Lecture 3:

The Camera Image Processing Pipeline
(part 2)

Visual Computing Systems
Stanford CS348K, Spring 2022

Previous class and today…

The pixels you see on screen are quite different than the values recorded by the sensor in a modern digital camera.

Computation is now a fundamental aspect of producing high-quality pictures.

Sensor output ("RAW") → **Computation** →

Beautiful image that impresses your friends on Instagram

Summary: simplified image processing pipeline

- Correct pixel defects
- Align and merge (to create high signal to noise ration RAW image)
- Correct for sensor bias (using measurements of optically black pixels)
- Vignetting compensation
- White balance
- Demosaic

Last time!

(10-12 bits per pixel)
1 intensity value per pixel
Pixel values linear in energy

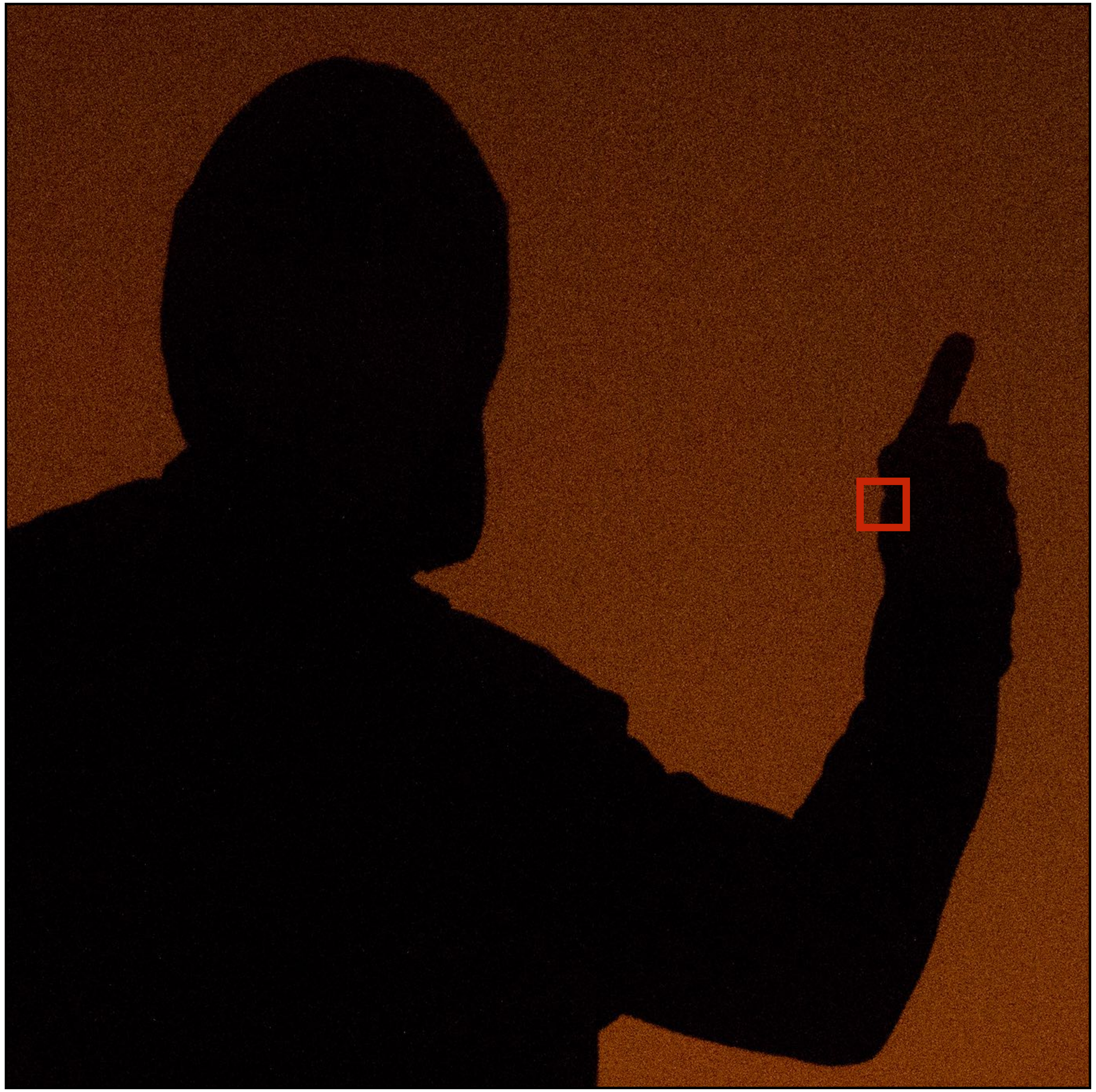
- Denoise
- Gamma Correction (non-linear mapping)
- Local tone mapping

