Lecture 16:

Image Compression and Basic Image Processing

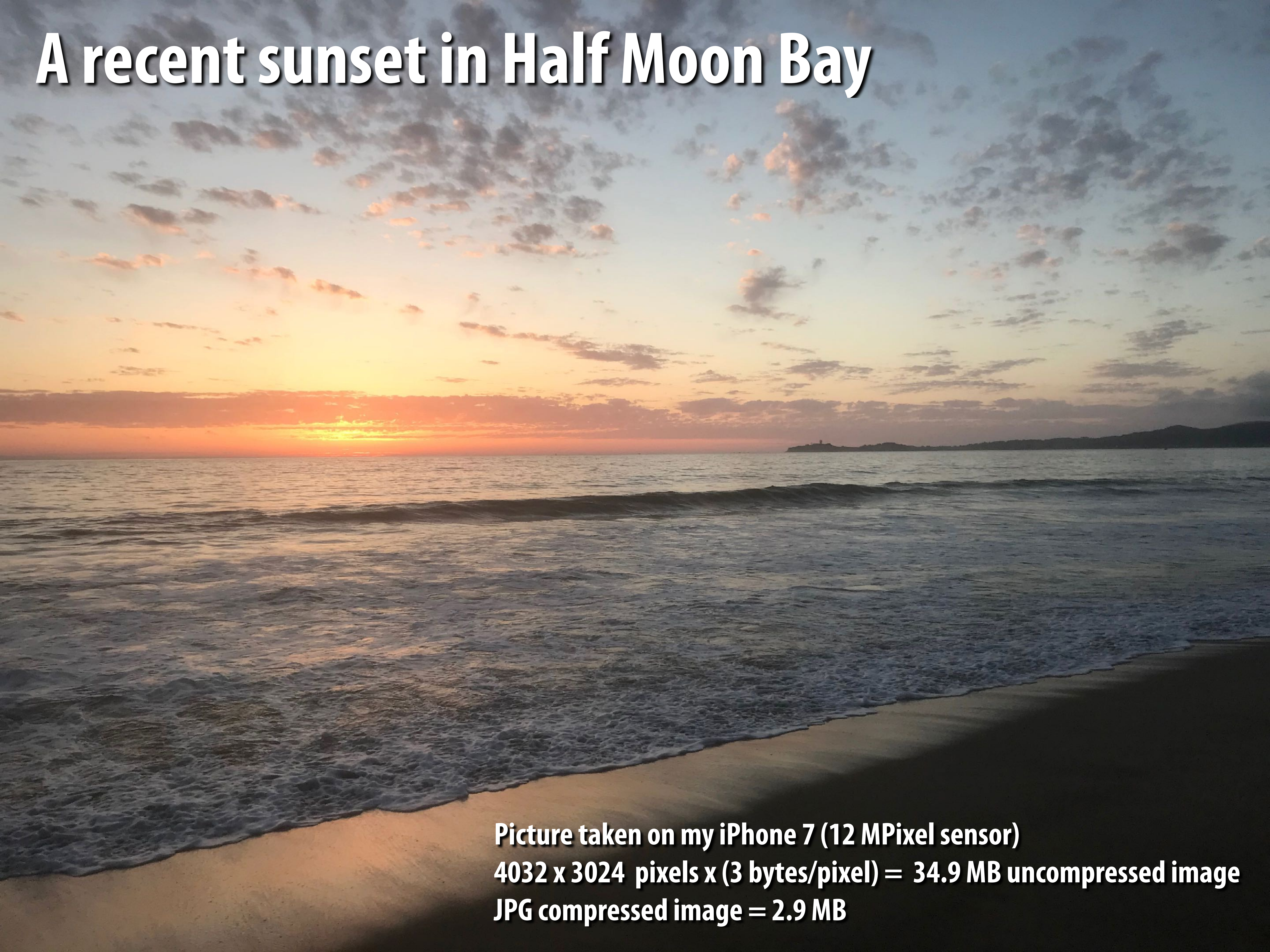
Interactive Computer Graphics
Stanford CS248, Spring 2018

Recurring themes in the course

- Choosing the right representation for a task
  - e.g., choosing the right basis
- Exploiting human perception for computational efficiency
  - Errors/approximations in algorithms can be tolerable if humans do not notice
- Convolution as a useful operator
  - To remove high frequency content from images
  - What else can we do with convolution?

Image Compression

A recent sunset in Half Moon Bay



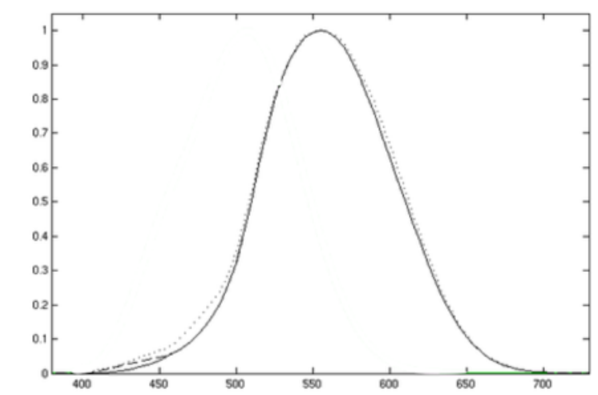
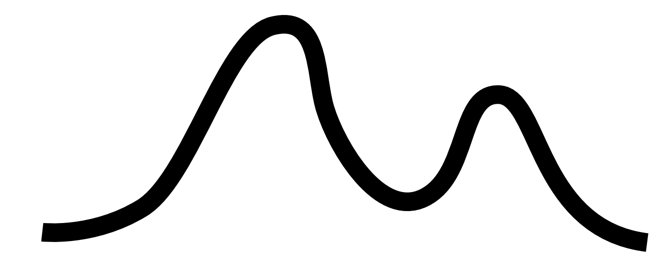
Picture taken on my iPhone 7 (12 MPixel sensor)
4032 x 3024 pixels x (3 bytes/pixel) = 34.9 MB uncompressed image
JPG compressed image = 2.9 MB

Idea 1:

- What is the most efficient way to encode intensity values as a byte?

- Encode based on how the brain perceives brightness not, based on actual response of eye

Lightness (perceived brightness) aka luma

Lightness (L*) ← ? **Luminance (Y)** $= \int_\lambda$  $*$ 

(Perceived by brain)

(Response of eye)

Spectral sensitivity of eye
(eye's response curve)

Radiance
(energy spectrum
from scene)

Dark adapted eye: $L^* \propto Y^{0.4}$

Bright adapted eye: $L^* \propto Y^{0.5}$

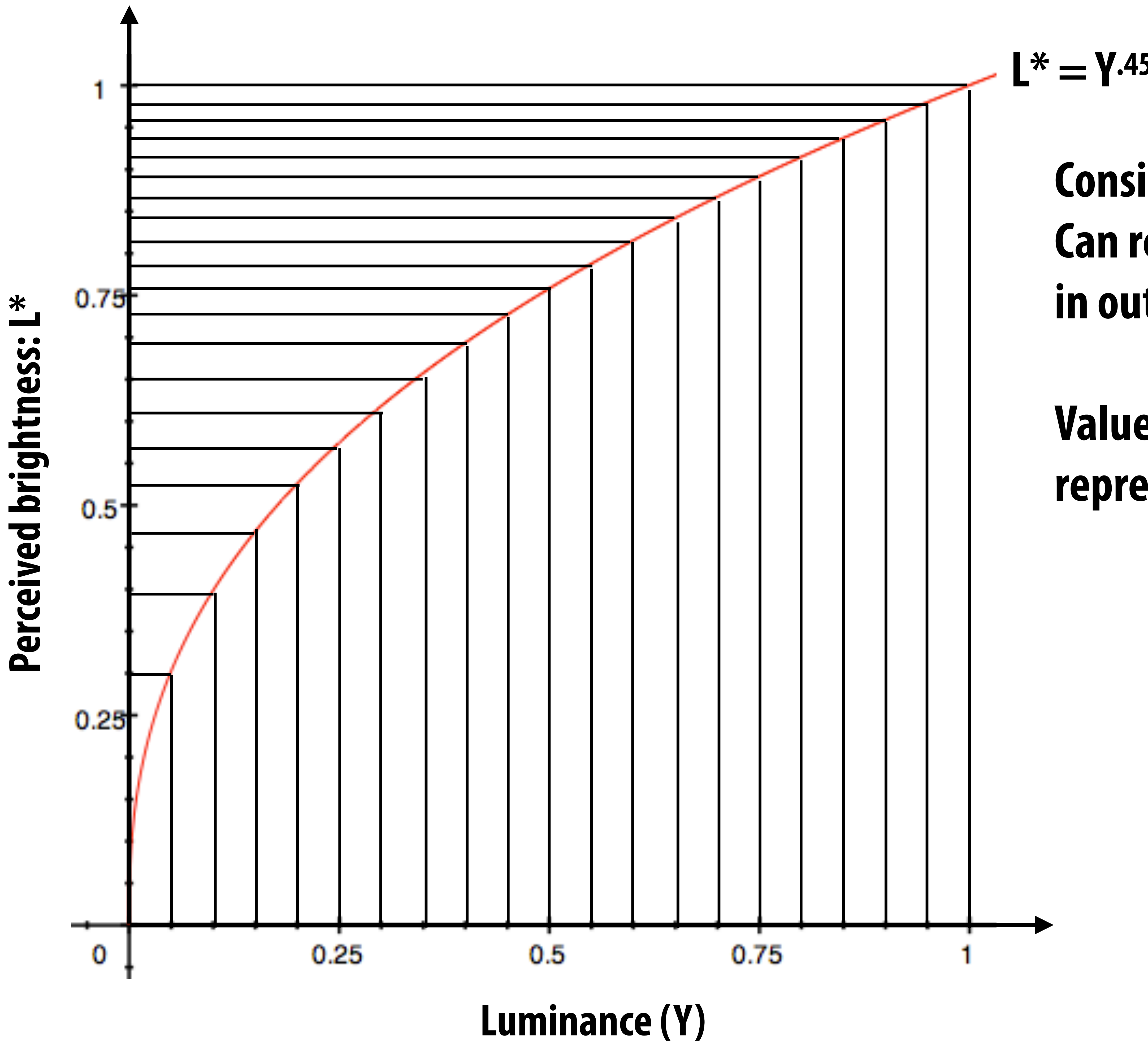
In a dark room, you turn on a light with luminance: Y_1

You turn on a second light that is identical to the first. Total output is now: $Y_2 = 2Y_1$

Total output appears $2^{0.4} = 1.319$ **times brighter to dark-adapted human**

Note: Lightness (L*) is often referred to as luma (Y')

Consider an image with pixel values encoding luminance (linear in energy hitting sensor)



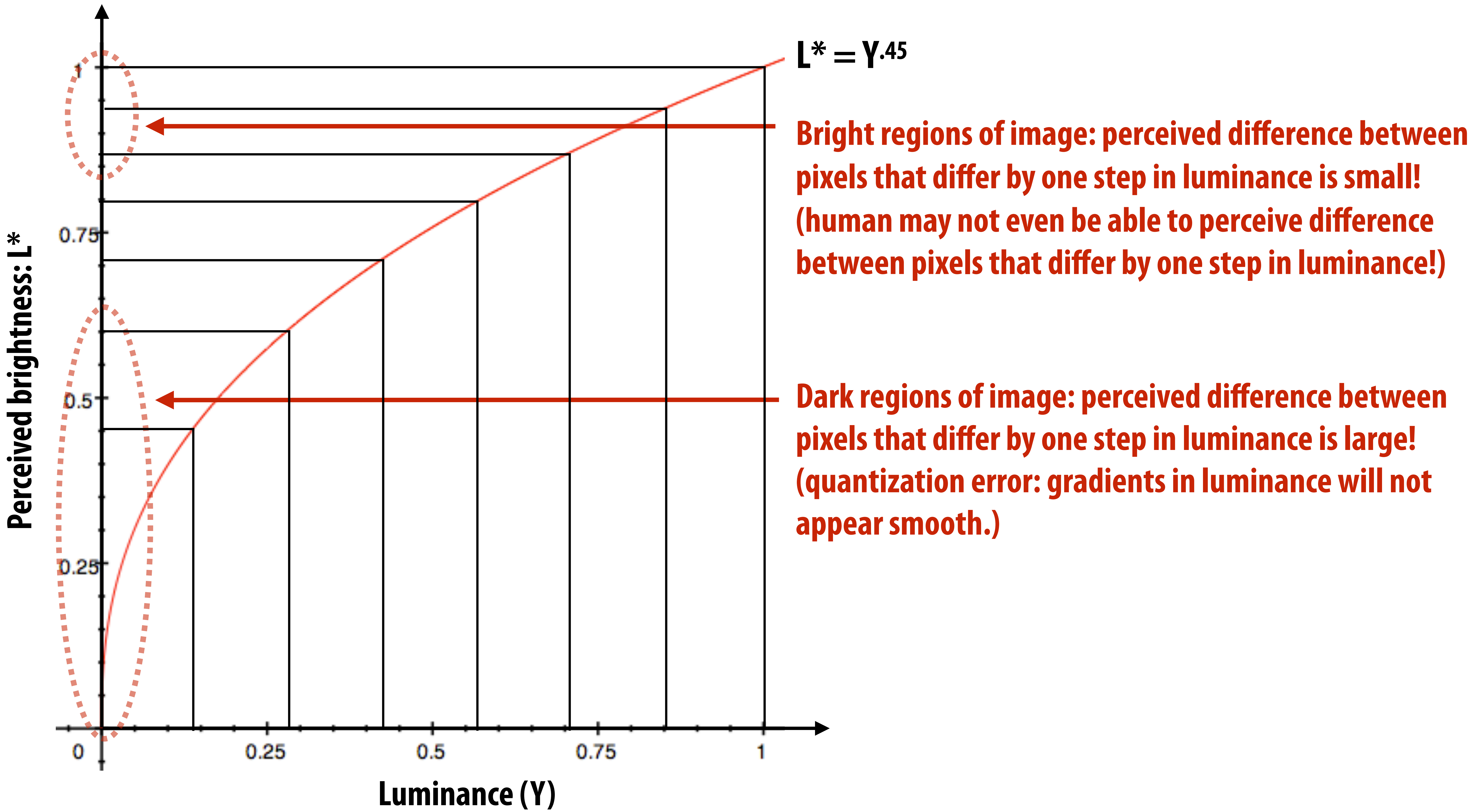
Consider 12-bit sensor pixel:
Can represent 4096 unique luminance values in output image

Values are ~ linear in luminance since they represent the sensor's response

Problem: quantization error

Many common image formats store 8 bits per channel (256 unique values)

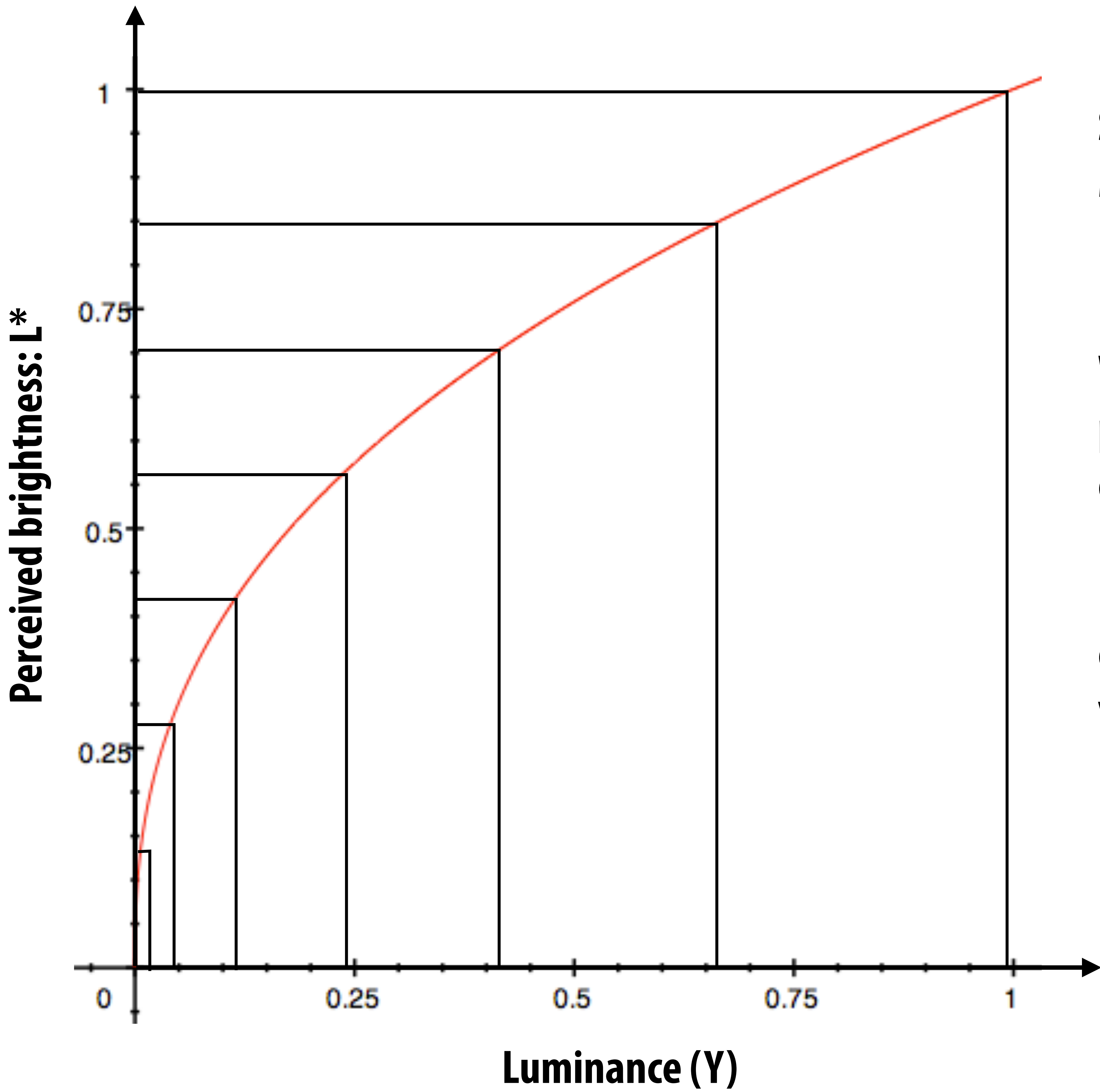
Insufficient precision to represent brightness in darker regions of image



Rule of thumb: human eye cannot differentiate <1% differences in luminance

Store lightness, not luminance

Idea: distribute representable pixel values evenly with respect to perceived brightness, not evenly in luminance (make more efficient use of available bits)



Solution: pixel stores $Y^{0.45}$

Must compute $(\text{pixel\_value})^{2.2}$ prior to display on LCD

Warning: must take caution with subsequent pixel processing operations once pixels are encoded in a space that is not linear in luminance.

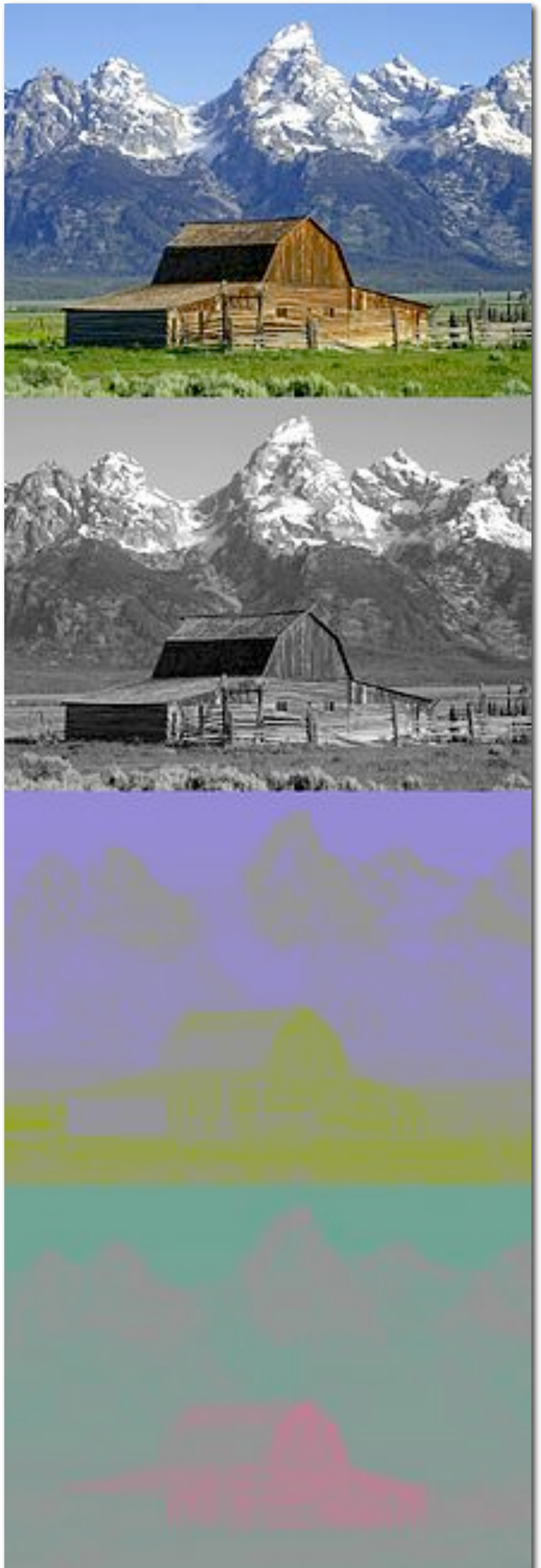
e.g., When adding images should you add pixel values that are encoded as lightness or as luminance?

