = \mathcal{H}\{\delta[n_1, n_2]\}$

## Two-dimensional filters

A linear, space-invariant 2D filter implements a 2D convolution:

$$(x * h)[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x[k_1, k_2]h[n_1 - k_1, n_2 - k_2]$$

## 2D convolution for separable signals

$$\text{If } h[n_1, n_2] = h_1[n_1]h_2[n_2]:$$

$$(x * h)[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} h_1[n_1 - k_1] \sum_{k_2=-\infty}^{\infty} x[k_1, k_2]h_2[n_2 - k_2]$$

- each column of $x[n_1, n_2]$ is a 1D signal $x_{n_1}[n]$
- convolve each column $x_{n_1}[n]$ with $h_2[n]$ to obtain the 2D intermediate signal $c[n_1, n_2]$
- each row of $c[n_1, n_2]$ is a 1D signal $c_{n_2}[n]$
- convolve each row $c_{n_2}[n]$ with $h_1[n]$ to obtain the final output

## 2D convolution for separable signals

If $h[n_1, n_2]$ is an $M_1 \times M_2$ finite-support signal:

- non-separable convolution: $M_1 M_2$ operations per output sample
- separable convolution: $M_1 + M_2$ operations per output sample!

## The two-dimensional Discrete Fourier Transform

Straightforward extension of th 1D-DFT to two dimensions:

- an $N_1 \times N_2$ signal $x[n_1, n_2]$ yields $N_1 N_2$ DFT coefficients
- DFT computes the similarity between $x[n_1, n_2]$ and the DFT basis vectors
- the $N_1 N_2$ basis vectors are $N_1 \times N_2$ sinusoidal signals

## 2D-DFT: analysis and synthesis

Analysis formula:

$$X[k_1, k_2] = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x[n_1, n_2] e^{-j \frac{2\pi}{N_1} n_1 k_1} e^{-j \frac{2\pi}{N_2} n_2 k_2}$$

Synthesis formula:

$$x[n_1, n_2] = \frac{1}{N_1 N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} X[k_1, k_2] e^{j \frac{2\pi}{N_1} n_1 k_1} e^{j \frac{2\pi}{N_2} n_2 k_2}$$

## 2D-DFT basis vectors

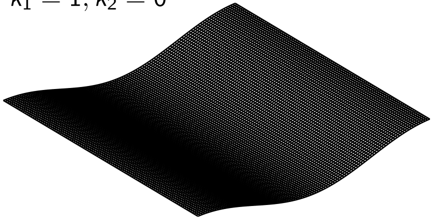
There are $N_1 N_2$ orthogonal basis vectors for an $N_1 \times N_2$ image:

$$w_{k_1,k_2}[n_1, n_2] = e^{j\frac{2\pi}{N_1}n_1 k_1} e^{j\frac{2\pi}{N_2}n_2 k_2}$$

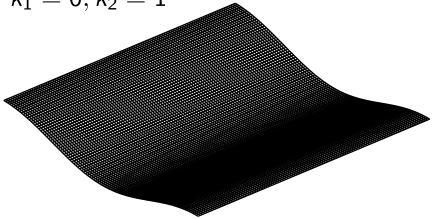
for $n_1, k_1 = 0, 1, \ldots, N_1 - 1$ and $n_2, k_2 = 0, 1, \ldots, N_2 - 1$

## 2D-DFT basis vectors (real part)

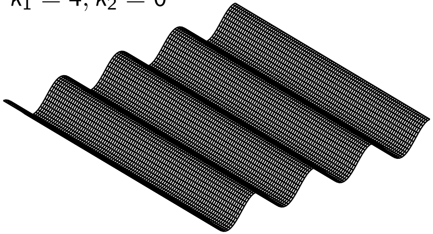
$k_1 = 1, k_2 = 0$



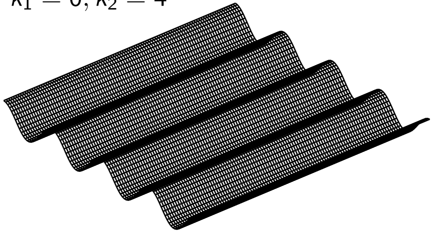
$k_1 = 0, k_2 = 1$



$k_1 = 4, k_2 = 0$

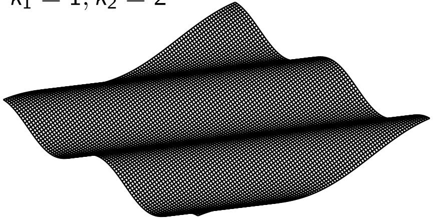


$k_1 = 0, k_2 = 4$

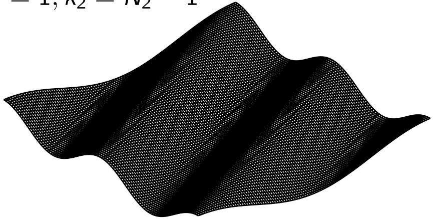


# 2D-DFT basis vectors (real part)

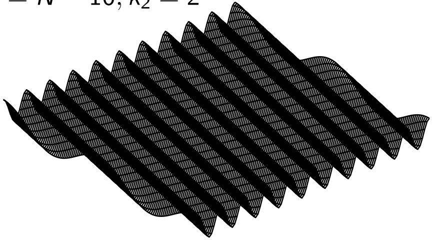
$k_1 = 1, k_2 = 2$



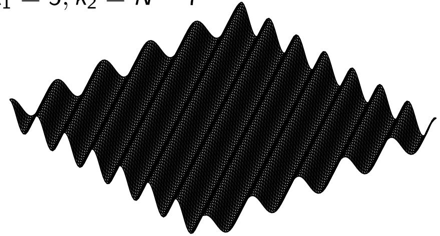
$k_1 = 1, k_2 = N_2 - 1$



$k_1 = N - 10, k_2 = 2$



$k_1 = 5, k_2 = N - 7$



## 2D DFT

2D-DFT basis vectors are separable, and so is the 2D-DFT:

$$X[k_1, k_2] = \boxed{\sum_{n_1=0}^{N_1-1} \left[ \boxed{\sum_{n_2=0}^{N_2-1} x[n_1, n_2] e^{-j\frac{2\pi}{N_2} n_2 k_2}} \right] e^{-j\frac{2\pi}{N_1} n_1 k_1}}$$

- 1D-DFT along $n_2$ (the columns)
- 1D-DFT along $n_1$ (the rows)

## image processing

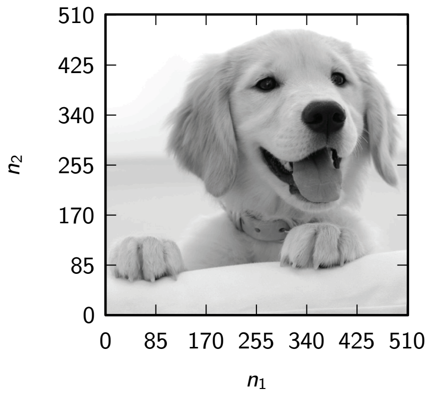
## The fundamental problem with image processing

among all possible 2D signals, images belong to a very small subset,
the set of 2D signals that "make sense" to the human visual processing system

- images are anisotropic: different areas carry extremely different data
- images are 2D projections of a 3D world: lots of information is lost
- images contain a lot of "visual semantics", which is extremely hard to deal with

## Displaying discrete-space signals as images

- coordinates $[n_1, n_2]$ point to a "pixel"
- $x[n_1, n_2]$ is the pixel's grayscale level
- the eye "interpolates" the individual dots into a smooth image (provided the pixel density is high enough)
- limiting factor is medium's dynamic range (how many levels of gray can be displayed)



## About dynamic ranges...

Images:

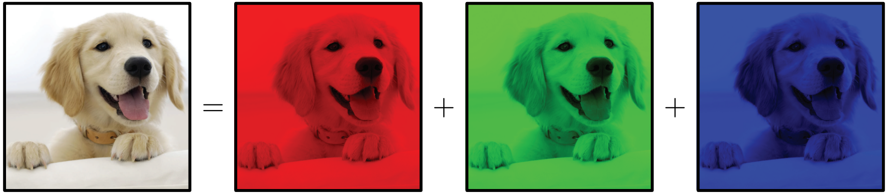
- human eye: 120dB
- prints: 12dB to 36dB
- LCD: 60dB
- digital cinema: 90dB

Sounds:

- human ear: 140dB
- speech: 40dB
- vinyl, tape: 50dB
- CD: 96dB

## Digital images: what about color?

- grayscale images: pixel values are scalars
- color images: pixel values are tuples in a given color space (RGB, HSV, YUV, etc)
- we can consider each color component separately:



## Image processing vs “standard” signal processing

From one to two dimensions…

- something still works
- something breaks down
- something is new

## Image processing: what still works

- Fourier transform formulas
- some key concepts and operators: linearity, *space*-invariance, convolution
- interpolation, sampling, quantization

## Image processing: what no longer works

- Fourier analysis is much less relevant
- filter design is hard, and IIRs are rare to non-existent
- only simple filters are useful because LTI filters are isotropic but images are not

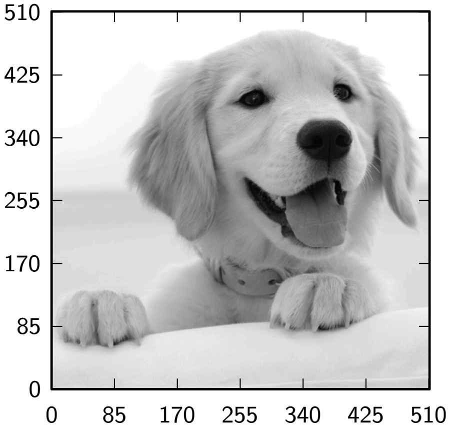
## Image processing: new concepts

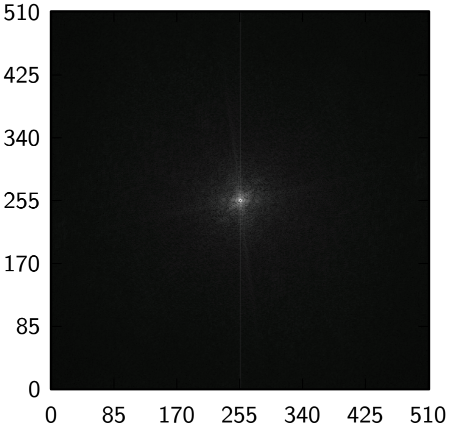
What's new:

- new class of signal operators: affine transforms
- images are finite-support and available in their entirety $\rightarrow$ filter causality becomes irrelevant
- images use up a lot of storage: we need compression

## images in the frequency domain

## How does a 2D-DFT look like?





## DFT of an image: magnitude vs phase

$$X[k_1, k_2] = \mathrm{DFT}\{x[n_1, n_2]\}$$

$$X[k_1, k_2] = A[k_1, k_2]e^{j\varphi[k_1,k_2]}, \quad A[k_1, k_2] \geq 0$$

- what happens if we discard the phase?