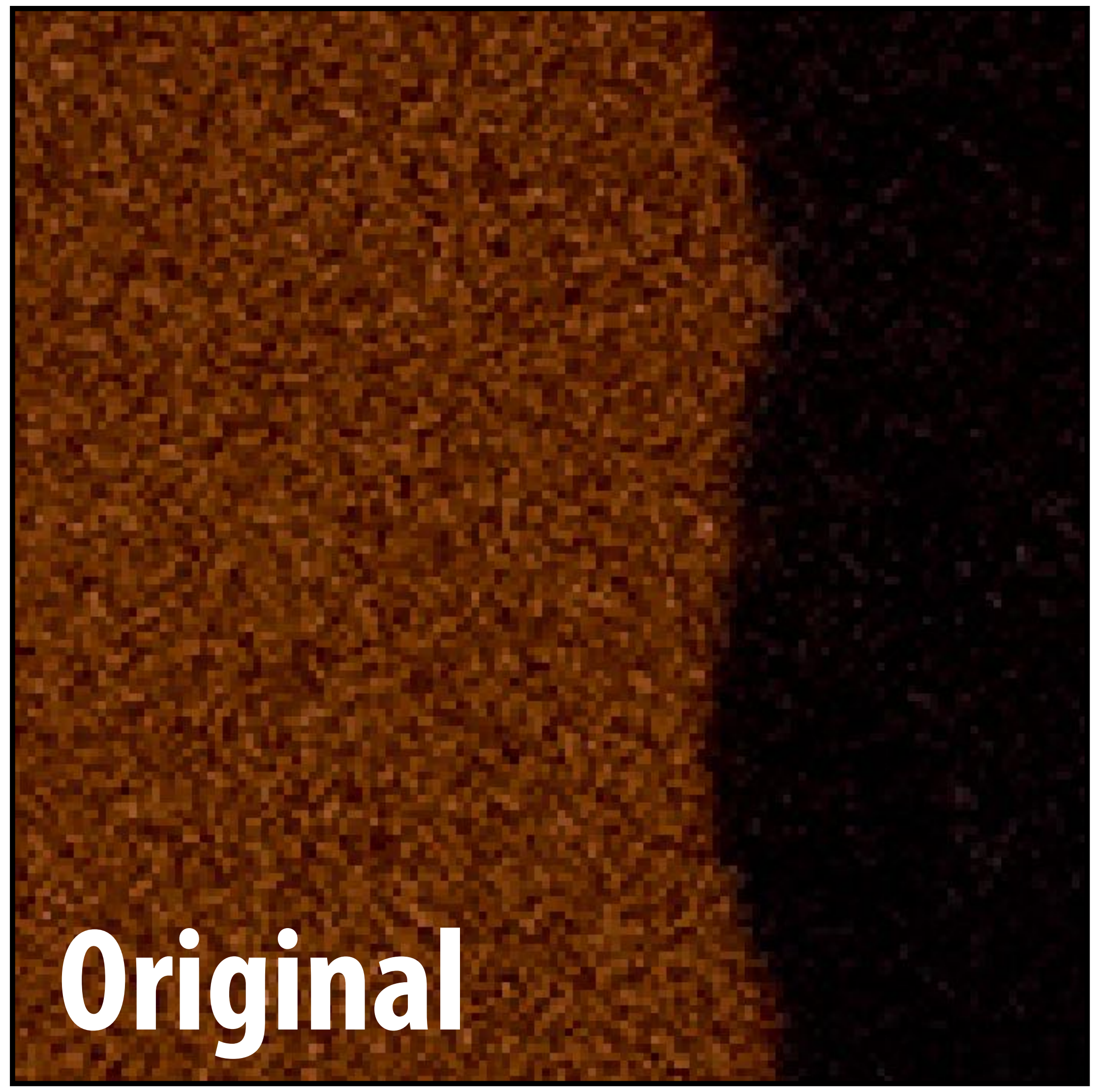
3 x (10-12) bits per pixel
RGB intensity per pixel
Pixel values linear in energy

- Final adjustments sharpen, fix chromatic aberrations, hue adjust, etc.