Idea 2:

- **Chrominance ("chroma") subsampling**

- **The human visual system is less sensitive to detail in chromaticity than in luminance**
  - **So it is sufficient to sample chroma at a lower rate**

Y’CbCr color space



Y’

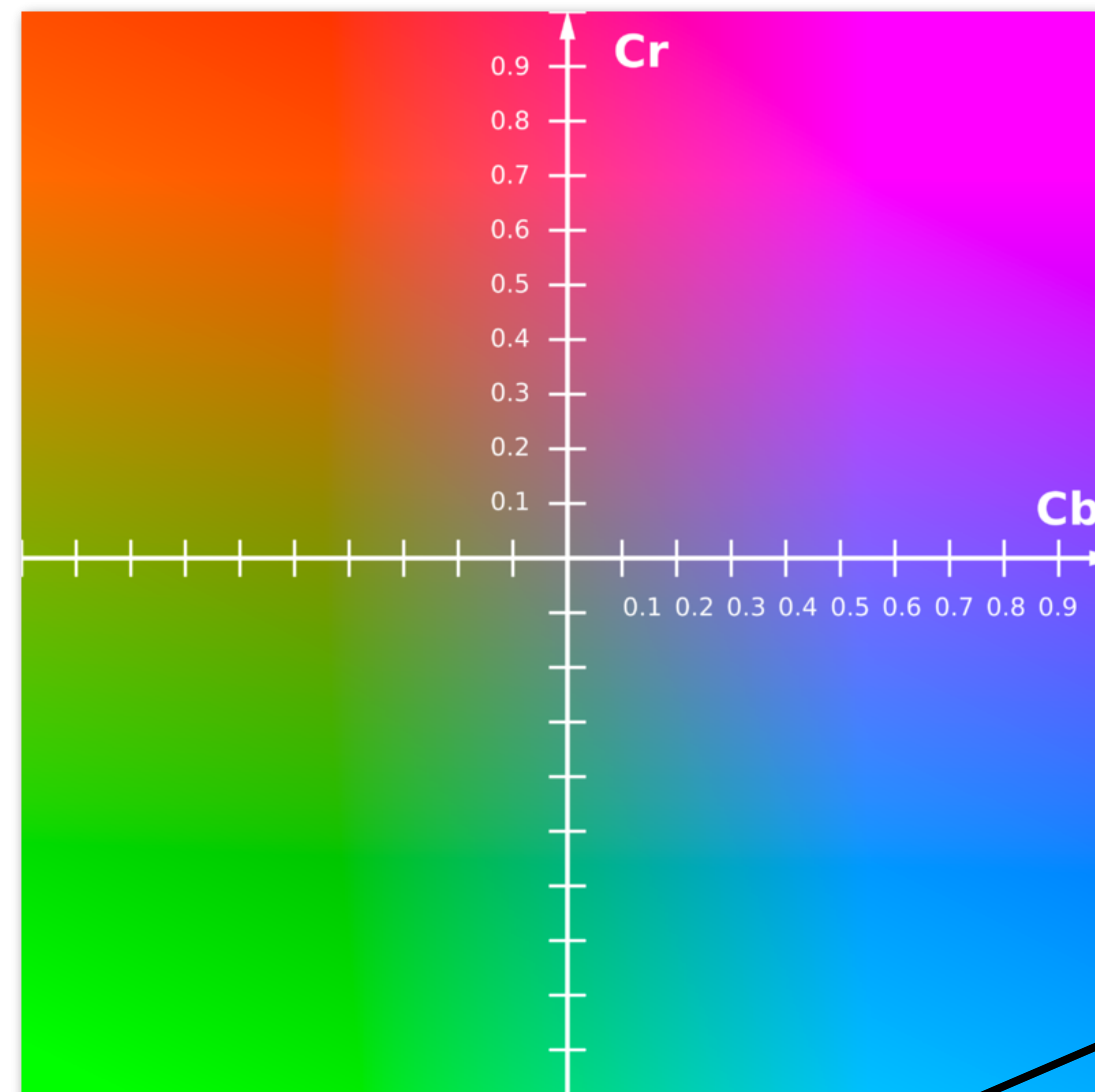
Cb

Cr

Y’ = luma: perceived luminance (non-linear)

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray



Non-linear RGB (primed notation indicates perceptual (non-linear) space)

Conversion from R’G’B’ to Y’CbCr:

$$Y' = 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256}$$

$$C_B = 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256}$$

$$C_R = 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}$$

Image credit: Wikipedia

Example: compression in Y'CbCr

Original picture of Kayvon

Example: compression in Y'CbCr

Contents of CbCr color channels downsampled by a factor of 20 in each dimension
(400x reduction in number of samples)

Example: compression in Y'CbCr

Full resolution sampling of luma (Y')

Example: compression in Y'CbCr

Reconstructed result
(looks pretty good)

Chroma subsampling

Y'CbCr is an efficient representation for storage (and transmission) because Y' can be stored at higher resolution than CbCr without significant loss in perceived visual quality

Y'_{00} Cb_{00} Cr_{00}	Y'_{10}	Y'_{20} Cb_{20} Cr_{20}	Y'_{30}
Y'_{01} Cb_{01} Cr_{01}	Y'_{11}	Y'_{21} Cb_{21} Cr_{21}	Y'_{31}

4:2:2 representation:

Store Y' at full resolution

Store Cb, Cr at full vertical resolution, but only half horizontal resolution

X:Y:Z notation:

- X = width of block
- Y = number of chroma samples in first row
- Z = number of chroma samples in second row

Y'_{00} Cb_{00} Cr_{00}	Y'_{10}	Y'_{20} Cb_{20} Cr_{20}	Y'_{30}
Y'_{01}	Y'_{11}	Y'_{21}	Y'_{31}

4:2:0 representation:

Store Y' at full resolution

Store Cb, Cr at half resolution in both dimensions

Real-world 4:2:0 examples:

most JPG images and H.264 video

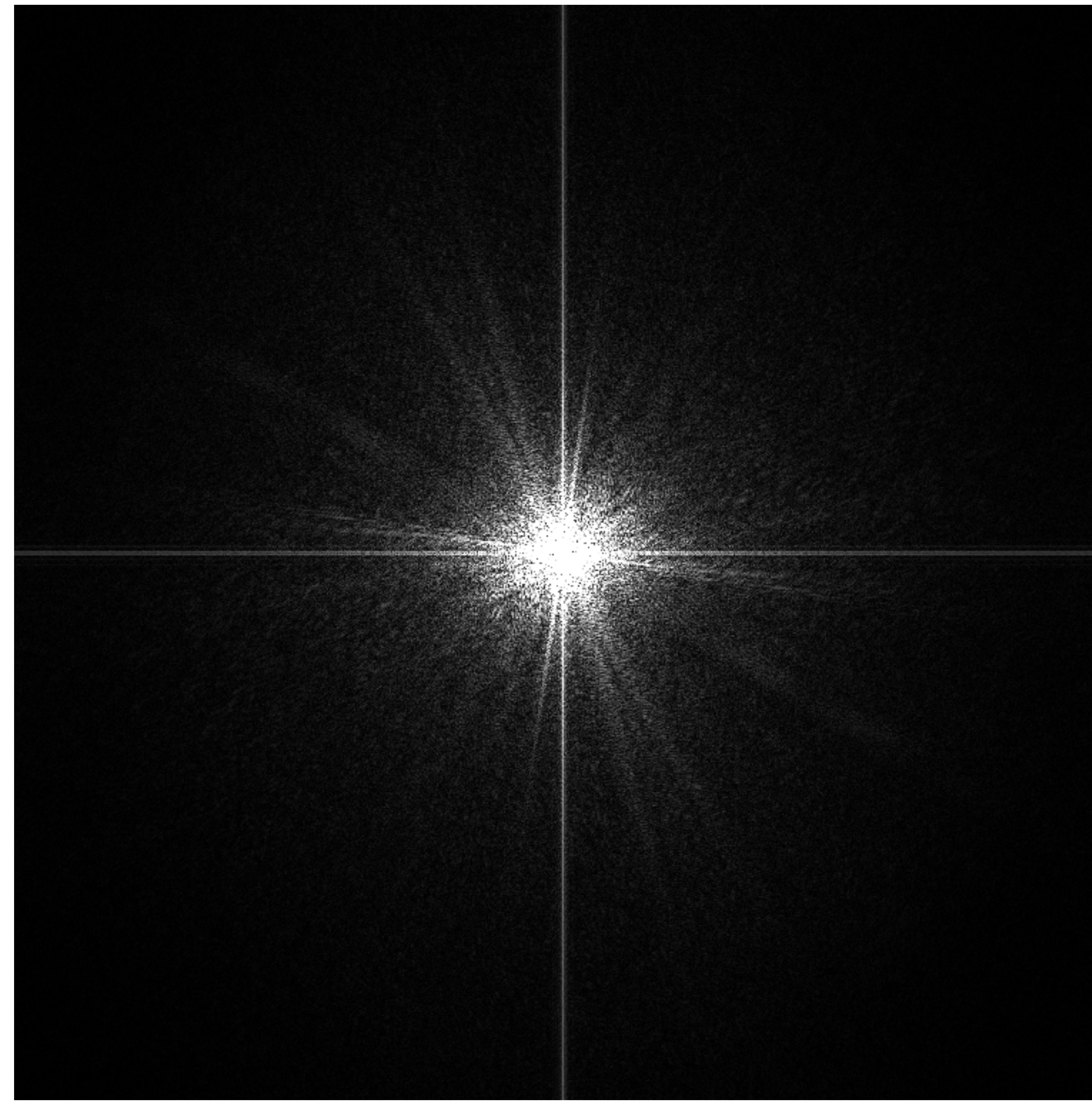
Blue-Ray

Idea 3:

- Low frequency content is predominant in the real world
- The human visual system is less sensitive to high frequency sources of error in images
- So a good compression scheme needs to accurately represent lower frequencies, but it can be acceptable to sacrifice accuracy in representing higher frequencies

Recall: frequency content of images

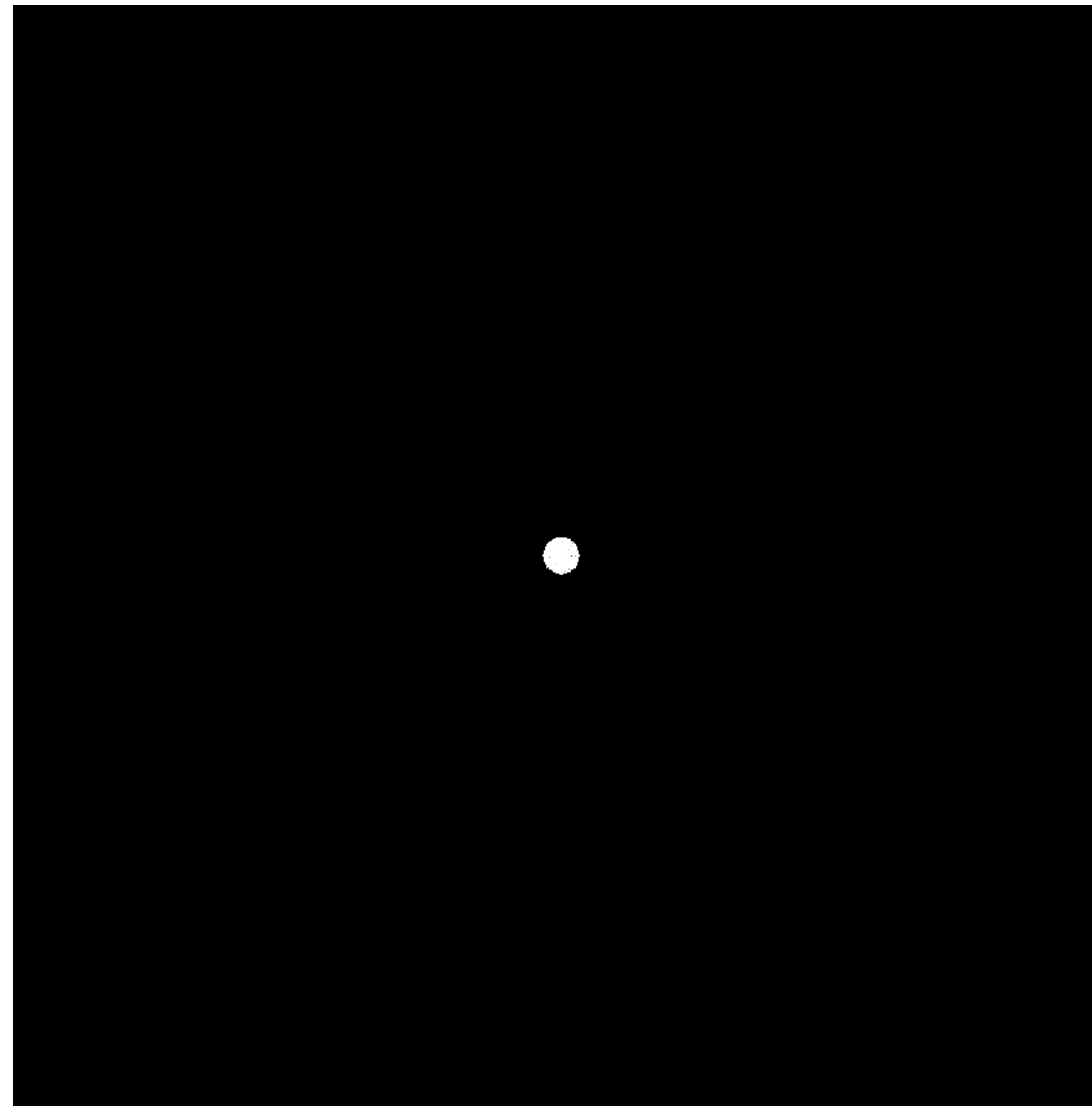
Spatial domain result



Spectrum

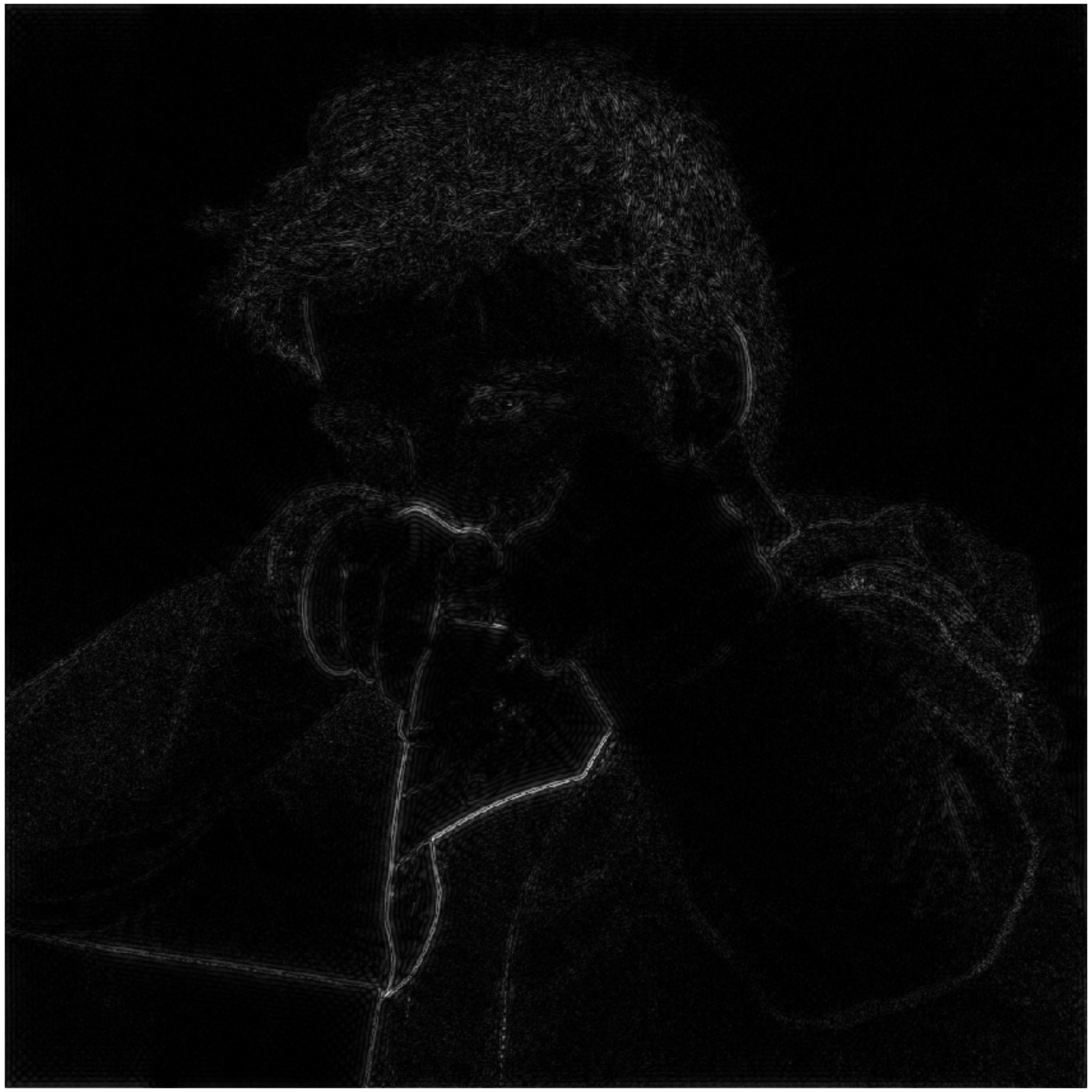
Recall: frequency content of images

Spatial domain result

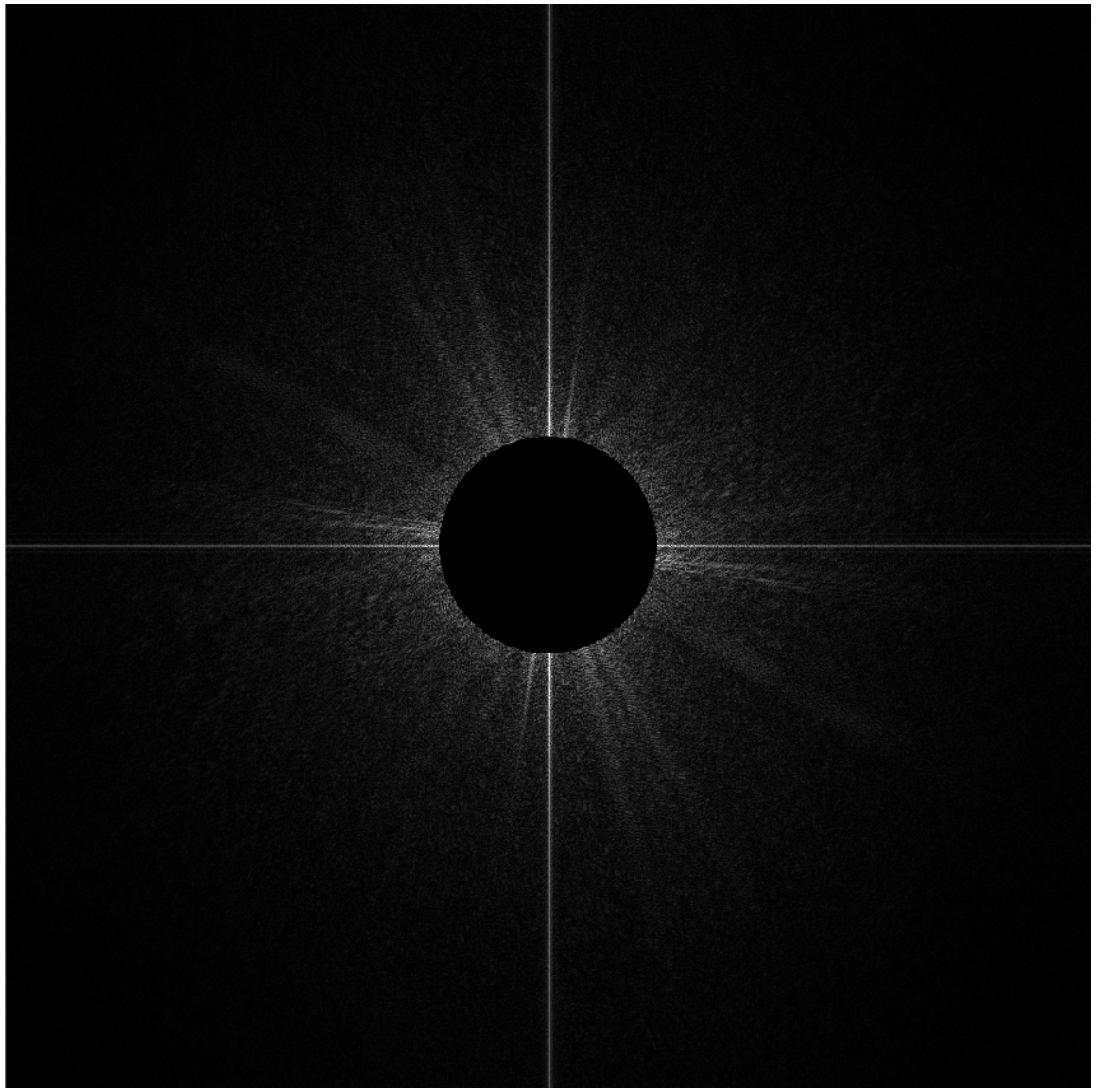


Spectrum (after low-pass filter)
All frequencies above cutoff have 0 magnitude

Recall: frequency content of images

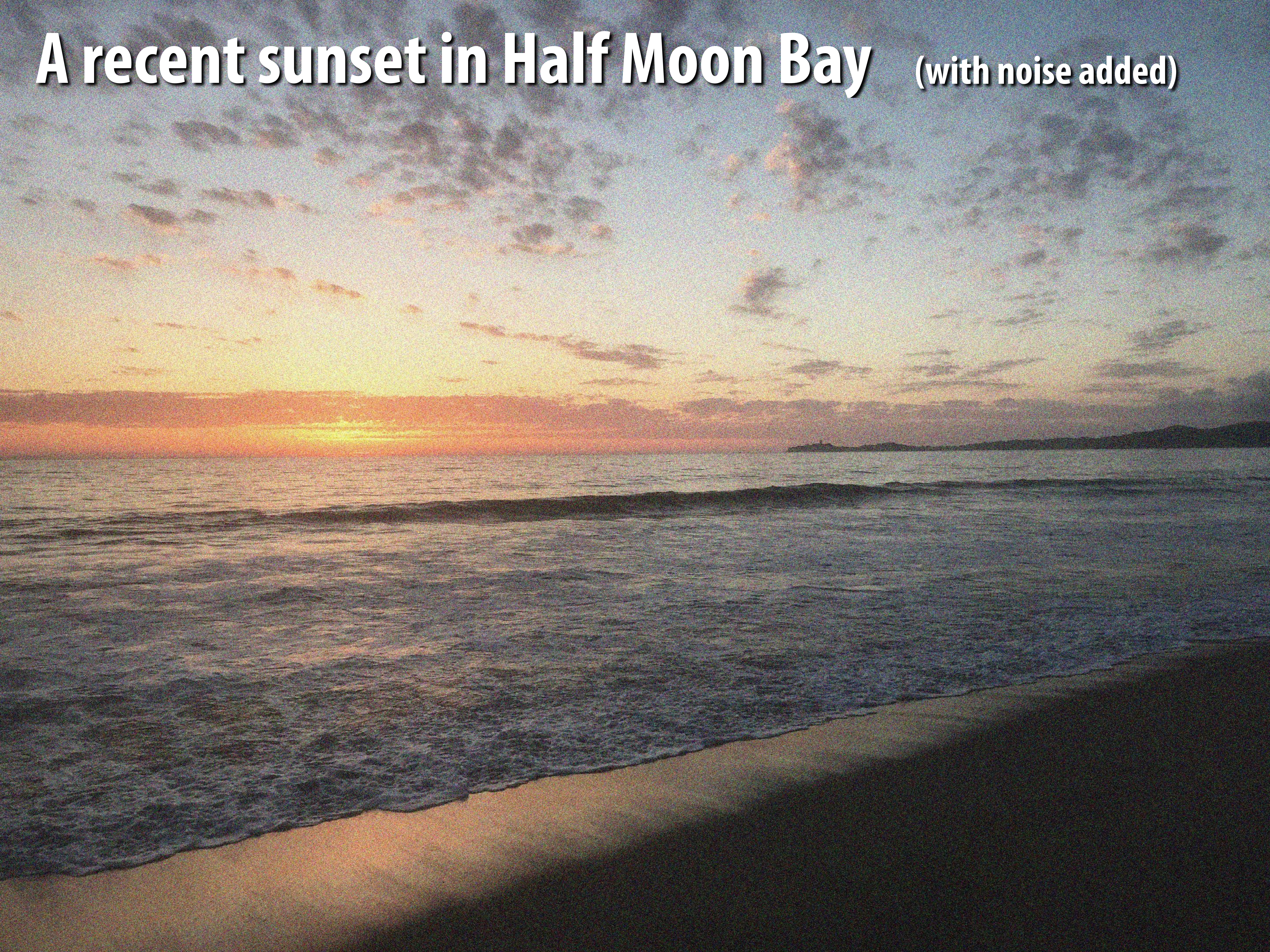


Spatial domain result
(strongest edges)