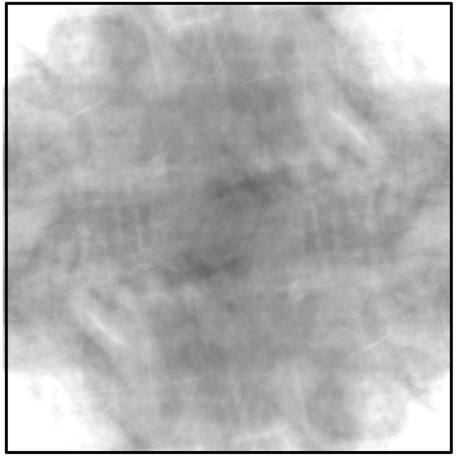
$$x_m[n_1, n_2] = \mathrm{IDFT}\{A[k_1, k_2]\}$$

- what happens if we discard the magnitude?

$$x_p[n_1, n_2] = \mathrm{IDFT}\left\{e^{j\varphi[k_1,k_2]}\right\}$$

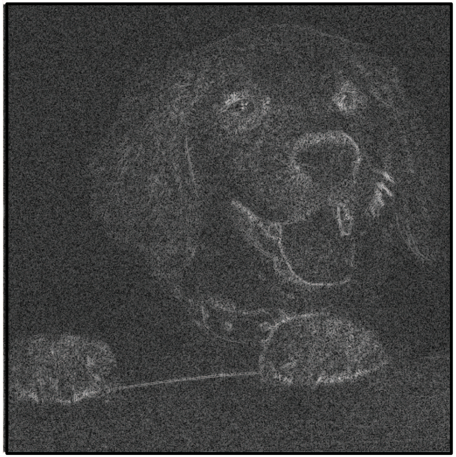
## DFT magnitude doesn't carry much visual information

$x_m[n_1, n_2]$



## DFT phase, on the other hand...

$x_p[n_1, n_2]$



## Images in the frequency domain

- key visual semantic information determined by *edges*
- edges are points of rapid amplitude change in space
- to create a “jump” we need precise phase alignment of basis vectors

in one dimension, the Fourier transform is a visual analysis tool...
...but images are already visual objects!

## filters for image processing

## Everyday 2D filter types

- you'll see mostly linear-phase FIRs (to preserve edges)
- zero-centered, odd-length impulse response to avoid shifts
- separable filters preferred for efficiency
- lowpass response for image smoothing (blurring)
- highpass response for edge detection

## 2D Moving Average

$$y[n_1, n_2] = \frac{1}{(2M+1)^2} \sum_{k_1=-M}^{M} \sum_{k_2=-M}^{M} x[n_1 - k_1, n_2 - k_2]$$

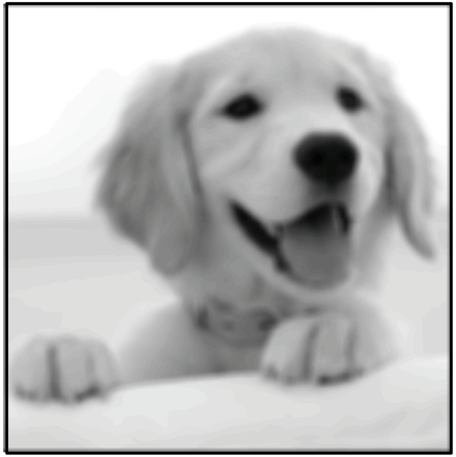
$$h[n_1, n_2] = \frac{1}{(2M+1)^2} \text{rect}\left(\frac{n_1}{2M}, \frac{n_2}{2M}\right)$$

## Example: 3x3 Moving Average

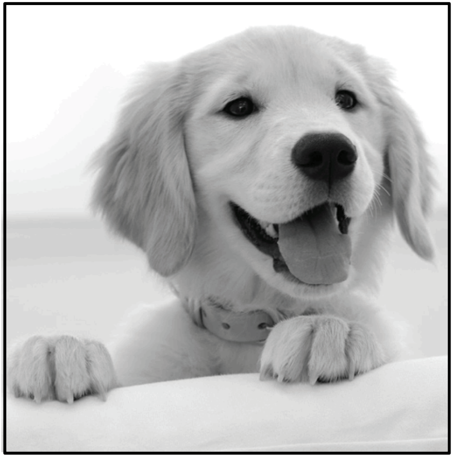
$$h[n_1, n_2] = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

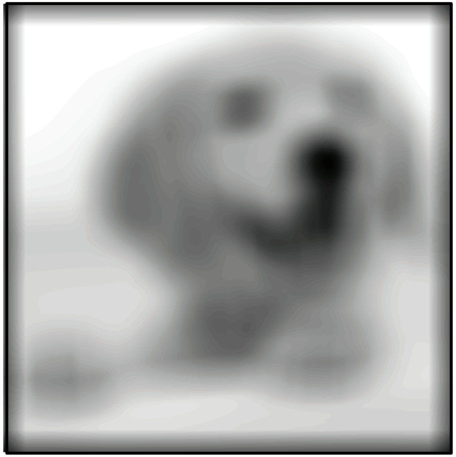
## 2D Moving Average

original


$11 \times 11$ MA

## 2D Moving Average


original


$51 \times 51$ MA

## Border effects

$N \times N$ image filtered with an $(2M+1) \times (2M+1)$ FIR:

$$(h * x)[n_1, n_2] = \sum_{k_1=-M}^{M} \sum_{k_2=-M}^{M} h[k_1, k_2] x[n_1 - k_1, n_2 - k_2]$$

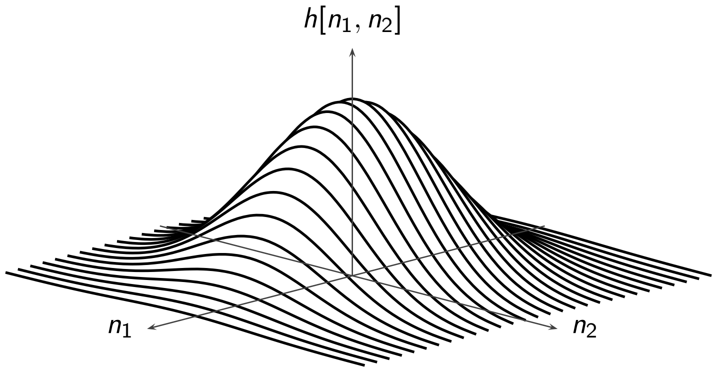
- when $n_i < M$ or $n_i > N - M$, convolution uses pixels outside of the $N \times N$ image
- normally we assume $x[n_1, n_2] = 0$ for $n_i \notin [0, N-1]$: this creates a dark band around the edges
- in 1D $2M$ samples would exhibit border effects
- in 2D $4M(N-M)$ pixels are affected!
- cosmetic solution is to mirror the image at each edge

## Gaussian Blur

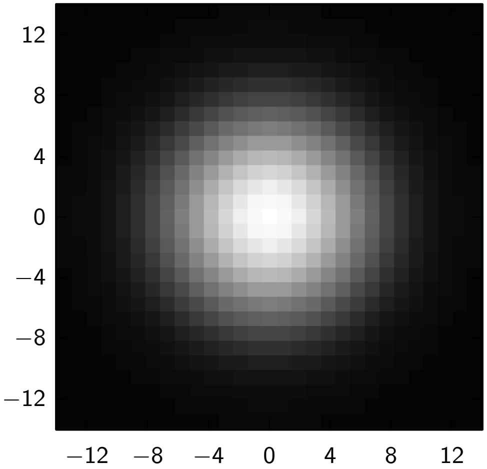
$$h[n_1, n_2] = \frac{1}{2\pi\sigma^2} e^{-\frac{n_1^2+n_2^2}{2\sigma^2}}, \qquad |n_1, n_2| < M$$

