Lecture 17:

Image Processing Basics

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2023

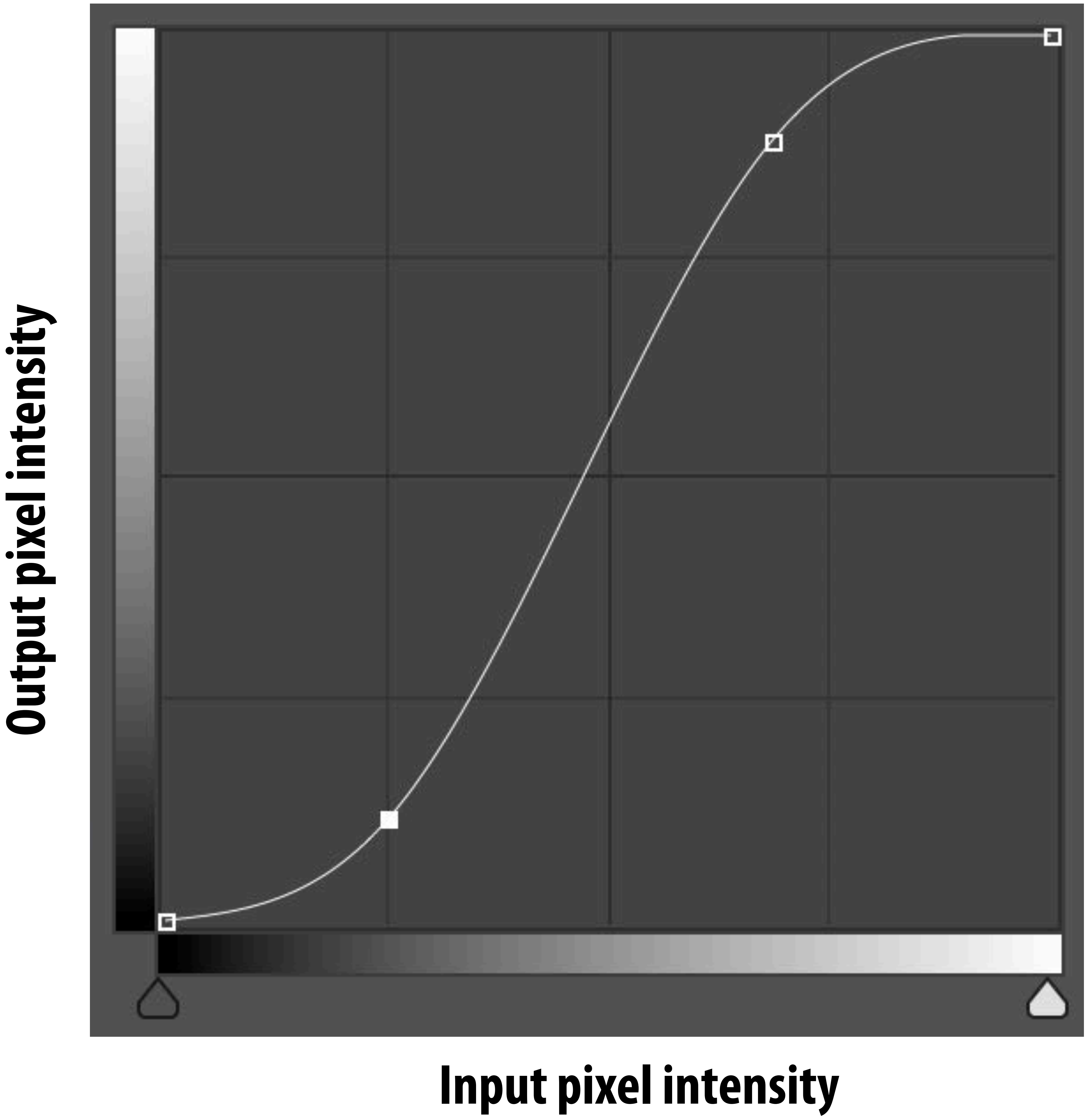
Example image processing operations

Increase contrast

Increasing contrast with "S curve"

Per-pixel operation:

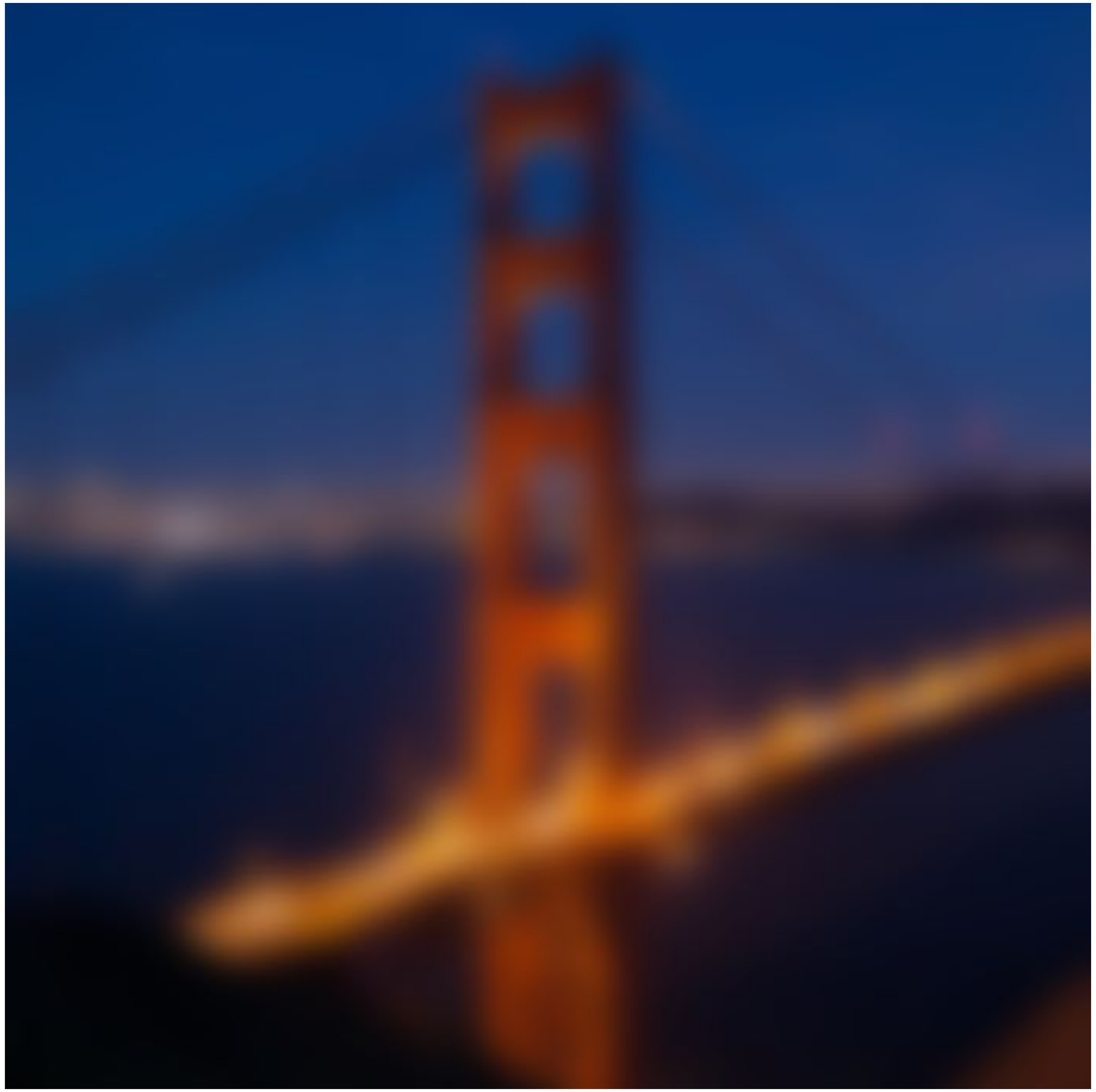
output(x,y) = f(input(x,y))



Invert

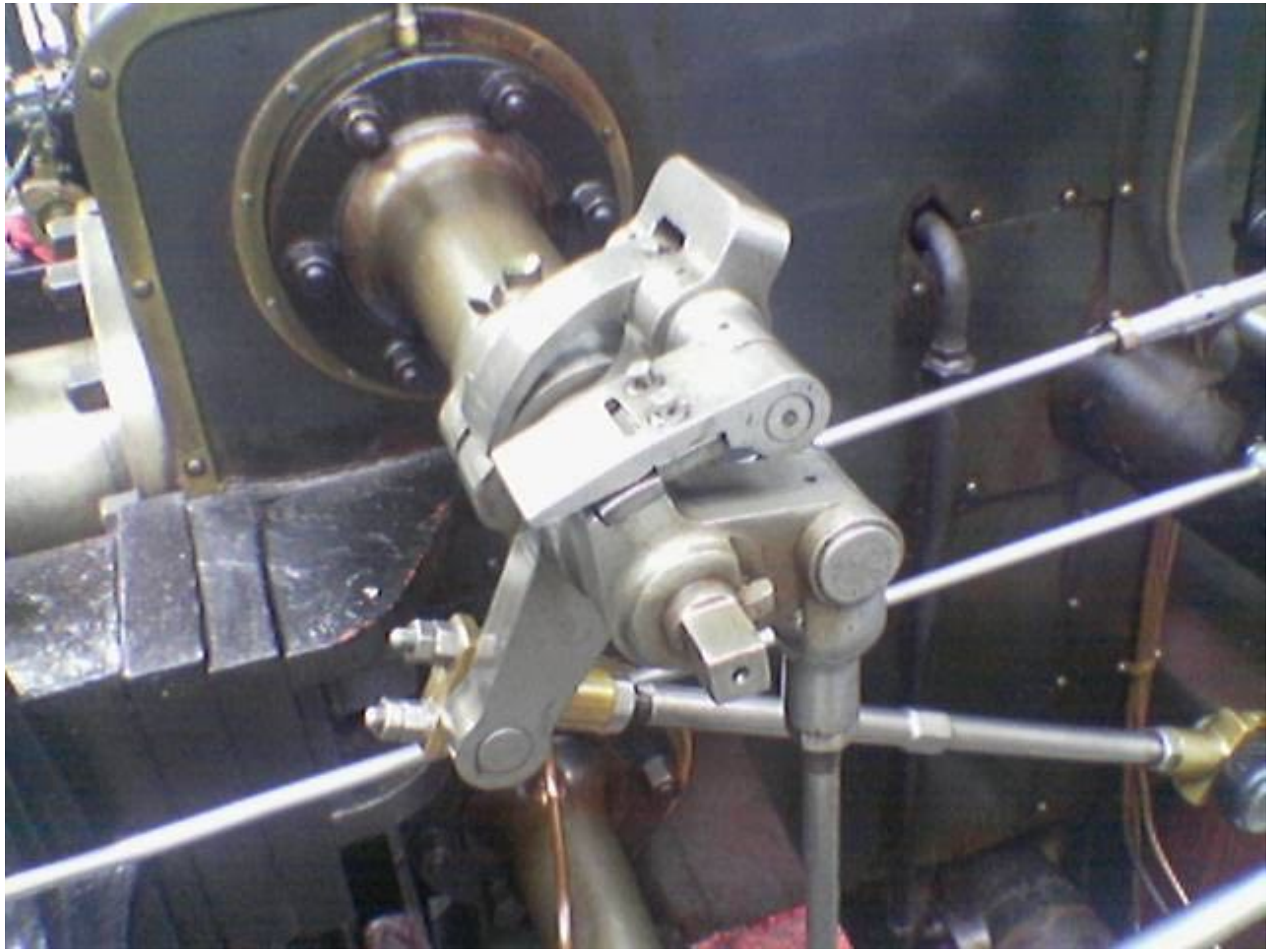
$$out(x,y) = 1 - in(x,y)$$

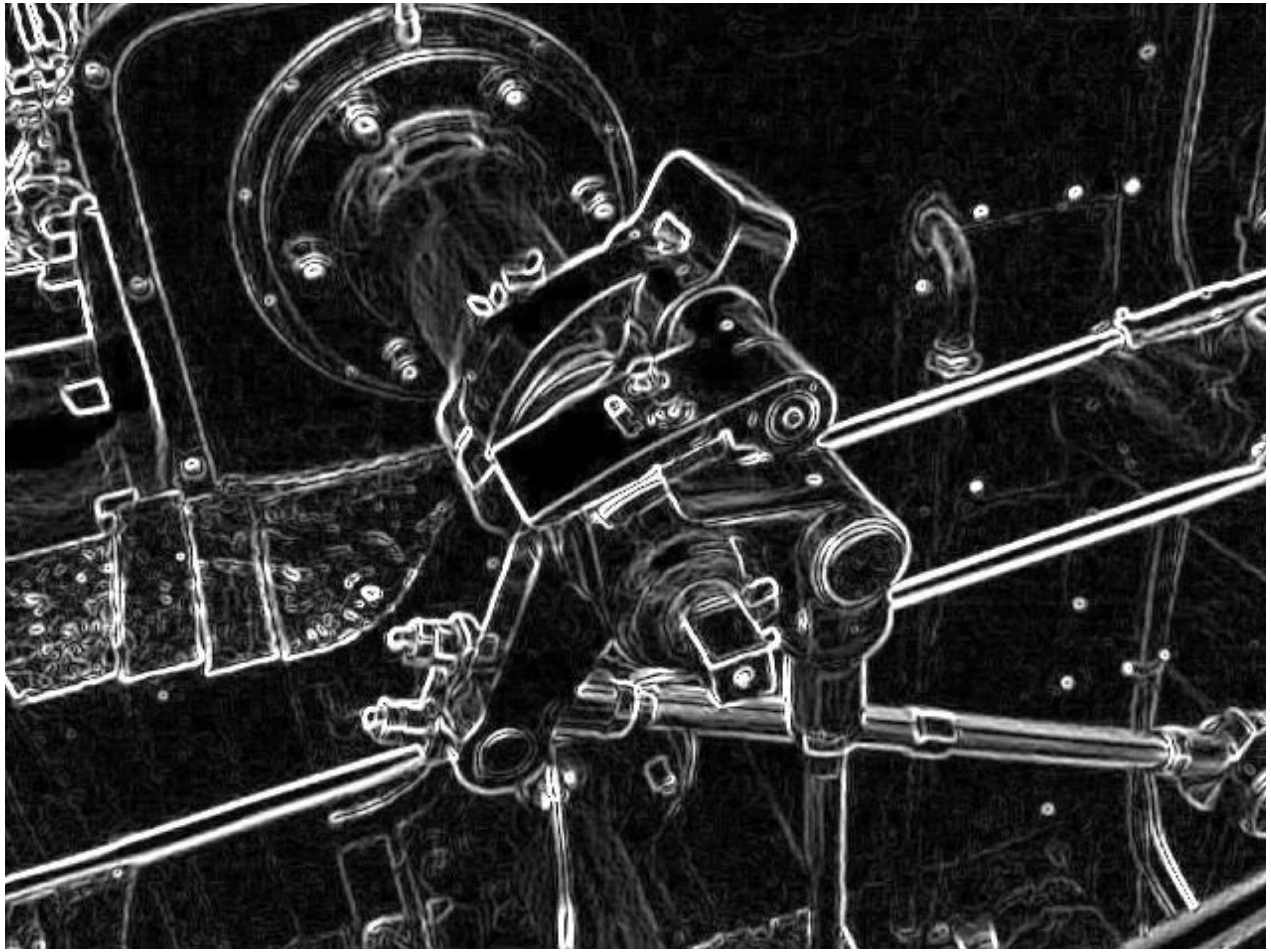
Blur



Sharpen

Edge detection





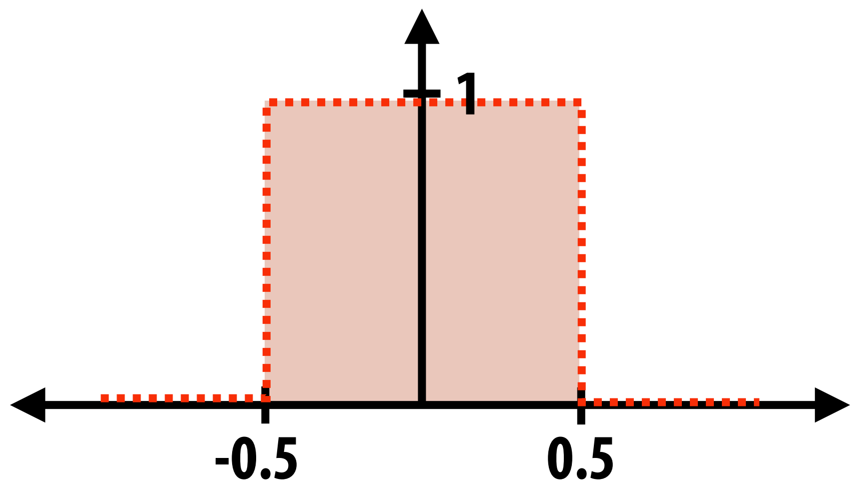
A “smarter” blur (doesn’t blur over edges)

Review: convolution

$$\underbrace{(f * g)(x)}_{\text{output signal}} = \int_{-\infty}^{\infty} \underbrace{f(y)}_{\text{filter (or ``kernel'')}}\underbrace{g(x-y)}_{\text{input signal (e.g. the input image)}}dy$$

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* where the output at x is the average value of the input between x-0.5 to x+0.5

Discrete 2D convolution

$$\underbrace{(f * g)(x, y)}_{\text{output image}} = \sum_{i,j=-\infty}^{\infty} \underbrace{f(i, j)}_{\text{filter}}\underbrace{I(x - i, y - j)}_{\text{input image}}$$

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

Then:

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

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")**

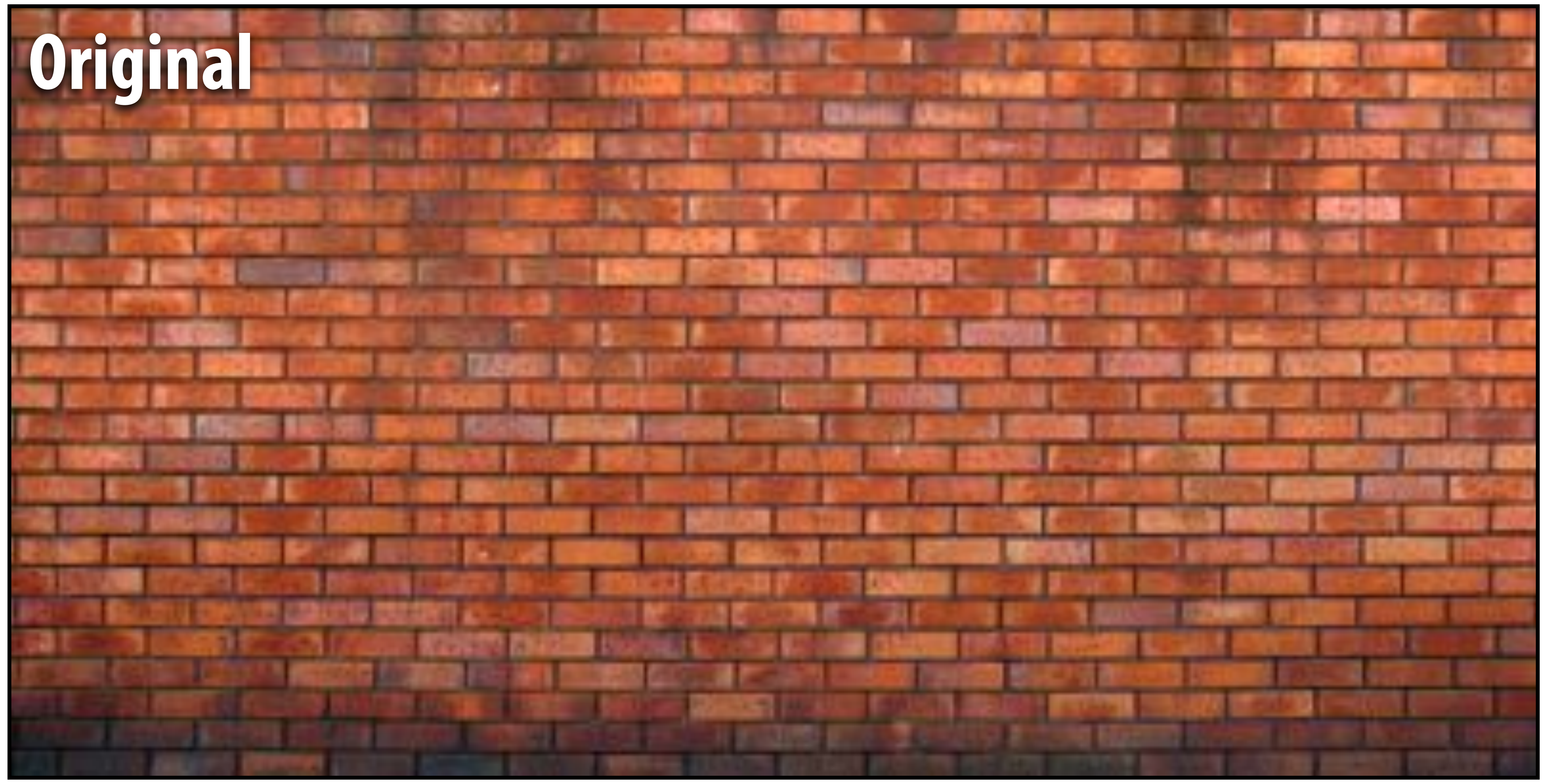
Simple 3x3 box blur

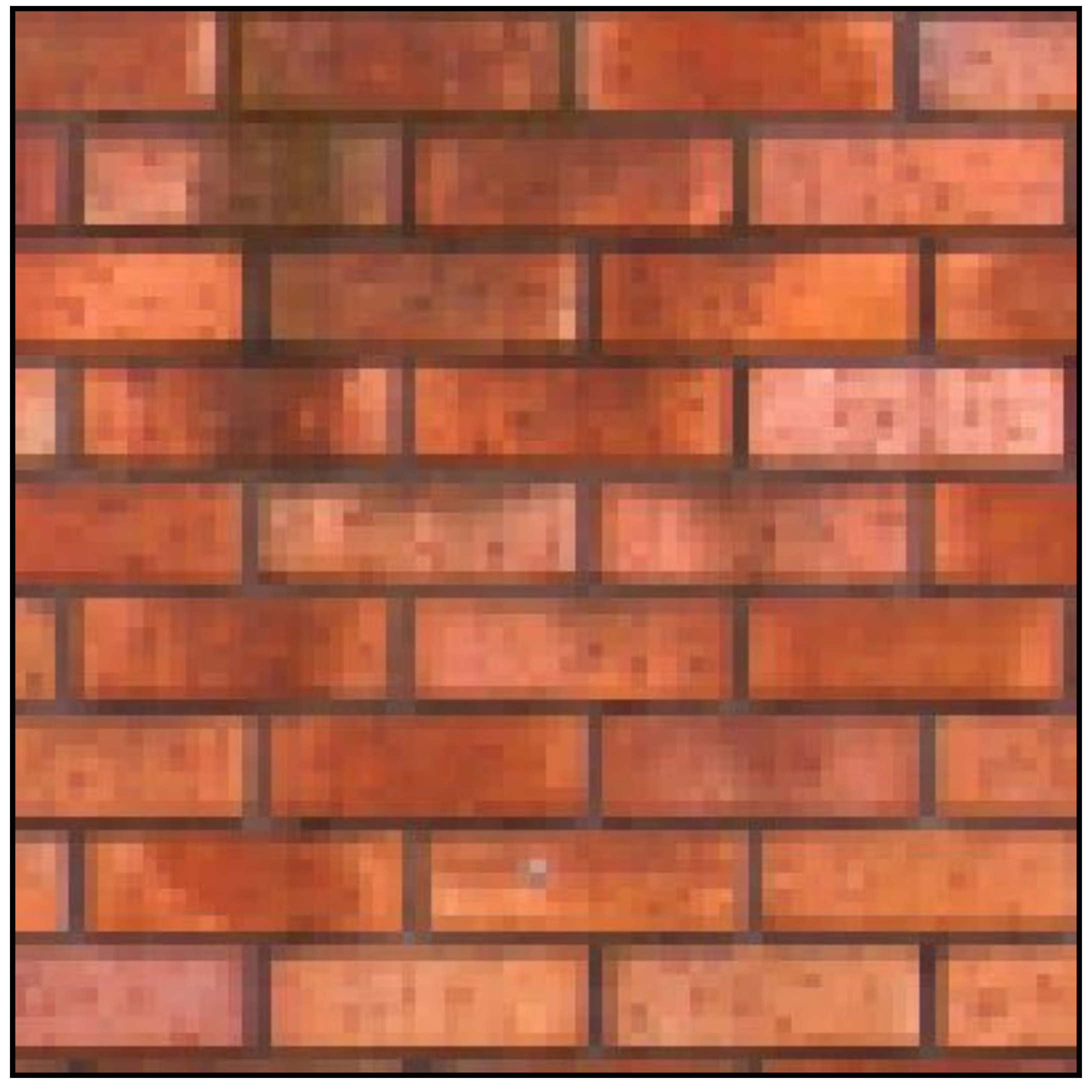
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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;
   }
}
```

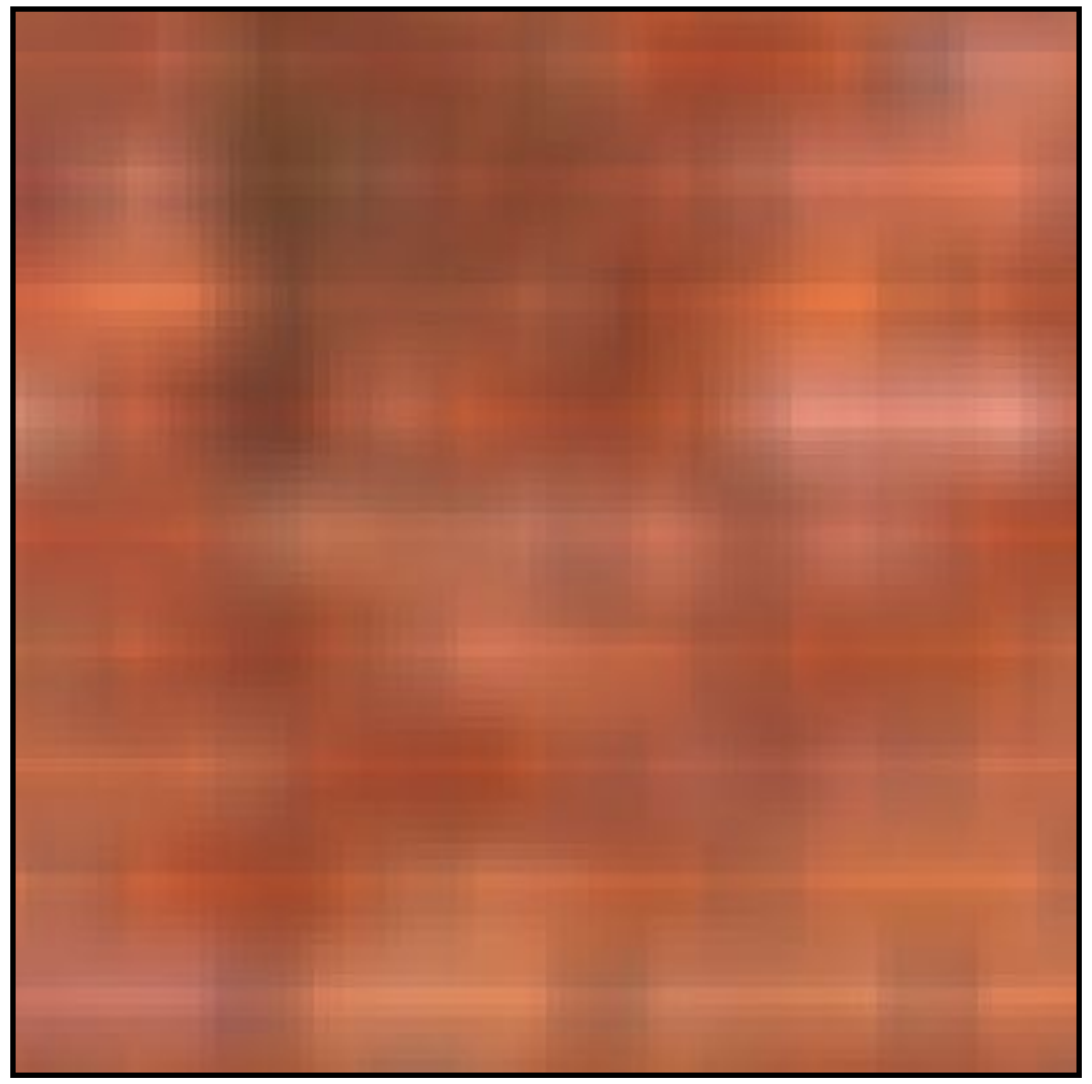
For now: ignore boundary pixels and assume output image is smaller than input (makes convolution loop bounds simpler to write)

7x7 box blur

Zoomed view







Gaussian blur

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

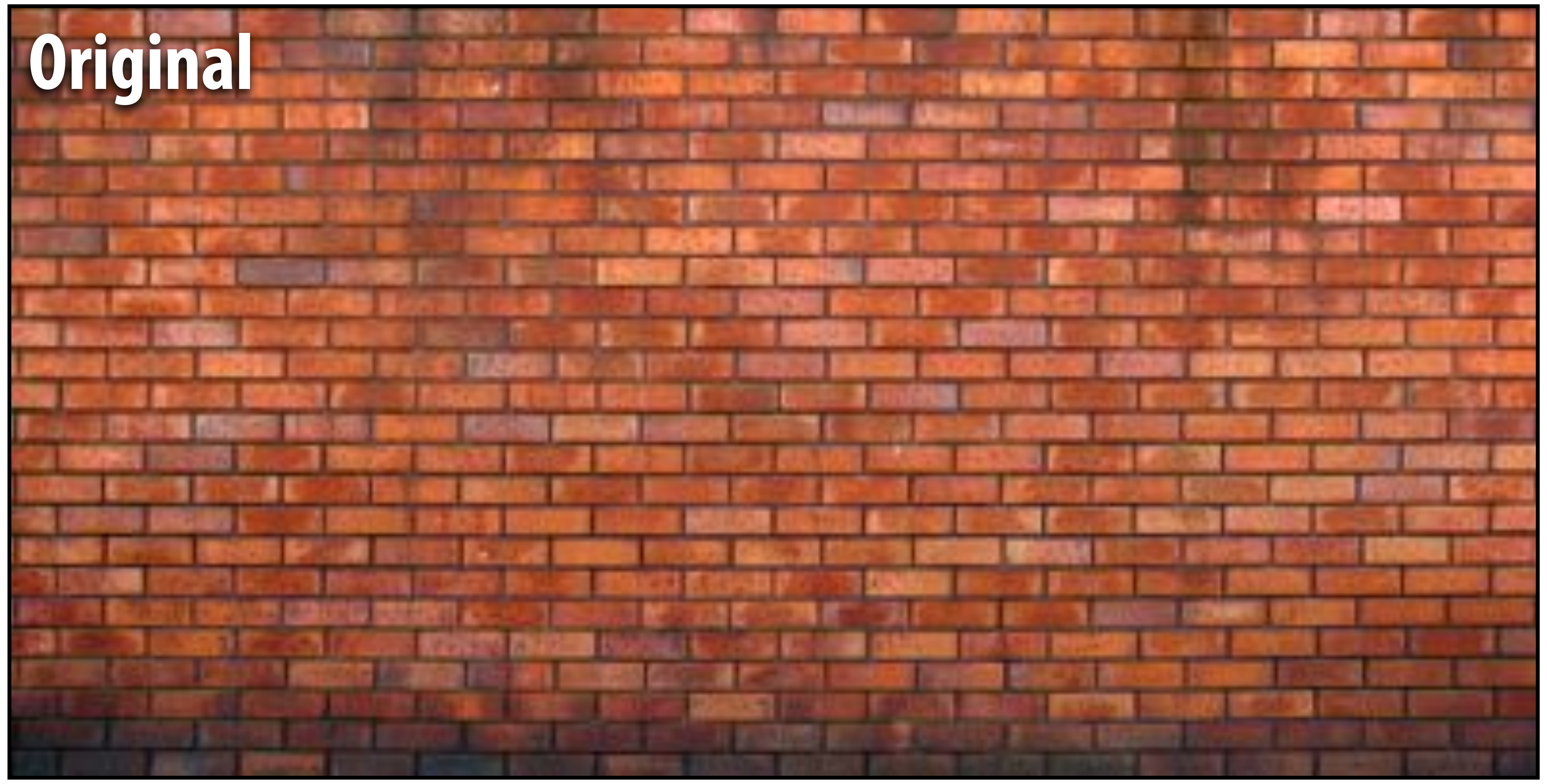
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

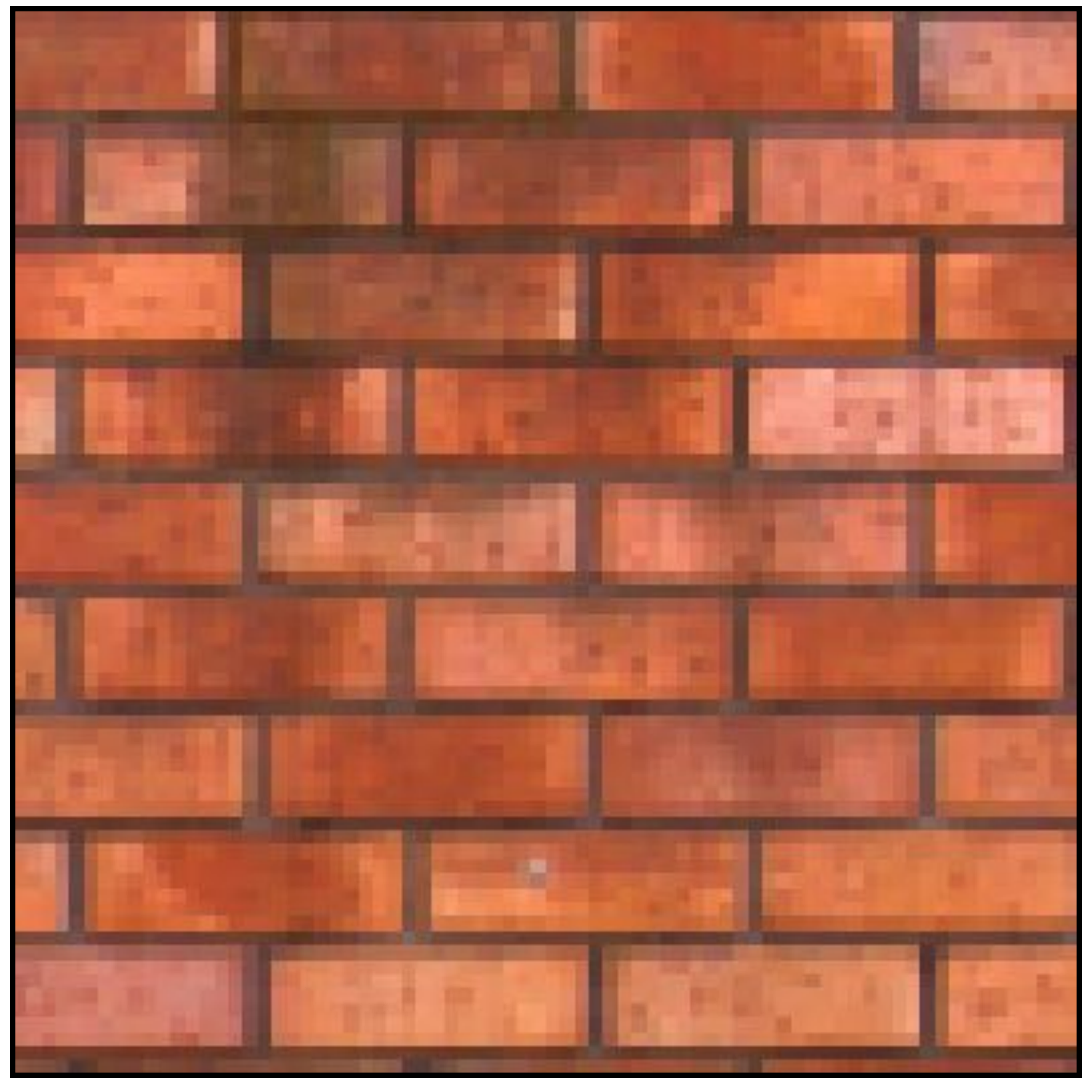
- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

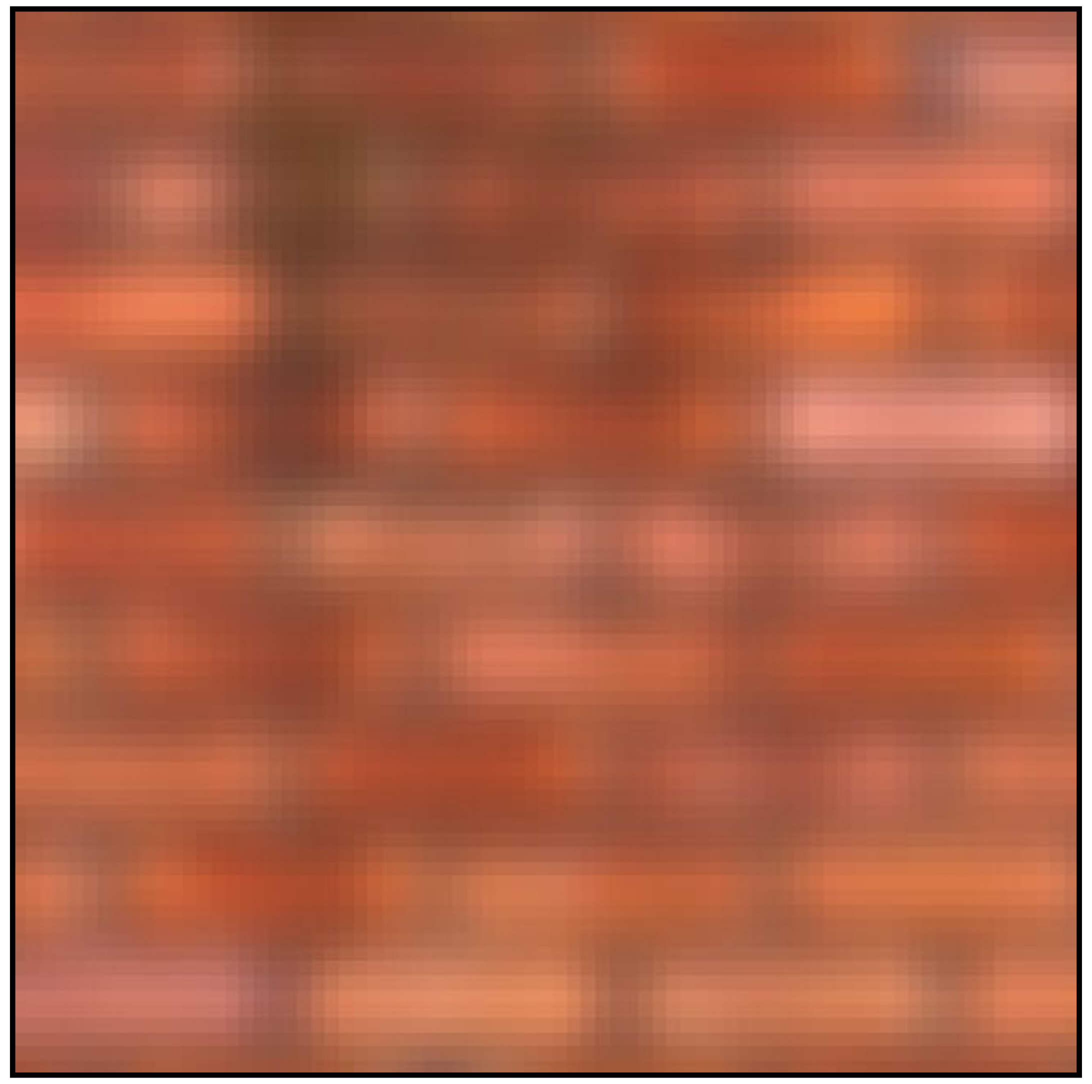
$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

7x7 gaussian blur

Zoomed view







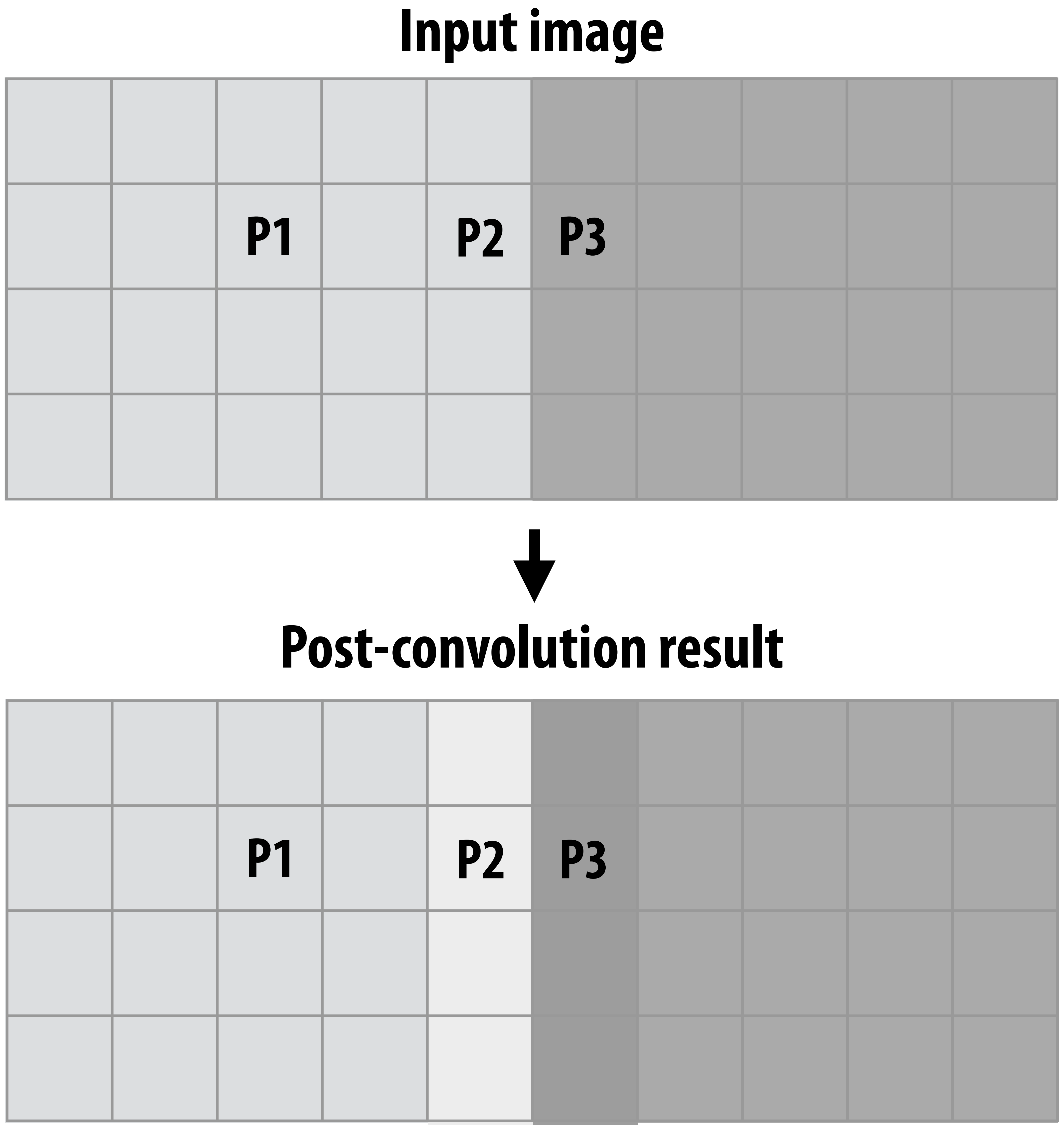
What does convolution with this filter do?

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

Sharpens image!

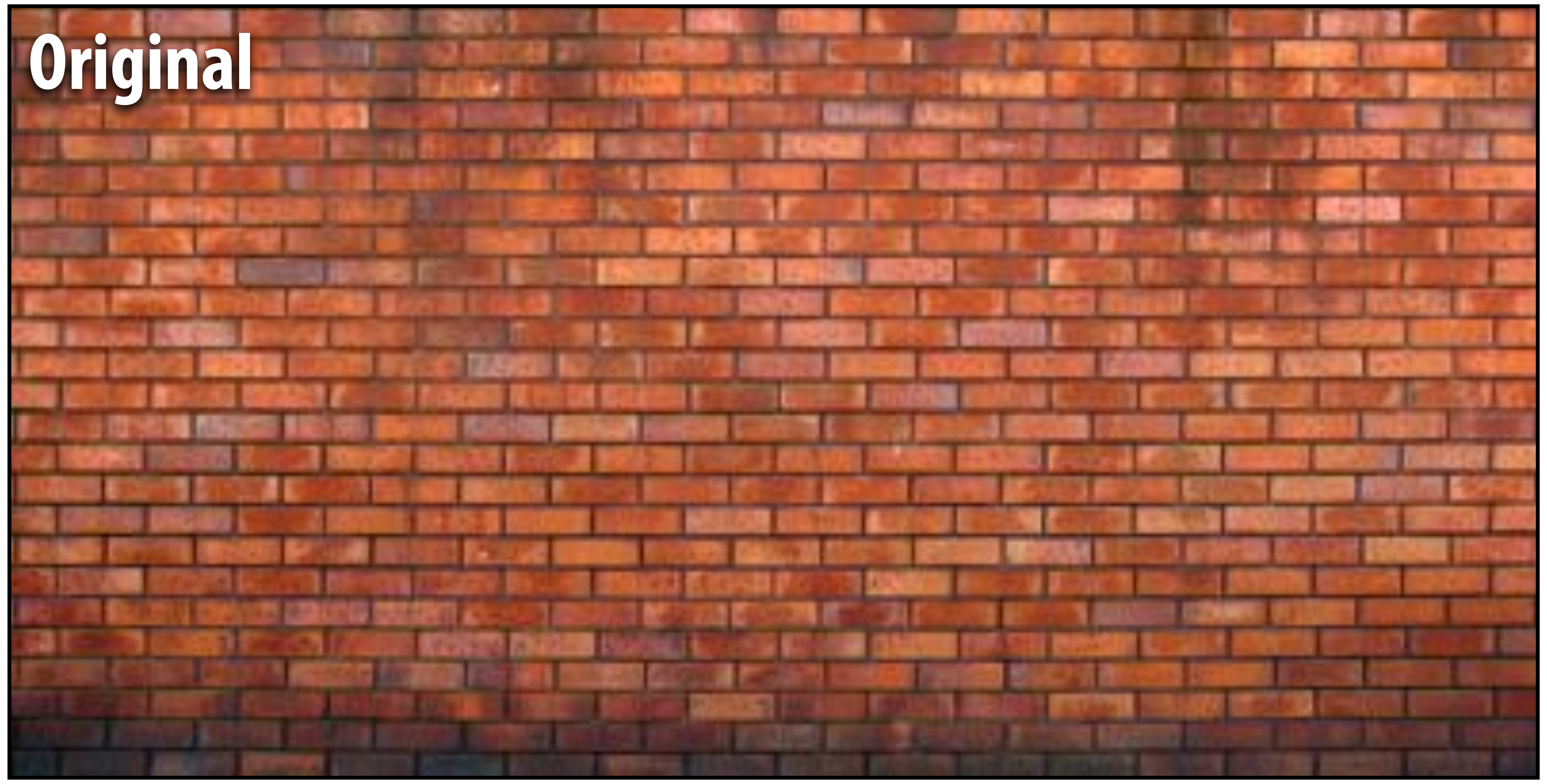
What does convolution with this filter do?

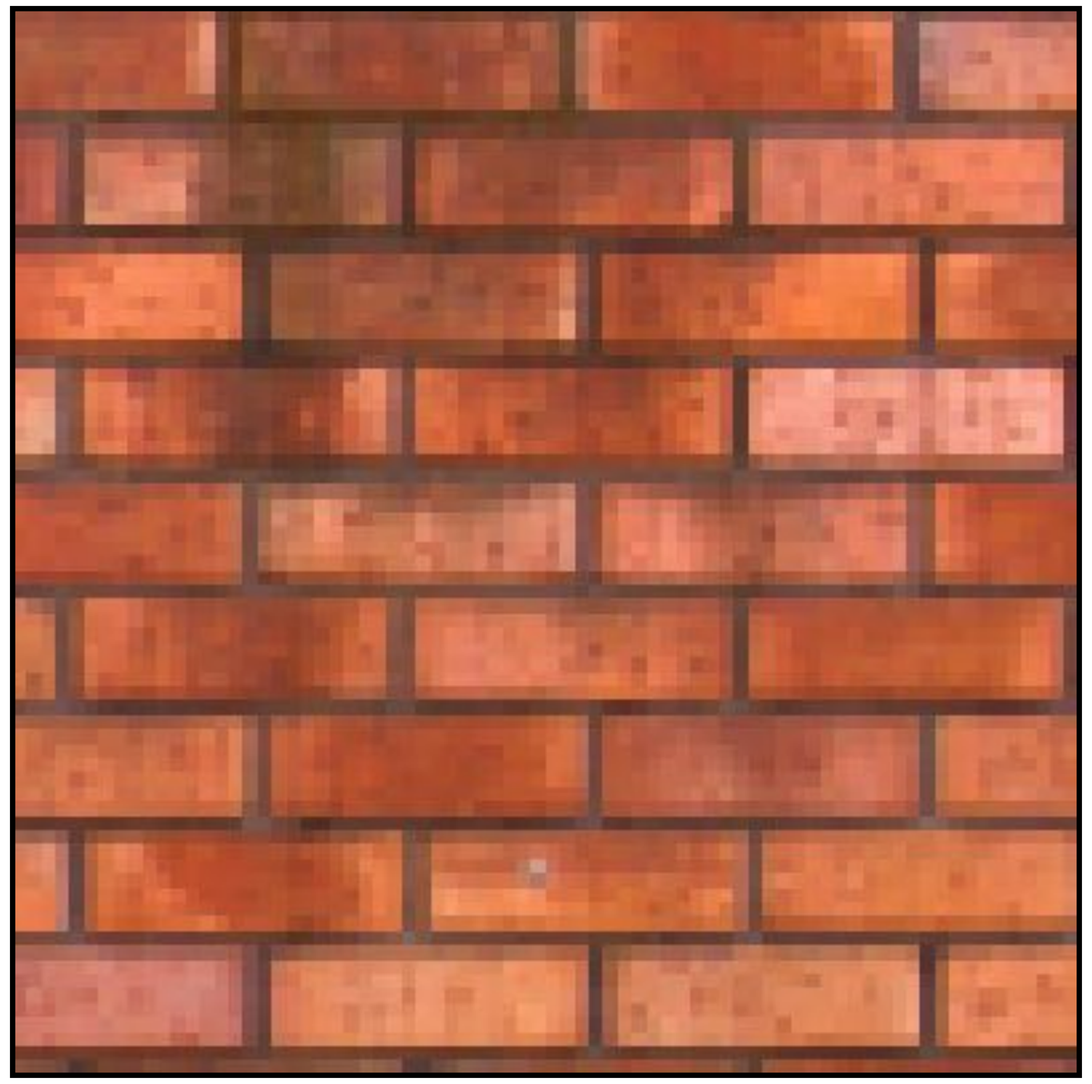
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

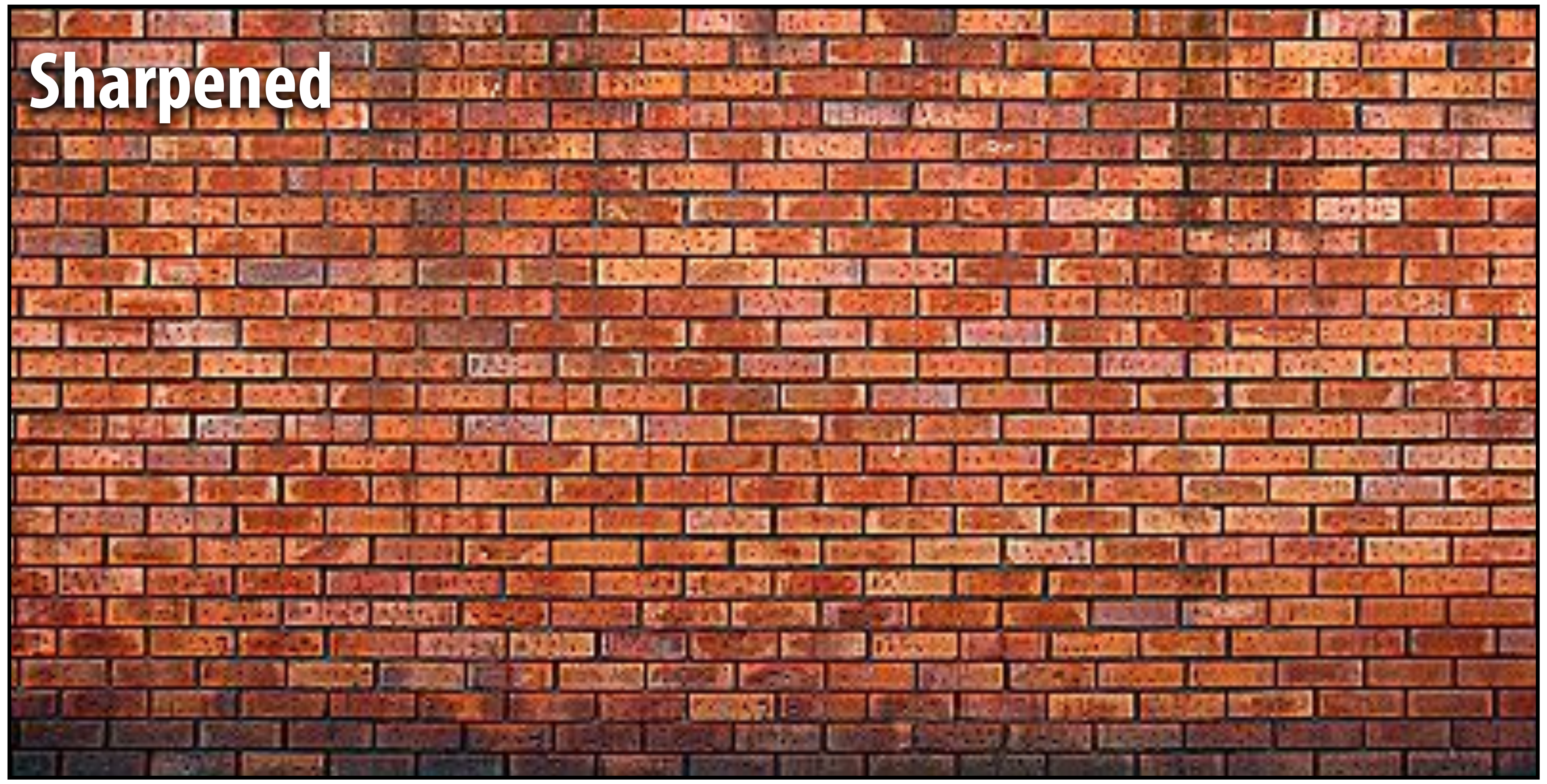