Spectrum (after high-pass filter)
All frequencies below threshold
have 0 magnitude

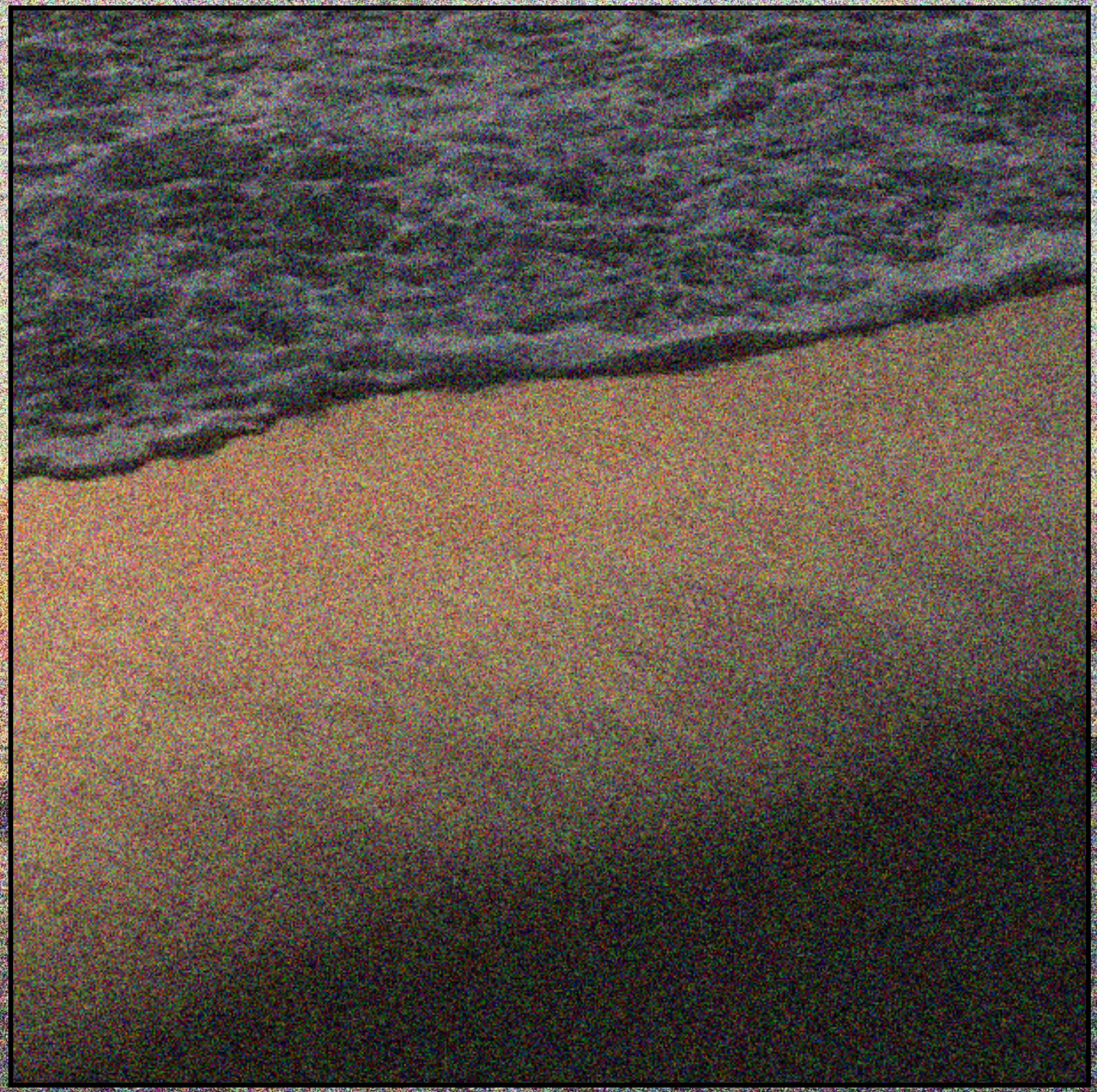
A recent sunset in Half Moon Bay


A recent sunset in Half Moon Bay
(with noise added)

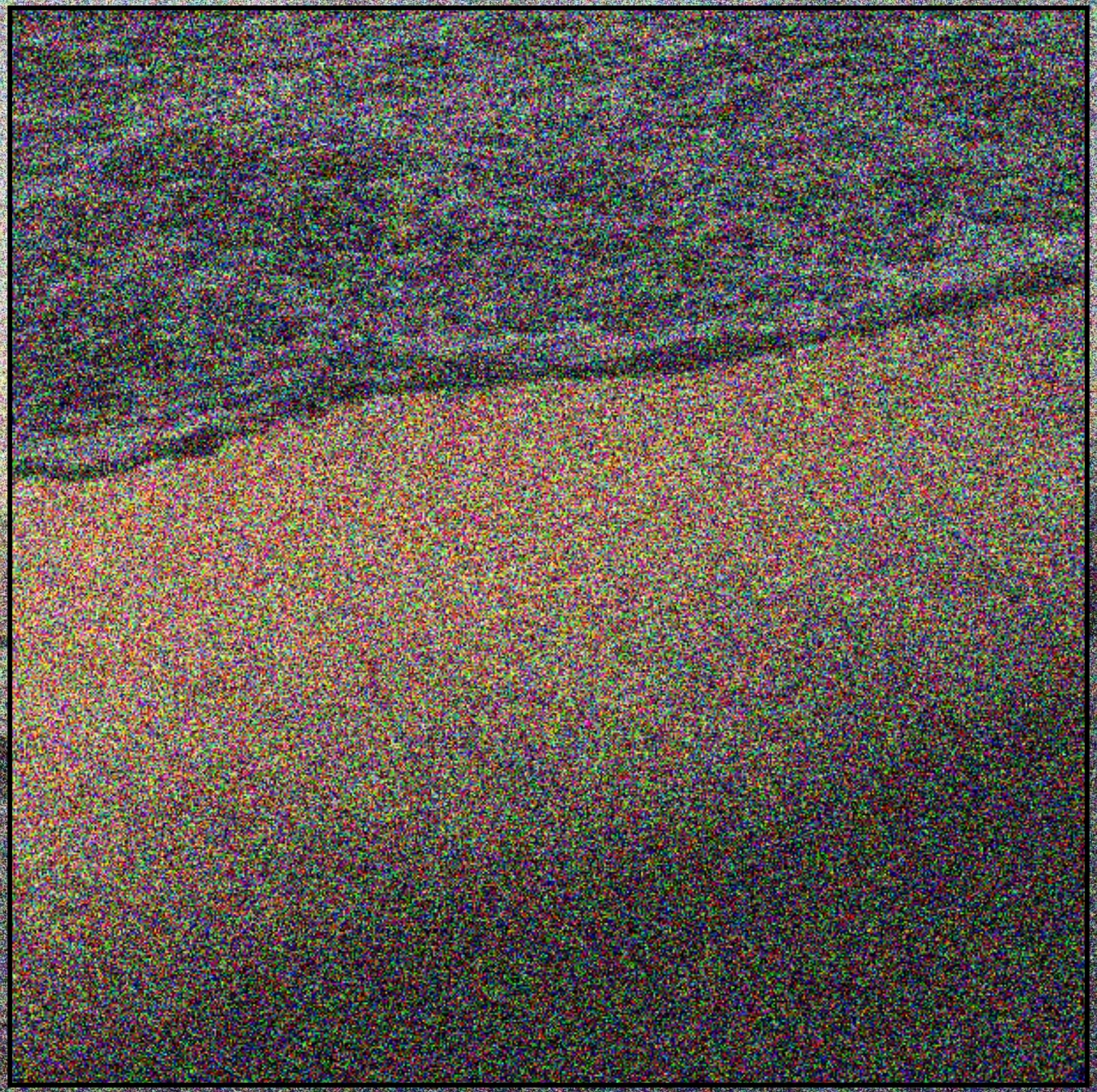
A recent sunset in Half Moon Bay (with more noise added)

A recent sunset in Half Moon Bay

Original image



Noise added
(increases high frequency content)



More noise added

What is a good representation for manipulating frequency content of images?

Hint:

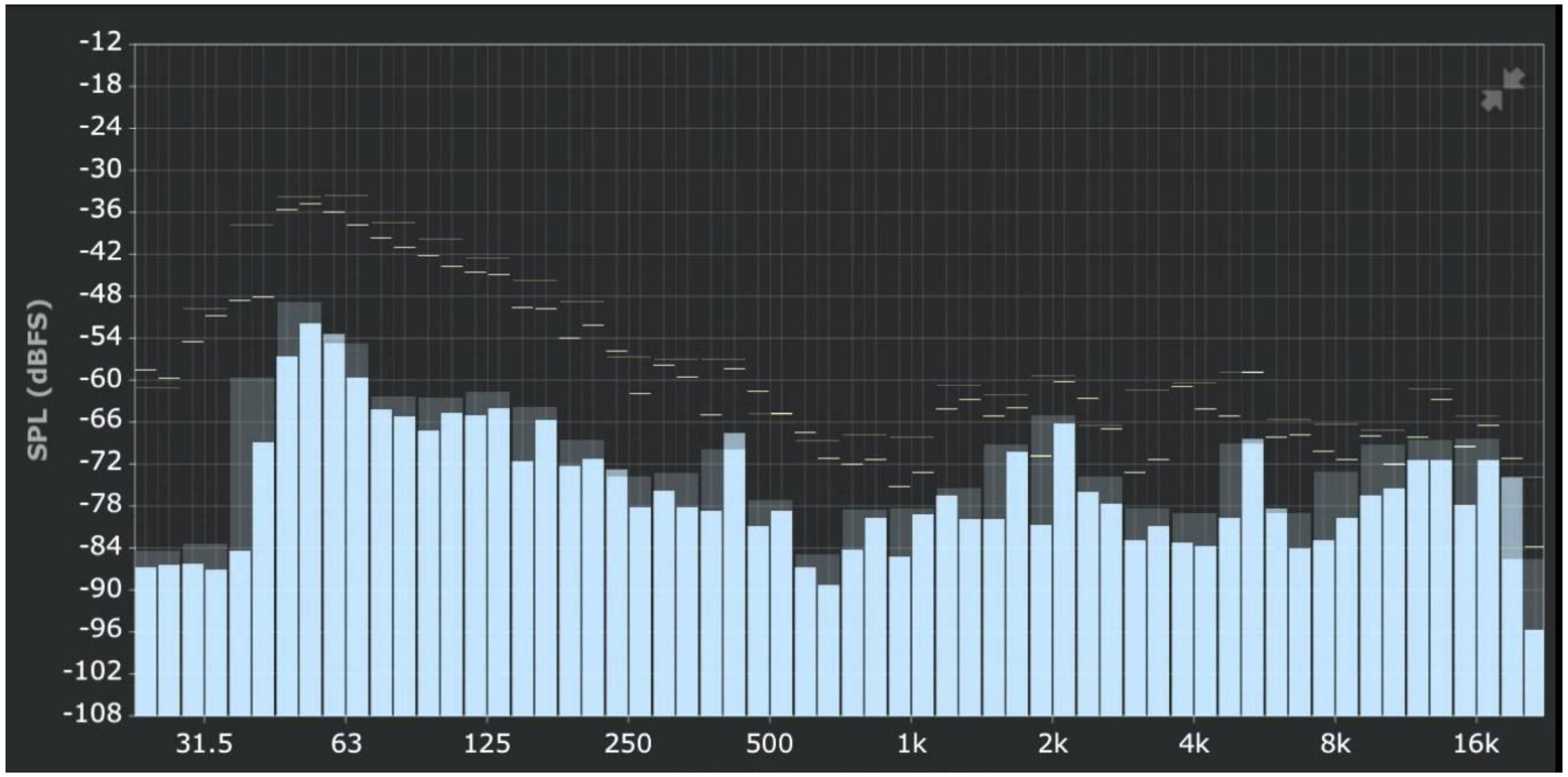
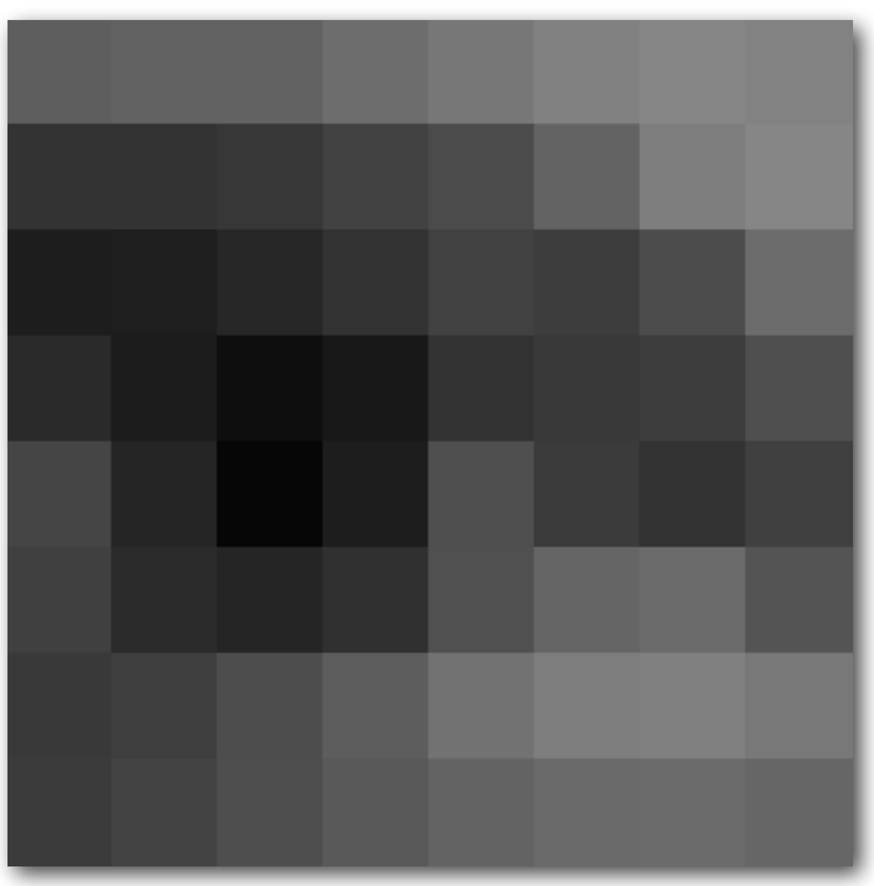


Image transform coding via discrete cosign transform (DCT)

8x8 pixel block (64 coefficients of signal in "pixel basis")



=

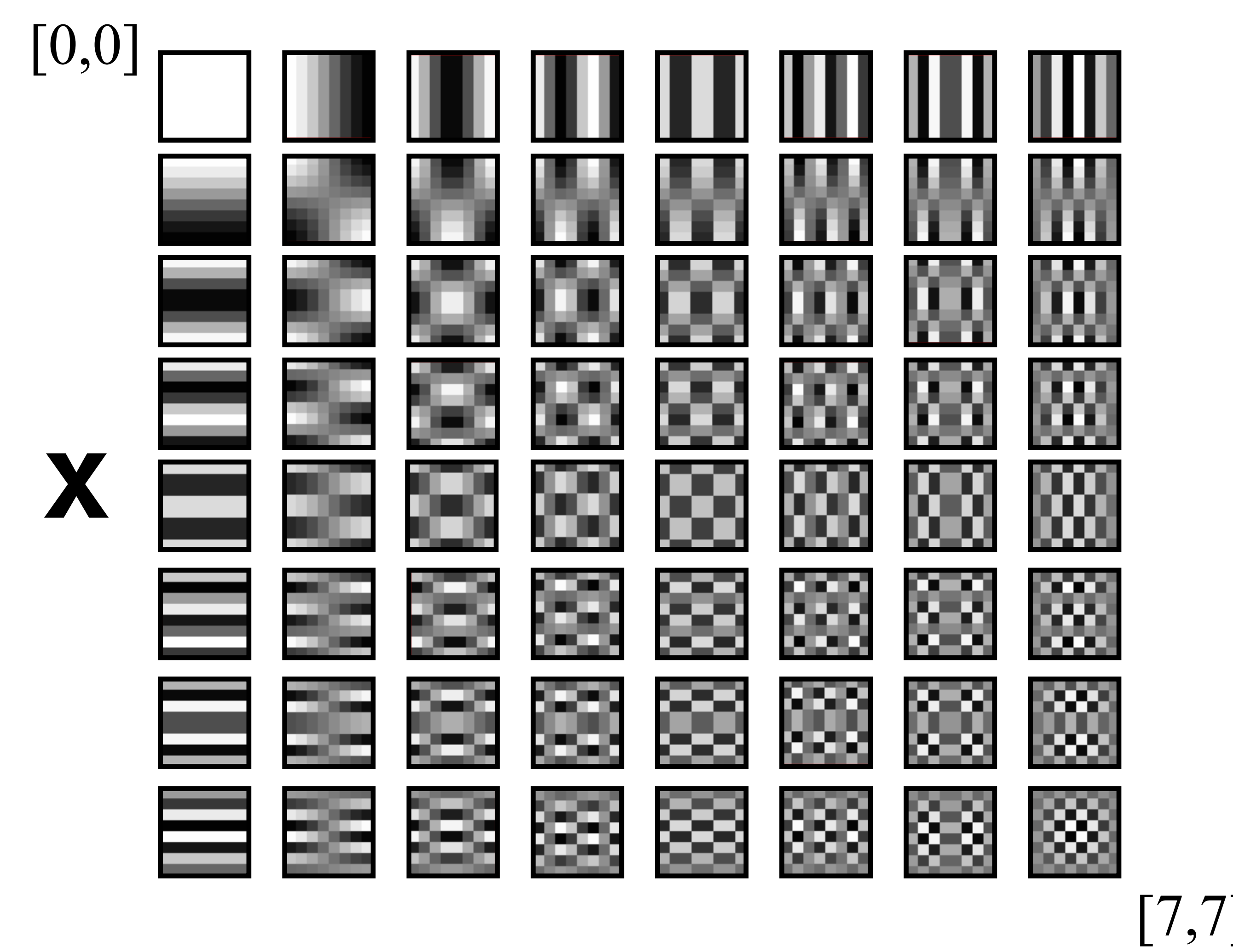
64 basis coefficients

$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

x

64 cosine basis vectors (each vector is 8x8 image)

$$\text{basis}[i,j] = \cos\left[\pi \frac{i}{N}\left(x + \frac{1}{2}\right)\right] \times \underset{j}{\cos}\left[\pi \frac{j}{N}\left(y + \frac{1}{2}\right)\right]$$