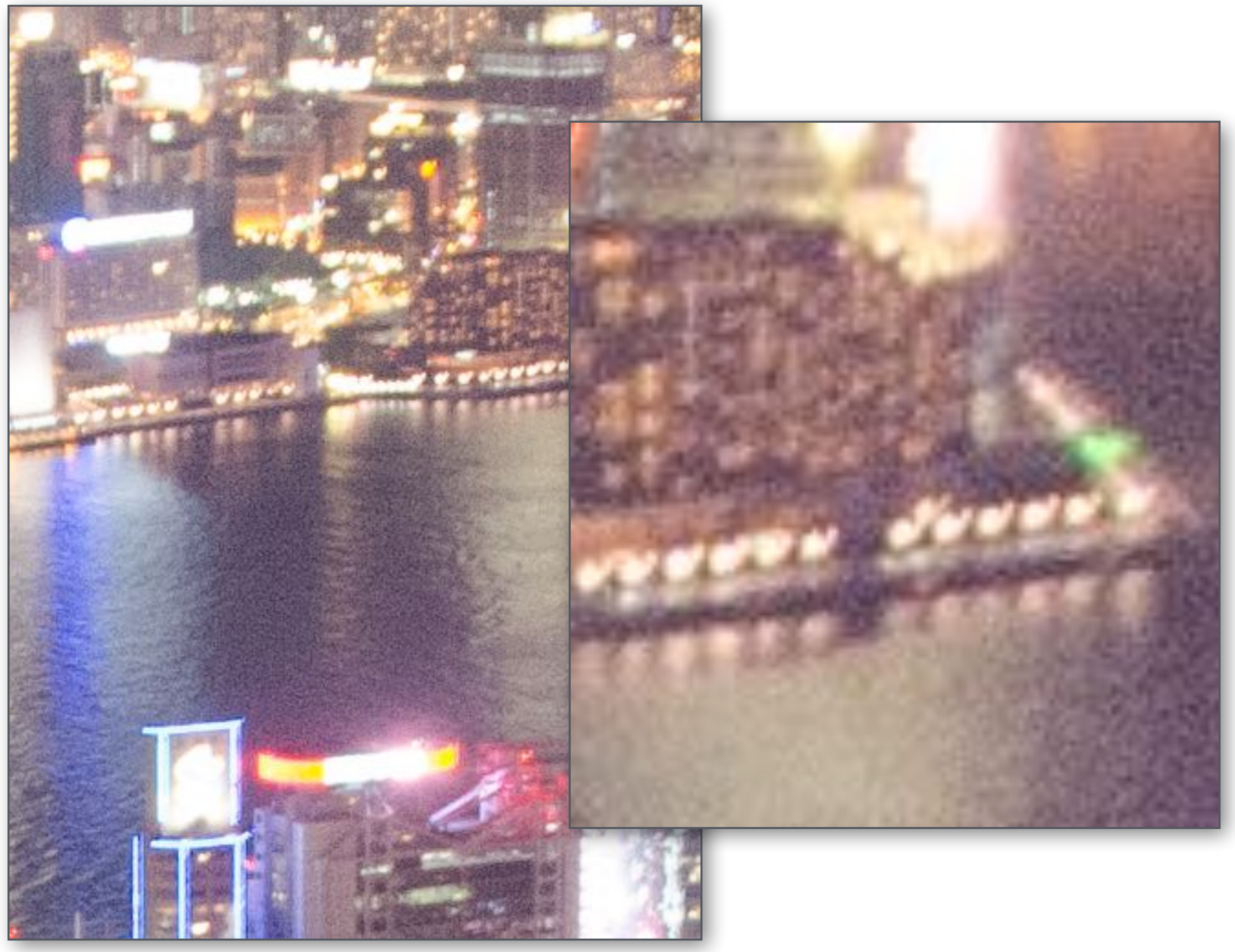
3x8-bits per pixel
Pixel values perceptually linear

Denoising





Reduce noise via image processing: denoising via downsampling



Downsample via point sampling

(noise remains)

Downsample via averaging 2x2 block of pixels

Noise reduced

Like a smaller number of bigger pixels!

Discrete 2D convolution

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

output image
(the result of convolving f with input image I)

filter

input image

Consider a $f(i,j)$ 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:

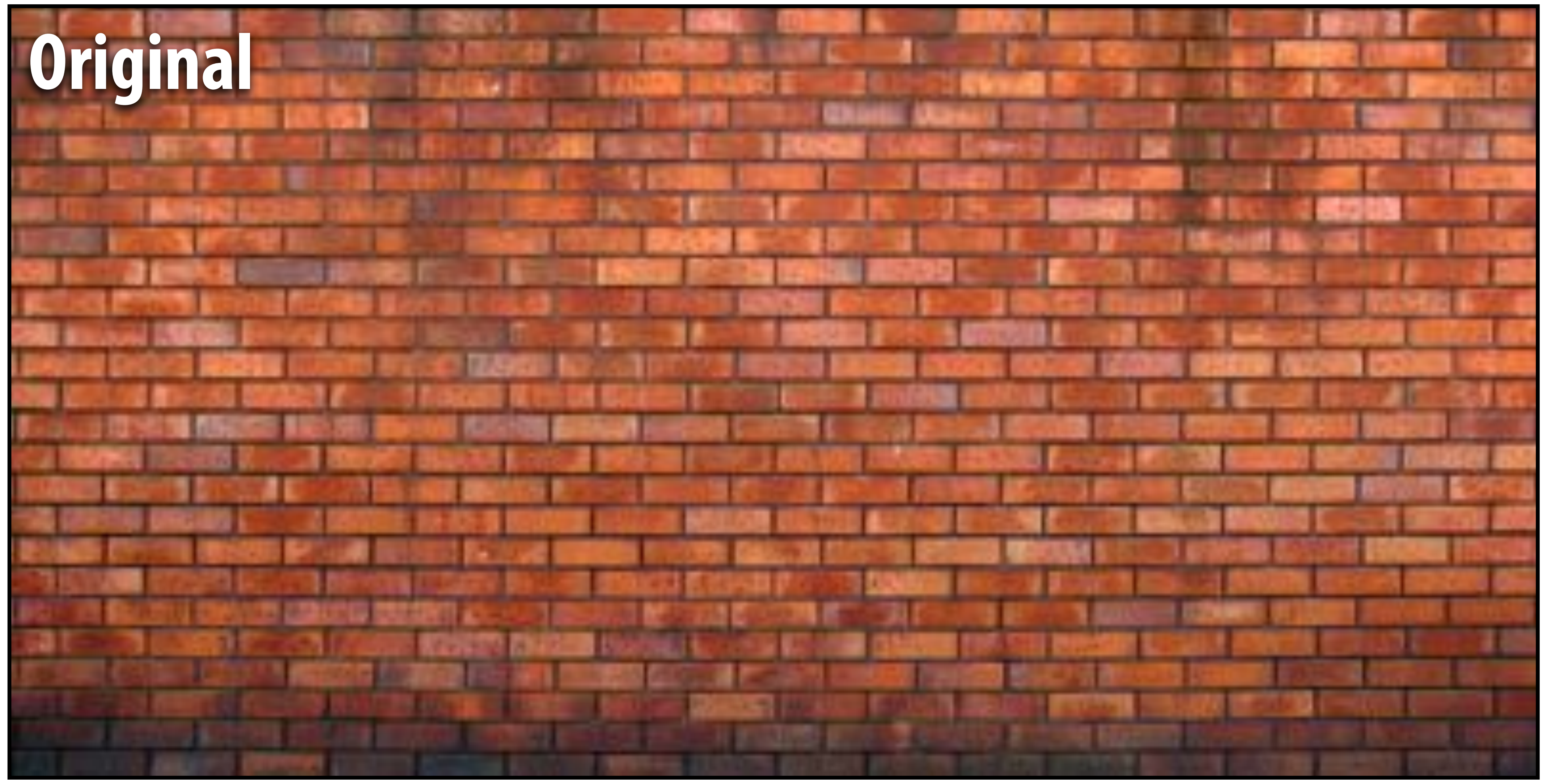
$f(i,j) = \mathbf{F}_{i,j}$ (often called: "filter weights", "filter kernel")