3x3 sharpen filter $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Zoomed view

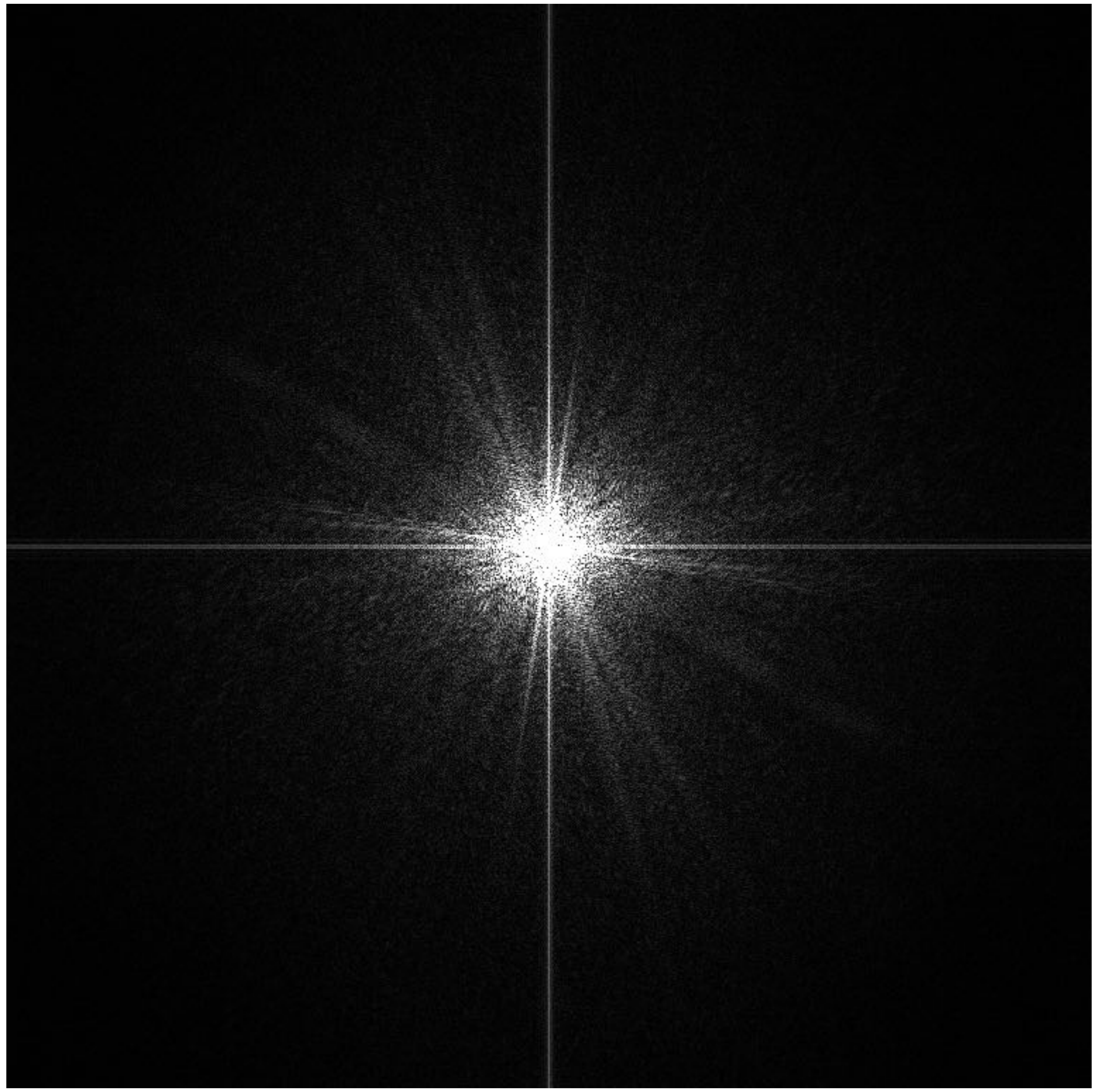






Recall: blurring is removing high frequency content

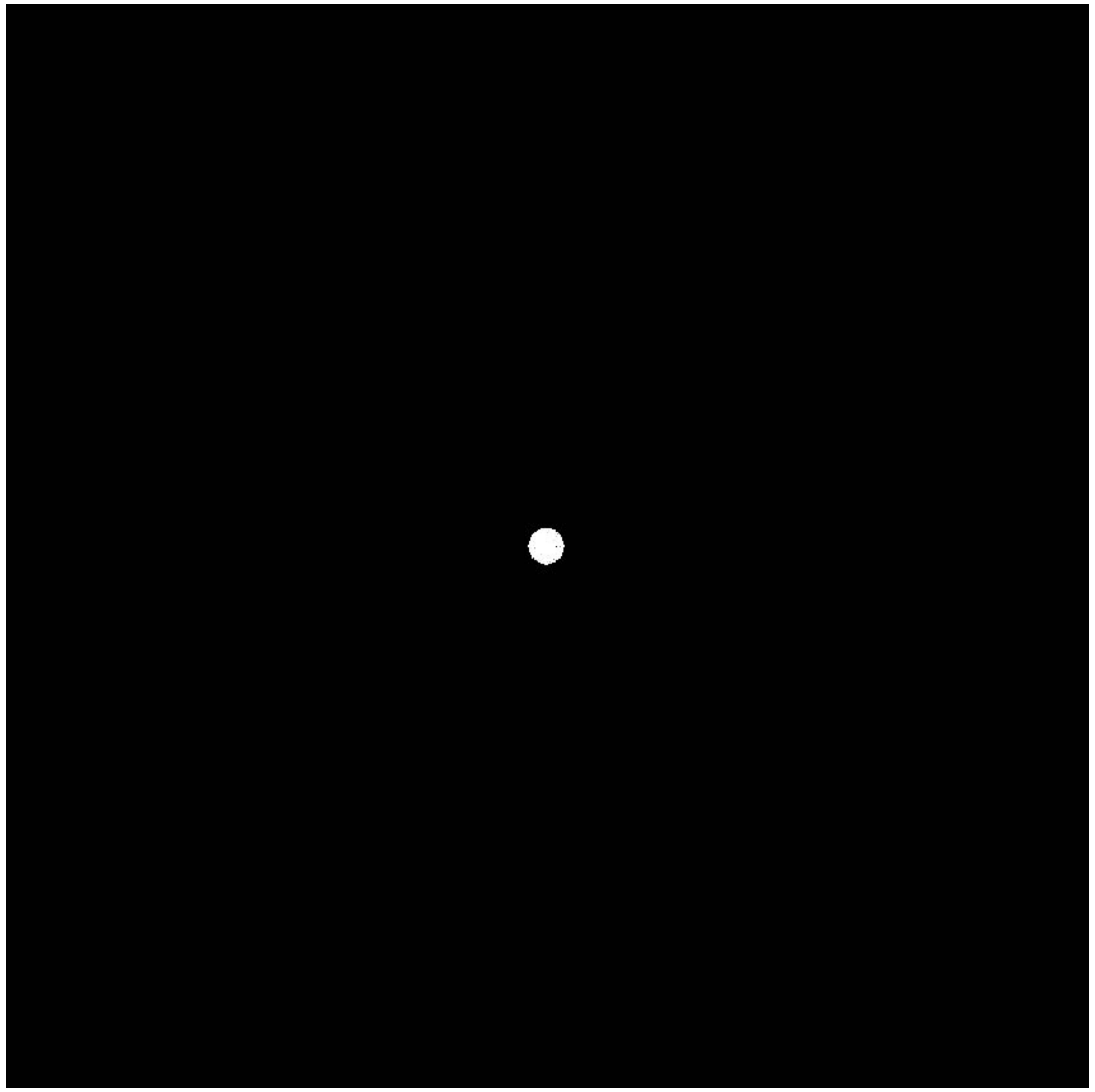
Spatial domain result



Spectrum

Recall: blurring is removing high frequency content

Spatial domain result



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

Sharpening is adding high frequencies

- Let I be the original image
- High frequencies in image $I = I$ - blur(I)
- Sharpened image $= I +$ (I-blur(I))

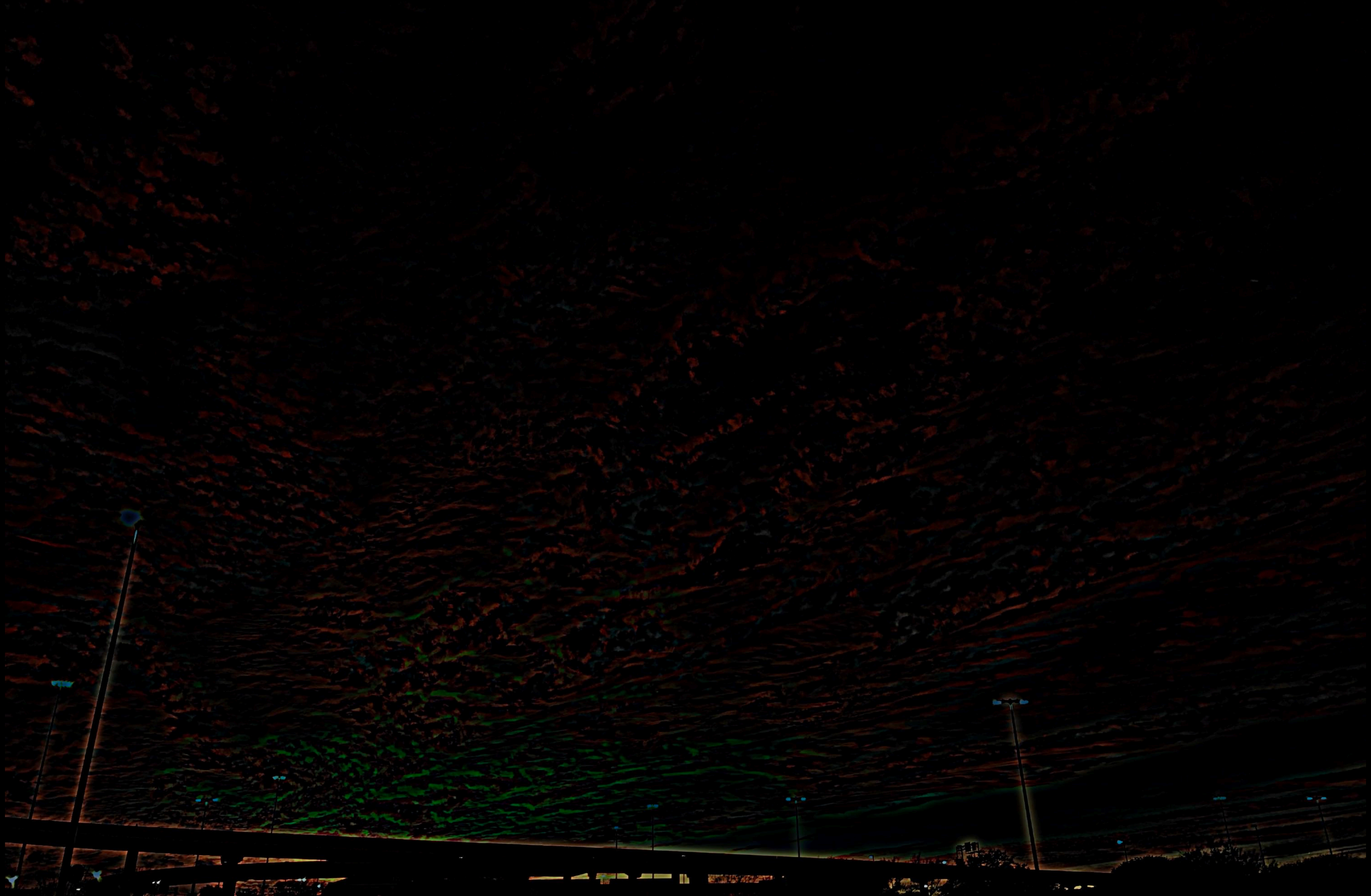
"Add high frequency content"

Original image (I)

Image credit:
Kayvon's parents

Blur(I)

I - blur(I)



Original image (I)

I + (I - blur(I))

What does convolution with these filters do?

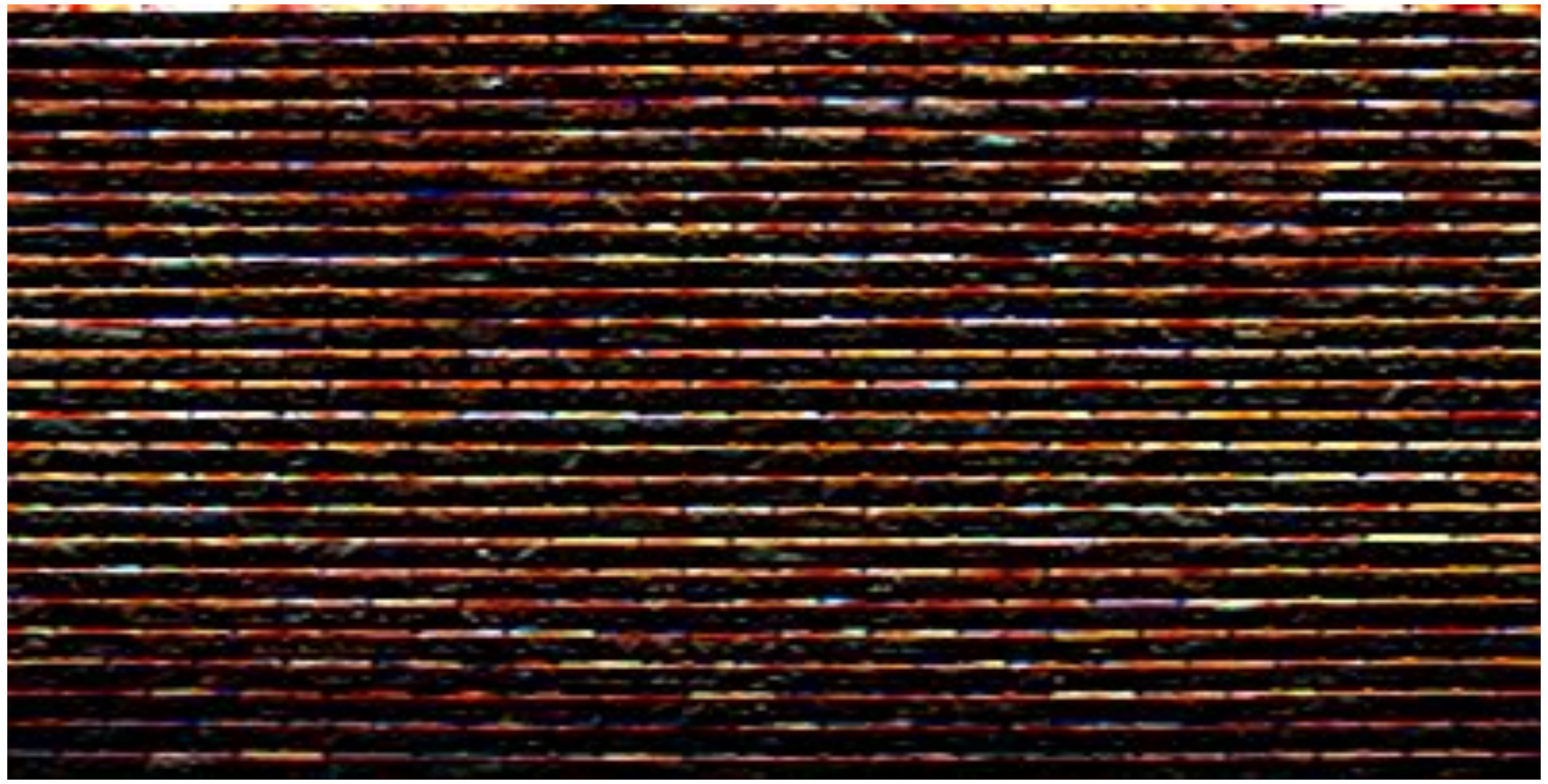
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Extracts horizontal gradients

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**

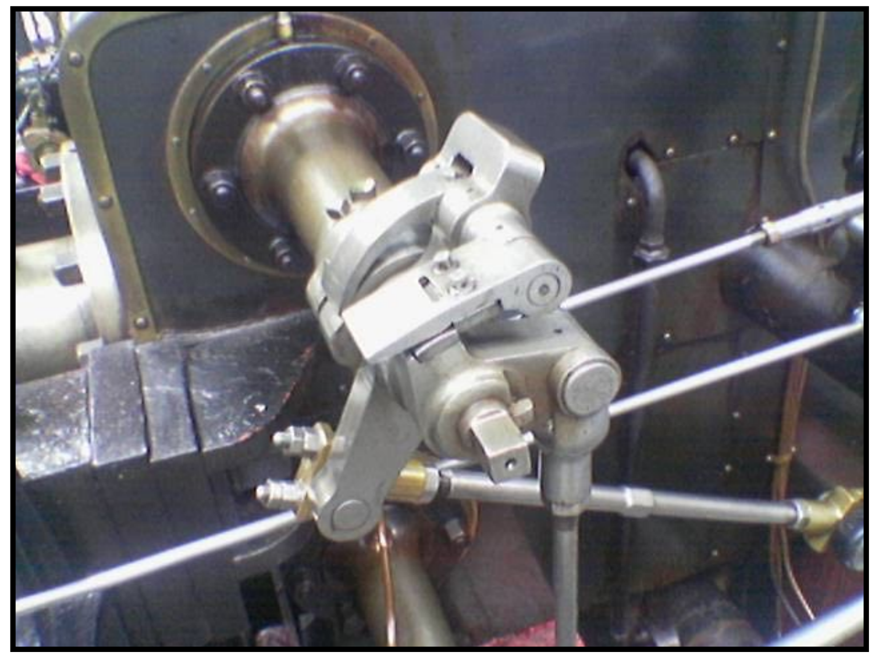
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

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

- **Find pixels with large gradients**

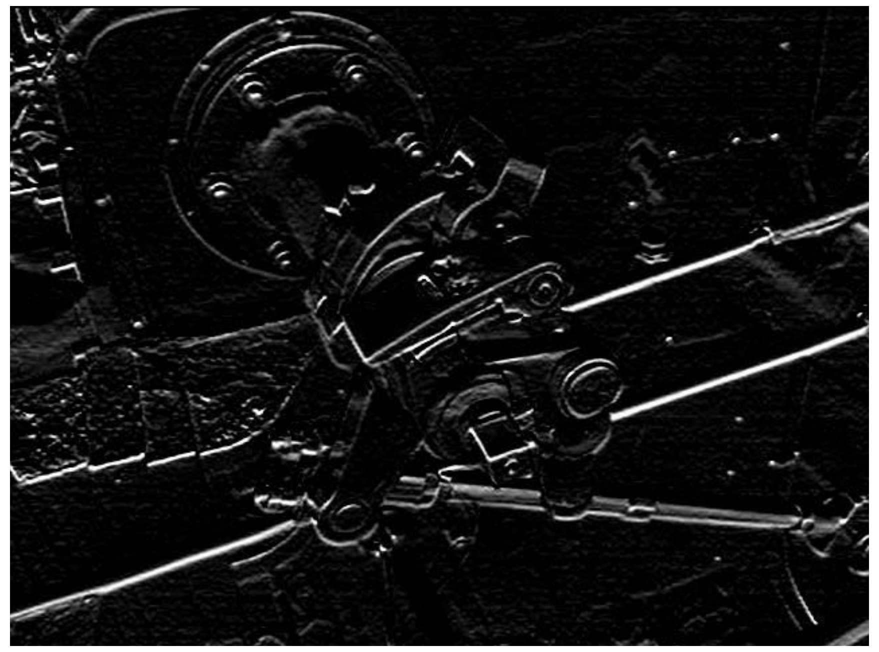
$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

Pixel-wise operation on images

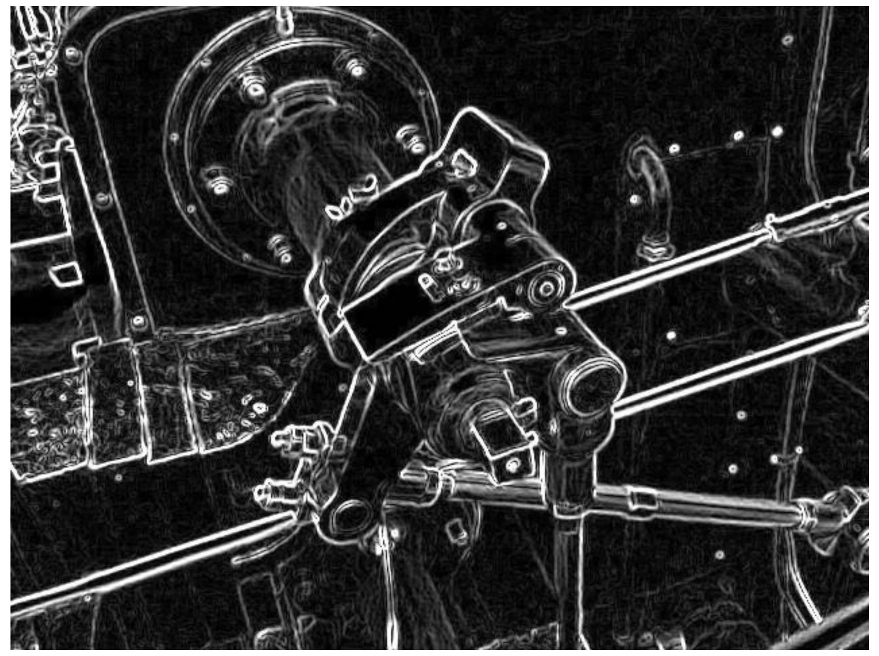