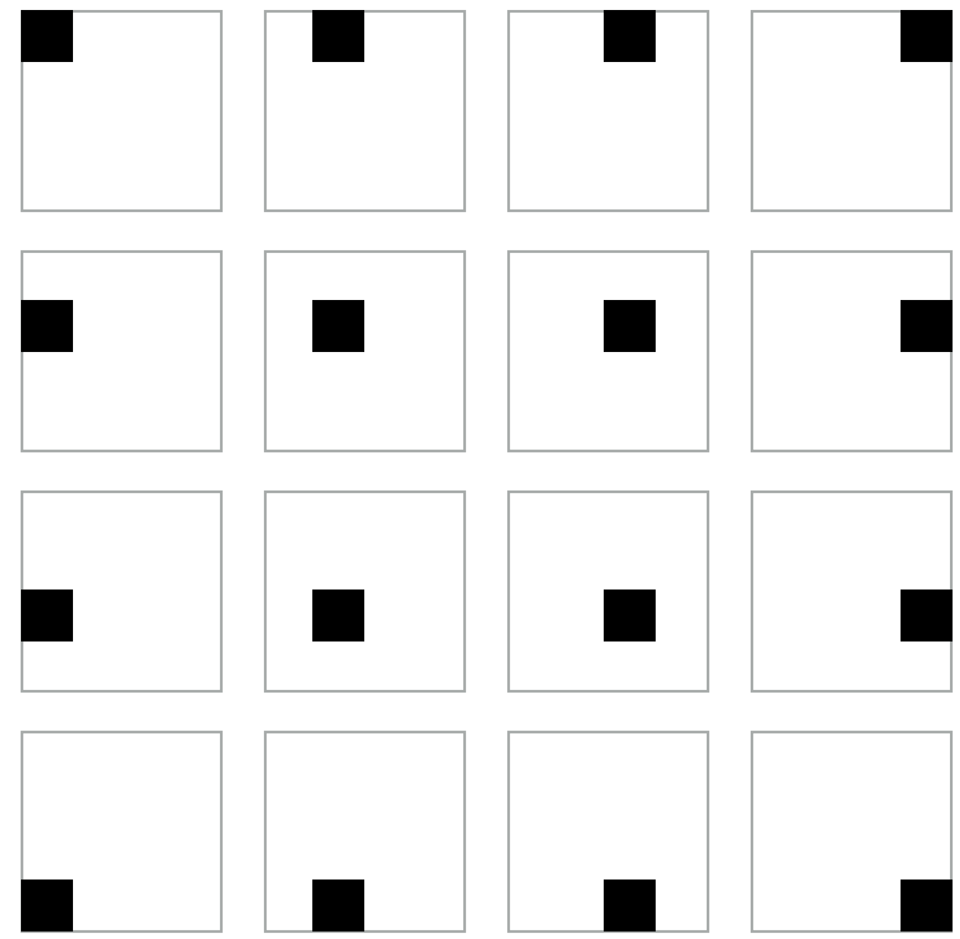


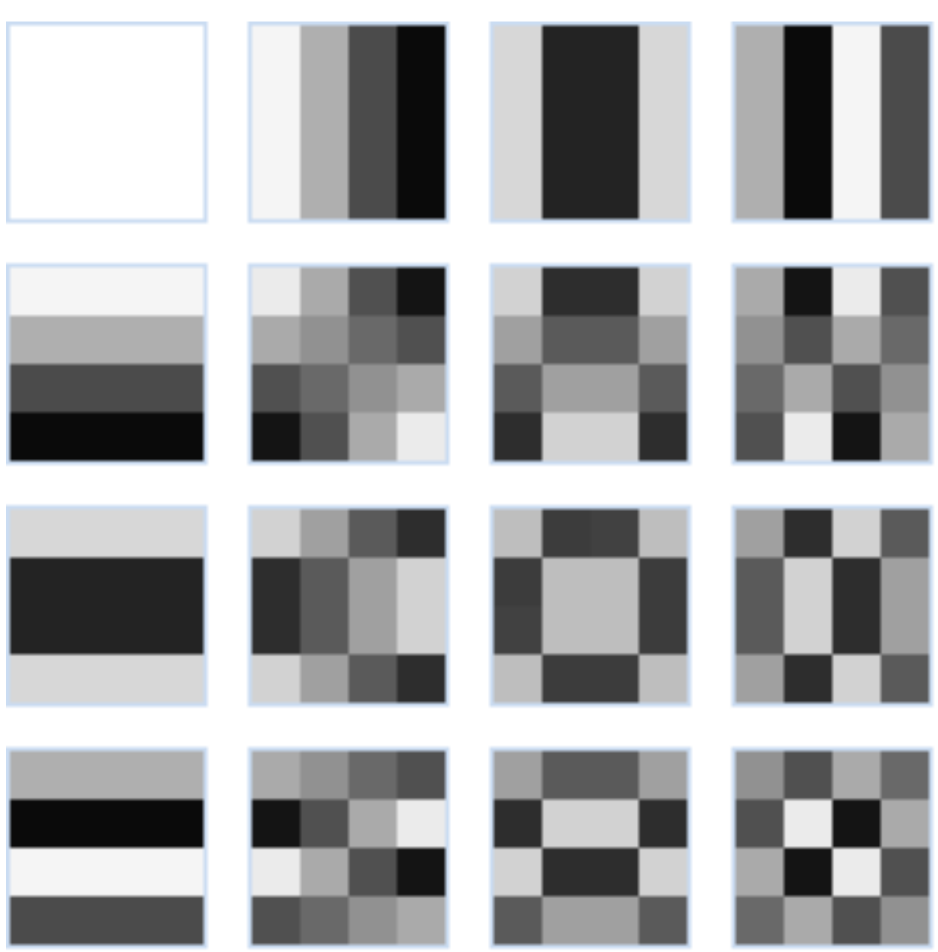
In practice: DCT applied to 8x8 pixel blocks of Y' channel, 16x16 pixel blocks of Cb, Cr (assuming 4:2:0)

Examples of other bases

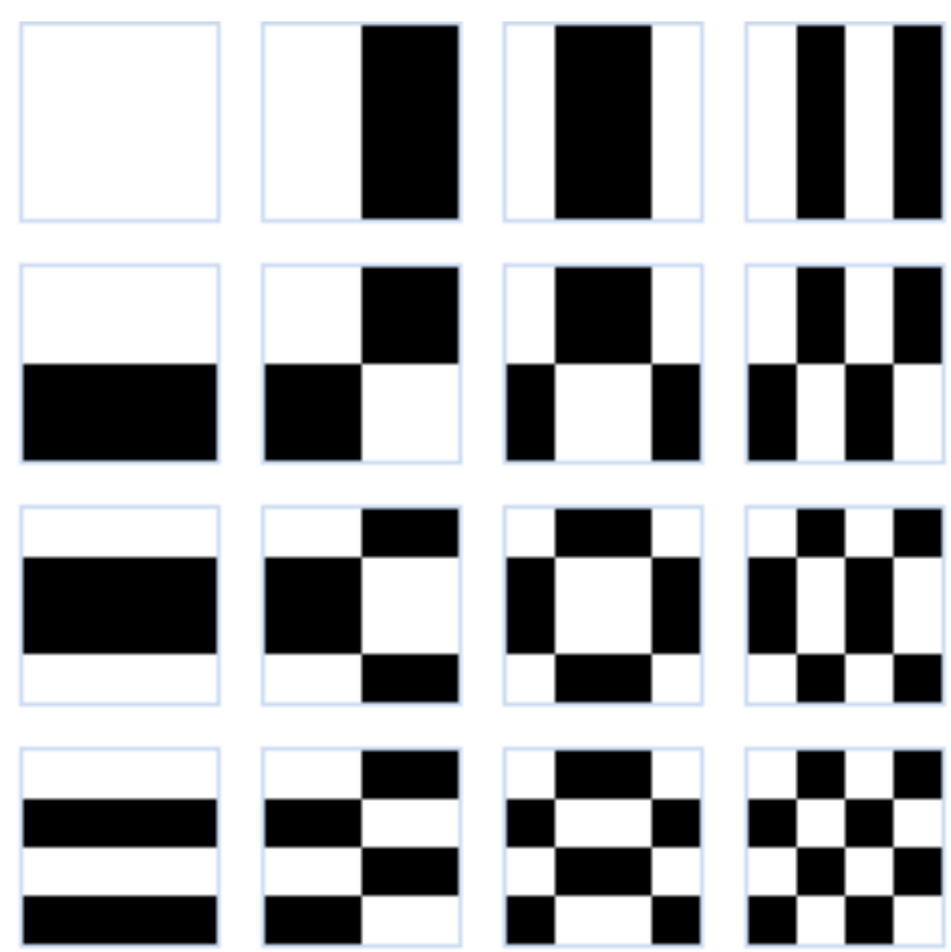
This slide illustrates basis images for 4x4 image block

Pixel Basis
(Compact: each coefficient in representation only effects a single pixel of output)

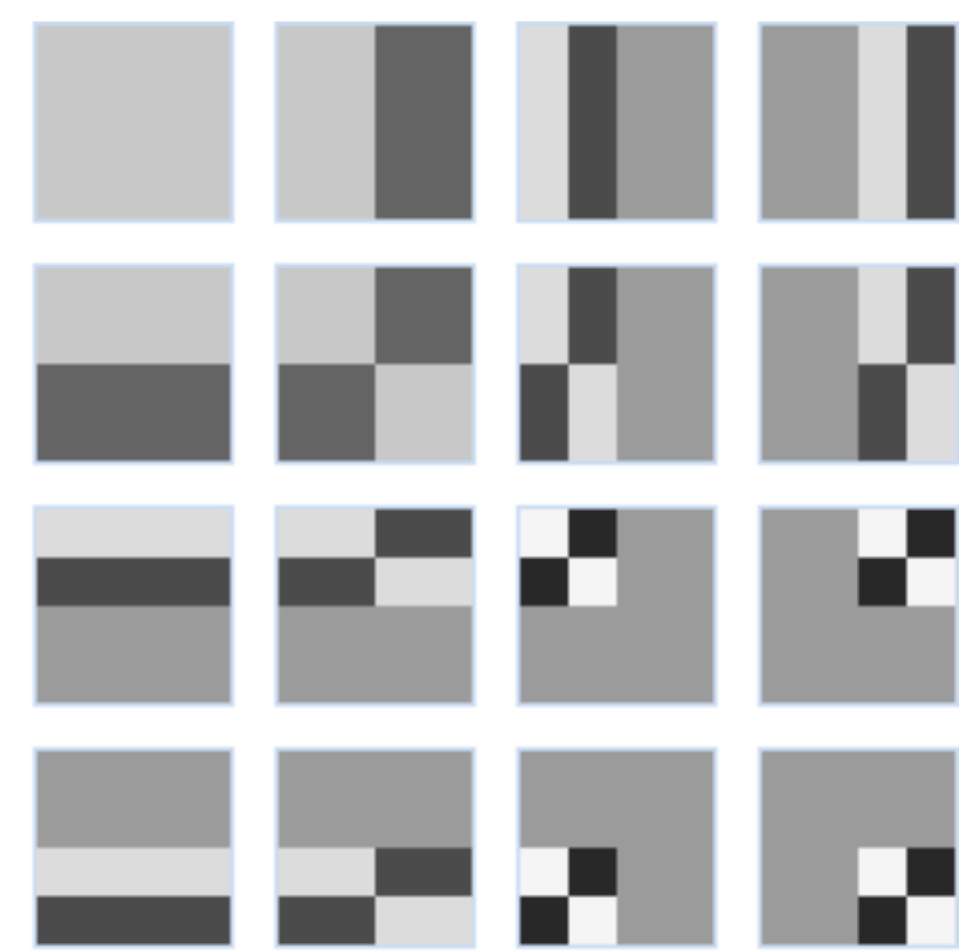




DCT



Walsh-Hadamard



Haar Wavelet

[Image credit: https://people.xiph.org/~xiphmont/demo/daala/demo3.shtml]

Quantization

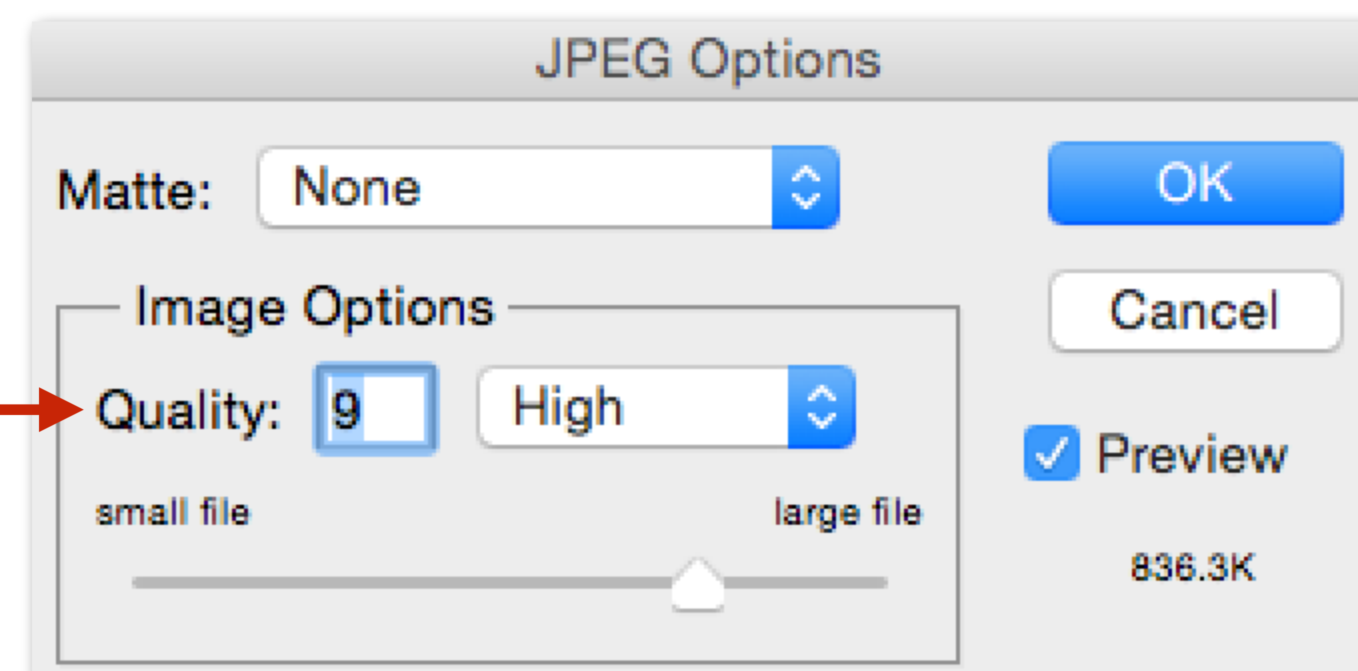
$$
\begin{bmatrix}
-415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\
4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\
-47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\
-49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\
12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\
-8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\
-1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\
0 & 0 & -1 & -4 & -1 & 0 & 1 & 2
\end{bmatrix}
\Bigg/
\begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99
\end{bmatrix}
$$

Result of DCT
(representation of image in cosine basis)

Quantization Matrix

$$
= \begin{bmatrix}
-26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\
0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\
-3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\
-4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Changing JPEG quality setting in your favorite photo app modifies this matrix ("lower quality" = higher values for elements in quantization matrix)



Quantization produces small values for coefficients (only few bits needed per coefficient)

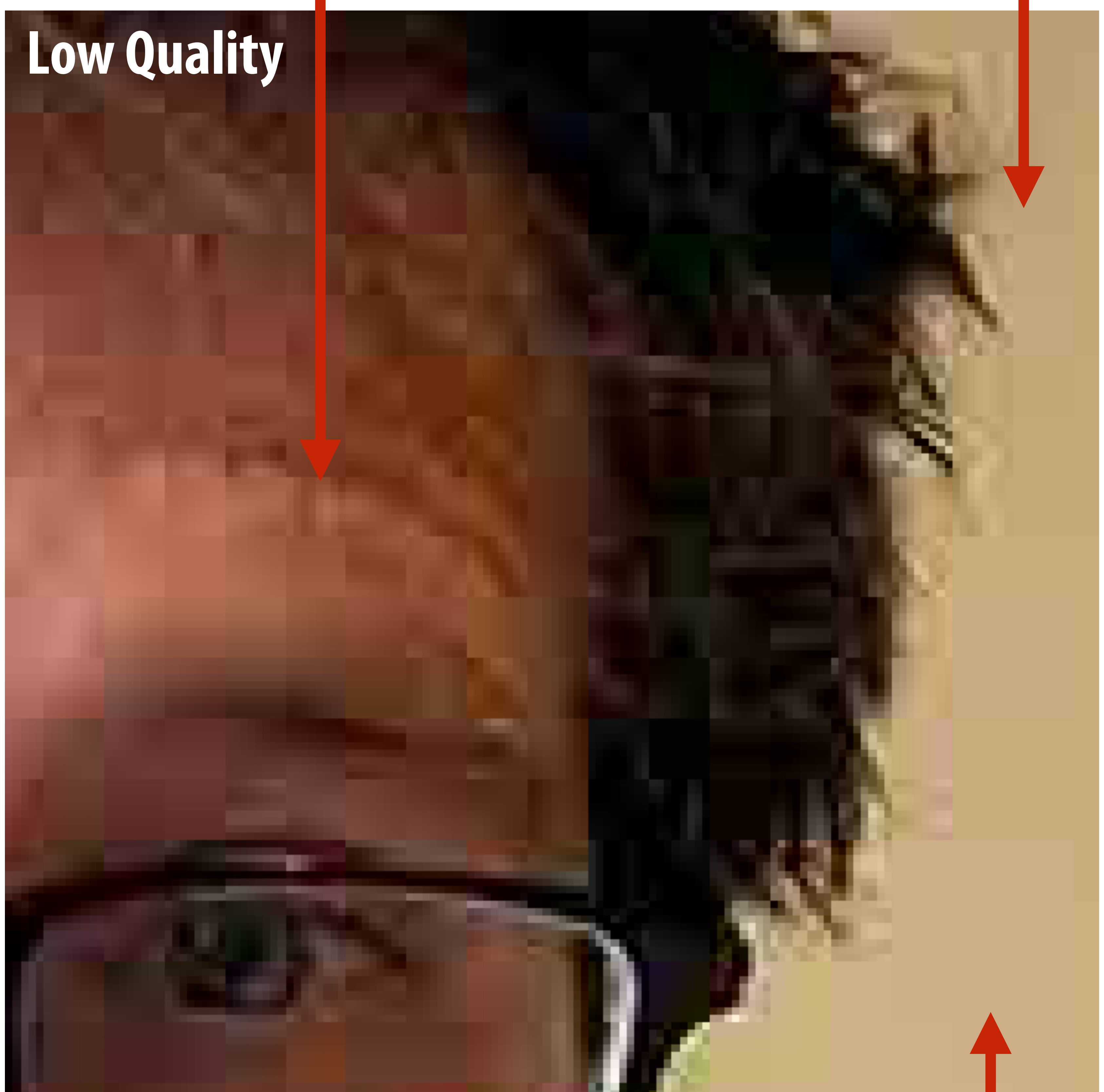
Quantization zeros out many coefficients

[Credit: Pat Hanrahan]

JPEG compression artifacts

Noticeable 8x8 pixel block boundaries

Noticeable error near high gradients



Low-frequency regions of image represented accurately even under high compression

JPEG compression artifacts

Original Image
(actual size)