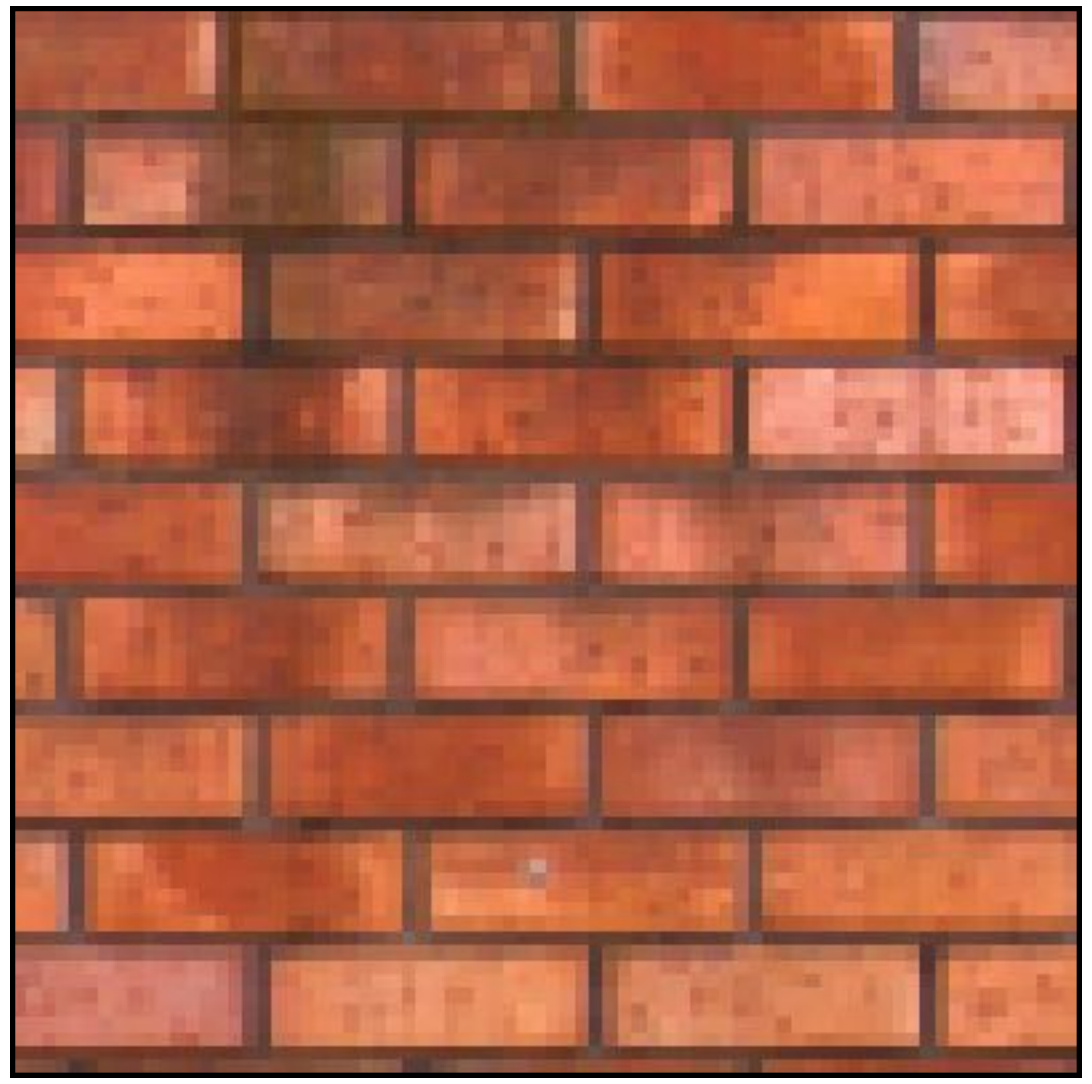
Simple 3x3 box blur in C code

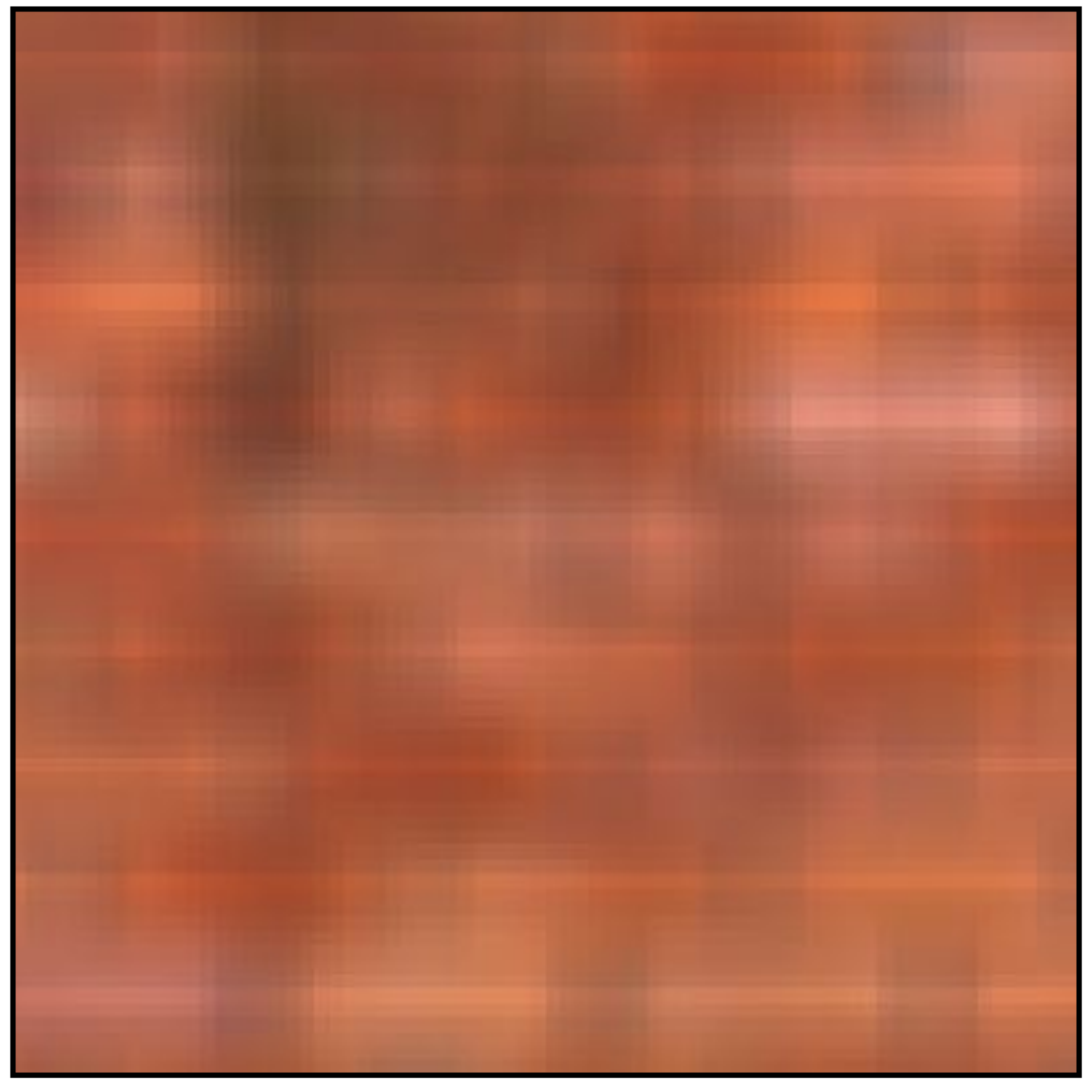
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