G_x

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 be 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

Bilateral filter

Do not smooth over hard edges, but smooth when there are not hard edges.

Original

After bilateral filter

Bilateral filter

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

Original

After bilateral filter

Bilateral filter

$$\mathrm{BF}[I](p) = \frac{1}{W_p} \sum_{i,j} \underbrace{f(|I(x-i,y-j) - I(x,y)|)}_{\text{Re-weight based on difference in input image pixel values}} \overbrace{G_\sigma(i,j)}^{\text{Gaussian blur kernel}} \overbrace{I(x-i,y-j)}^{\text{Input image}}$$

Normalization
(weights should sum to 1)

For all pixels in support region of Gaussian kernel

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

- **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 $x > \textit{threshold}$, 1 *otherwise*

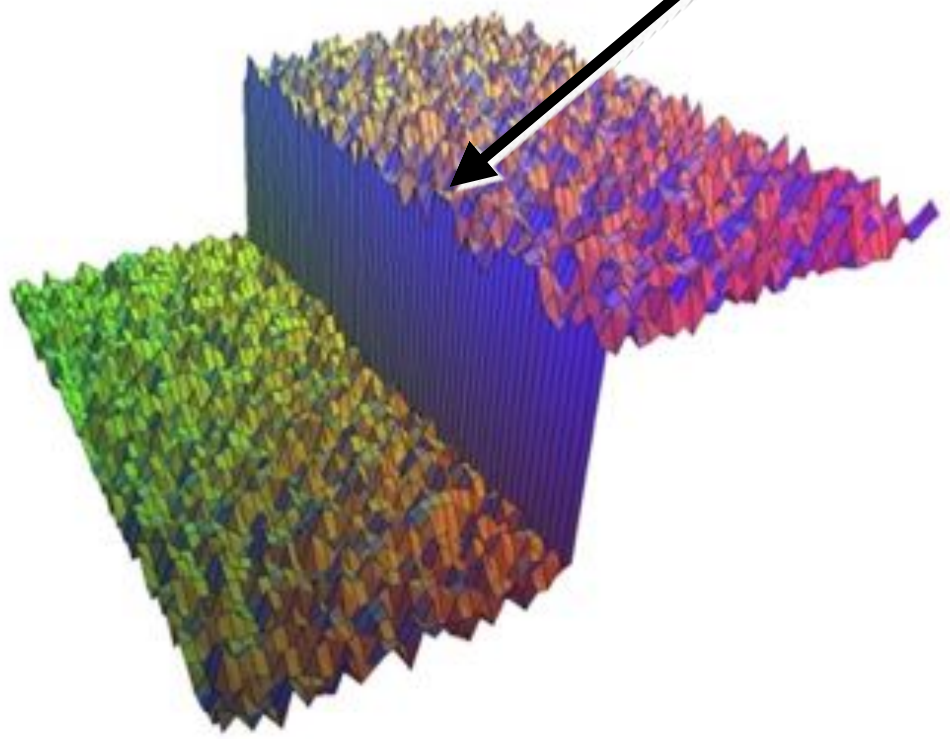
Value of output pixel (x,y) is the weighted sum of all pixels in the support region of a truncated gaussian kernel

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

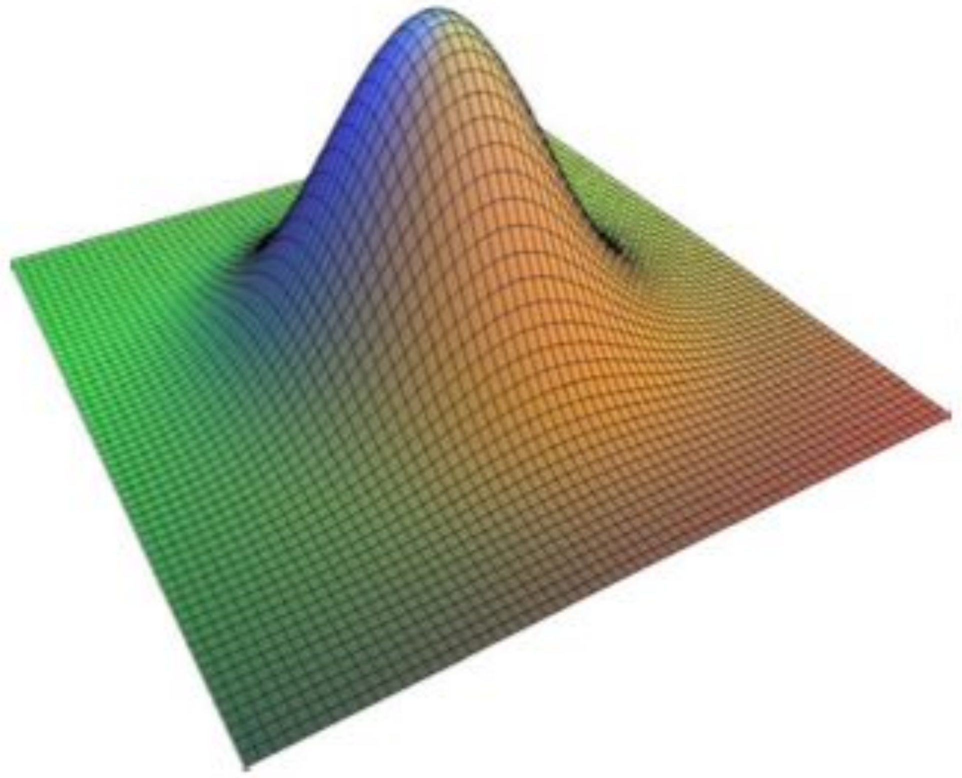
Visualization of bilateral filter

Input pixel *p*

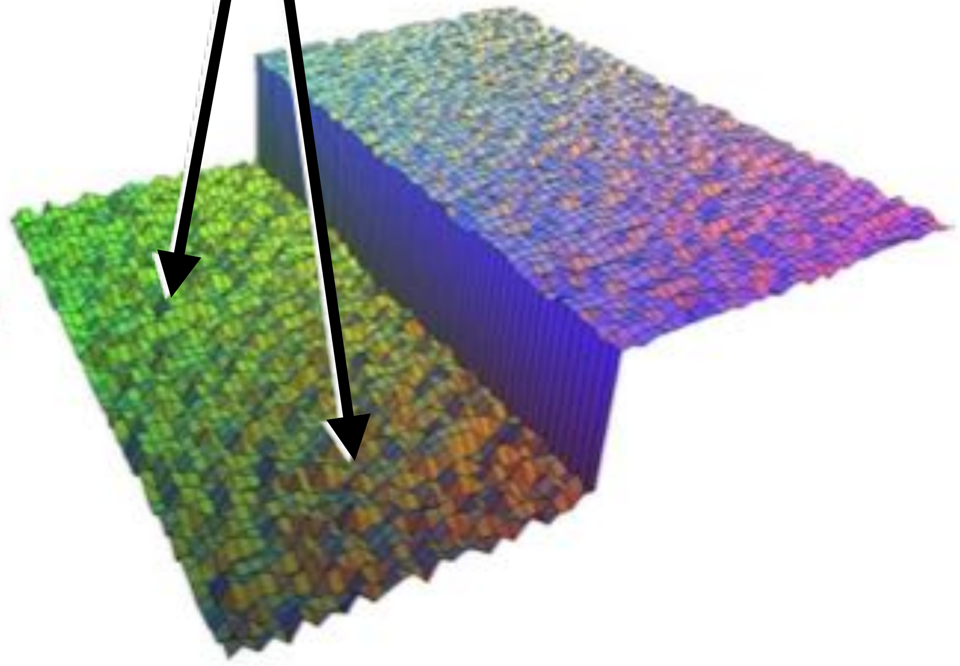
Pixels with significantly different intensity as *p* contribute little to filtered result (they are "on the "other side of the edge"