Original Image

Quality Level 9

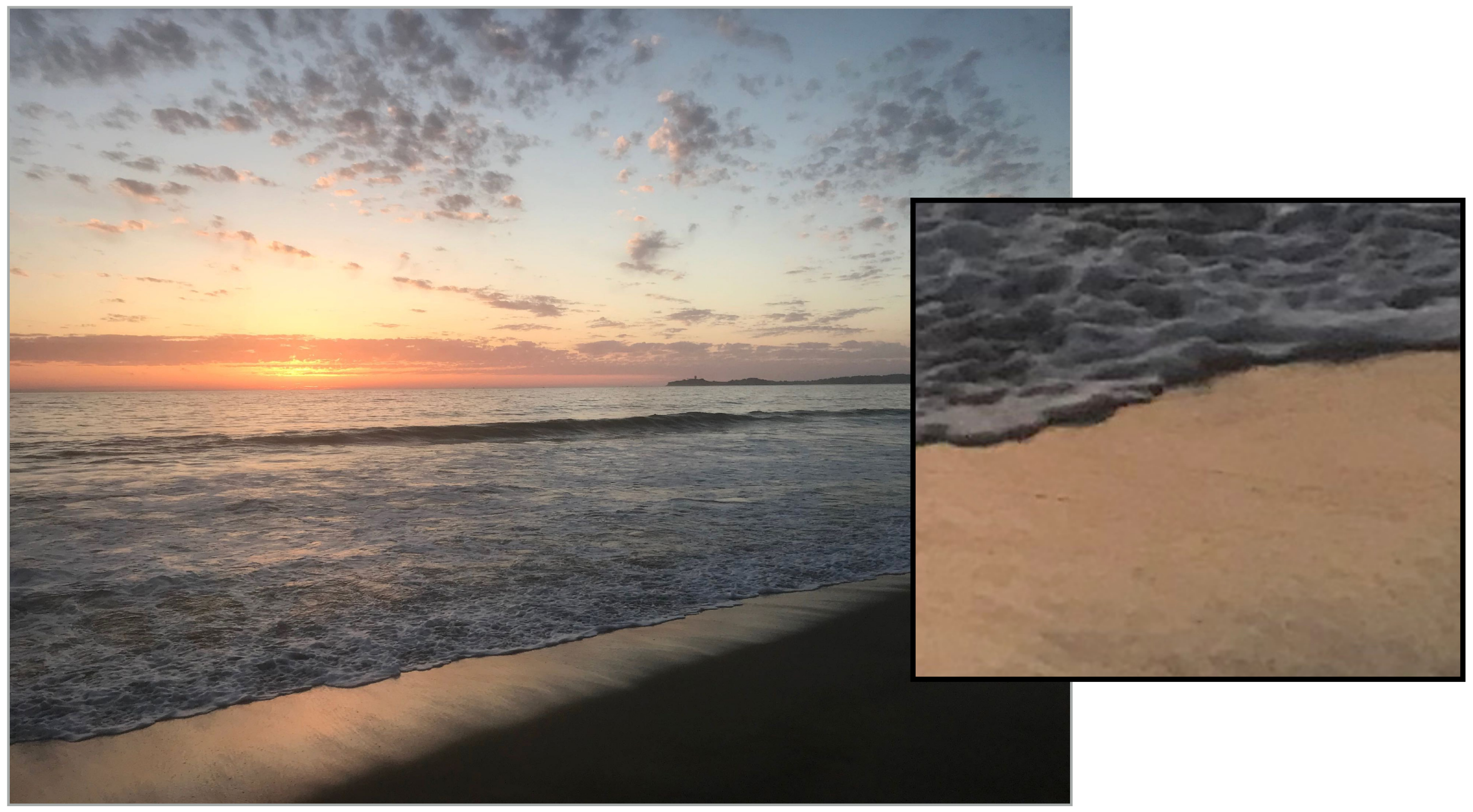
Quality Level 6

Quality Level 3

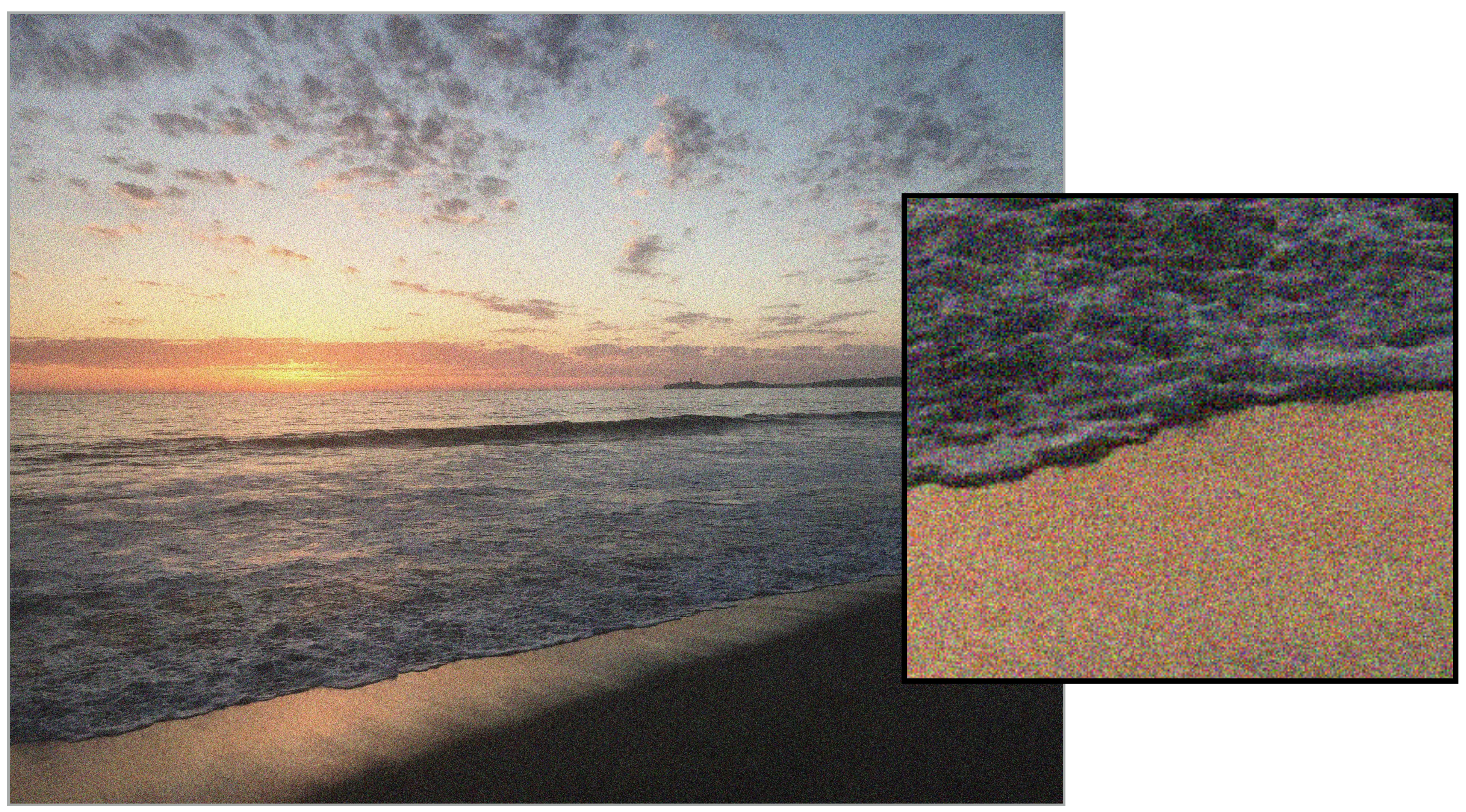
Quality Level 1

Why might JPEG compression not be a good compression scheme for illustrations and rasterized text?

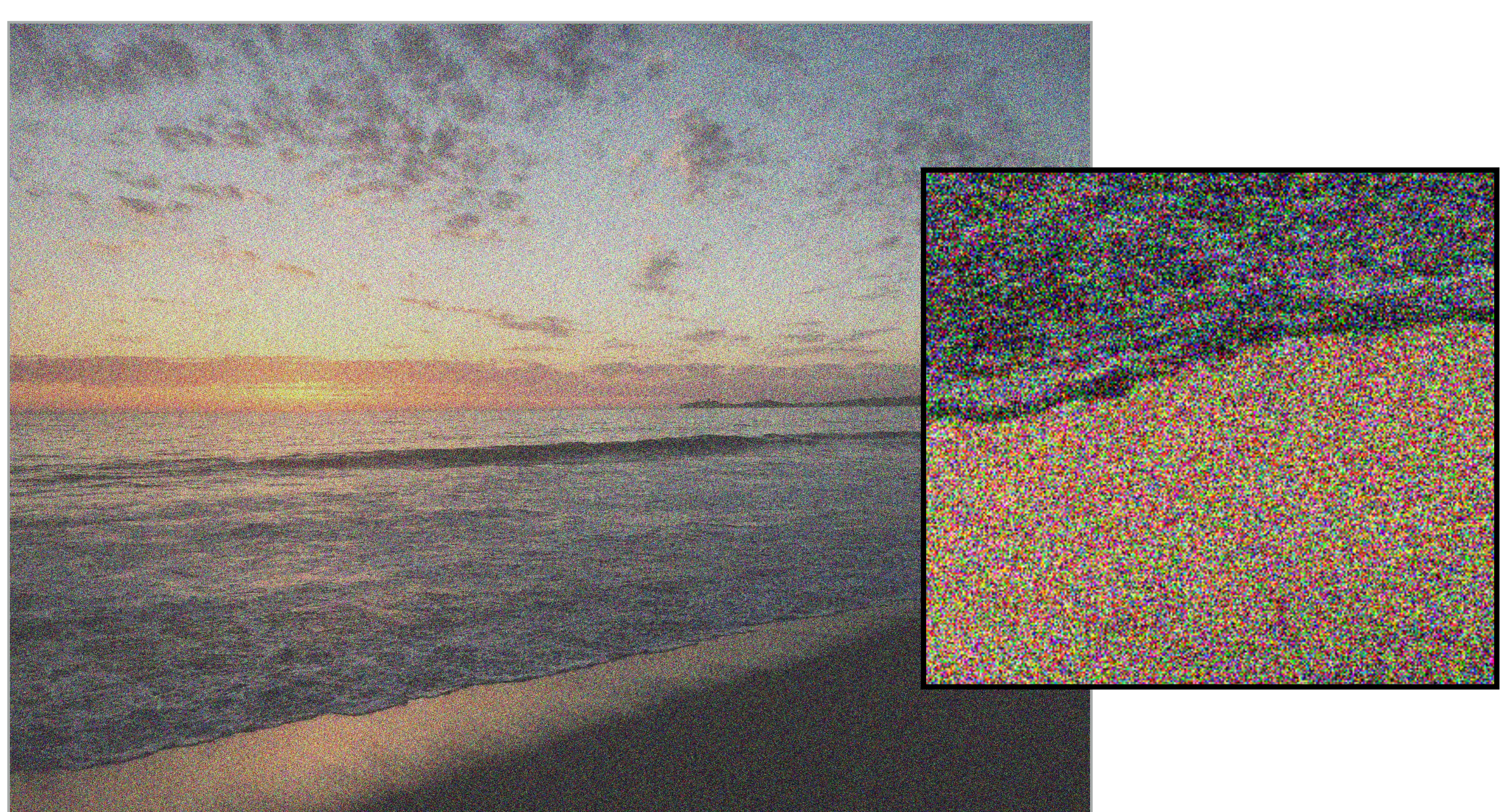
Images with high frequency content do not exhibit as high compression ratios. Why?



Original image: 2.9MB JPG



Medium noise: 22.6 MB JPG



High noise: 28.9 MB JPG

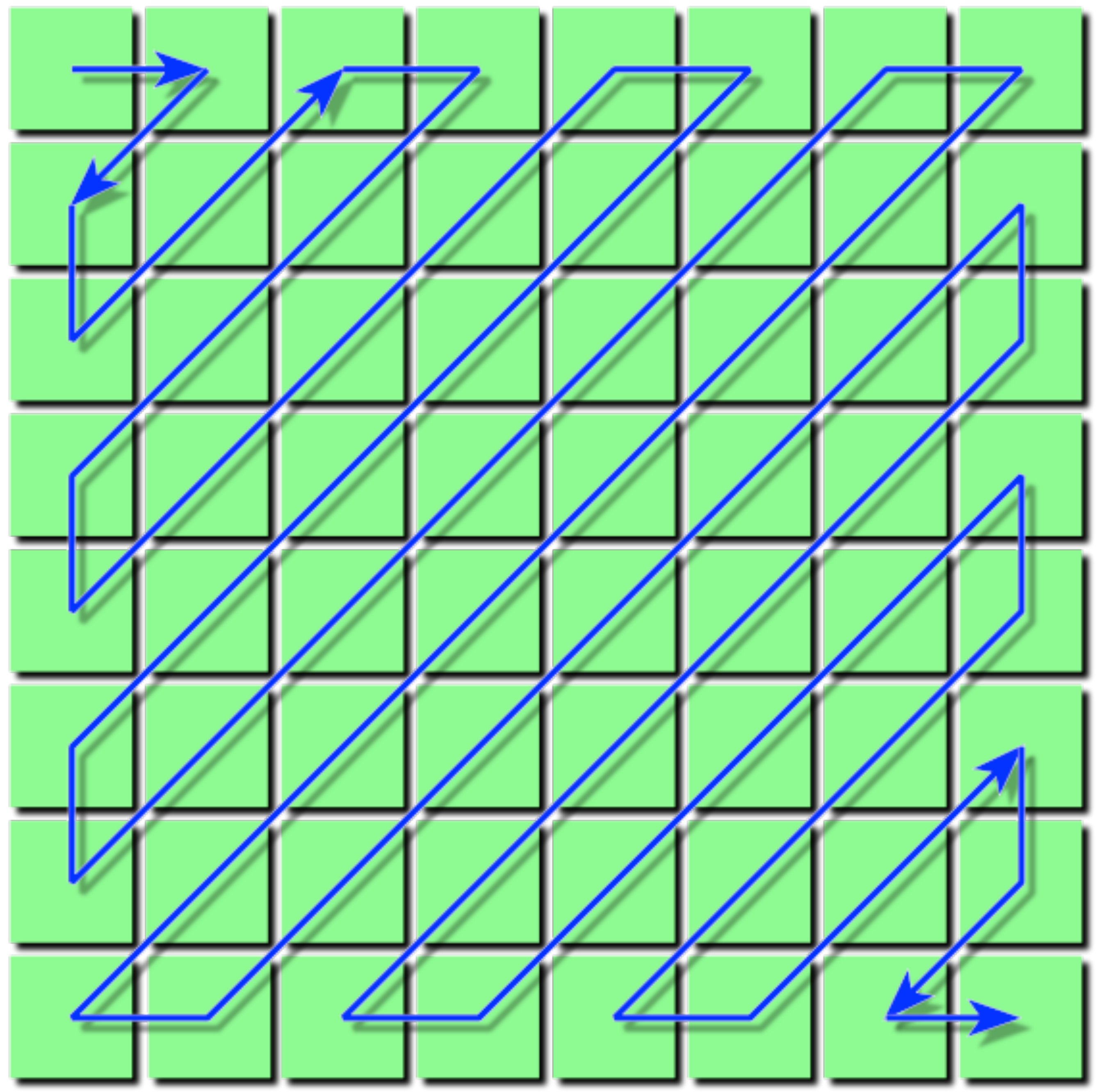
Photoshop JPG compression level = 10 used for all compressed images

Uncompressed image:
4032 x 3024 x 24 bytes/pixel = 36.6 MB

Lossless compression of quantized DCT values

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Quantized DCT Values



Reordering

Entropy encoding: (lossless)

- **Reorder values**
- **Run-length encode (RLE) 0's**
- **Huffman encode non-zero values**

Image credit: Wikipedia

JPEG compression summary

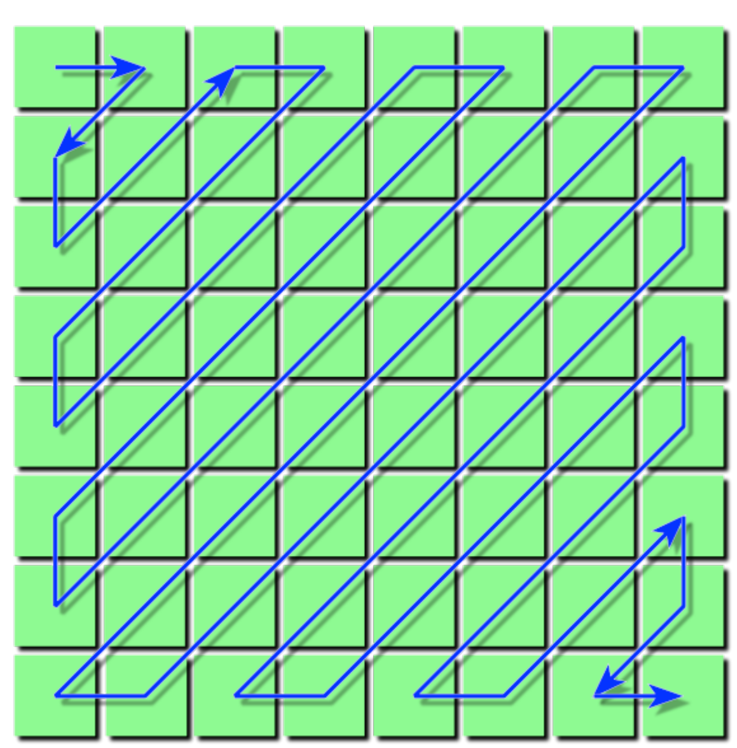
$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix} \Big/ \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

DCT / **Quantization Matrix**

$$= \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Quantized DCT

Quantization loses information (lossy compression!)



Coefficient reordering

RLE compression of zeros

Entropy compression of non-zeros

Lossless compression!

Compressed bits

Credit: Pat Hanrahan

JPEG compression summary

Convert image to Y'CbCr

Downsample CbCr (to 4:2:2 or 4:2:0) (information loss occurs here)

For each color channel (Y', Cb, Cr):

- For each 8x8 block of values
  - Compute DCT
  - Quantize results (information loss occurs here)
  - Reorder values
  - Run-length encode 0-spans
  - Huffman encode non-zero values

Key idea: exploit characteristics of human perception to build efficient image storage and image processing systems

- **Separation of luminance from chrominance in color representation (Y'CrCb) allows reduced resolution in chrominance channels (4:2:0)**
- **Encode pixel values linearly in lightness (perceived brightness), not in luminance (distribute representable values uniformly in perceptual space)**
- **JPEG compression significantly reduces file size at cost of quantization error in high spatial frequencies**
  - **Human brain is more tolerant of errors in high frequency image components than in low frequency ones**
  - **Images of the real world are dominated by low-frequency components**

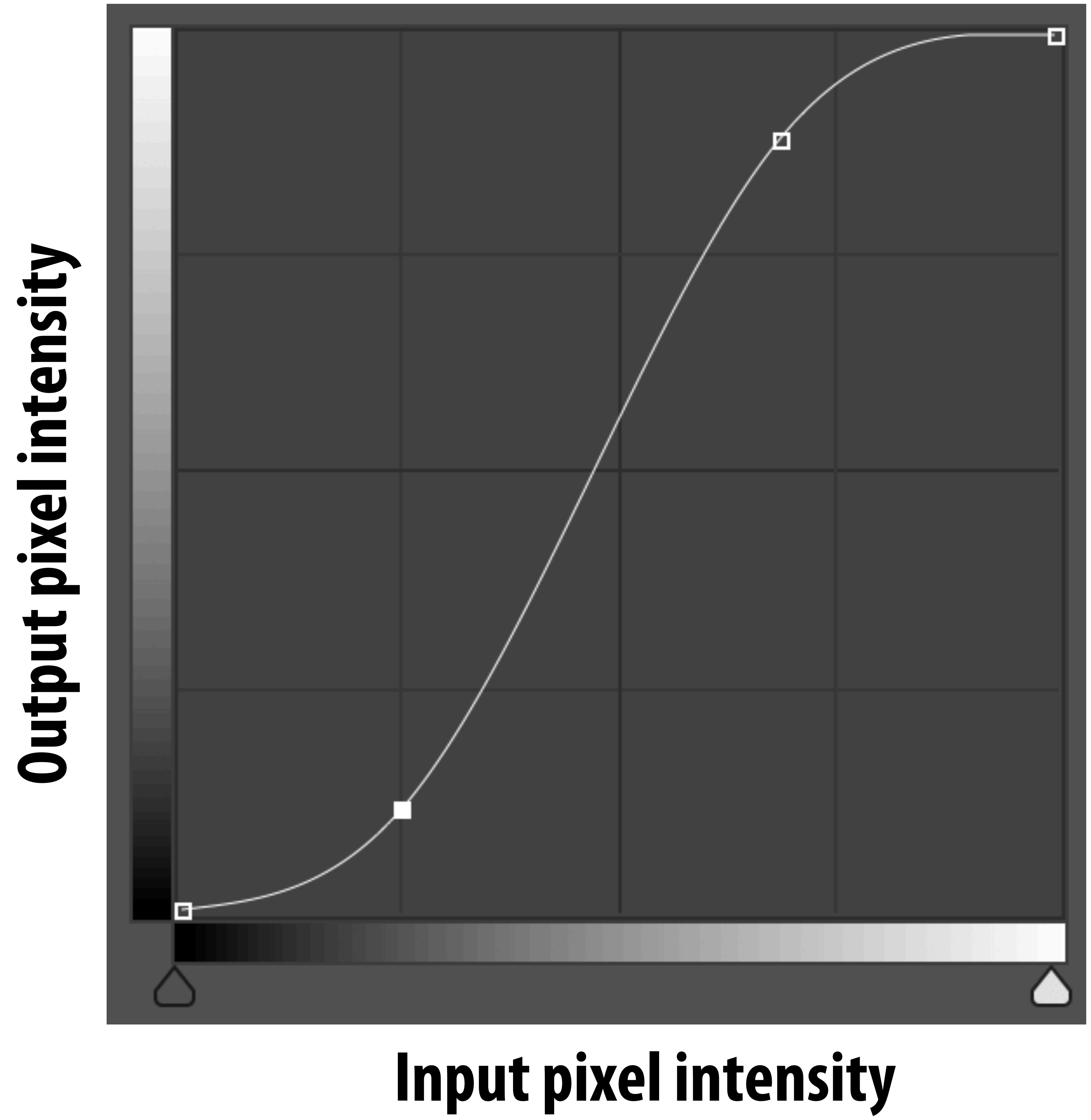
Image processing basics

Example image processing operations

Increase contrast

Increasing contrast with “S curve”

- Per-pixel operation
- output(x,y) = f(input(x,y))



Example image processing operations

Image Invert:

out(x,y) = 1 - in(x,y)