with $M \approx 3\sigma$

## Gaussian Blur



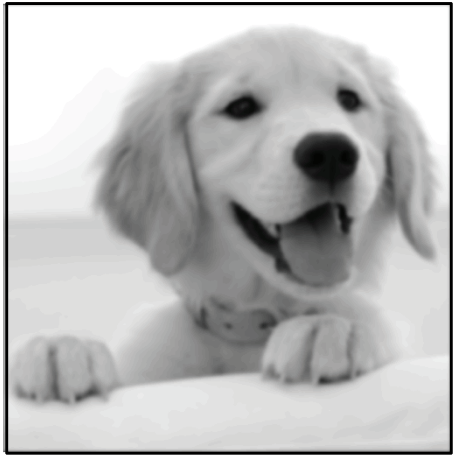
## Gaussian Blur



$\sigma = 5, M = 14$

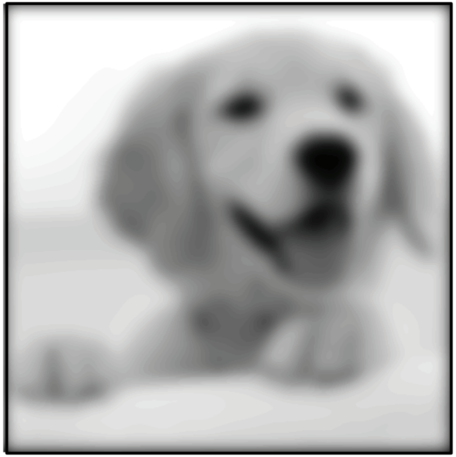
## Gaussian Blur

original

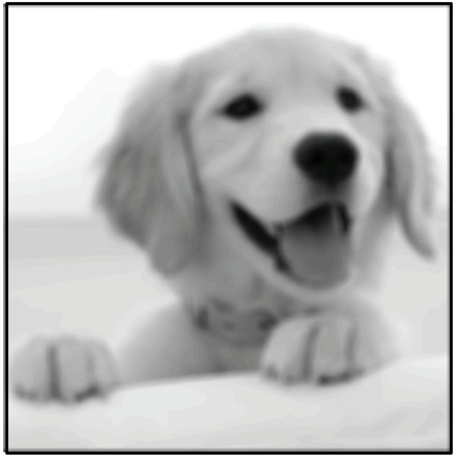

$\sigma = 1.8, 11 \times 11$ blur

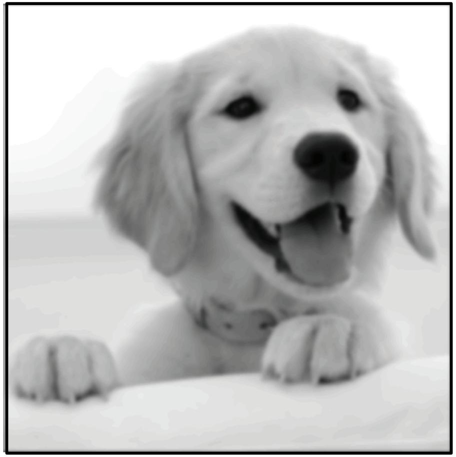
## Gaussian Blur

original

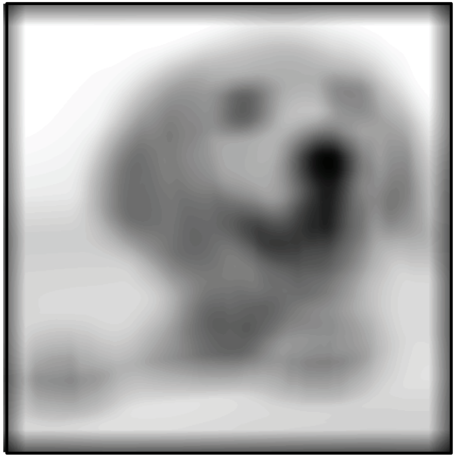

$\sigma = 8.7, 51 \times 51$ blur

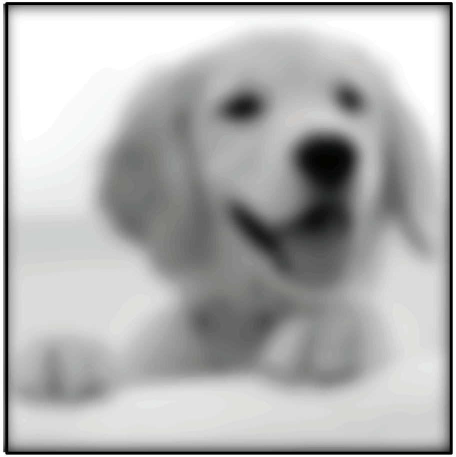
## Gaussian blur more “photographic” than moving average


$11 \times 11$ MA


$\sigma = 1.8, 11 \times 11$ blur

## Gaussian blur more “photographic” than moving average


$51 \times 51$ MA


$\sigma = 8.7, 51 \times 51$ blur

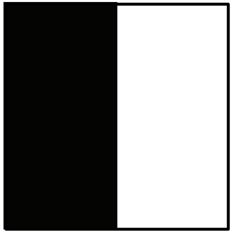
## edge detection

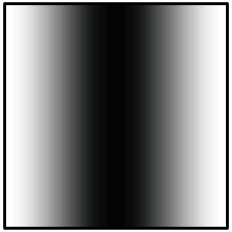
## Filters for edge detection

What is an edge? In general very complicated since dependent on semantic scene analysis.

Candidate edge points are found where the signal's amplitude

- has a discontinuity
- has an inflection





## Edge detection

Edge detection algorithms work in 2 passes:

1. selection of candidate edge points via a filtering operation
2. refinement of edges using geometric consideration, heuristics, and machine learning

- the second step is highly nonlinear and quite complicated
- we will look at the first phase only
- candidate points are located by estimating the derivatives of the signal

## Gradient and Laplacian

for points of discontinuity we look for large values of the gradient

$$\nabla f(t_1, t_2) = \begin{bmatrix} \frac{\partial f}{\partial t_1} & \frac{\partial f}{\partial t_2} \end{bmatrix}$$

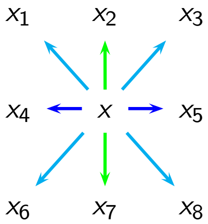
for points of inflection we look for the zeros of the Laplacian

$$\Delta f(t_1, t_2) = \frac{\partial^2 f}{\partial t_1^2} + \frac{\partial^2 f}{\partial t_2^2}$$

## Gradient and Laplacian for discrete images

- the gradient is approximated using the Sobel filter
- the Laplacian is approximated by the Laplacian discrete-space operator
- both are implemented using simple separable filters

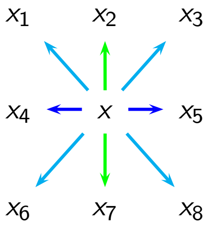
## Approximating the gradient with first differences



- derivatives on the discrete grid are approximated by first-order differences
- use also diagonal differences with appropriate scaling:

$$p_i = \frac{x_i - x}{d_i}, \qquad d_i = \begin{cases} 1 & i = 2,4,5,7 \\ \sqrt{2} & i = 1,3,6,8 \end{cases}$$

## Approximating the gradient with first differences



- combine differences with proper sign (left to right and down to up)
- diagonal first differences contribute to both horizontal and vertical derivatives:

$$\frac{\partial x}{\partial n_1} \approx p_5 - p_4 - p_1 + p_3 - p_6 + p_8$$

$$\frac{\partial x}{\partial n_2} \approx p_2 - p_7 + p_1 + p_3 - p_6 - p_8$$

## Approximating the gradient with first differences

to make things computationally easy, set

$$d_i = \begin{cases} 1/2 & i = 2,4,5,7 \\ 1 & i = 1,3,6,8 \end{cases}$$

$$\frac{\partial x}{\partial n_1} \approx (x_3 - x_1) + 2(x_5 - x_4) + (x_8 - x_6)$$
$$\frac{\partial x}{\partial n_2} \approx (x_1 - x_6) + 2(x_2 - x_7) + (x_3 - x_8)$$

## The Sobel filter

$$\frac{\partial x[n_1, n_2]}{\partial n_1} \approx (s_h * x)[n_1, n_2]$$
$$\frac{\partial x[n_1, n_2]}{\partial n_2} \approx (s_v * x)[n_1, n_2]$$

$$s_h[n_1, n_2] = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$s_v[n_1, n_2] = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

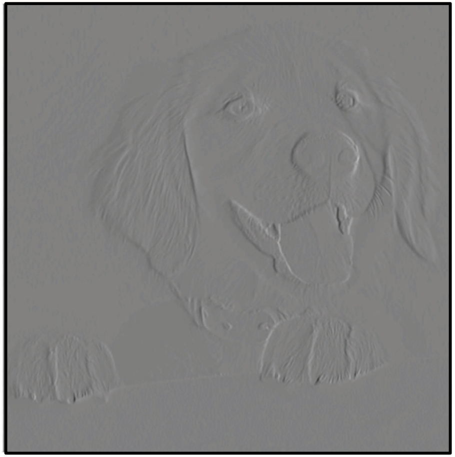
## The Sobel filter

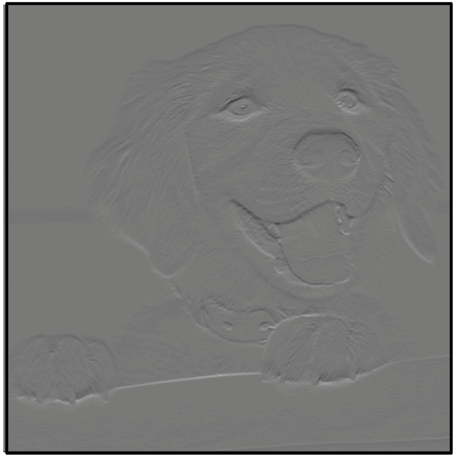
the Sobel filter is separable:

$$s_h[n_1, n_2] = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

interpretation: horizontal gradient = vertical averaging before horizontal differentiation