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**

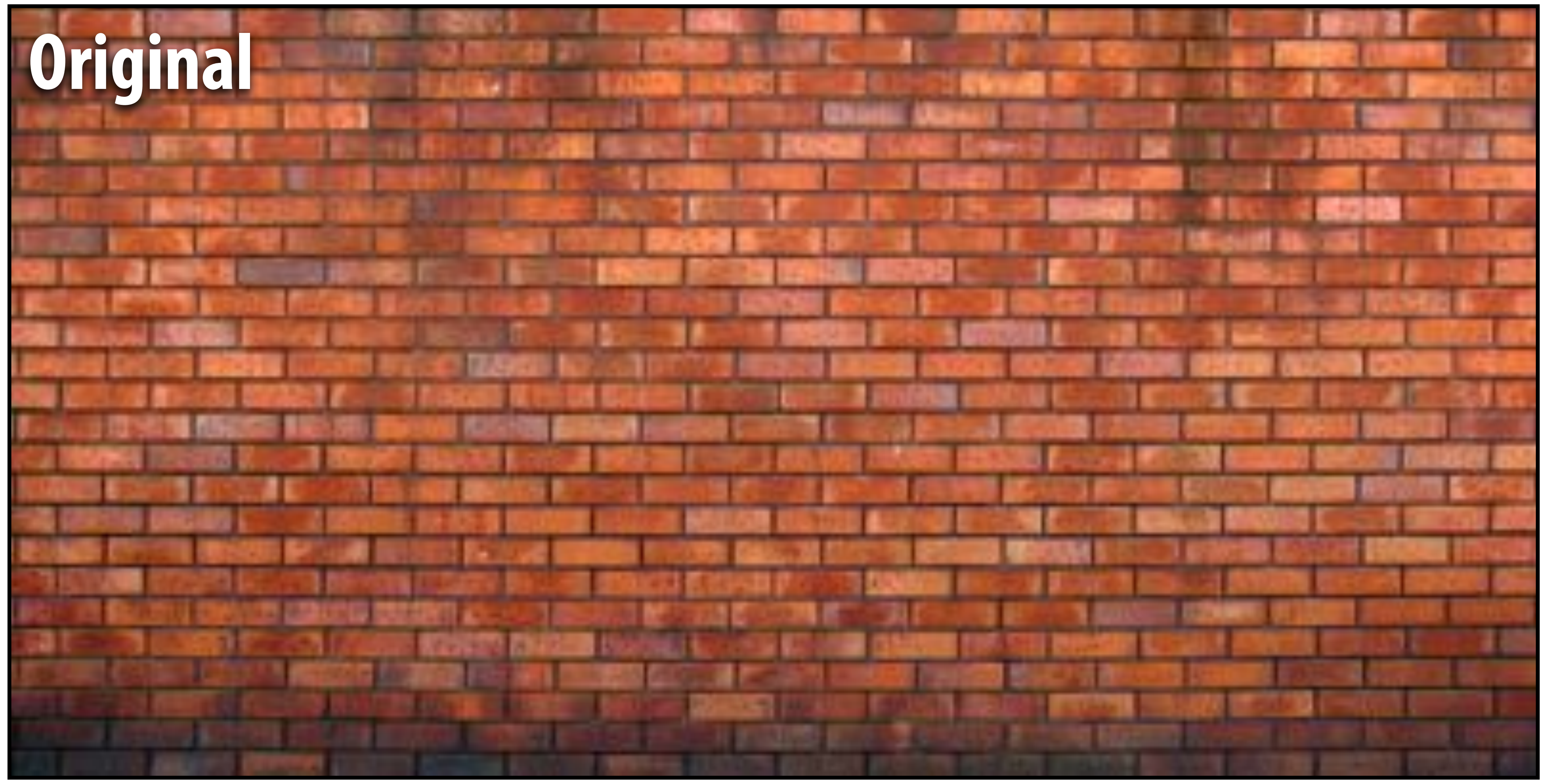
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