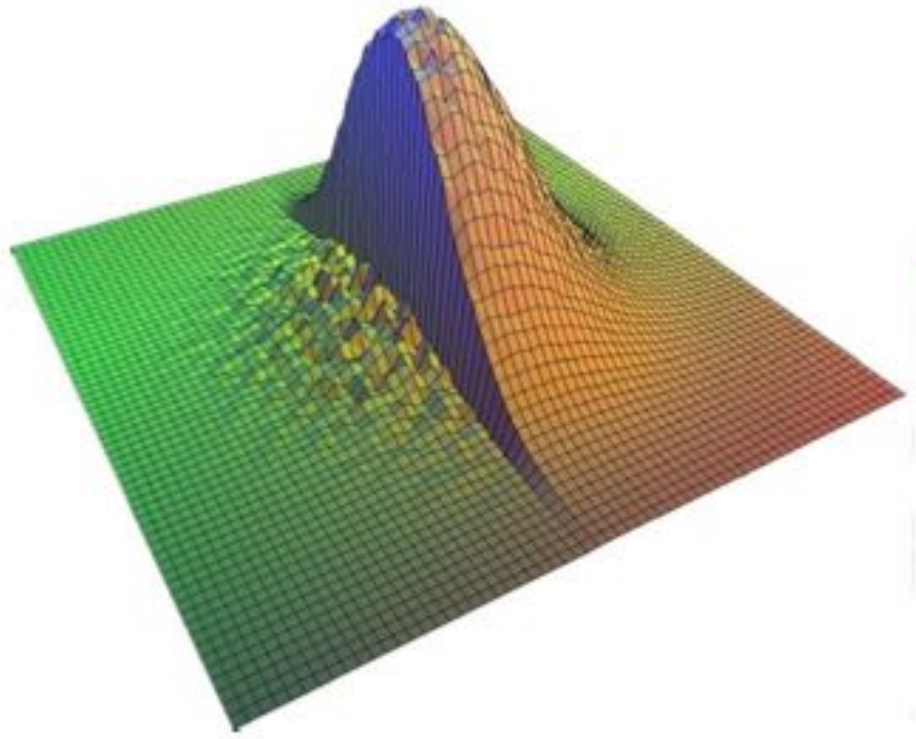
Input image



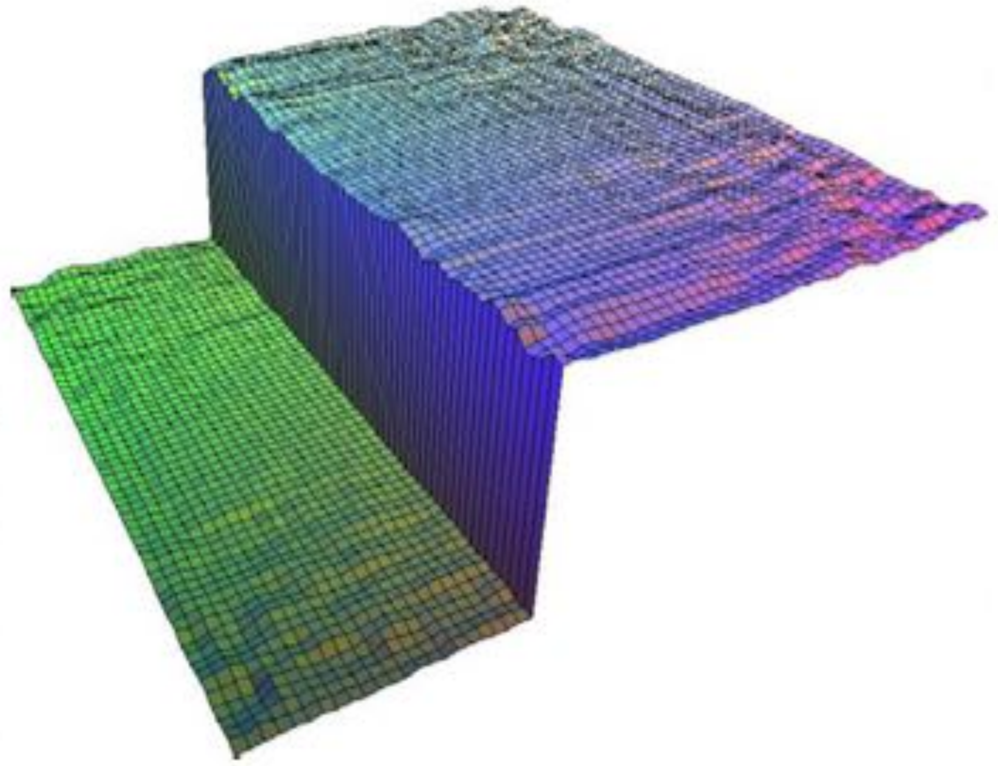
G(): gaussian about input pixel *p*



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

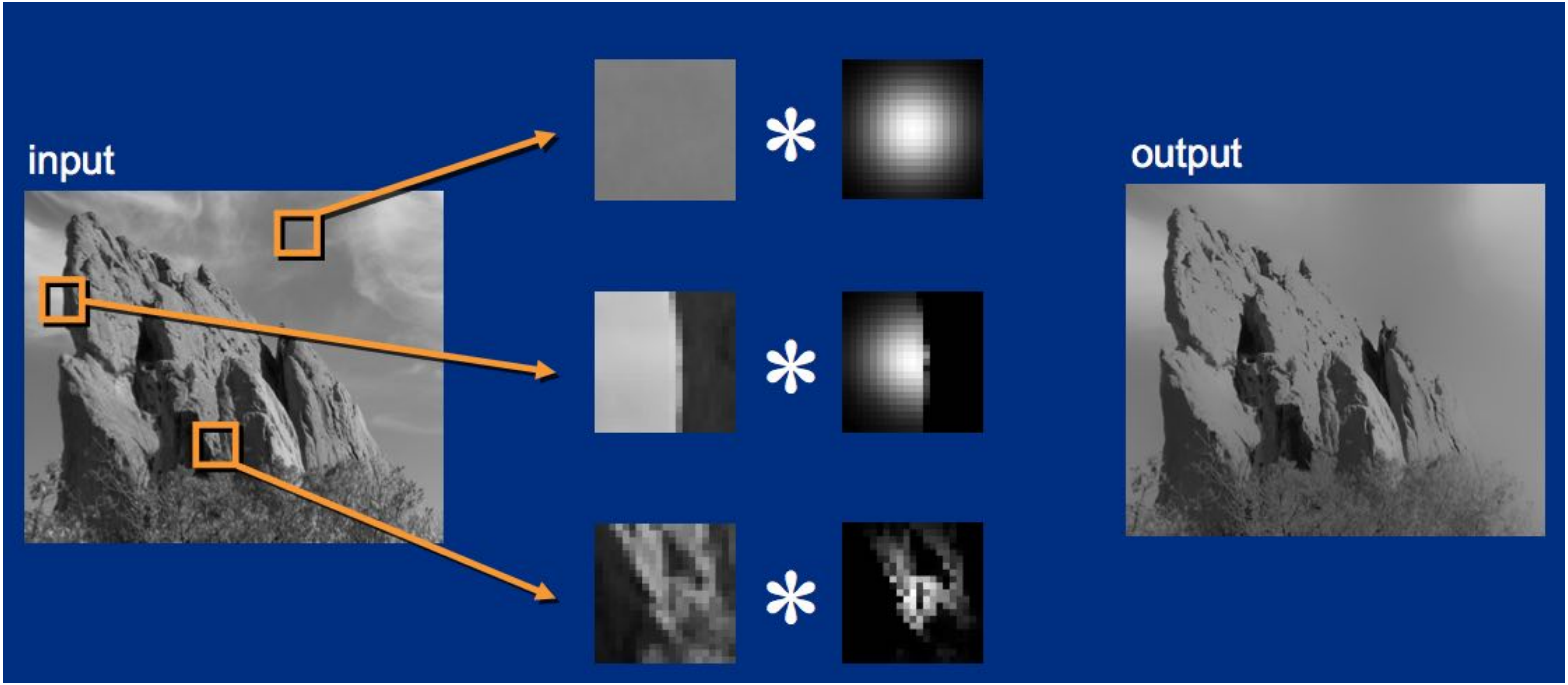


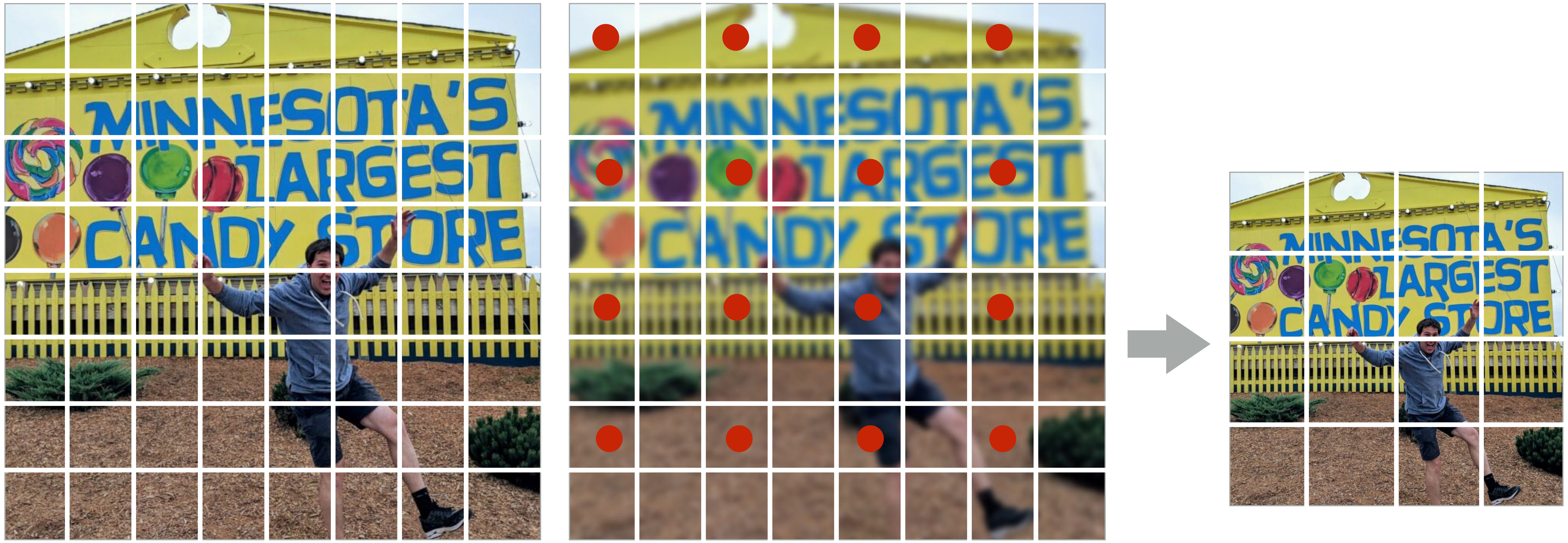
Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

What if we wish to localize image edits both in space and in frequency?

(Adjust certain frequency content of image… in a particular region of the image)

Downsample

- Step 1: Remove high frequencies (aka blur)
- Step 2: Sparsely sample pixels (in this example: every other pixel)



Downsample

- **Step 1: Remove high frequencies**
- **Step 2: Sparsely sample pixels (in this example: every other pixel)**

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

float weights[] = {1/64, 3/64, 3/64, 1/64,    // 4x4 blur (approx Gaussian)
                   3/64, 9/64, 9/64, 3/64,
                   3/64, 9/64, 9/64, 3/64,
                   1/64, 3/64, 3/64, 1/64};

for (int j=0; j<HEIGHT/2; j++) {
   for (int i=0; i<WIDTH/2; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<4; jj++)
         for (int ii=0; ii<4; ii++)
            tmp += input[(2*j+jj)*(WIDTH+2) + (2*i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH/2 + i] = tmp;
  }
}
```