## The Sobel filter


horizontal Sobel filter


vertical Sobel filter

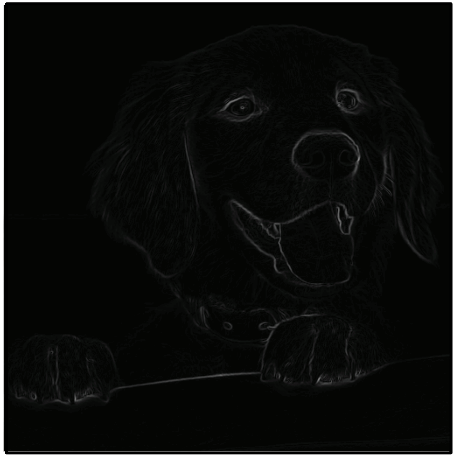
## The Sobel operator

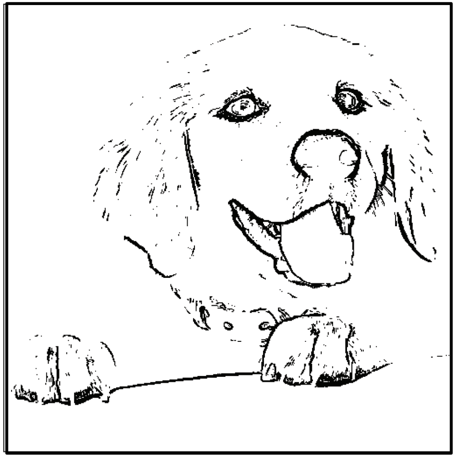
to find all extrema, use the square magnitude of the gradient:

$$|\nabla x[n_1, n_2]|^2 = |(s_h * x)[n_1, n_2]|^2 + |(s_v * x)[n_1, n_2]|^2$$

(we call this an “operator” instead of a filter because it is nonlinear)

## Edge detection with the Sobel operator


output of Sobel operator


thresholded output

## The Laplacian operator

The Laplacian of a bivariate function is defined as

$$\Delta f(t_1, t_2) = \frac{\partial^2 f}{\partial t_1^2} + \frac{\partial^2 f}{\partial t_2^2}$$

zero crossings of the Laplacian are potential inflection points

## Approximating the second derivative

for a smooth function $f$, the first three terms of the Taylor expansion in $(t+\tau)$ and $(t-\tau)$

$$f(t+\tau) = \sum_{n=0}^{\infty} \frac{f^{(n)}(t)}{n!}\,\tau^n$$

$$f(t+\tau) \approx f(t) + f'(t)\tau + \frac{1}{2}f''(t)\tau^2$$

$$f(t-\tau) \approx f(t) - f'(t)\tau + \frac{1}{2}f''(t)\tau^2$$

so that

$$f''(t) \approx \frac{1}{\tau^2}(f(t-\tau) - 2f(t) + f(t+\tau))$$

## Approximating the second derivative

$$f''(t) \approx \frac{1}{\tau^2}(f(t-\tau) - 2f(t) + f(t+\tau))$$

on the discrete grid $(t \leftarrow n,\ \tau = 1)$:

$$\begin{aligned}\frac{\partial^2 x[n]}{\partial n^2} &\approx x[n-1] - 2x[n] + x[n+1] \\ &= (h * x)[n] \\ h[n] &= \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}\end{aligned}$$

## The Laplacian filter

for images, there are two alternate definitions

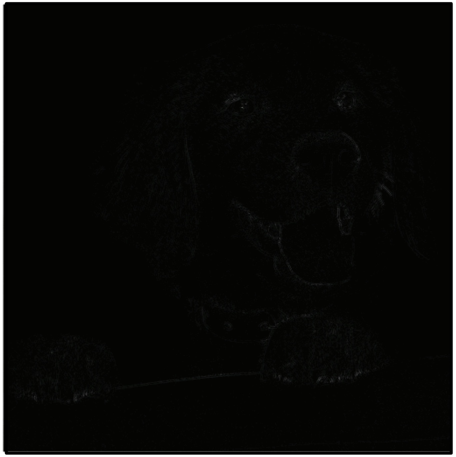
- using only the horizontal and vertical derivatives:

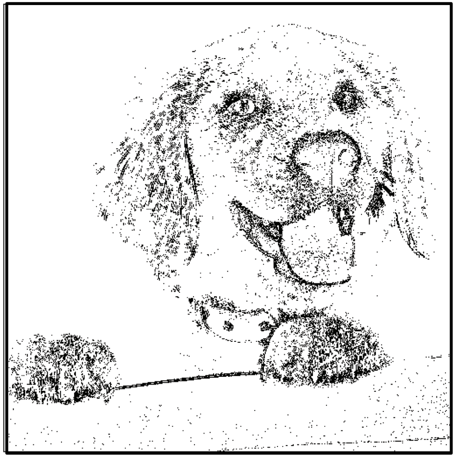
$$h[n_1, n_2] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- using also the diagonals

$$h[n_1, n_2] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

## Edge detection with the Laplacian filter


output of Laplacian filter


thresholded output

## affine transforms

## Affine transforms in continuous space

mapping $\mathbb{R}^2 \to \mathbb{R}^2$ that reshapes the coordinate system:

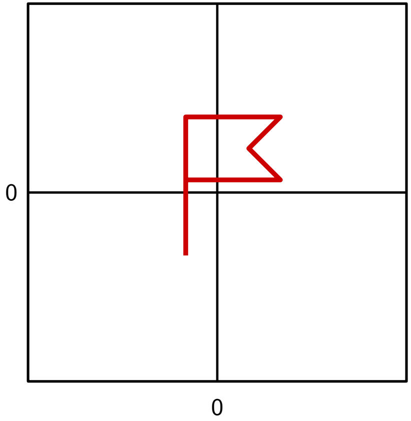
$$\begin{bmatrix} t'_1 \\ t'_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} - \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

$$\begin{bmatrix} t'_1 \\ t'_2 \end{bmatrix} = \mathbf{A} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} - \mathbf{d}$$

## Translation

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}$$

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

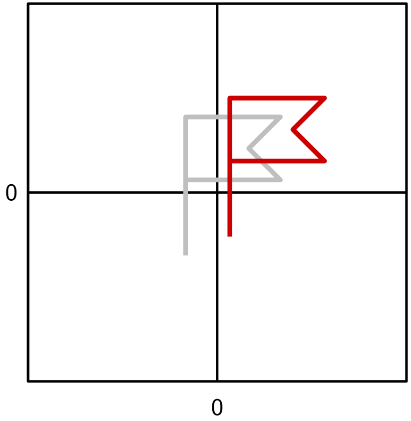


## Translation

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}$$

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

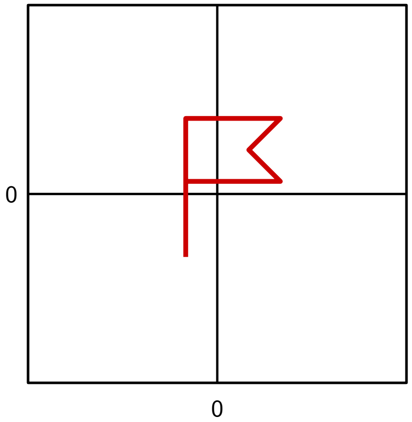
$d_1 = -0.7, d_2 = -0.3$



## Scaling

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix}$$

$$\mathbf{d} = 0$$



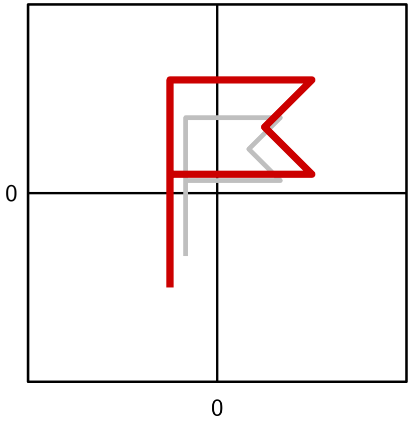
## Scaling

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix}$$

$$\mathbf{d} = 0$$

if $a_1 = a_2$ the *aspect ratio* is preserved

$a_1 = a_2 = 1.5$

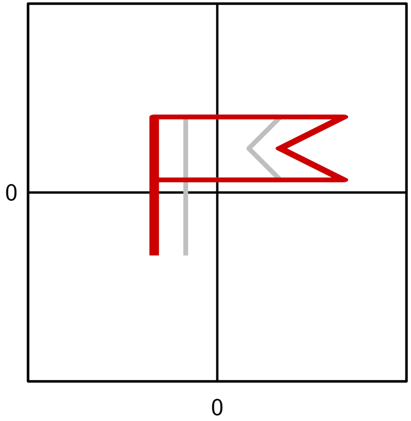


## Scaling

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix}$$

$$\mathbf{d} = 0$$

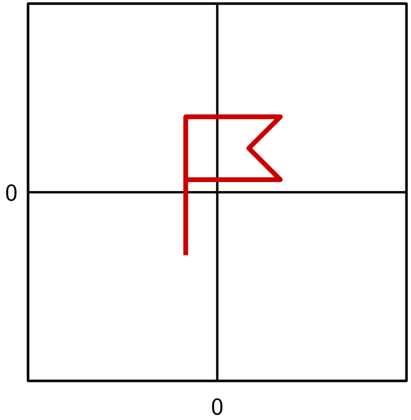
$a_1 = 2, a_2 = 1$



## Rotation

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{d} = 0$$

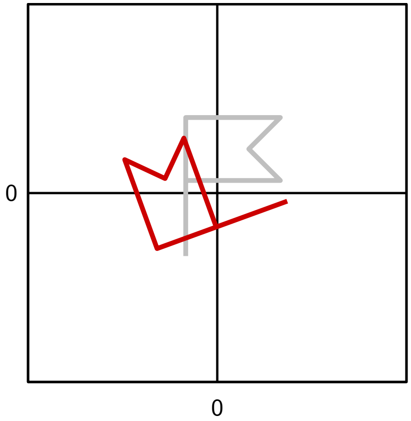


## Rotation

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{d} = 0$$

$\theta = -0.6\pi$

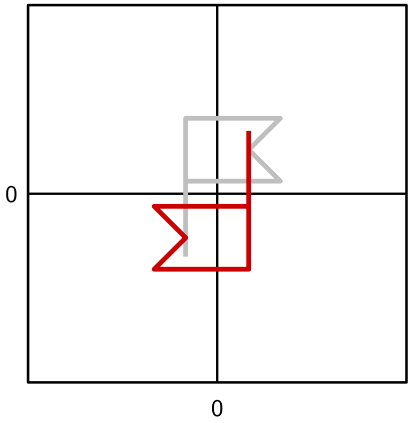


## Rotation

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{d} = 0$$

$\theta = \pi$



## Flips

Horizontal:

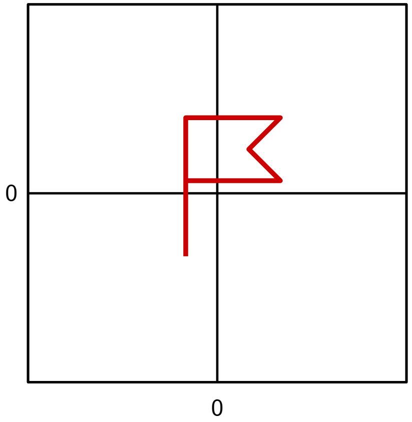
$$\mathbf{A} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$

Vertical:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\mathbf{d} = 0$$



## Flips

Horizontal:

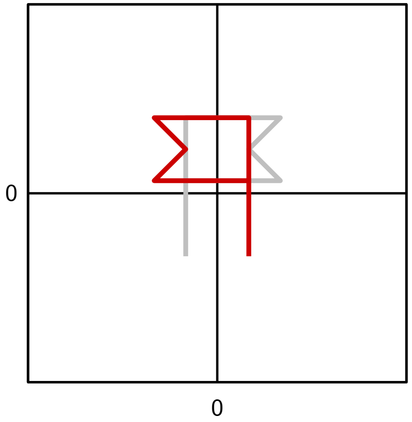
$$\mathbf{A} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$

Vertical:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\mathbf{d} = 0$$



## Shear

Horizontal:

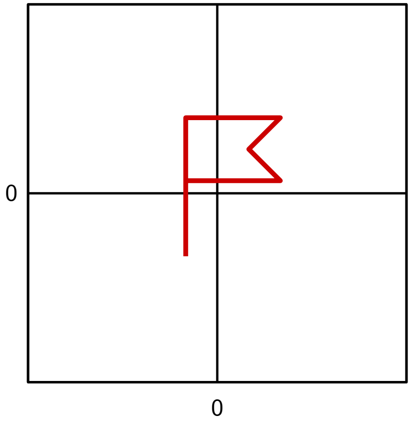
$$\mathbf{A} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$

Vertical:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$



## Shear

Horizontal:

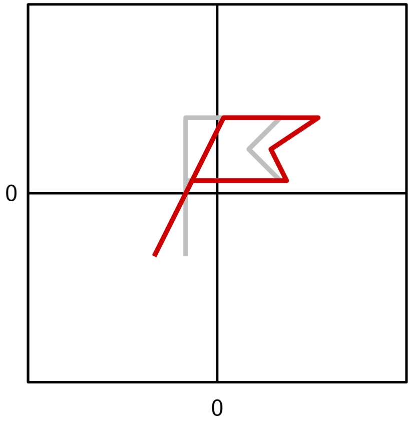
$$\mathbf{A} = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$

Vertical:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

$$\mathbf{d} = 0$$



## Affine transforms in discrete space

$$\begin{bmatrix} t'_1 \\ t'_2 \end{bmatrix} = \mathbf{A} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} - \mathbf{d} \quad \in \mathbb{R}^2 \neq \mathbb{Z}^2$$

## Solution: work backwards!

- take each *output* coordinate $y[m_1, m_2]$
- apply the *inverse* transform to $[m_1, m_2]$ and find the *source* coordinates:

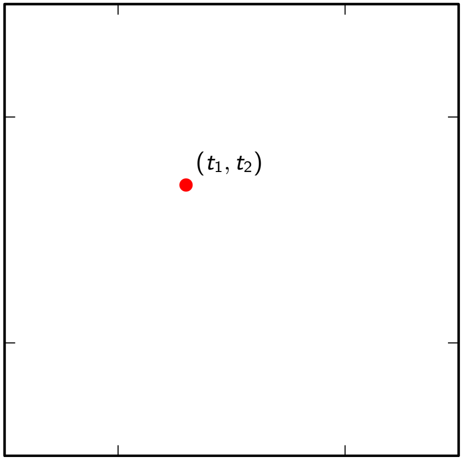
$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} m_1 + d_1 \\ m_2 + d_2 \end{bmatrix};$$

- if source point not on source grid, write

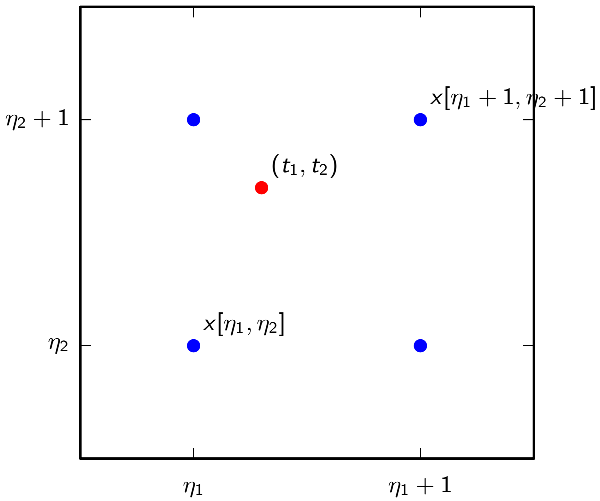
$$(t_1, t_2) = (\eta_1 + \tau_1, \eta_2 + \tau_2), \qquad \eta_{1,2} \in \mathbb{Z}, \quad 0 \leq \tau_{1,2} < 1$$

and interpolate from the surrounding original grid points

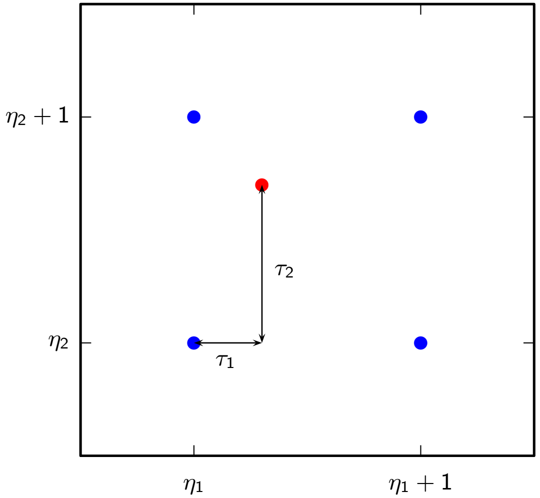
## Bilinear Interpolation



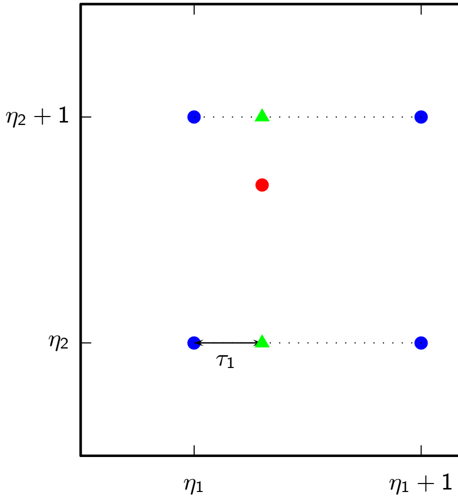
## Bilinear Interpolation



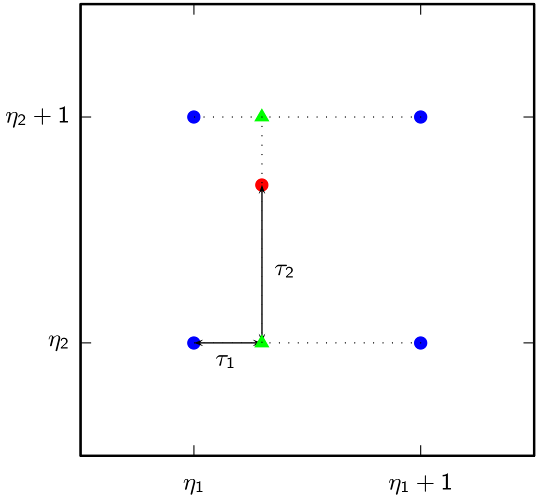
## Bilinear Interpolation



## Bilinear Interpolation



## Bilinear Interpolation

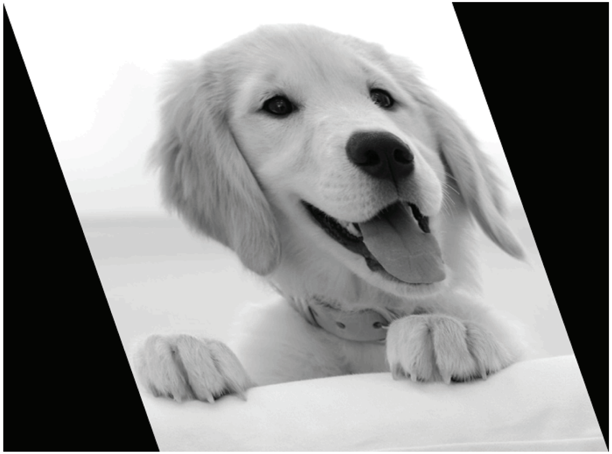


## Bilinear Interpolation

- we are free to use any interpolation scheme
- a common and easy choice is linear interpolation for both direction
- this leads to a closed-form expression using four pixel values:

$$\begin{aligned} y[m_1, m_2] = {} & (1-\tau_1)(1-\tau_2)x[\eta_1, \eta_2] + \tau_1(1-\tau_2)x[\eta_1+1, \eta_2] \\ & + (1-\tau_1)\tau_2 x[\eta_1, \eta_2+1] + \tau_1\tau_2 x[\eta_1+1, \eta_2+1] \end{aligned}$$