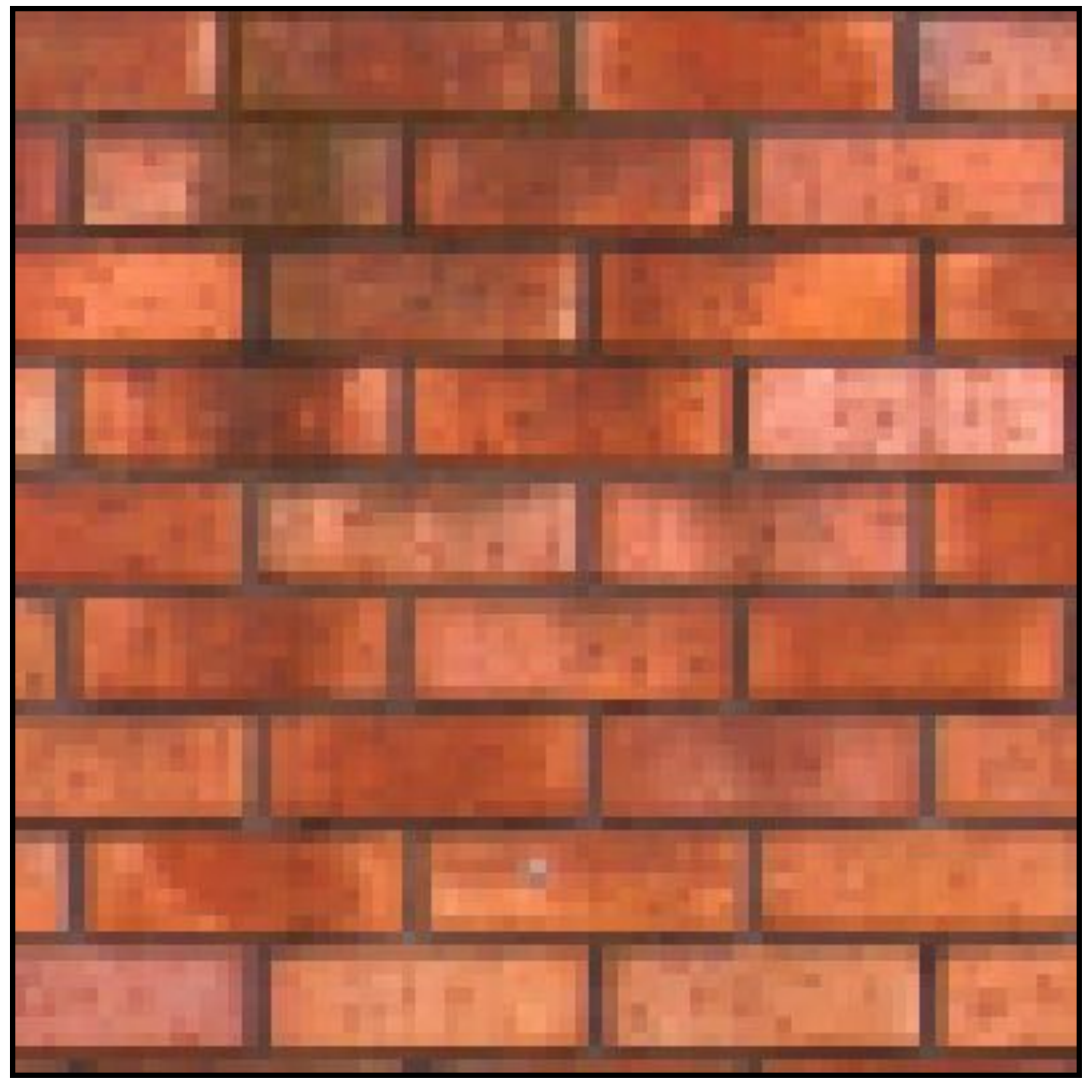
- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