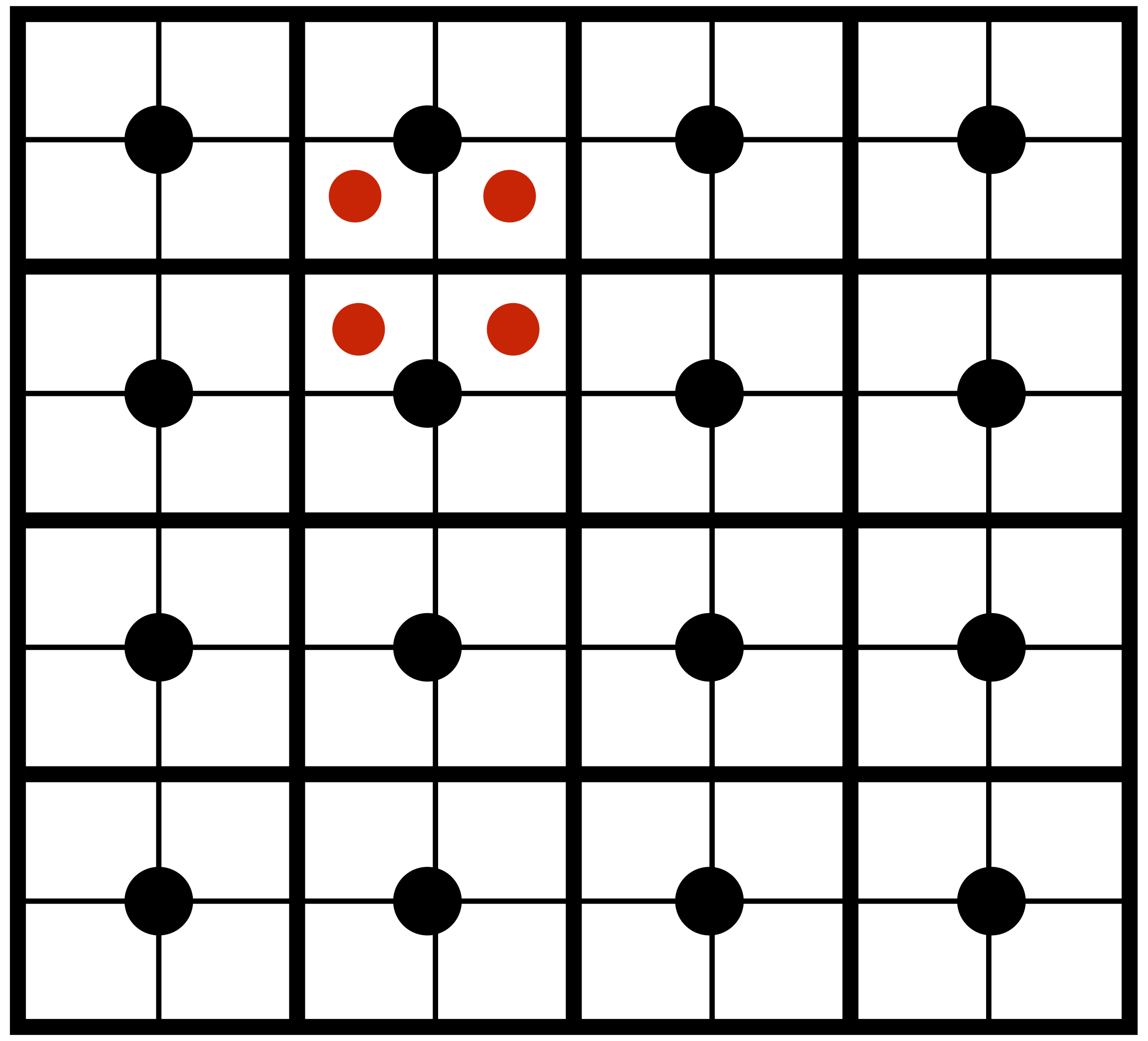
Upsample

Via bilinear interpolation of samples from low resolution image

Upsample

Via bilinear interpolation of samples from low resolution image

```
float input[WIDTH * HEIGHT];
float output[2*WIDTH * 2*HEIGHT];

for (int j=0; j<2*HEIGHT; j++) {
   for (int i=0; i<2*WIDTH; i++) {
      int row = j/2;
      int col = i/2;
      float w1 = (i%2) ? .75f : .25f;
      float w2 = (j%2) ? .75f : .25f;

      output[j*2*WIDTH + i] = w1 * w2 * input[row*WIDTH + col] +
                              (1.0-w1) * w2 * input[row*WIDTH + col+1] +
                              w1 * (1-w2) * input[(row+1)*WIDTH + col] +
                              (1.0-w1)*(1.0-w2) * input[(row+1)*WIDTH + col+1];
  }
}
```



Gaussian pyramid

$G_0 = \text{image}$

$G_1 = \text{down}(G_0)$

$G_2 = \text{down}(G_1)$

Each image in pyramid contains increasingly low-pass filtered signal

down() = downsample operation

Gaussian pyramid

G_0

Gaussian pyramid

G_1

Gaussian pyramid

G_2

Gaussian pyramid

G_3

Gaussian pyramid

G_4

Gaussian pyramid

G_5

Laplacian pyramid

$L_0 = G_0 - up(G_1)$

G_0

$G_1 = down(G_0)$

Each (increasingly numbered) level in Laplacian pyramid represents a band of (increasingly lower) frequency information in the image

[Burt and Adelson 83]

Laplacian pyramid

$L_0 = G_0 - \text{up}(G_1)$

$L_1 = G_1 - \text{up}(G_2)$

Laplacian pyramid

$L_0 = G_0$ - up(G_1)

$L_1 = G_1$ - up(G_2)

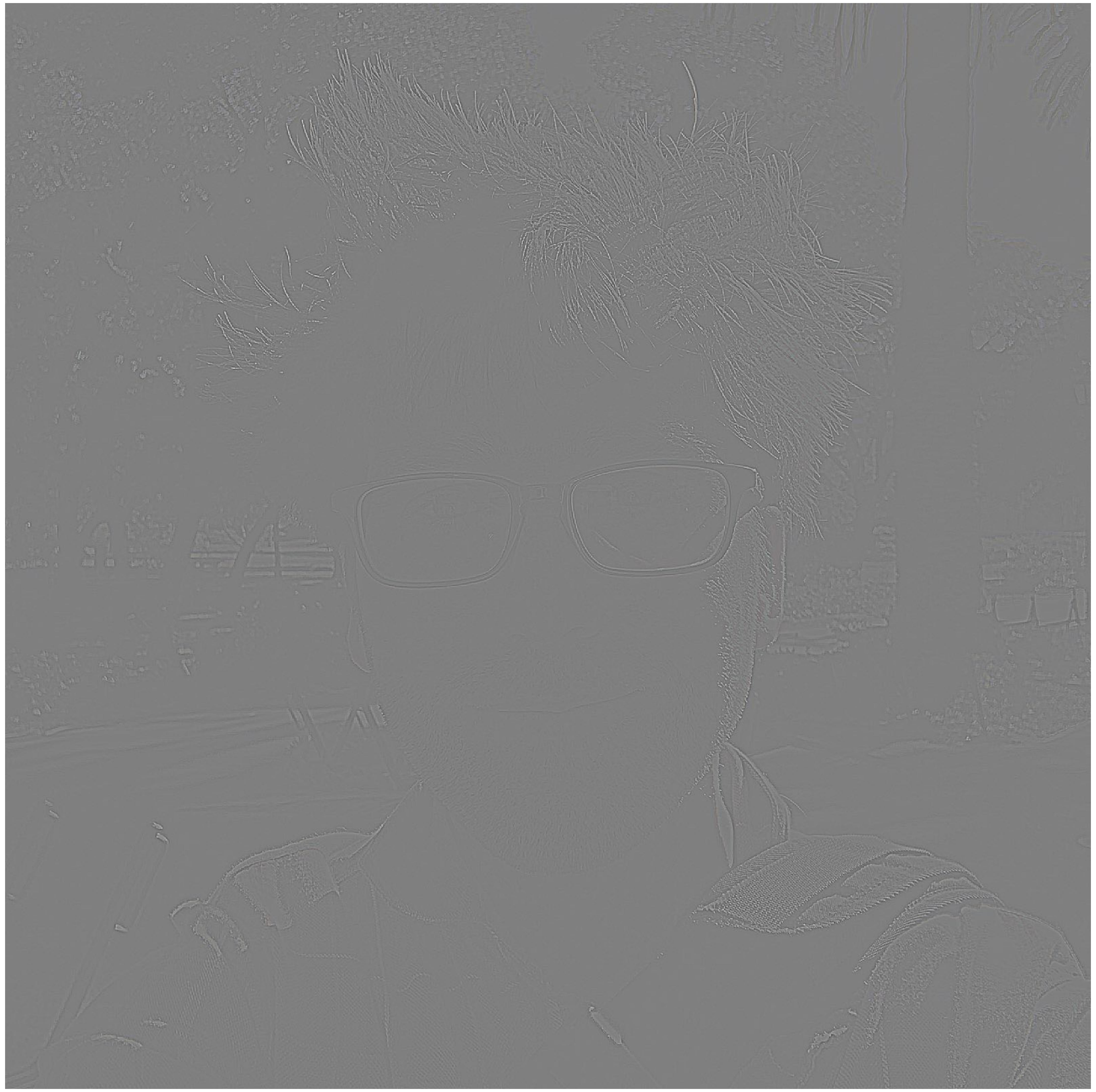
$L_2 = G_2$ - up(G_3)

$L_3 = G_3$ - up(G_4)

$L_4 = G_4$

Question: how do you reconstruct original image from its Laplacian pyramid?

Laplacian pyramid



$L_0 = G_0 - up(G_1)$

Laplacian pyramid

$L_1 = G_1 - up(G_2)$

Laplacian pyramid

$L_2 = G_2 - \mathrm{up}(G_3)$

Laplacian pyramid

$L_3 = G_3 - \mathrm{up}(G_4)$

Laplacian pyramid

$L_4 = G_4 - \text{up}(G_5)$

Laplacian pyramid

$L_5 = G_5$

Gaussian/Laplacian pyramid summary

- Gaussian and Laplacian pyramids are image representations where each pixel maintains information about frequency content in a region of the image

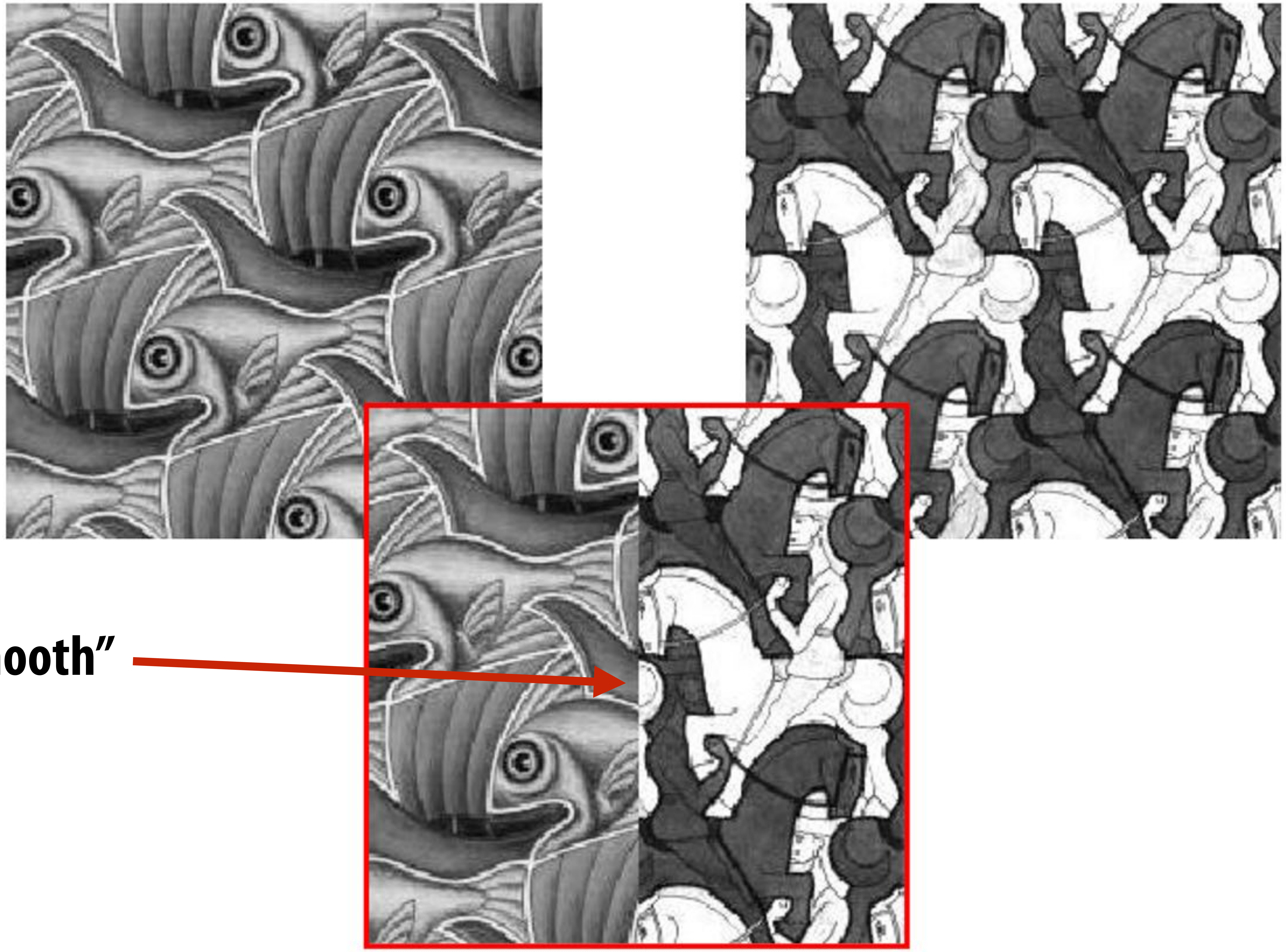
- $G_i(x,y)$ — frequencies up to limit given by *i*

- $L_i(x,y)$ — frequencies added to G_{i+1} to get G_i

- Notice: to boost the band of frequencies in image around pixel (x,y), increase coefficient $L_i(x,y)$ in Laplacian pyramid

Application: image blending

Consider a simple case where we want to blend two patterns:

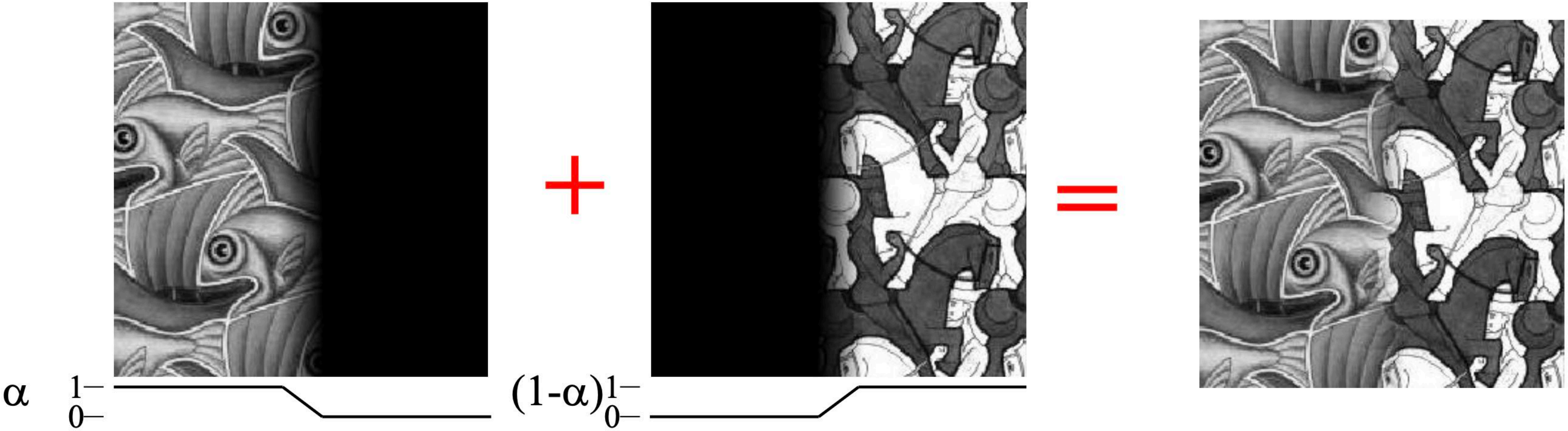


Problem: not “smooth”

Slide credit: Efros

"Feather" the alpha mask

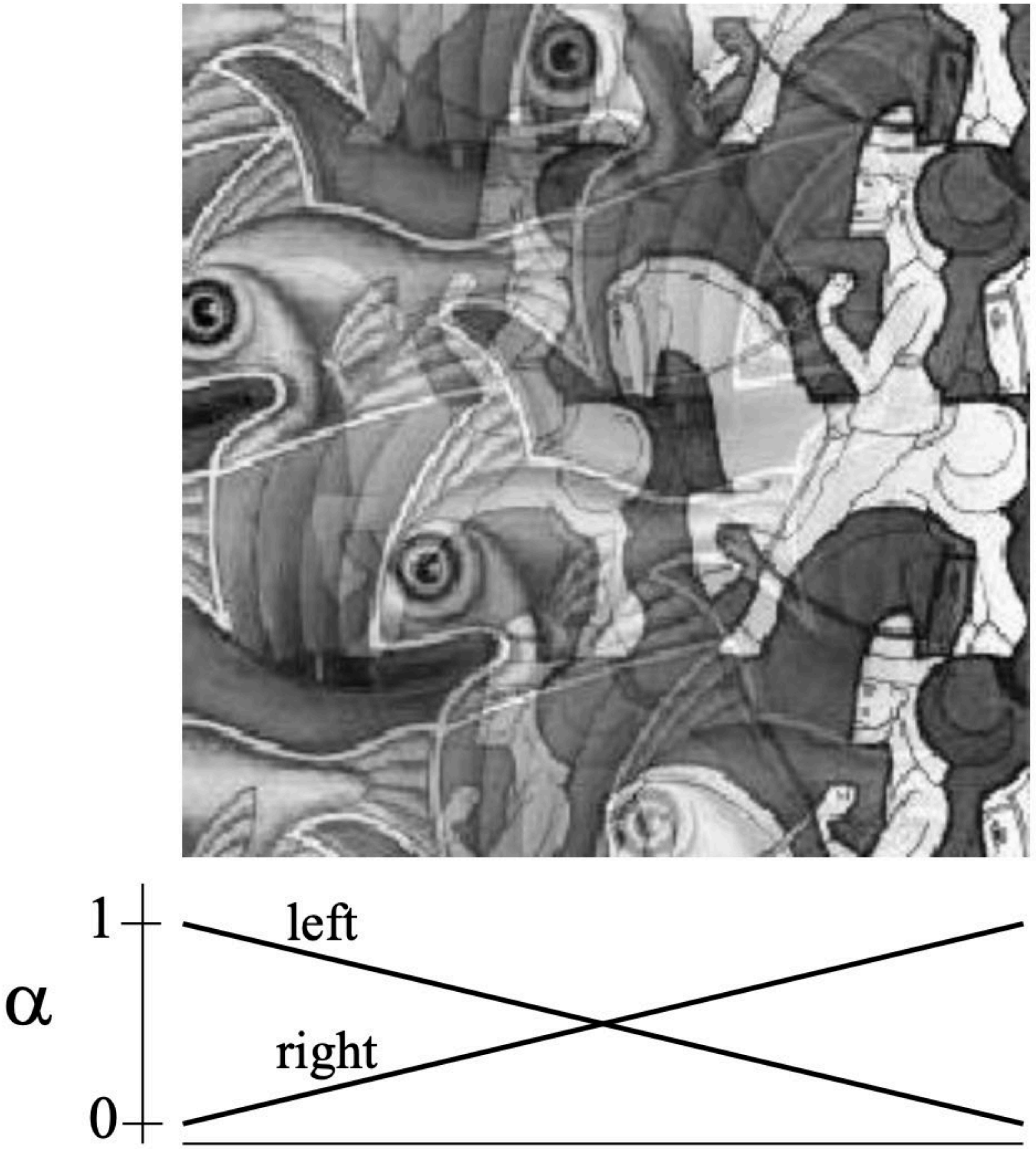
For a "smoother" look…

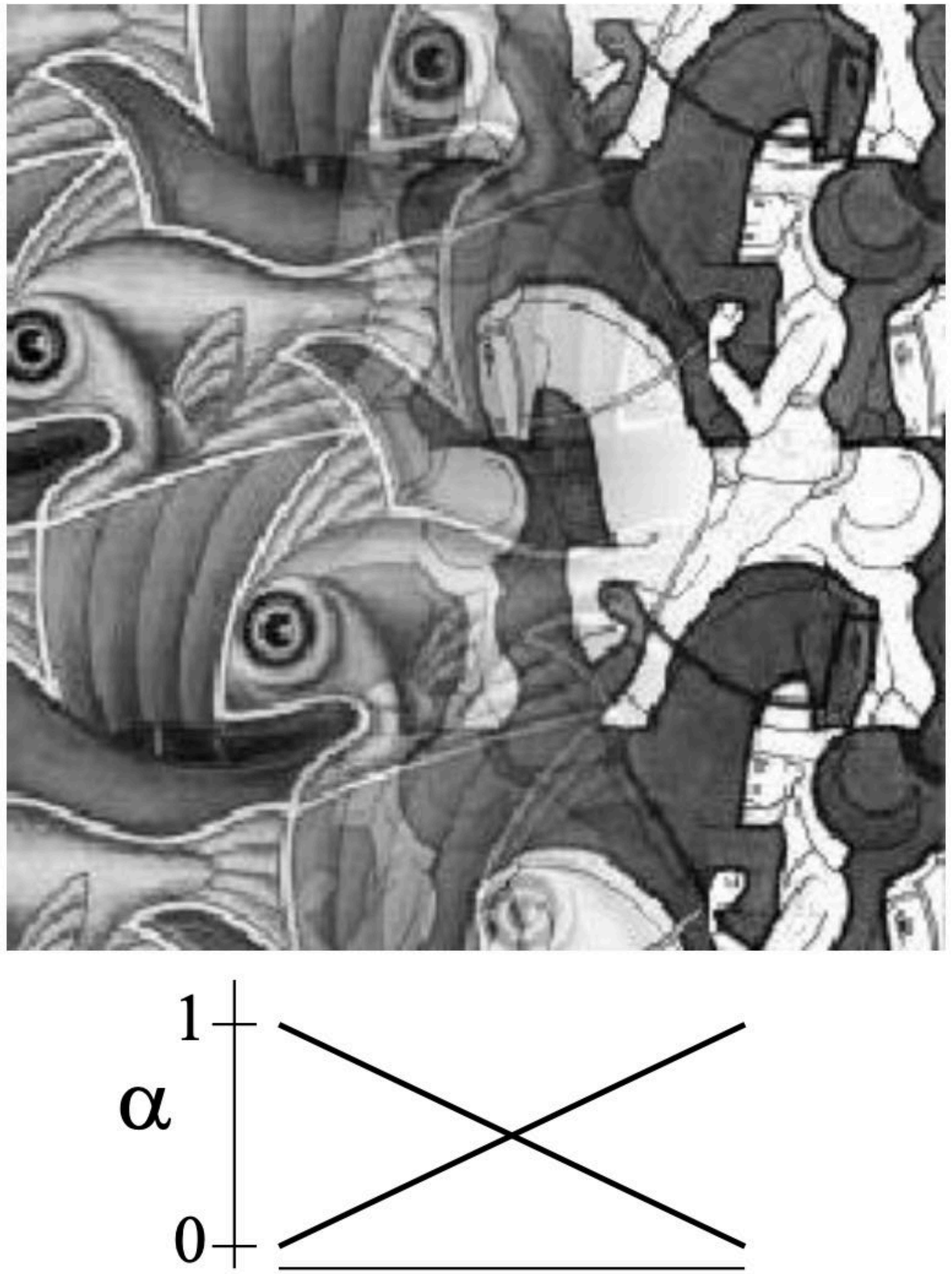


$$I_{\text{blend}} = \alpha\, I_{\text{left}} + (1-\alpha)\, I_{\text{right}}$$

Slide credit: Efros

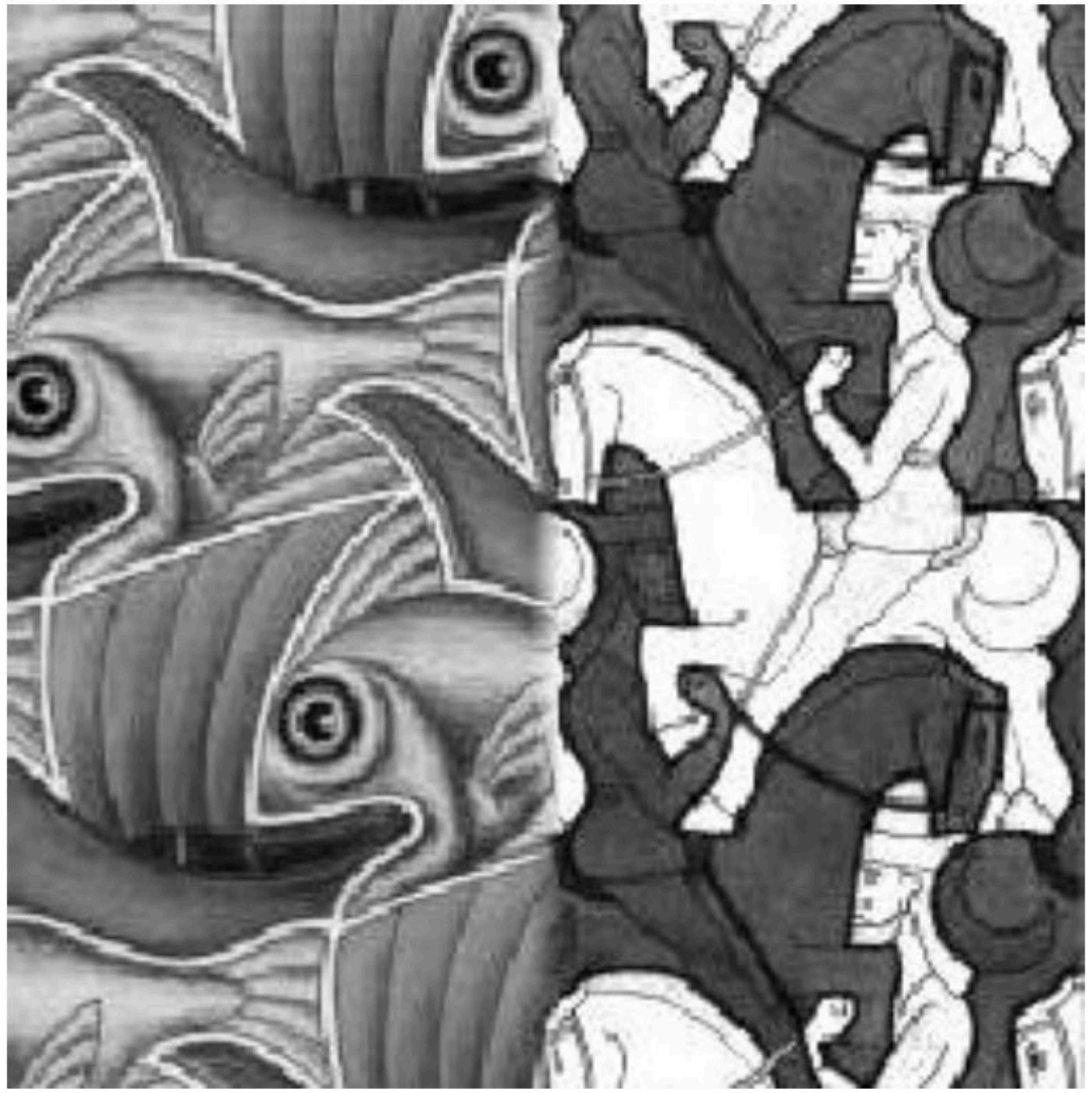
Effect of feather window size

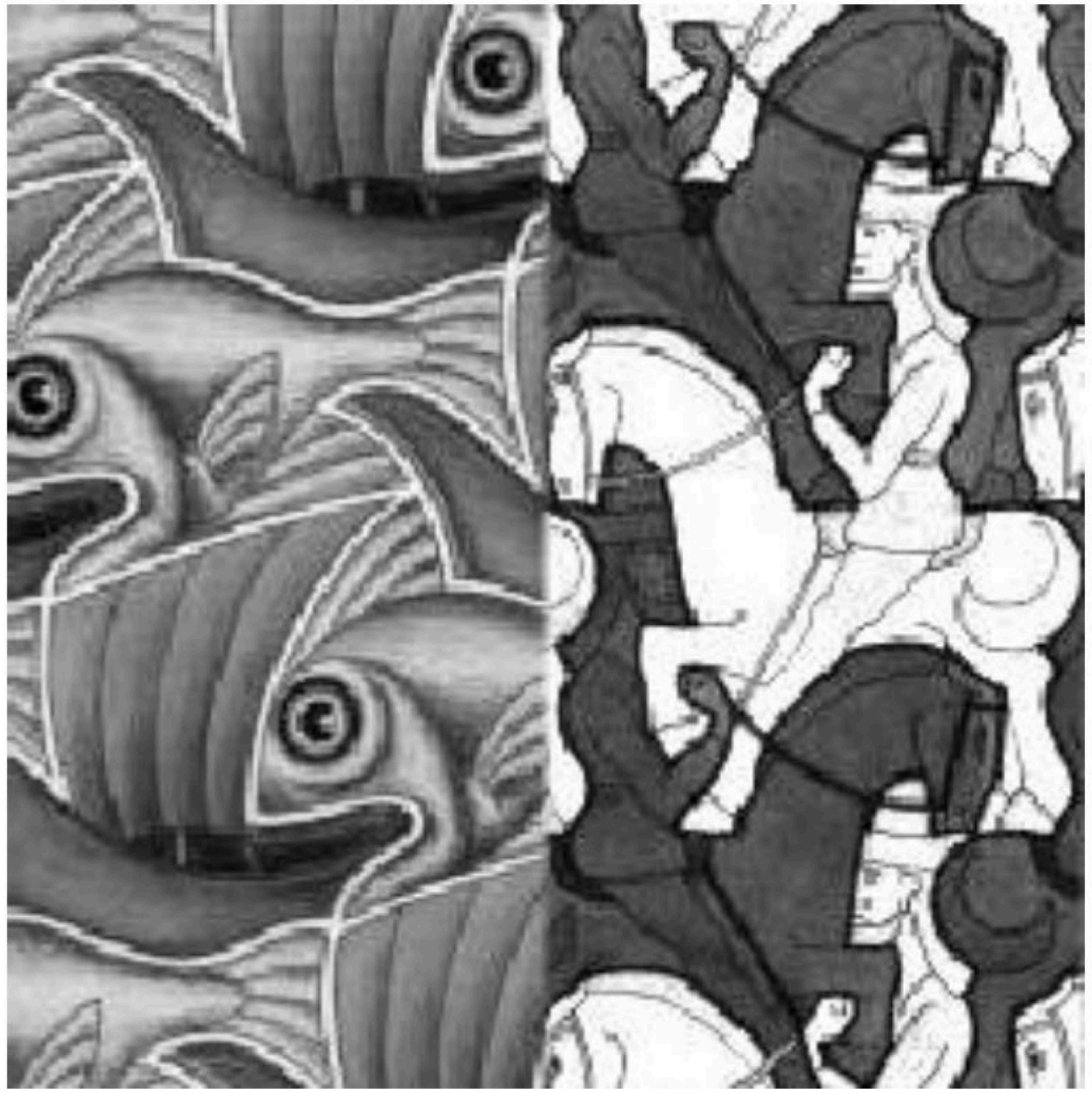


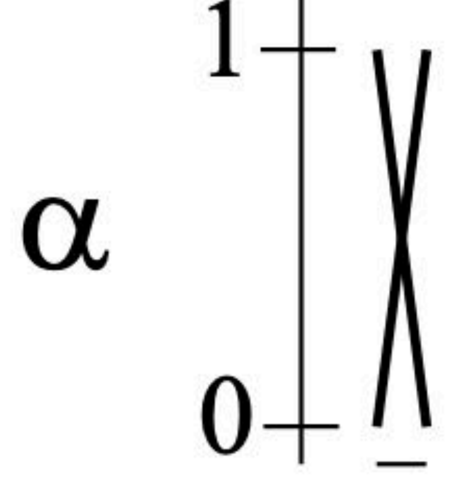


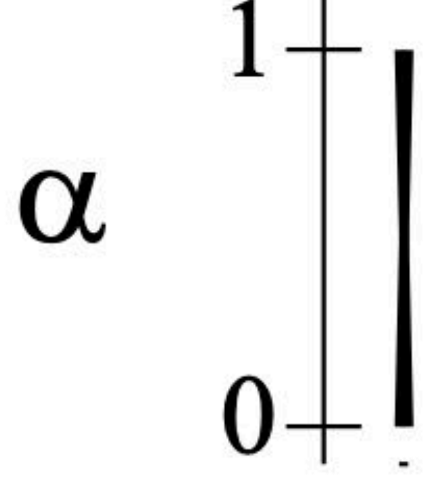
“Ghosting” visible is feather window (transition) is too large