## Shearing



## Bilinear Interpolation for resizing

- start from the coordinates of each pixel in output resized image
- use the inverse scaling operator $\mathbf{A}^{-1} = \begin{bmatrix} 1/a_1 & 0 \\ 0 & 1/a_2 \end{bmatrix}$
- use bilinear interpolation to find value of output pixel from target input pixels

## the JPEG image compression standard

## A thought experiment

- consider all possible $256 \times 256$, 8bpp "images"
- each image is 524,288 bits

- total number of possible images: $2^{524,288} \approx 10^{157,826}$
- number of atoms in the universe: $10^{82}$

## Image compression

Goal: reduce the storage required by an image

- lossless compression: store the information exactly, just more compactly
  - zip files, PNG
  - exploits patterns in bitstream (not image-specific)
  - mostly a problem for Information Theory
- lossy compression: accept some degradation in return for larger storage gains
  - JPG, MP3
  - designed specifically for a class of signals (images, music)
  - relies on known “blind spots” in our perceptual systems

## Lossy image compression

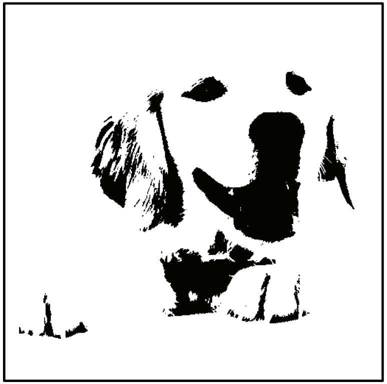
- exploit natural redundancy in images (eg: large blue sky)
- use psychovisual experiments to determine what impacts perceived quality
- allocate most bits to the things that matter
- hide the losses where they are hard to see

## Key ingredients of the JPEG standard

1. compressing at block level
2. using a suitable transform (i.e., a change of basis)
3. smart quantization
4. entropy coding

## Compressing at pixel level

- reduce number bits per pixel
- equivalent to coarser quantization
- the lowest we can go is one bit per pixel
  i.e. pixels are either black or white



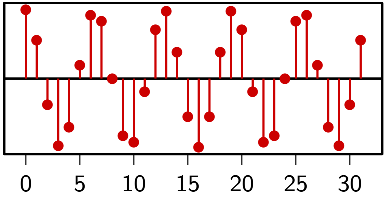
## Compressing at block level

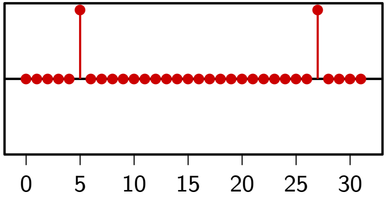
- split image into small blocks
- keep only average pixel value in blocks
- shown here:
  - $3 \times 3$ blocks, i.e. 9 pixels each
  - average value coded using 8 bits
  - storage less than 1 bpp
- where's the catch? zoom in to see

## Transform coding

A simple 1D example:

- signal encoded at $R$ bits per sample
- storing $N$ samples requires $NR$ bits
- look at the DFT: if many DFT values are close to zero we can discard them and store only the nonzero ones
- many natural signals are sparse once properly transformed!



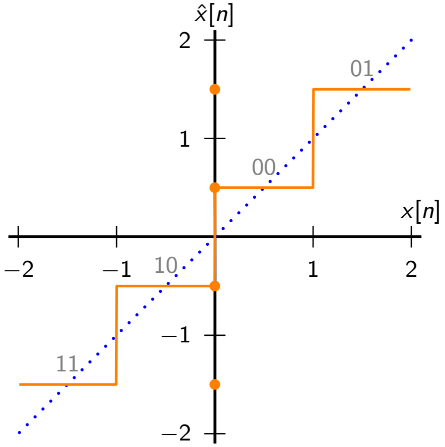


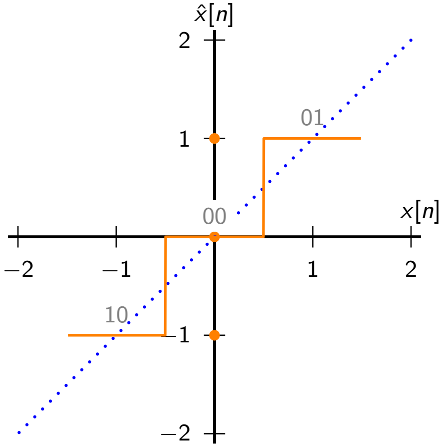
## Smart quantization

After computing the transform of each block:

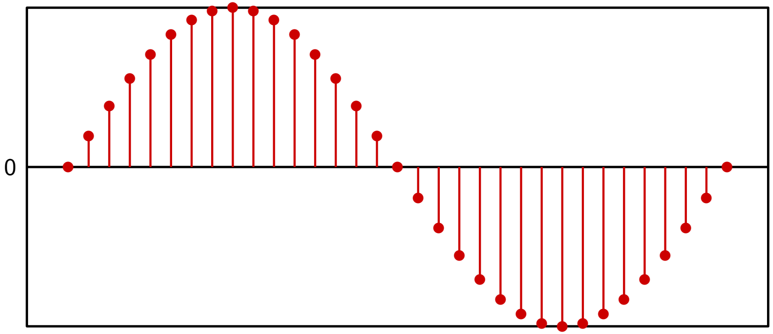
- set to zero all small coefficients
  - use a “deadzone” quantizer
  - lots of consecutive zeros can be encoded efficiently
- use more bits for the “important” coefficients
  - more bits means more quantization levels, i.e. lower quantization error
  - visual importance determined experimentally using subjective tests

## Standard vs Deadzone Quantization (2 bps)

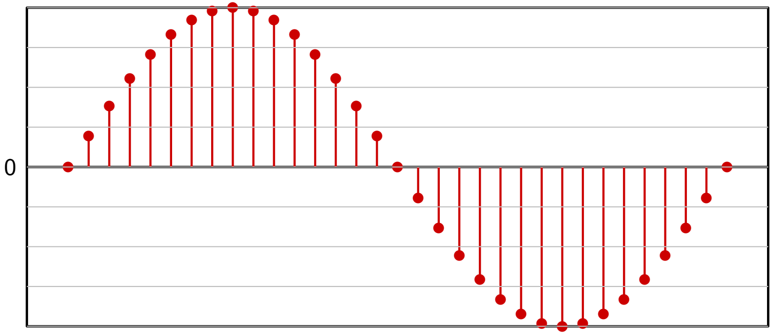




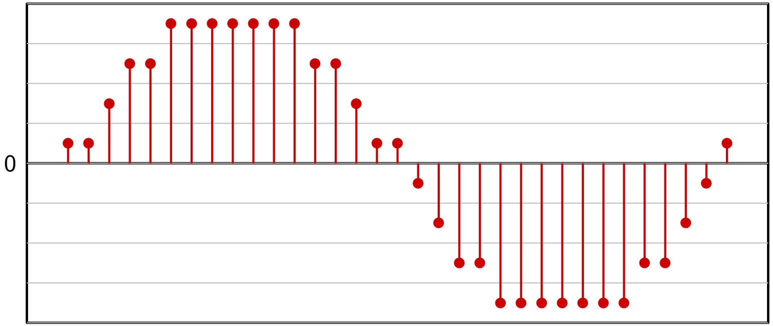
## Standard quantization: full-amplitude signal



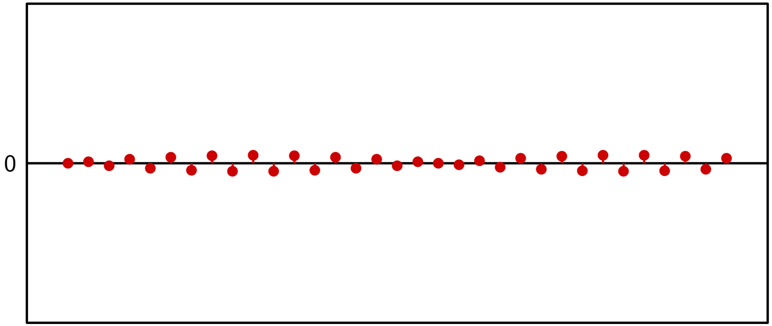
## Standard quantization: full-amplitude signal



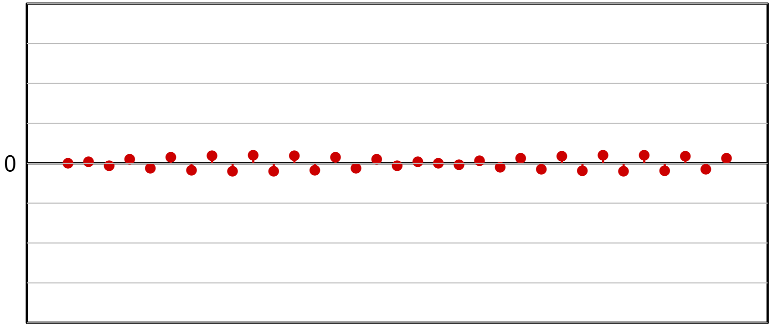
## Standard quantization: full-amplitude signal



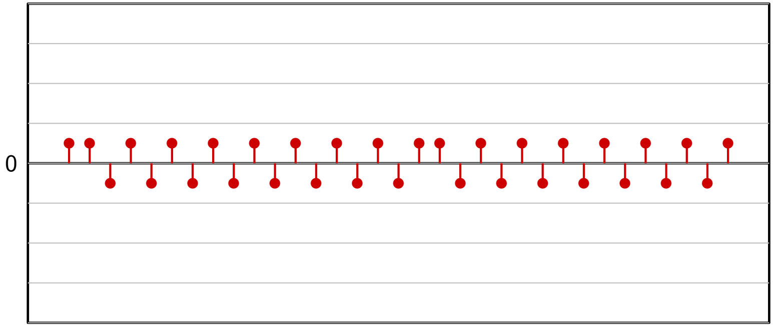
## Standard quantization: small amplitude values