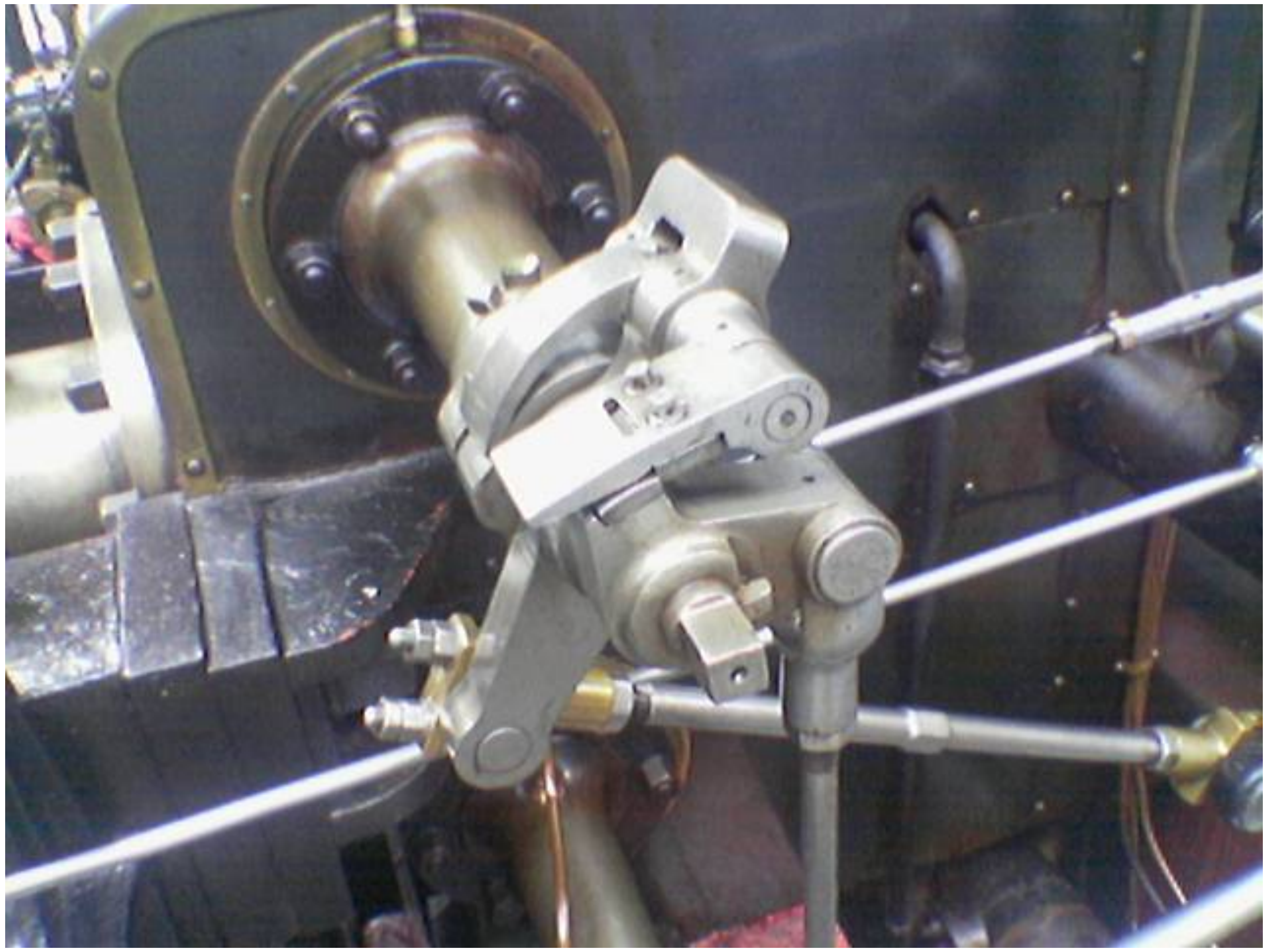
Example image processing operations

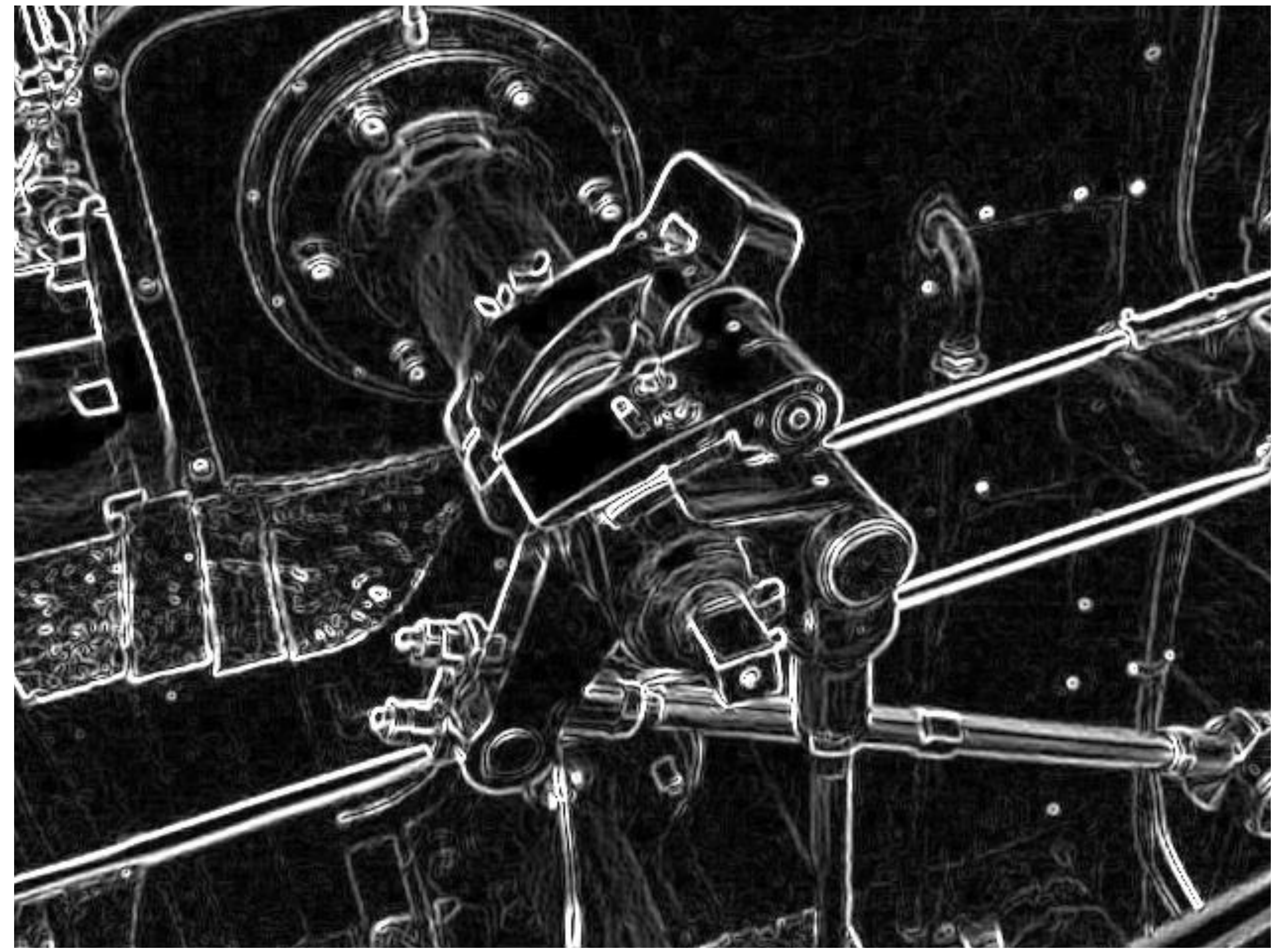
Blur

Example image processing operations

Sharpen

Edge detection





A “smarter” blur (doesn’t blur over edges)

Review: convolution

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y) g(x-y) dy$$

output signal

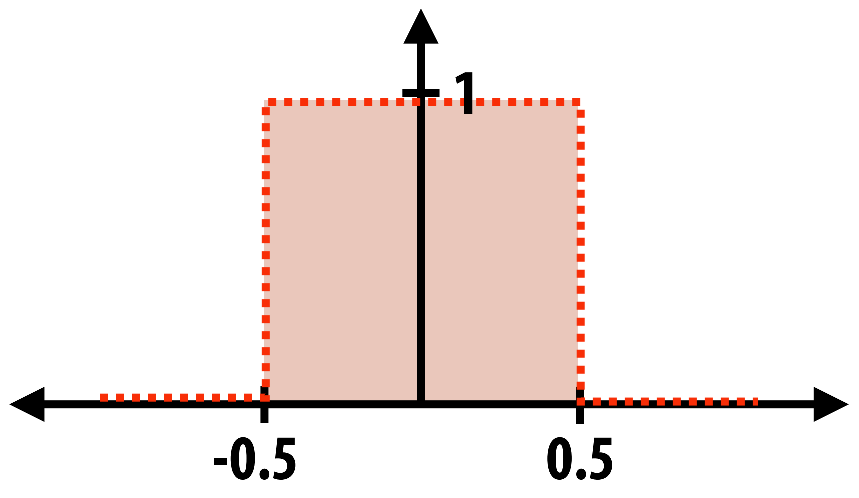
filter

input signal

It may be helpful to consider the effect of convolution with the simple unit-area "box" function:

$$f(x) = \begin{cases} 1 & |x| \leq 0.5 \\ 0 & otherwise \end{cases}$$

$$(f * g)(x) = \int_{-0.5}^{0.5} g(x-y) dy$$



f * *g* is a "blurred" version of *g*

Discrete 2D convolution

$$(f * g)(x, y) = \sum_{i,j=-\infty}^{\infty} f(i,j) I(x-i, y-j)$$

output image — **filter** — **input image**

Consider $f(i,j)$ **that is nonzero only when:** $-1 \leq i, j \leq 1$

Then:

$$(f * g)(x, y) = \sum_{i,j=-1}^{1} f(i,j) I(x-i, y-j)$$

And we can represent f(i,j) as a 3x3 matrix of values where:

$$f(i,j) = \mathbf{F}_{i,j}$$

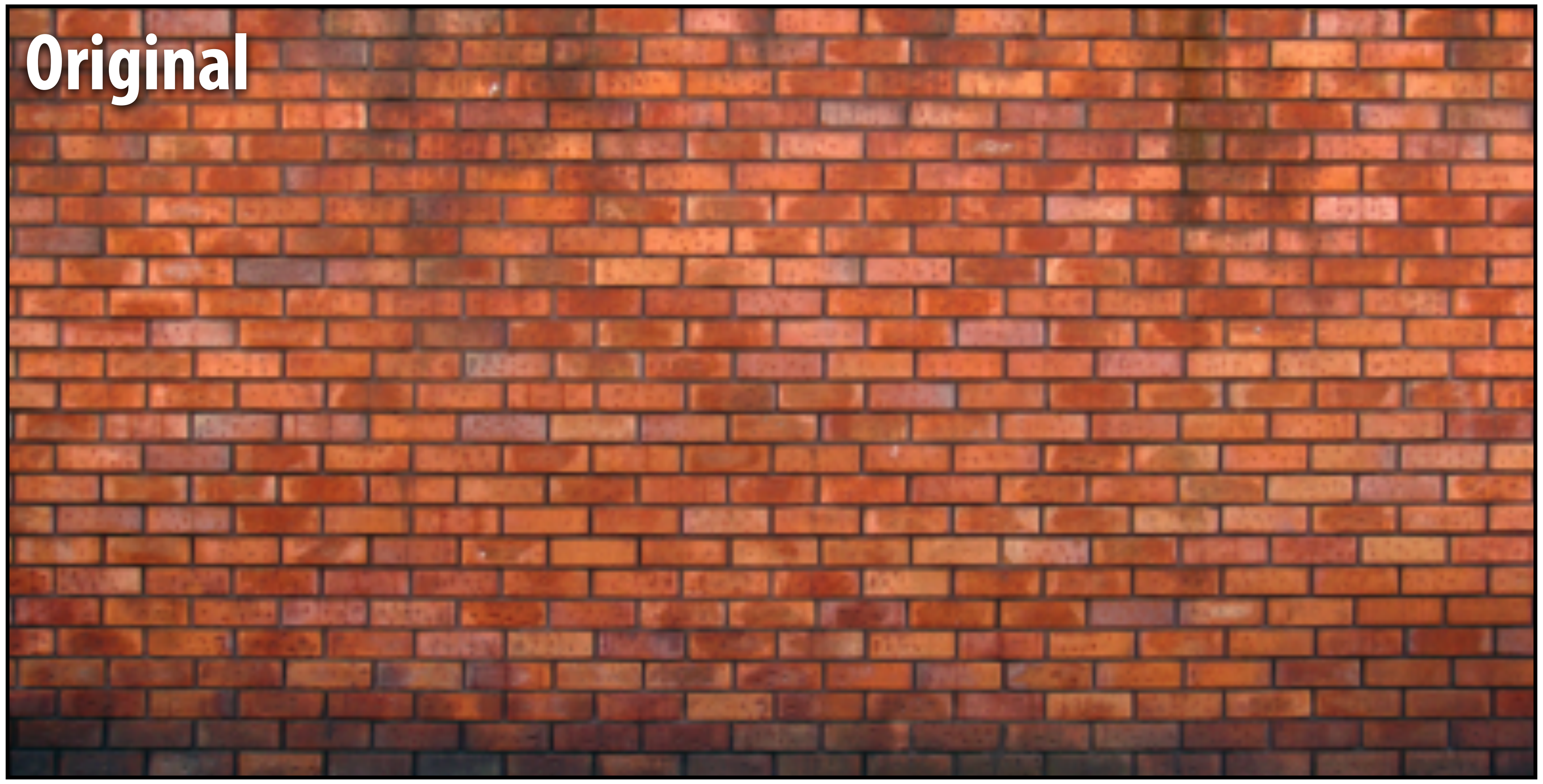
(often called: "filter weights", "filter kernel")

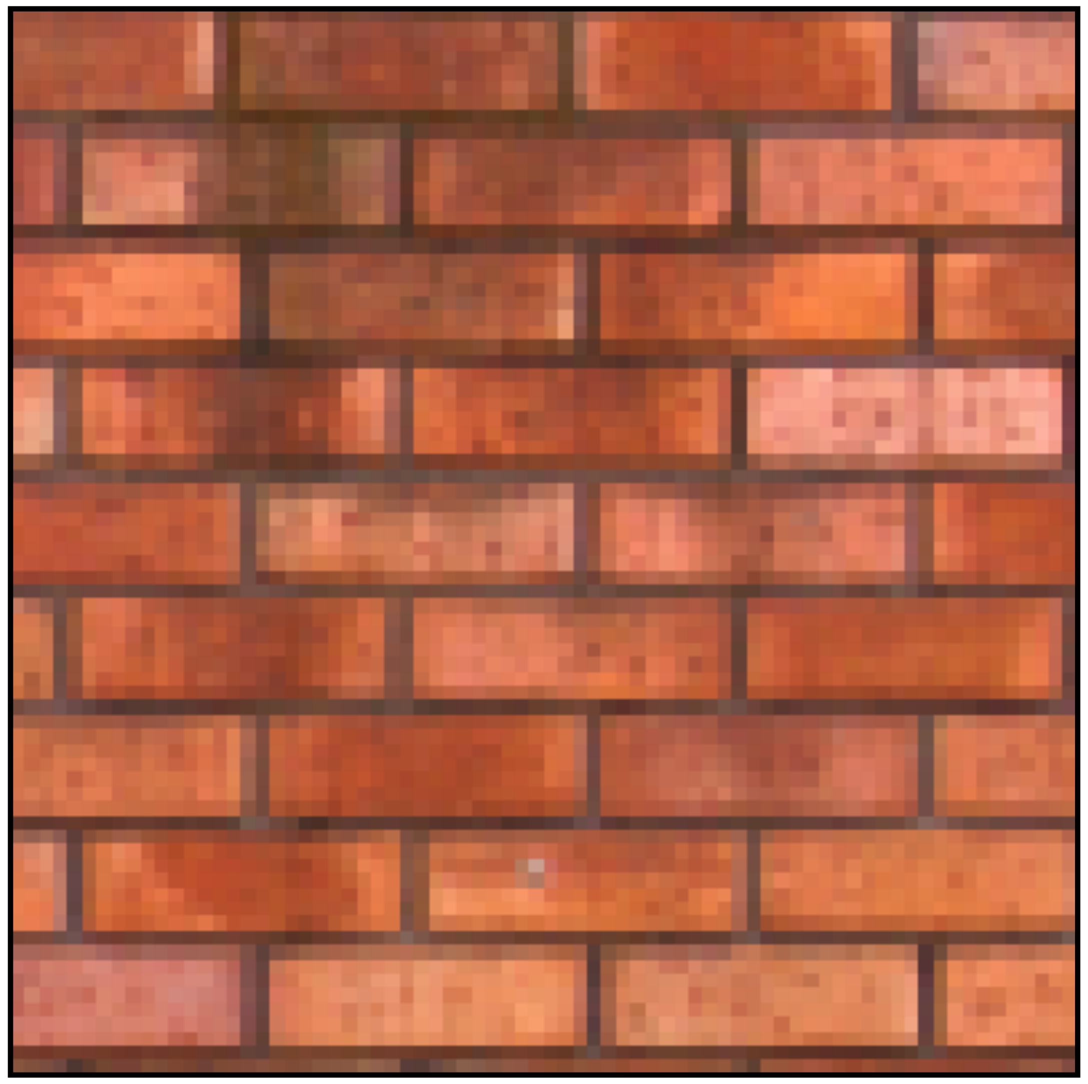
Simple 3x3 box blur

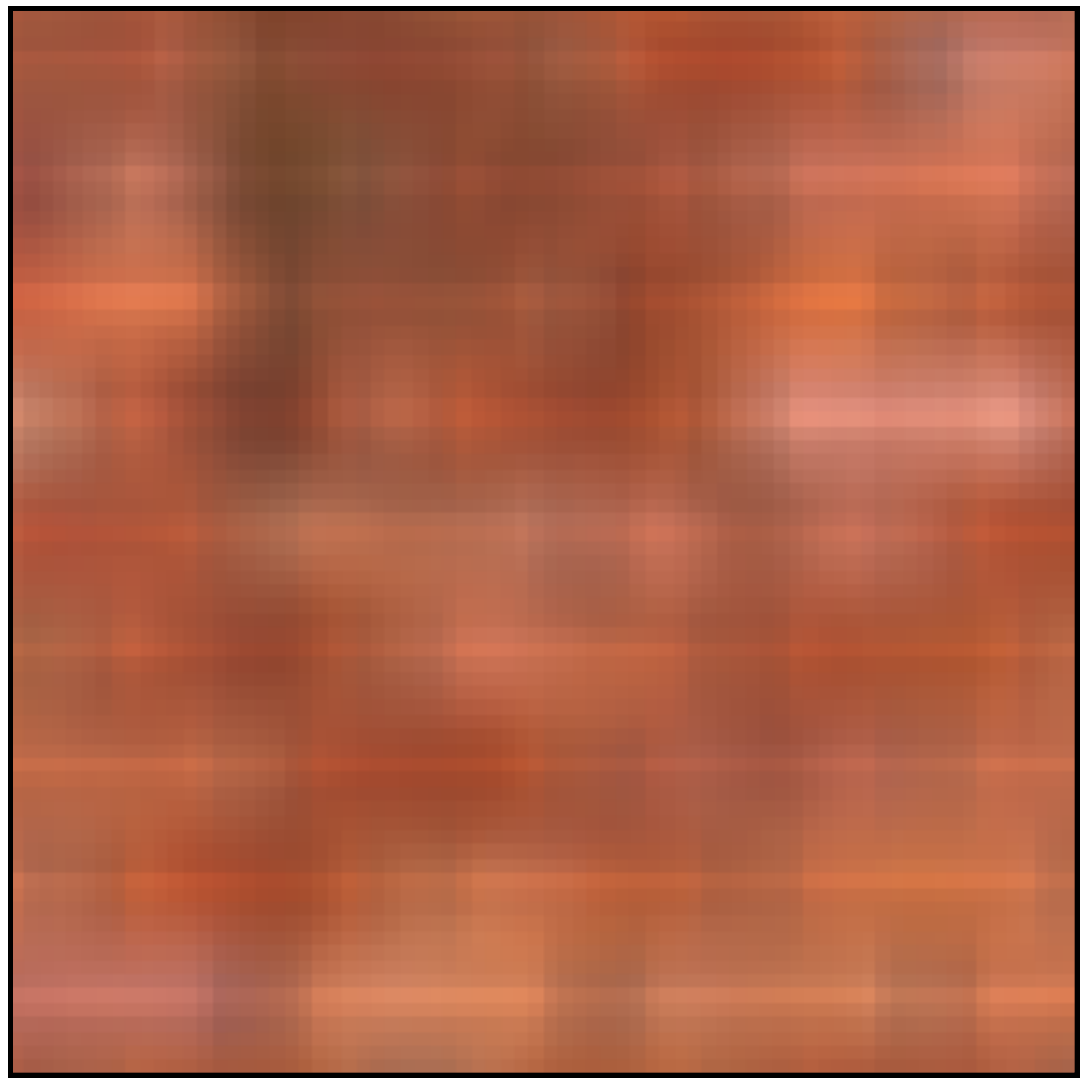
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9};