Effect of feather window size









Seams visible is feather window (transition) is too small

Slide credit: Efros

What do we want

- To avoid seams, transition window should be >= size of largest prominent feature

- To avoid ghosting, transition window should be smaller than ~ 2X smallest prominent feature

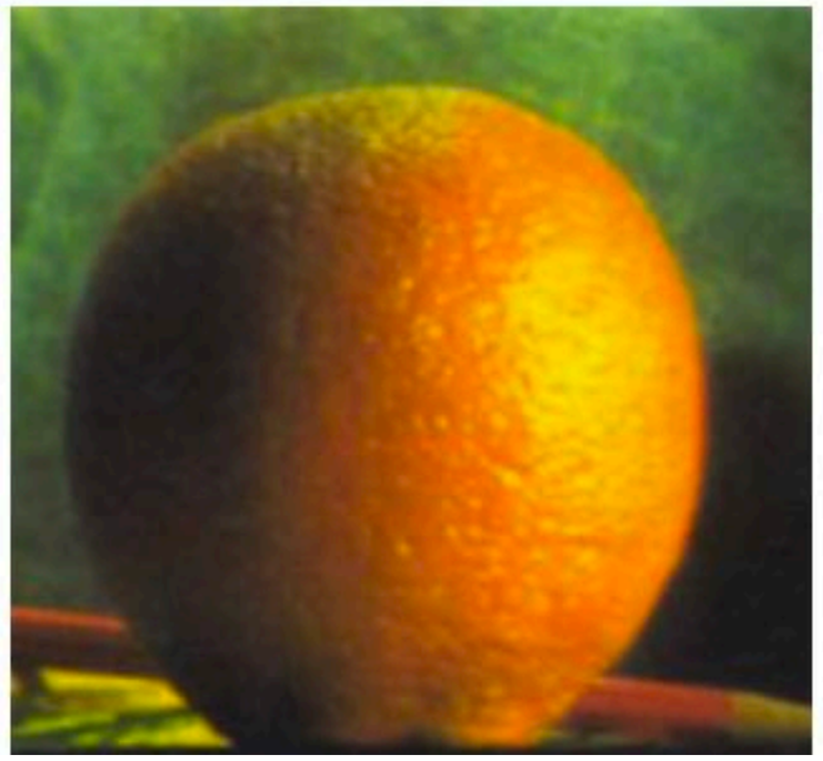
- In other words, the largest and smallest features need to be within a factor of two for feathering to generate good results

- Intuition:
  - Coarse structure of images (large features) should transition slowly between images
  - Fine structure should blend quickly!

Slide credit: Efros, Guerzhoy

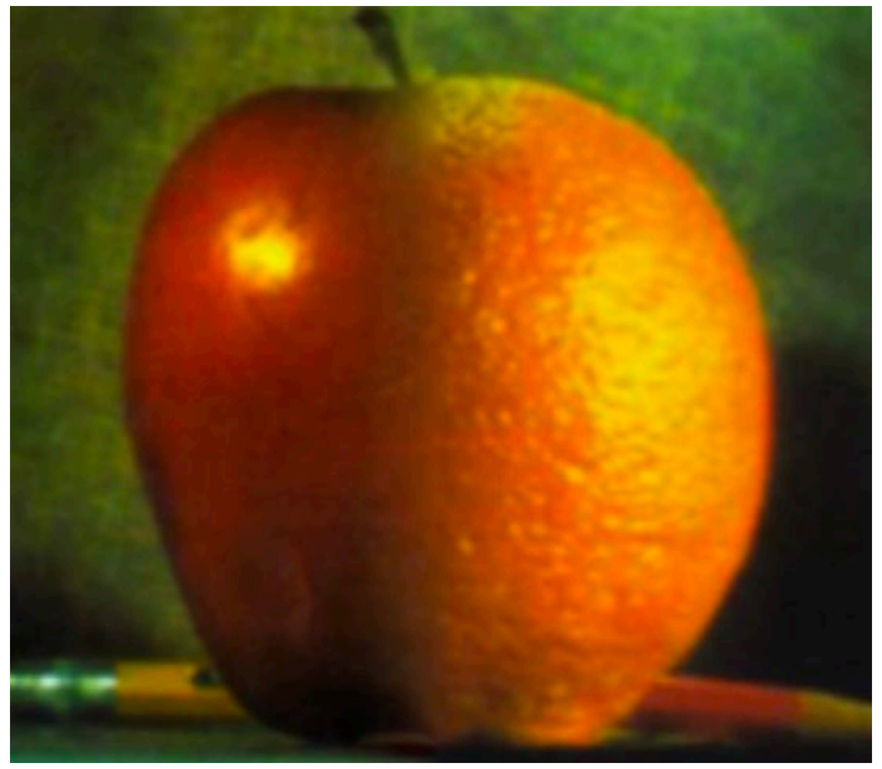
Idea: blend laplacian pyramids (not pixels) according to gaussian pyramid of alpha mask

Source apple

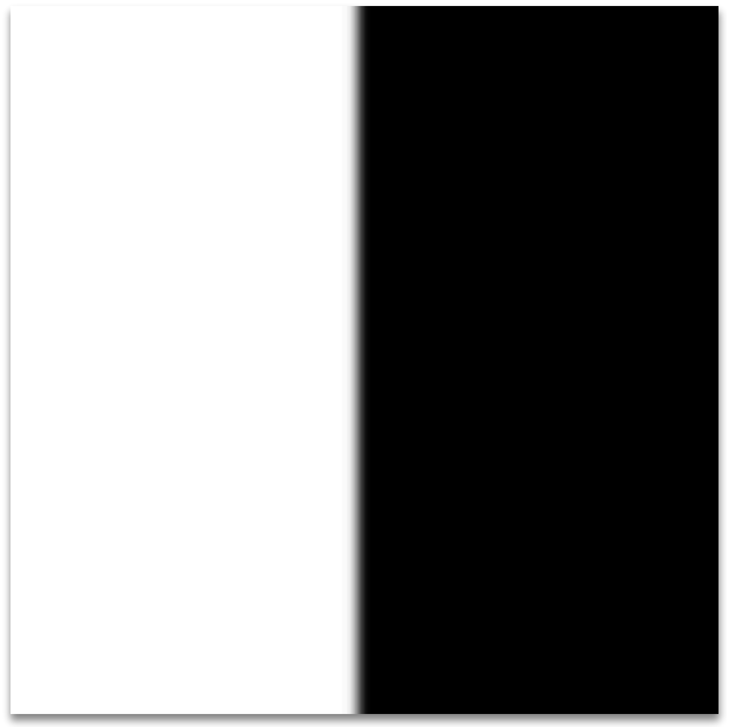


Source orange

Mask



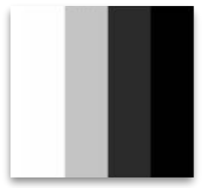
Blended Laplacian pyramid



Mask G0

Mask G1

Mask G2



Mask G3

Mask G3

Modern application: HDR photography

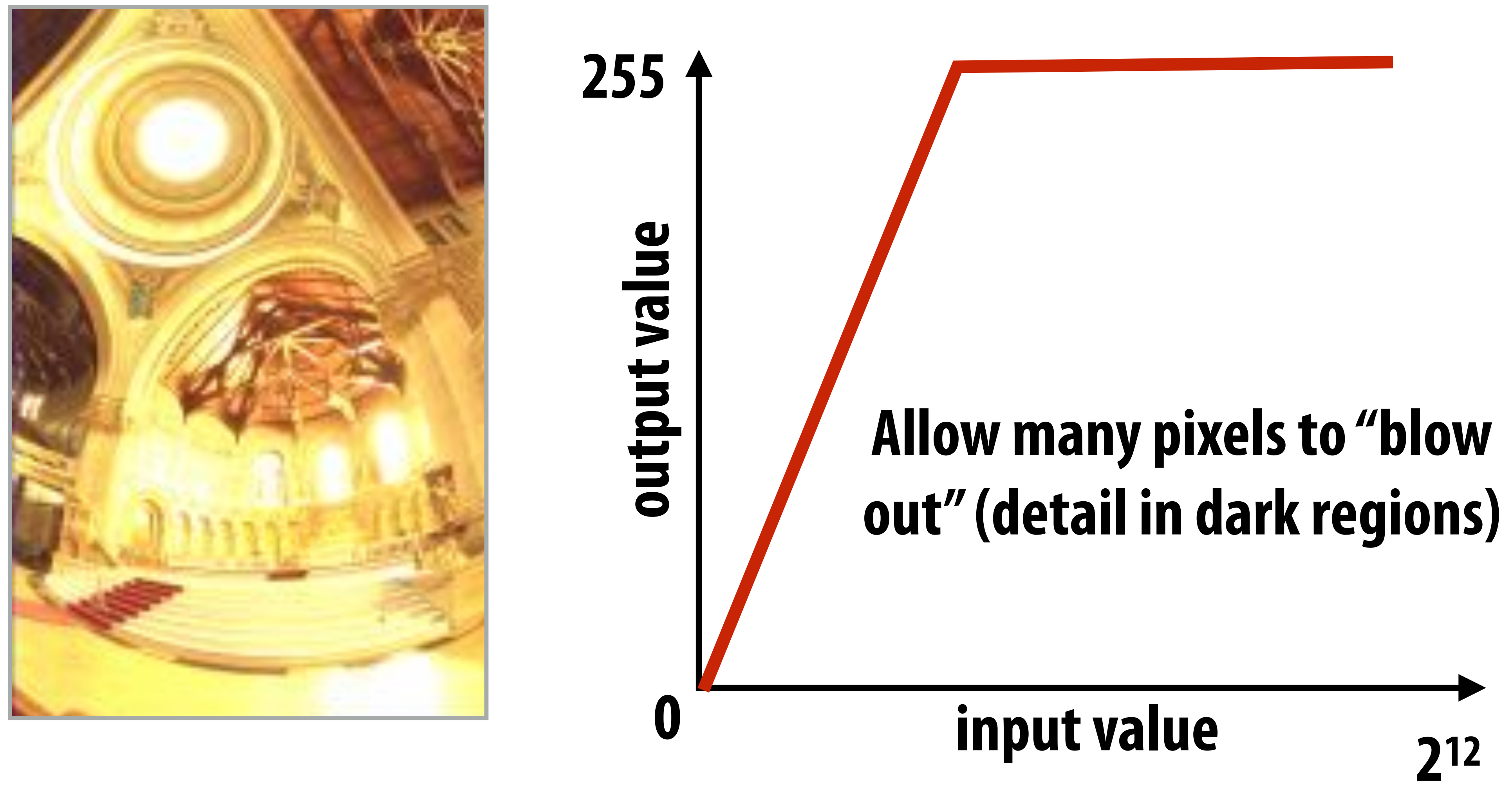
Saturated pixels

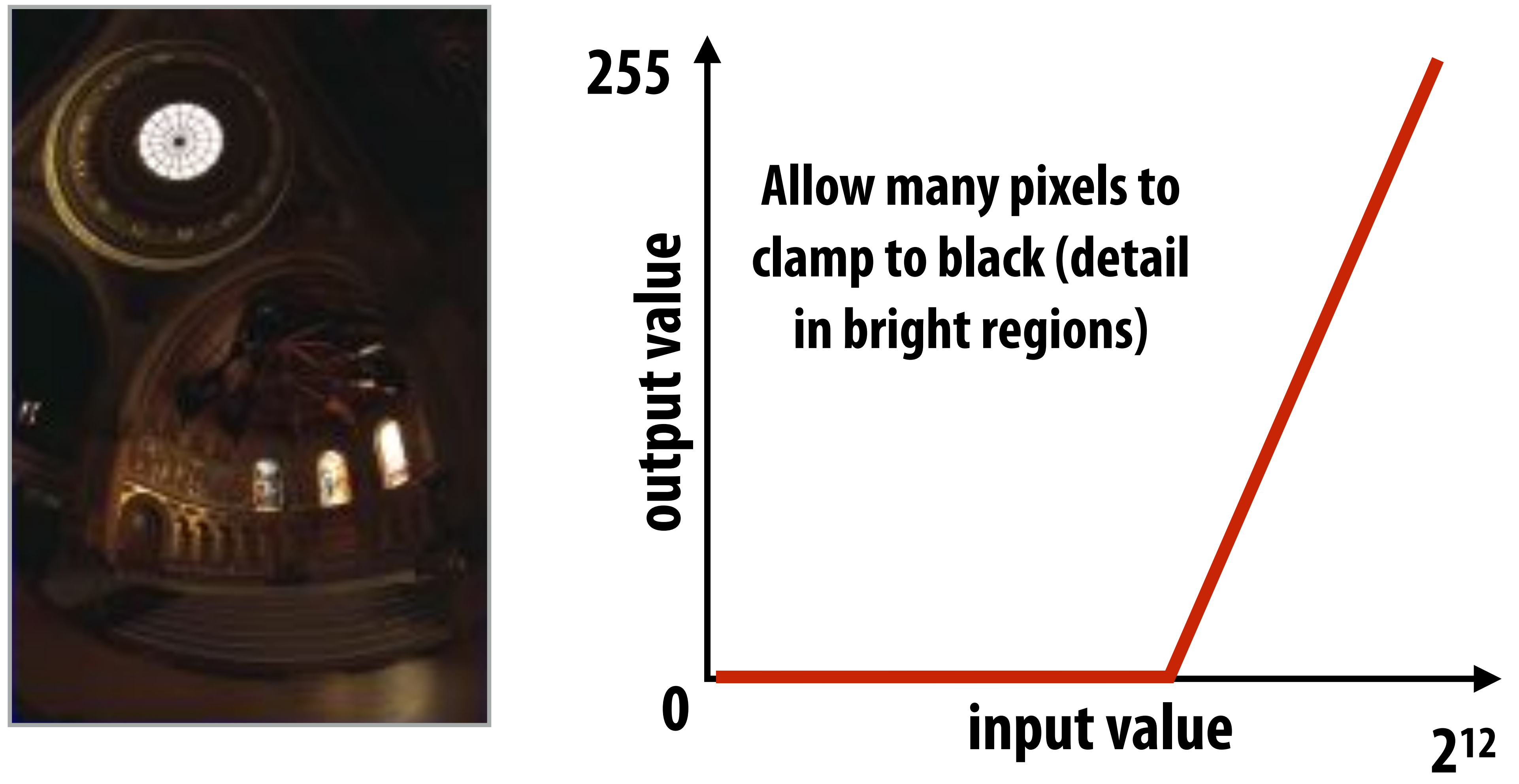
Saturated pixels

Credit: P. Debevec

Global tone mapping

- Measured image values (by camera's sensor): 10-12 bits / pixel, but common image formats are 8-bits/pixel
- How to convert 12 bit number to 8 bit number?





High dynamic range image (HDR)
Detail in dark and light images

Image credit: Wikipedia

Local tone adjustment

Pixel values

Weights

Improve picture's aesthetics by locally adjusting contrast, boosting dark regions, decreasing bright regions

(no physical basis for this)

Combined image
(unique weights per pixel)

Image credit: Mertens 2007

Challenge of merging images

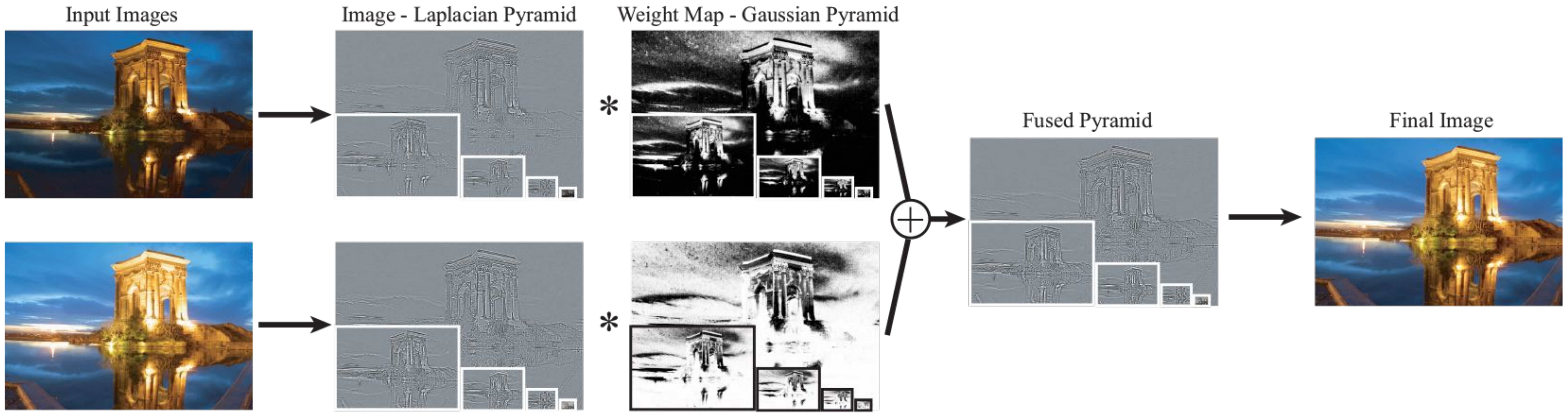
Four exposures (weights not shown)

Merged result (based on weight masks)
Notice heavy “banding” since absolute intensity
of different exposures is different

Merged result
(after blurring weight mask)
Notice “halos” near edges

Use of Laplacian pyramid in local tone mapping

- Compute weights for all Laplacian pyramid levels
- Merge pyramids (image features) not image pixels
- Then "flatten" merged pyramid to get final image



Merging Laplacian pyramids

Four exposures (weights not shown)

Merged result
(after blurring weight mask)
Notice "halos" near edges

Merged result
(based on multi-resolution pyramid merge)

Why does merging Laplacian pyramids work better than merging image pixels?

Summary

- Convolution is a powerful image processing operation
  - Different kernels = different effects

- Data-dependent kernels for edge-aware image processing (Laplacian filter)

- Gaussian and Laplacian pyramids: data structures for enabling edits localized to both spatial regions and frequency components of images