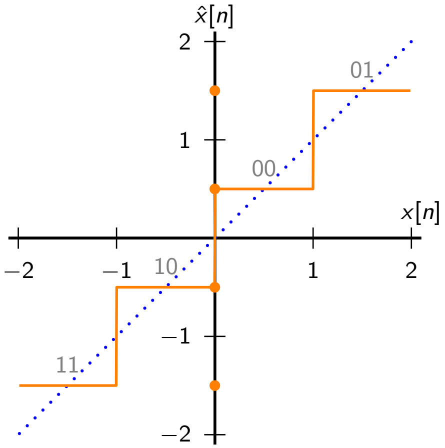
## Standard quantization: small amplitude values

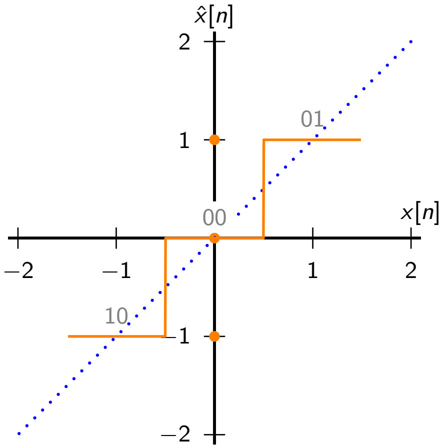


## Standard quantization: small amplitude values

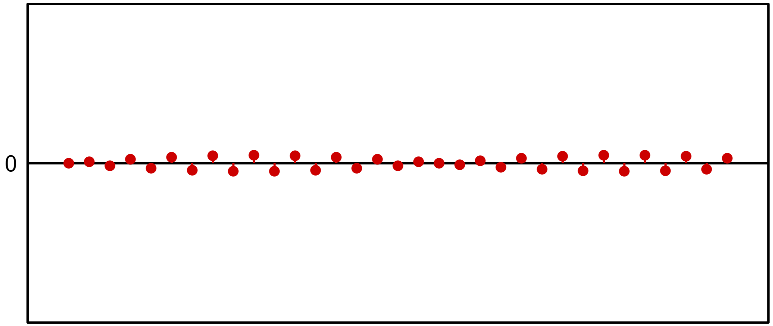


## Standard vs Deadzone Quantization (2 bps)

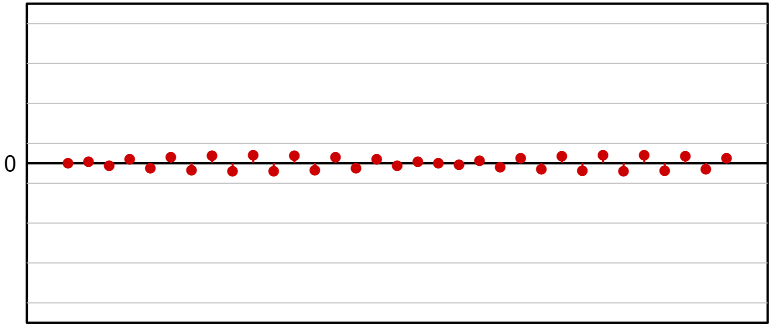




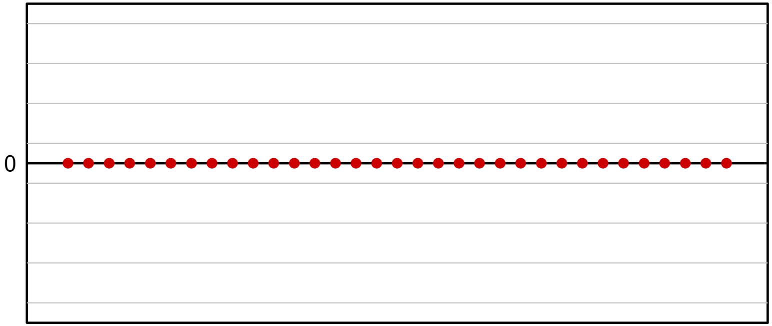
## Standard quantization: small amplitude values



## Standard quantization: small amplitude values



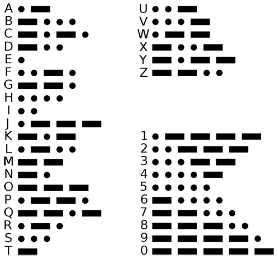
## Standard quantization: small amplitude values



## Final bitstream encoding

Smart quantization returns a series of integers that must be encoded efficiently:

- not all values occur with the same probability:
  - use short binary "codewords" for very frequent values
  - use longer binary codewords for rarely occurring values
- this is the principle of Morse code!



## Key ingredients in the JPEG standard

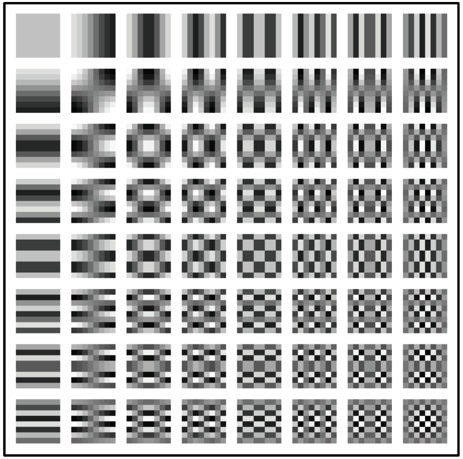
- compressing at block level: split image into $8 \times 8$ non-overlapping blocks
- using a suitable transform: compute the Discrete Cosine Transform of each block
- smart quantization: round DCT coefficients according to psycovisually-tuned tables
- bitstream encoding: run-length encoding plus Huffman entropy coding

## The Discrete Cosine Transform (DCT)

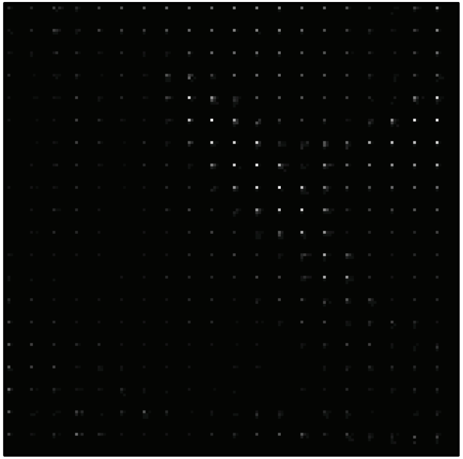
pretty much the same as the DFT:

- each coefficient inner product between image block and DCT basis vectors
- basis vectors are a bit different from the DFT but still orthogonal
- DCT coefficients are real-valued
- DCT minimizes border effects

## DCT basis vectors for an $8 \times 8$ image



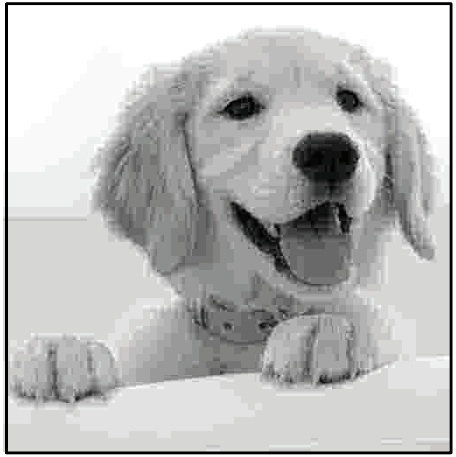
## DCT coefficients of image blocks

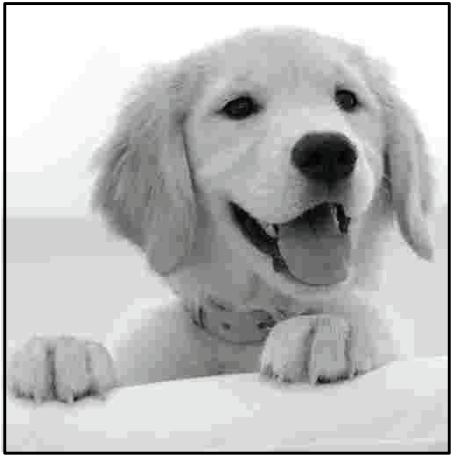


## Psycovisually-tuned quantization

- most of the 64 DCT coefficients in each block are small and rounded to zero
- coefficient with high visual impact are encoded first
- remaining bit budget allocated to the rest

## Impact of smart quantization at 0.2bpp


uniform


tuned

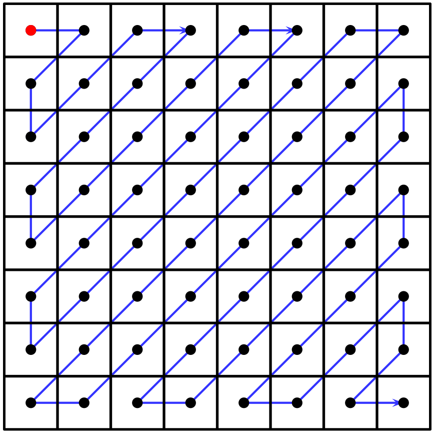
## Runlength encoding

The matrix of quantized DCT coefficients will contain a lot of zeros:

- scan the matrix in a zig-zag pattern to obtain long sequence of zeros
- encode each nonzero value as a triple:
    - number of zeros preceding the value
    - number of bits used to encode the value
    - the value itself
- each triple is a *symbol* that we must efficiently encode in binary format

$$(0,0,0,0,0,6) \quad \longrightarrow \quad [(5,4),6]$$

## Zigzag scan



## Example

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

[100],

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

$[100],\ [(0,4),\ -6],$

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

$[100]$, $[(0,4),\ -6]$, $[(4,2),\ 1]$,

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

$[100]$, $[(0, 4), -6]$, $[(4, 2), 1]$, $[(3, 3), 2]$,

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

$[100], \ [(0,4), \ -6], \ [(4,2), \ 1], \ [(3,3), \ 2], \ [(8,2), \ -1],$

## Runlength encoding

$$\begin{bmatrix} 100 & -6 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

100, -6, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

$[100],\ [(0,4),\ -6],\ [(4,2),\ 1],\ [(3,3),\ 2],\ [(8,2),\ -1],\ [(0,0)]$

## Bitstream encoding

goal: minimize storage requirement by encoding each symbol with a binary *codeword*

- *variable-length* entropy coding: frequent symbols use shorter codewords and vice-versa
- *prefix-free* code: no need for explicit separators between codewords
- JPEG provides a “general-purpose” code but you can build your own
- given the symbol frequencies, the Huffman algorithm builds an optimal prefix-free enropy coder

## Variable-length encoding

if codewords have different lengths, we must know how to parse them

- in western languages we use spaces and punctuation (we need extra symbols)
- in Morse code we use short pauses for letters and long for words (wasteful)
- can we do away with separators?