for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
         for (int ii=0; ii<3; ii++)
            tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH + i] = tmp;
  }
}
```

For now: ignore boundary pixels and assume output image is smaller than input (makes convolution loop bounds much simpler to write)

7x7 box blur







Gaussian blur

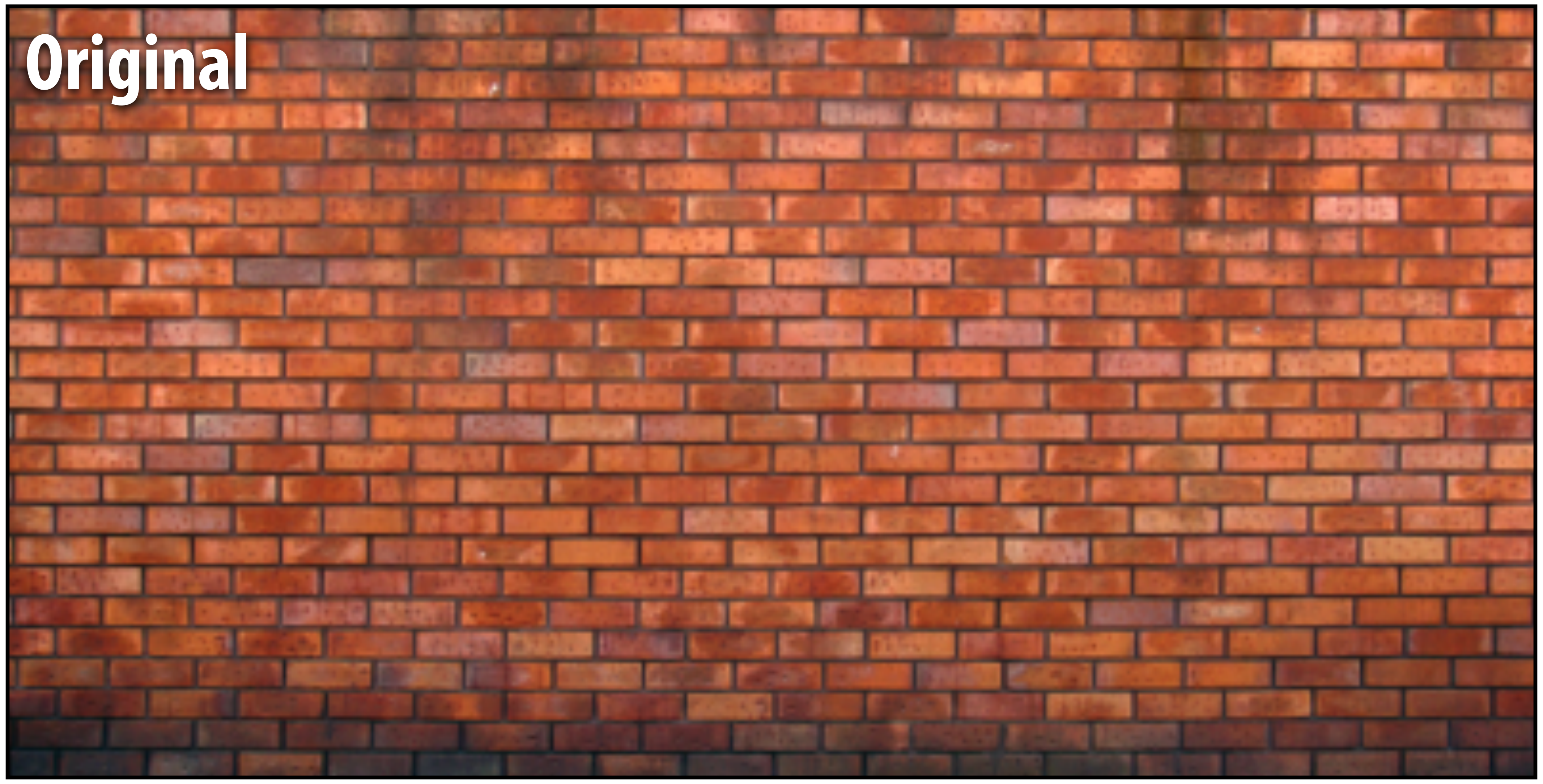
- **Obtain filter coefficients from sampling 2D Gaussian function**

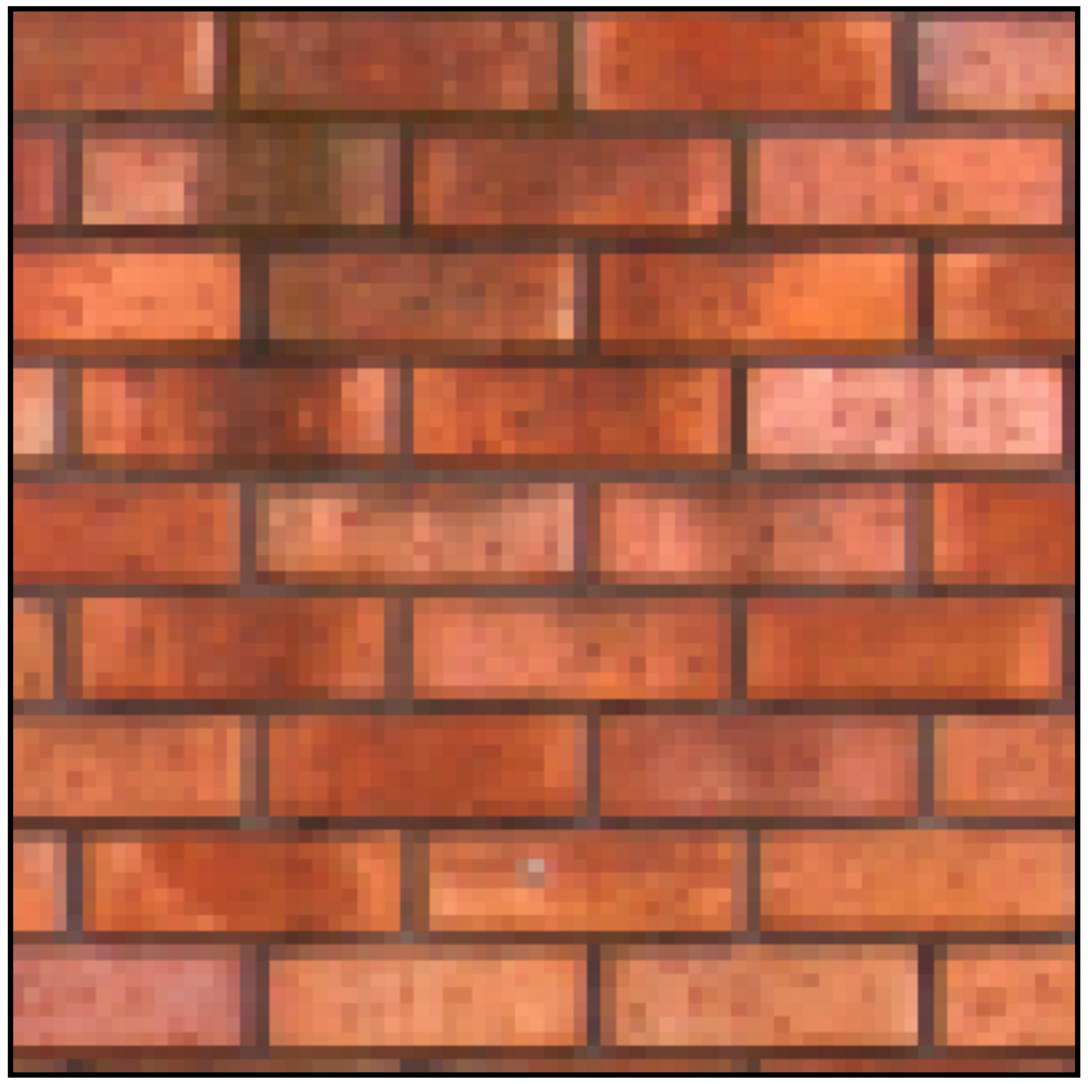
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

7x7 gaussian blur



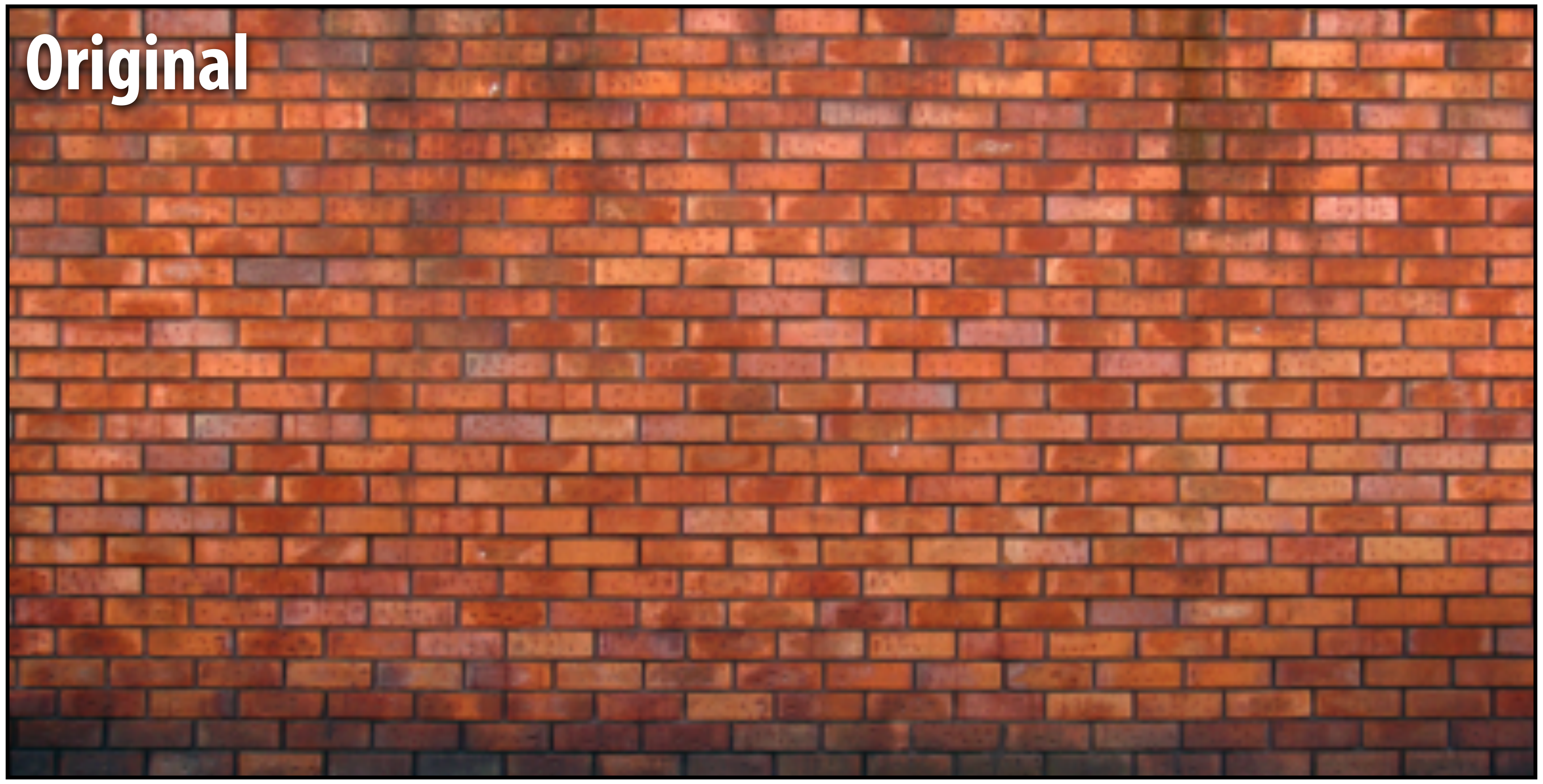


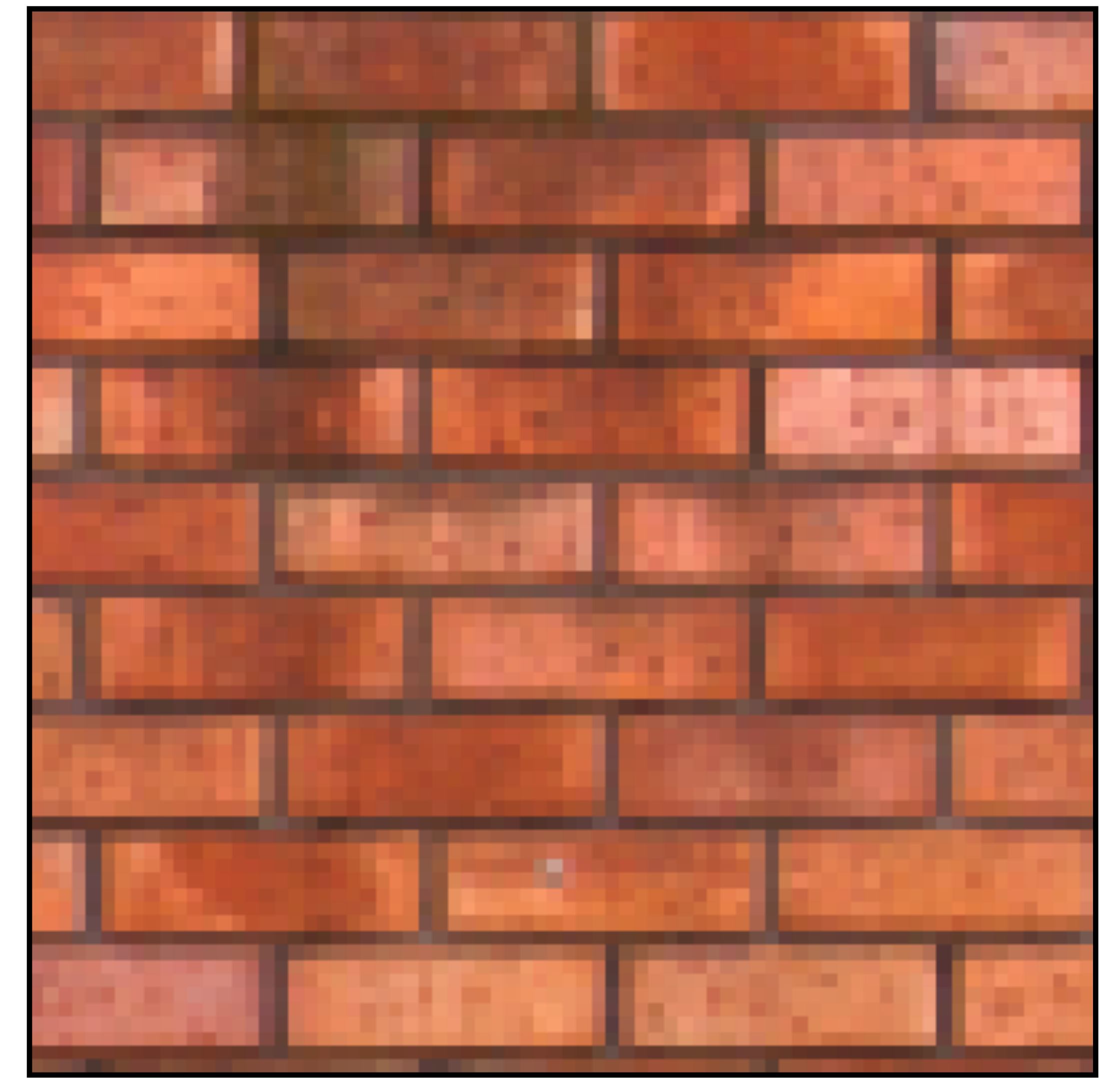
What does convolution with this filter do?

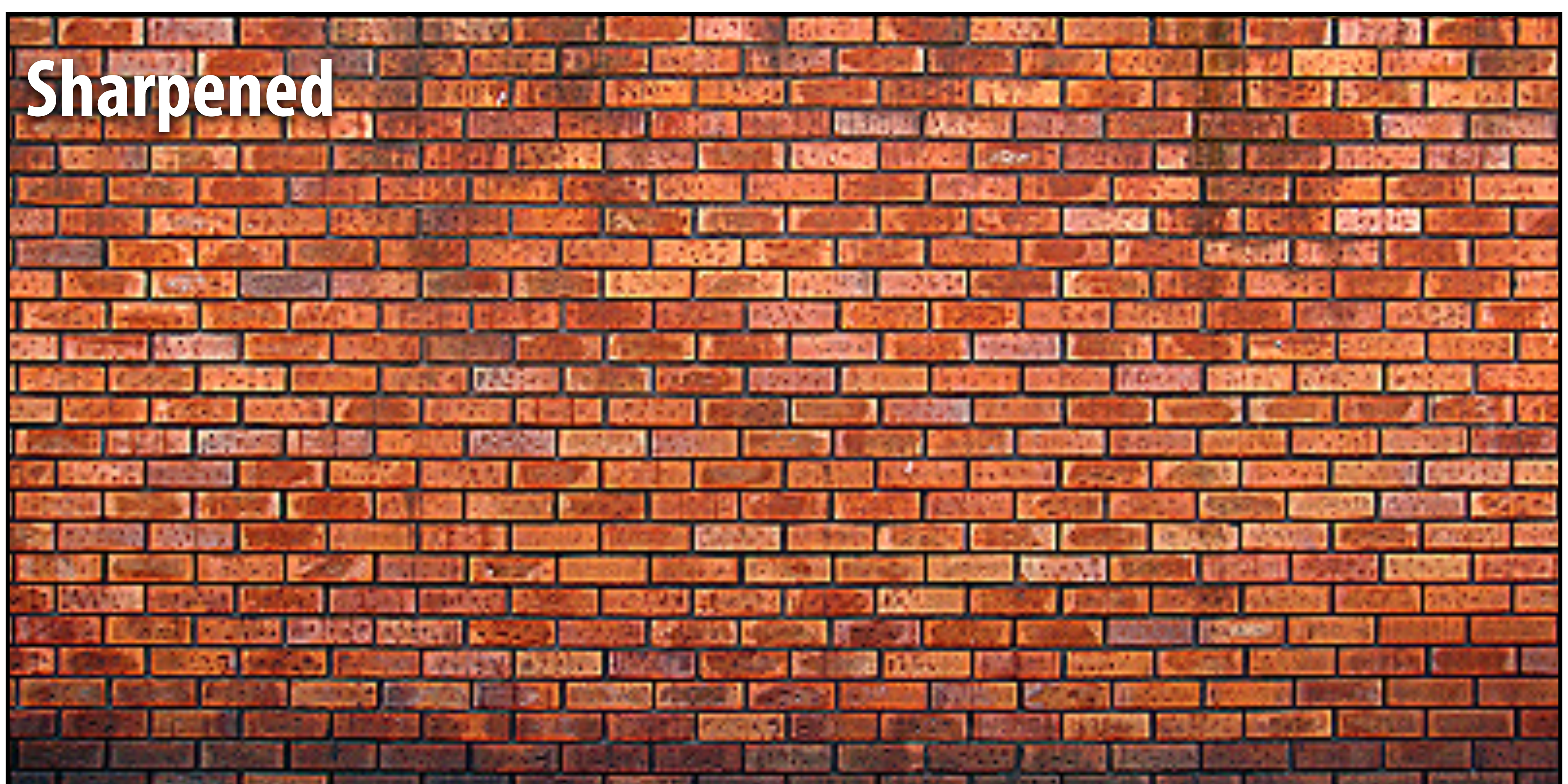
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

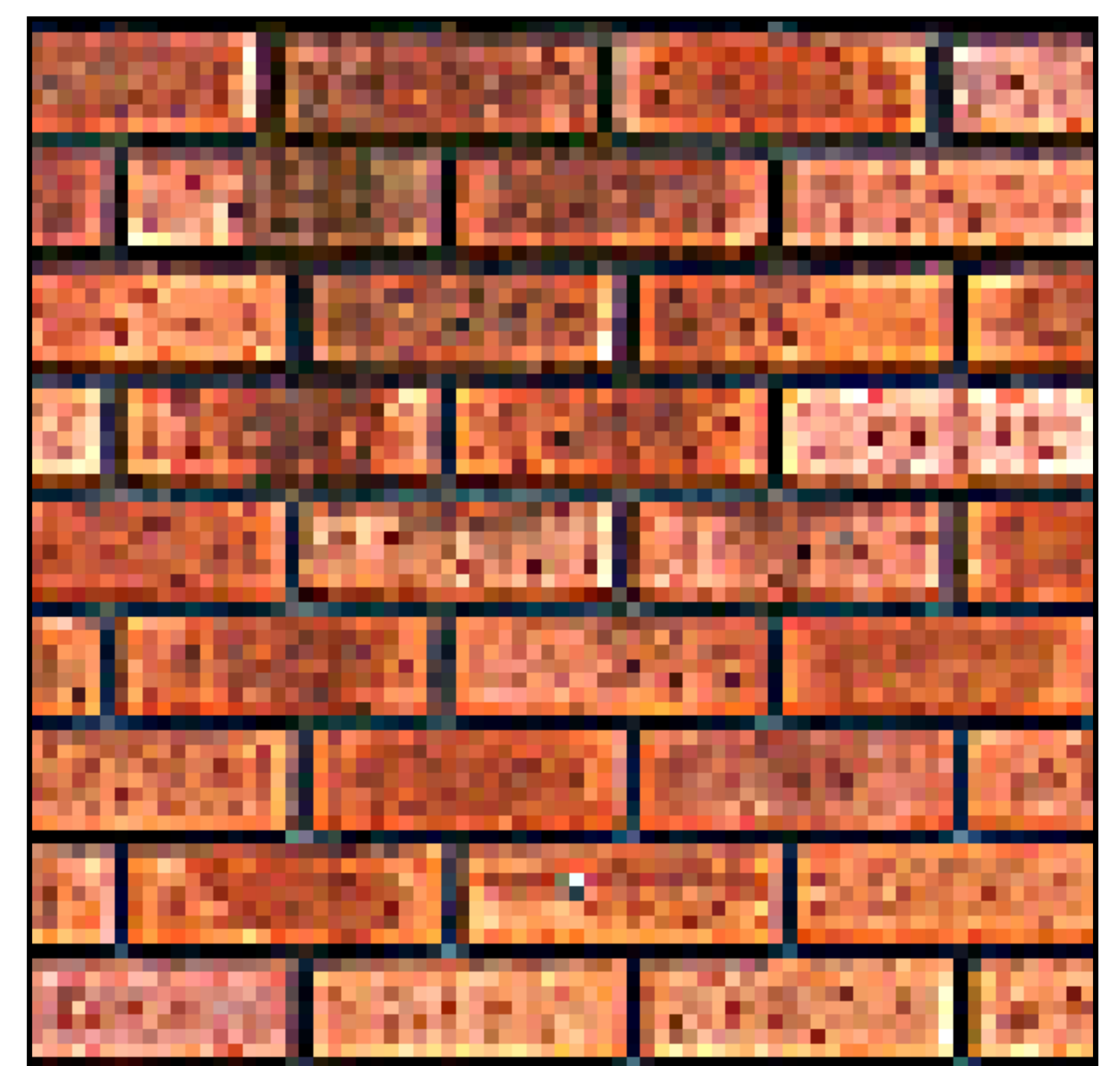
Sharpens image!

3x3 sharpen filter









What does convolution with these filters do?

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

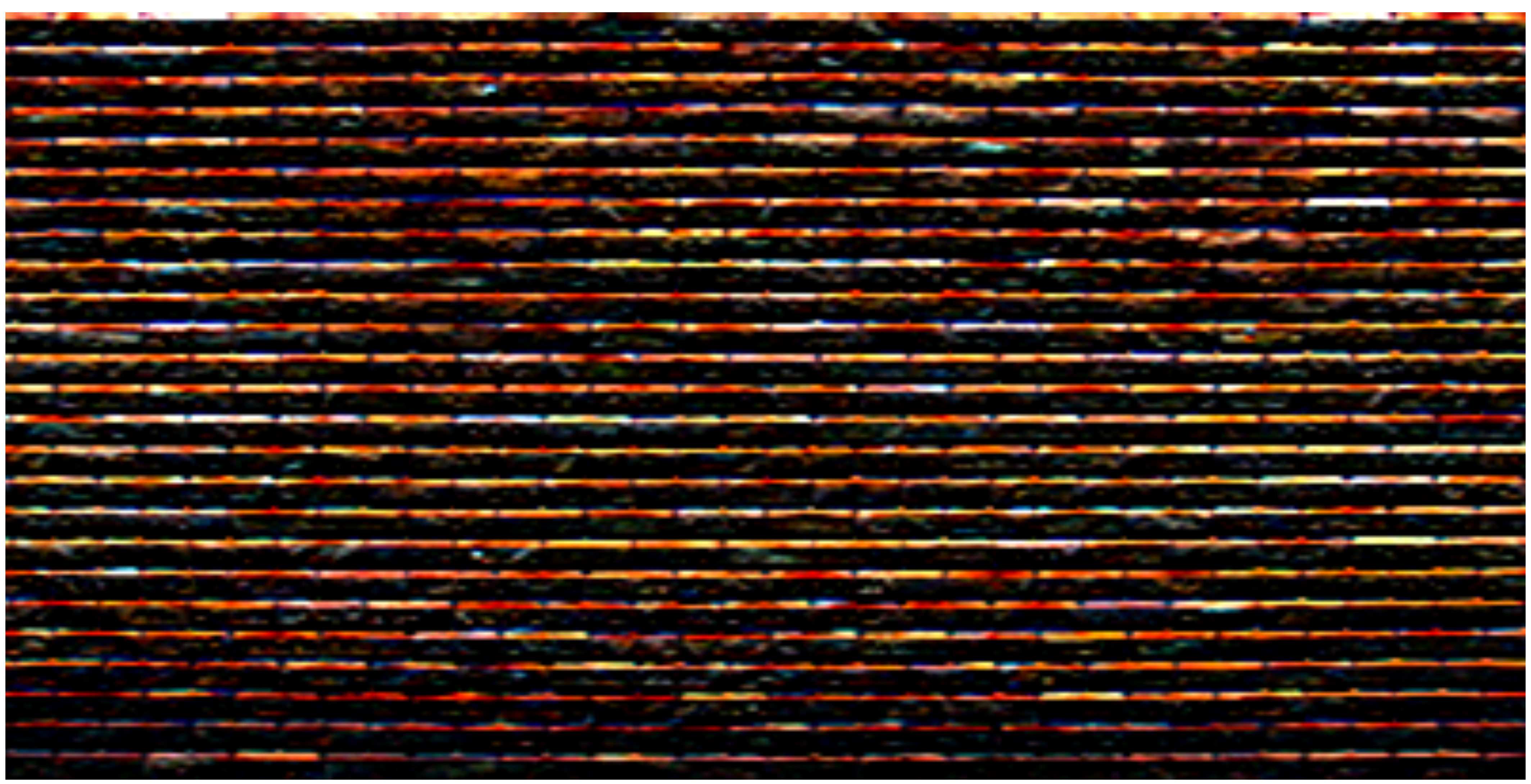
Extracts horizontal gradients

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

Extracts vertical gradients

Gradient detection filters

Horizontal gradients



Vertical gradients

Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image (this is a common interpretation in computer vision)

Sobel edge detection

- **Compute gradient response images**

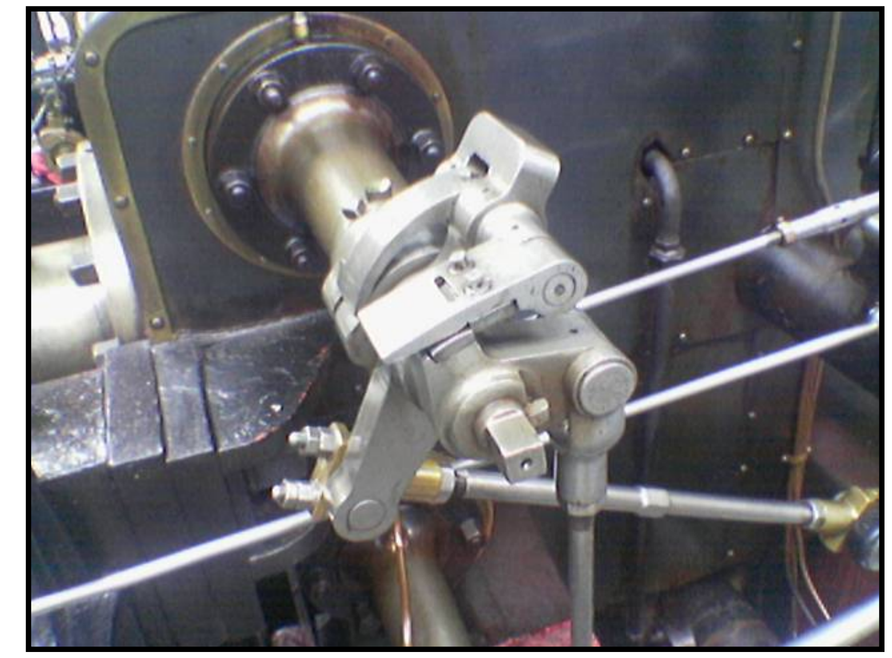
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

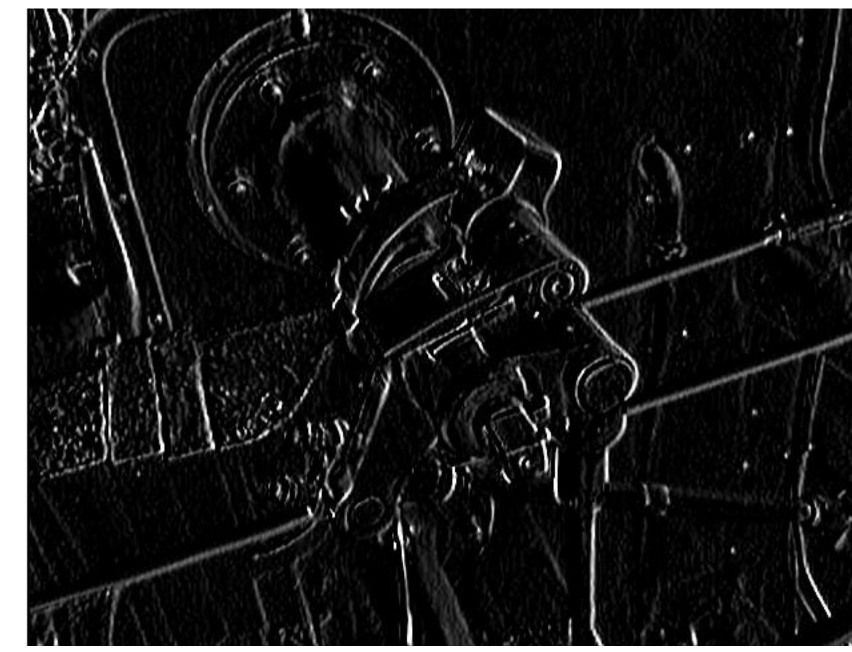
- **Find pixels with large gradients**

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

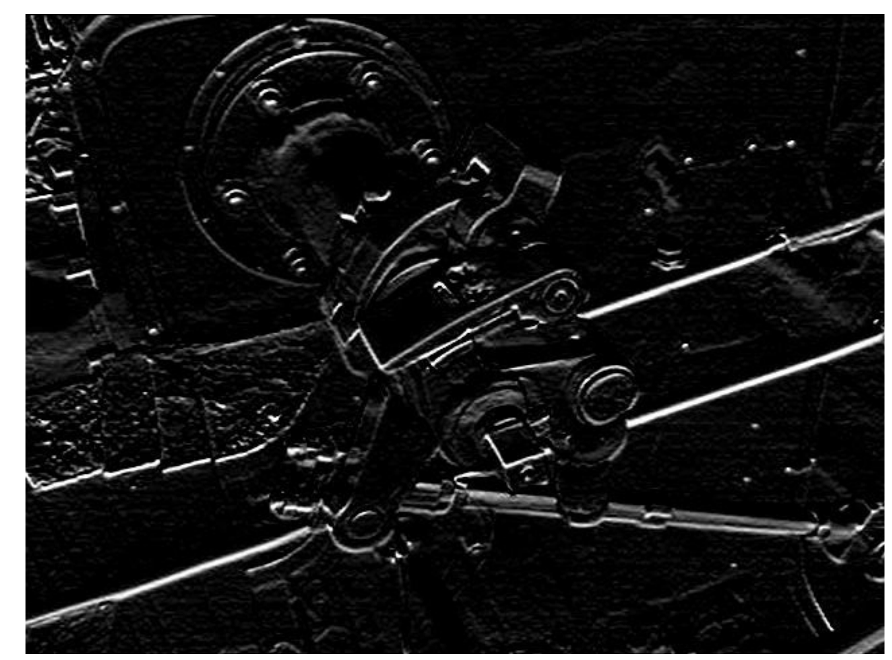
Pixel-wise operation on images



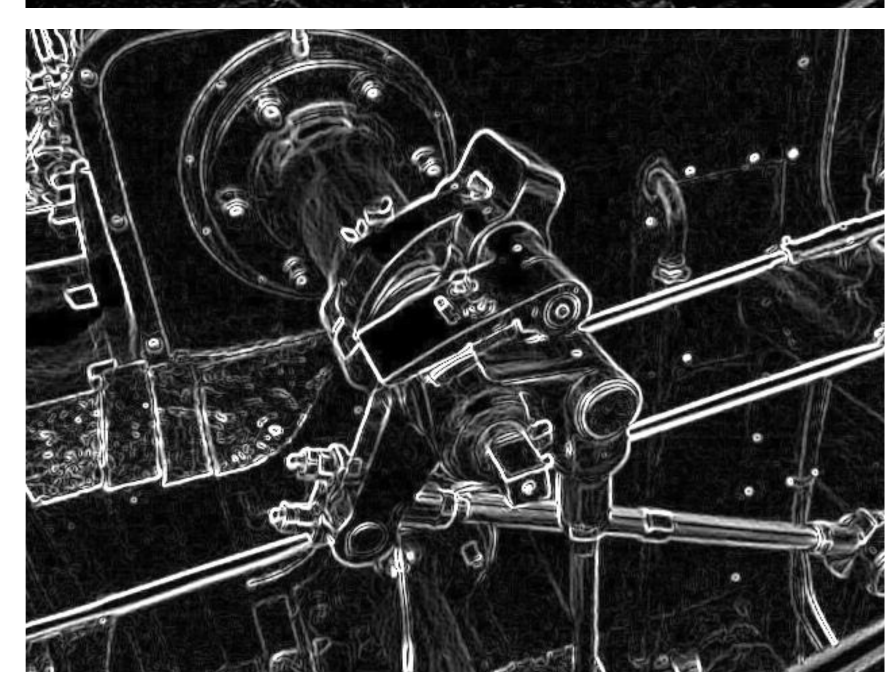
G_x



G_y



G



Cost of convolution with N x N filter?