## Prefix-free codes

in the bitstream generated by a prefix-free encoder:

- no valid codeword is the beginning of another valid codeword
- the bitstream can be parsed sequentially with no look-ahead
- easiest way to understand is graphically, using binary trees

## Building a Huffman code: toy example

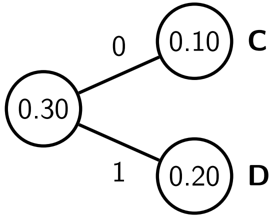
- four symbols: A, B, C, D
- probability table:

$$p(A) = 0.38 \qquad p(B) = 0.32$$

$$p(C) = 0.1 \qquad p(D) = 0.2$$

## Building a Huffman code

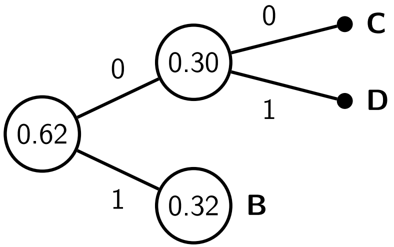
$$p(A) = 0.38 \quad p(B) = 0.32 \quad p(C) = 0.1 \quad p(D) = 0.2$$



- at each step, select the two least probable symbols
- set them as descendant of an intermediate node
- arbitrarily assign binary digits to the branches

## Building a Huffman code

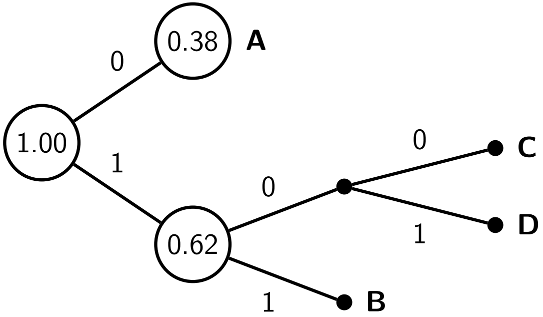
$p(A) = 0.38 \quad p(B) = 0.32 \quad p(C + D) = 0.3$



- probability of intermediate nodes is the sum of descendants
- repeat the previous step with remaining symbols and intermediate nodes

## Building a Huffman code

$$p(A) = 0.38 \quad p(B + C + D) = 0.62$$



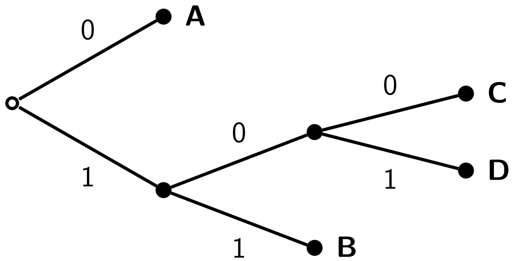
## Building a Huffman code: toy example

- four symbols: A, B, C, D
- probabilities: $p(A) = 0.38,\ p(B) = 0.32,\ p(C) = 0.1,\ p(D) = 0.2$
- prefix-free codewords:

$$A \longrightarrow 0 \qquad B \longrightarrow 11$$

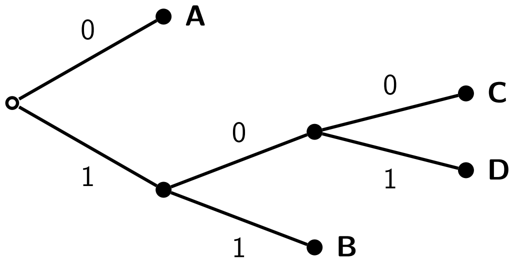
$$C \longrightarrow 100 \qquad D \longrightarrow 101$$

## Decoding a prefix-free bitstream



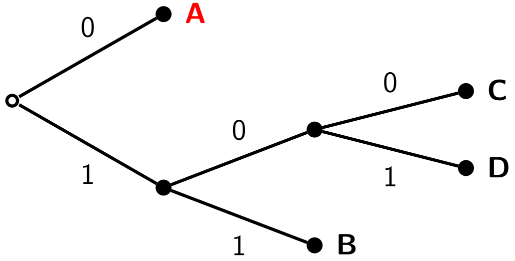
001100110101100

## Decoding a prefix-free bitstream



001100110101100

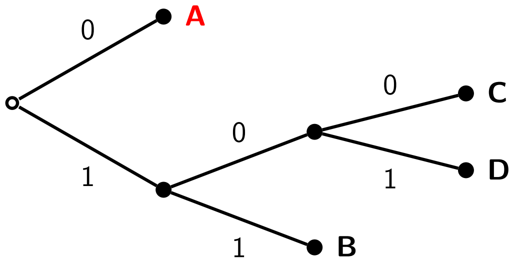
## Decoding a prefix-free bitstream



001100110101100

A

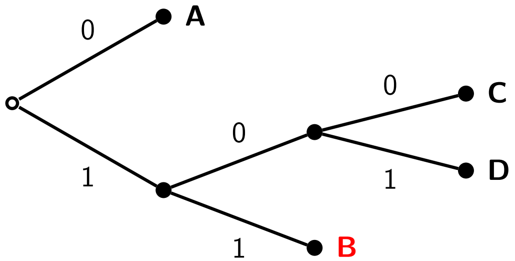
## Decoding a prefix-free bitstream



001100110101100

AA

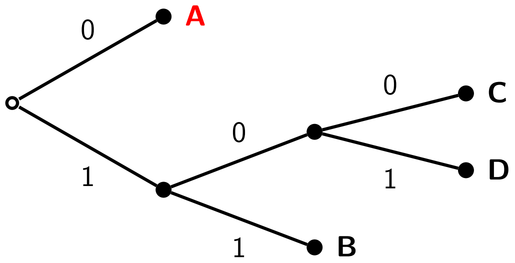
## Decoding a prefix-free bitstream



0011001101010110 0

AAB

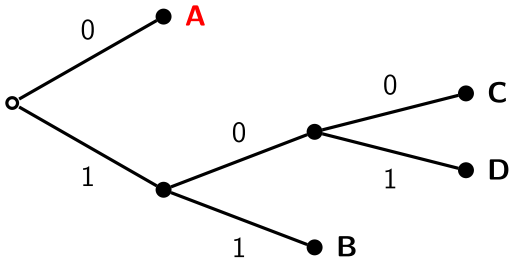
## Decoding a prefix-free bitstream



0011**0**0110101100

AABA

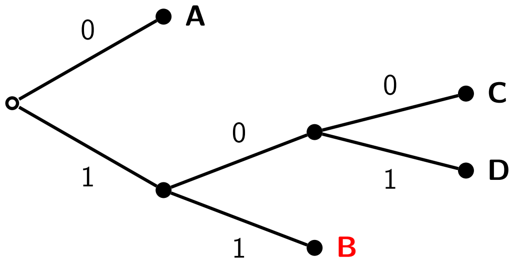
## Decoding a prefix-free bitstream



001100110101100

AABAA

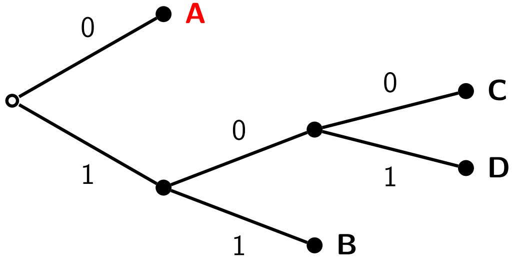
## Decoding a prefix-free bitstream



001100110101100

AABAAB

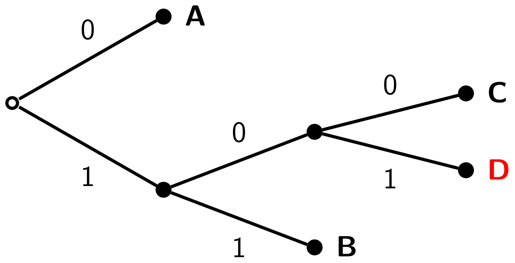
## Decoding a prefix-free bitstream



001100110101100

AABAABA

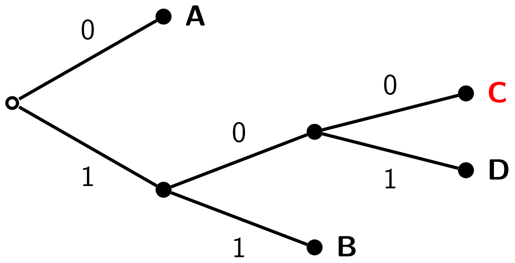
## Decoding a prefix-free bitstream



001100110101100

AABAABAD

## Decoding a prefix-free bitstream



001100110101100

AABAABADC

## Conclusions

- JPEG is a very complex and comprehensive standard:
  - lossless, lossy
  - color, B&W
  - progressive encoding
  - HDR (12bpp) for medical imaging
- JPEG is VERY good:
  - compression factor of 10:1 virtually indistinguishable
  - rates of 1bpp for RGB images acceptable (25:1 compression ratio)
- other important compression schemes:
  - TIFF, JPEG2000
  - MPEG (MP3)