```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9};

for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
         for (int ii=0; ii<3; ii++)
            tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH + i] = tmp;
  }
}
```

In this 3x3 box blur example:

Total work per image = 9 x WIDTH x HEIGHT

For N x N filter: N^2 x WIDTH x HEIGHT

Separable filter

- **A filter is separable if can bee written as the outer product of two other filters. Example: a 2D box blur**

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \frac{1}{3}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$$

  - **Exercise: write 2D gaussian and vertical/horizontal gradient detection filters as product of 1D filters (they are separable!)**

- **Key property: 2D convolution with separable filter can be written as two 1D convolutions!**

Implementation of 2D box blur via two 1D convolutions

```
int WIDTH = 1024
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1./3, 1./3, 1./3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Total work per image for NxN filter:
2N x WIDTH x HEIGHT

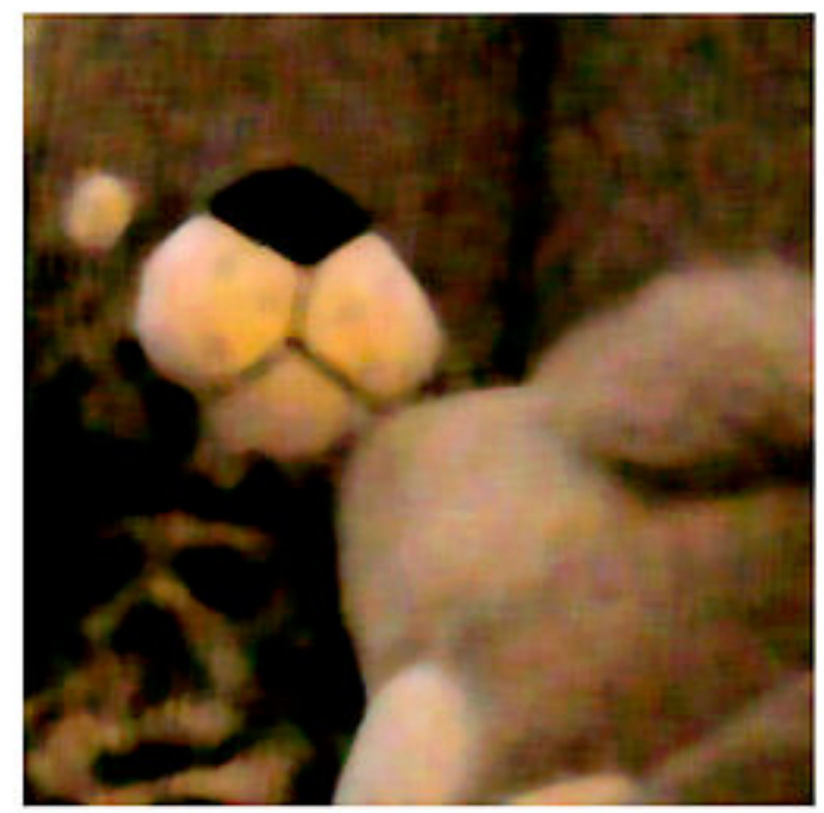
Median filter

- Replace pixel with median of its neighbors
  - Useful noise reduction filter: unlike gaussian blur, one bright pixel doesn't drag up the average for entire region
- Not linear, not separable
  - Filter weights are 1 or 0 (depending on image content)

```
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];
for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      output[j*WIDTH + i] =
           // compute median of pixels
           // in surrounding 5x5 pixel window
   }
}
```


original image


1px median filter


3px median filter


10px median filter

- Basic algorithm for NxN support region:
  - Sort N^2 elements in support region, then pick median: $O(N^2 \log(N^2))$ work per pixel
  - Can you think of an $O(N^2)$ algorithm? What about $O(N)$?

Bilateral filter

Example use of bilateral filter: removing noise while preserving image edges

Bilateral filter

Gaussian blur kernel

Input image

$$\mathrm{BF}[I](p) = \frac{1}{W_p} \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|) G_\sigma(i,j) I(x-i,y-j)$$

Normalization

For all pixels in support region of Gaussian kernel

Re-weight based on difference in input image pixel values

$$W_p = \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|) G_\sigma(i,j) I(x-i,y-j)$$

- **The bilateral filter is an "edge preserving" filter: down-weight contribution of pixels on the "other side" of strong edges. $f(x)$ defines what "strong edge means"**
- **Spatial distance weight term $f(x)$ could itself be a gaussian**
  - **Or very simple:** $f(x) = 0$ if $x > \mathit{threshold}$, 1 *otherwise*

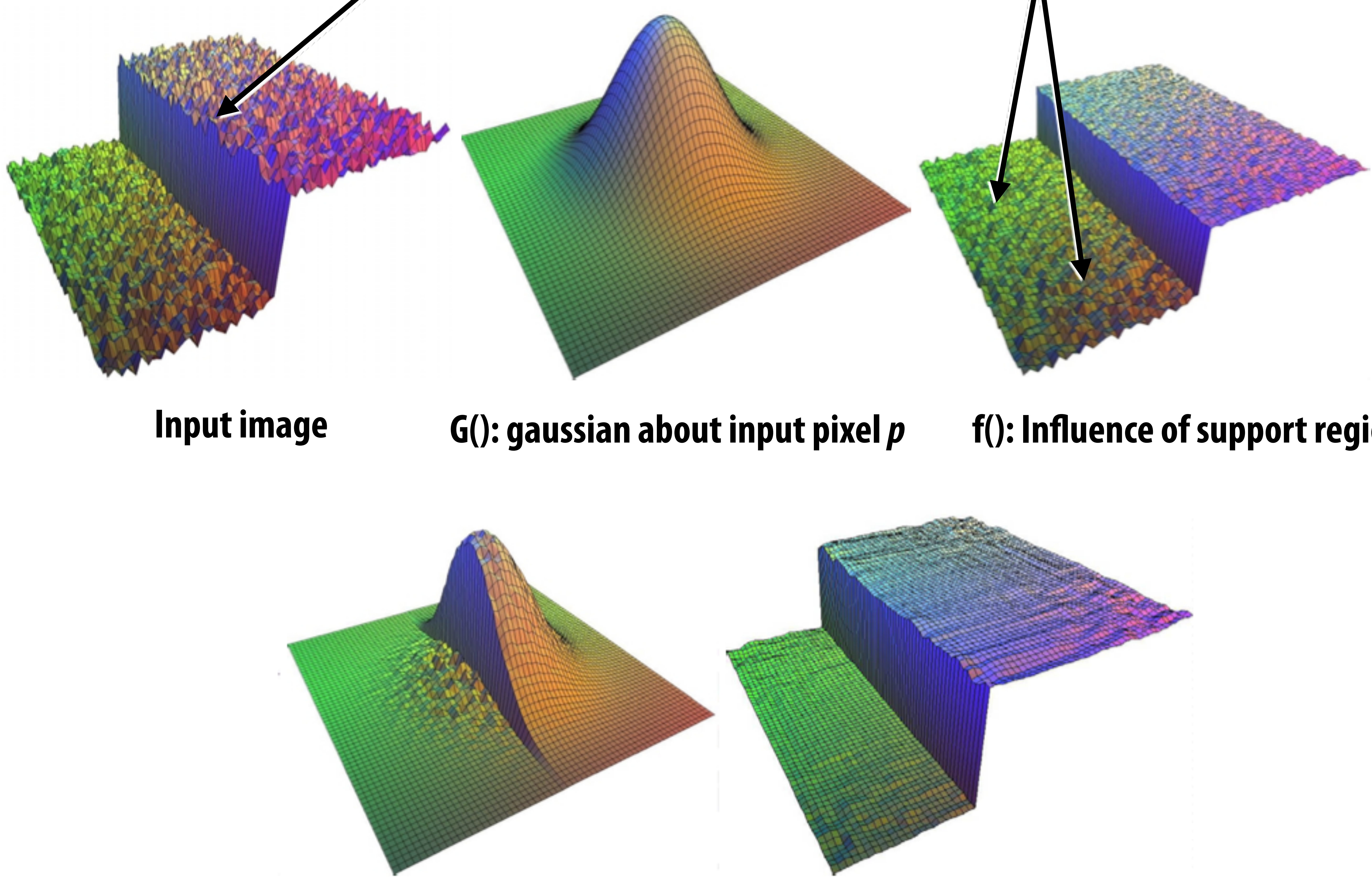
Value of output pixel (x,y) is the weighted sum of all pixels in the support region of a truncated gaussian kernel

But weight is combination of <u>spatial distance</u> and <u>input image pixel intensity</u> difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

Bilateral filter

Pixels with significantly different intensity as *p* contribute little to filtered result (they are "on the "other side of the edge"

Input pixel *p*



Input image

G(): gaussian about input pixel *p*

f(): Influence of support region

G x f: filter weights for pixel *p*

Filtered output image

Figure credit: Durand and Dorsey, "Fast Bilateral Filtering for the Display of High-Dynamic-Range Images", SIGGRAPH 2002

Bilateral filter: kernel depends on image content

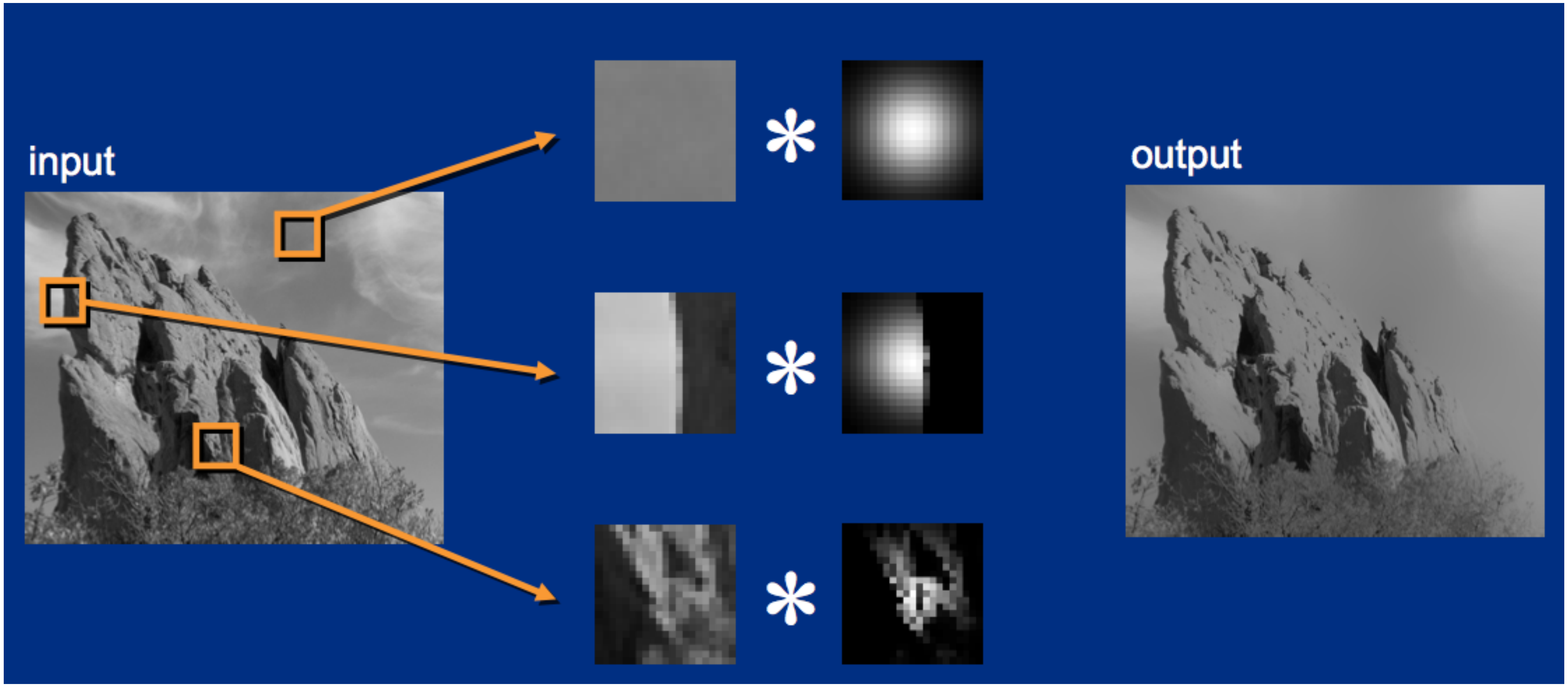


Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

Summary

- **Last two lectures: representing images**
  - Choice of color space (different representations of color)
  - Store values in perceptual space (non-linear in energy)
  - JPG image compression (tolerate loss due to approximate representation of high frequency components)

- **Basic image processing operations**
  - Per-pixel operations out(x,y) = f(in(x,y)) (e.g., contrast enhancement)
  - Image filtering via convolution (e.g., blur, sharpen, simple edge-detection)
  - Non-linear, data-dependent filters (median filter, avoid blurring over strong edges, etc.)