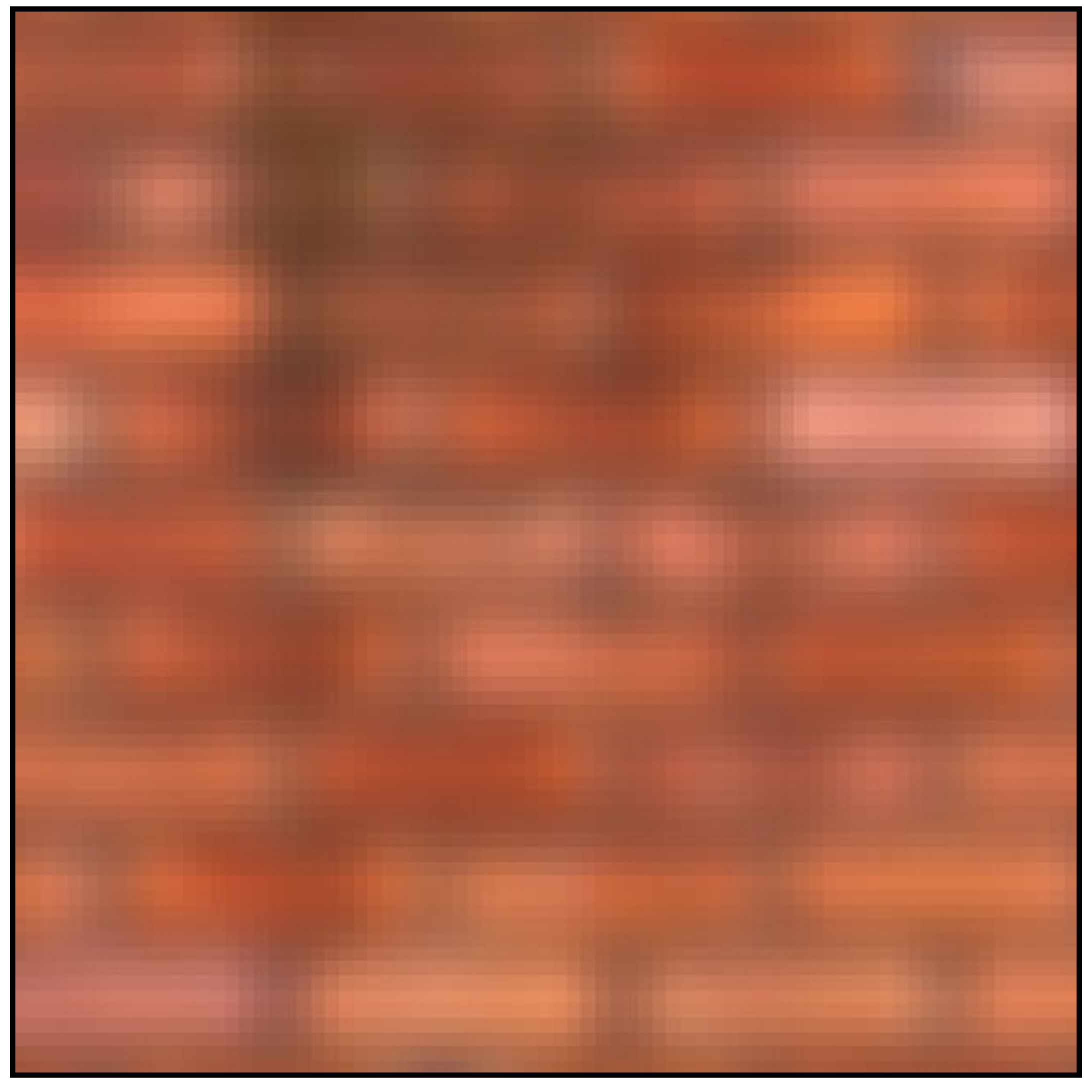
$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Note: this is a 5x5 truncated Gaussian filter

7x7 gaussian blur

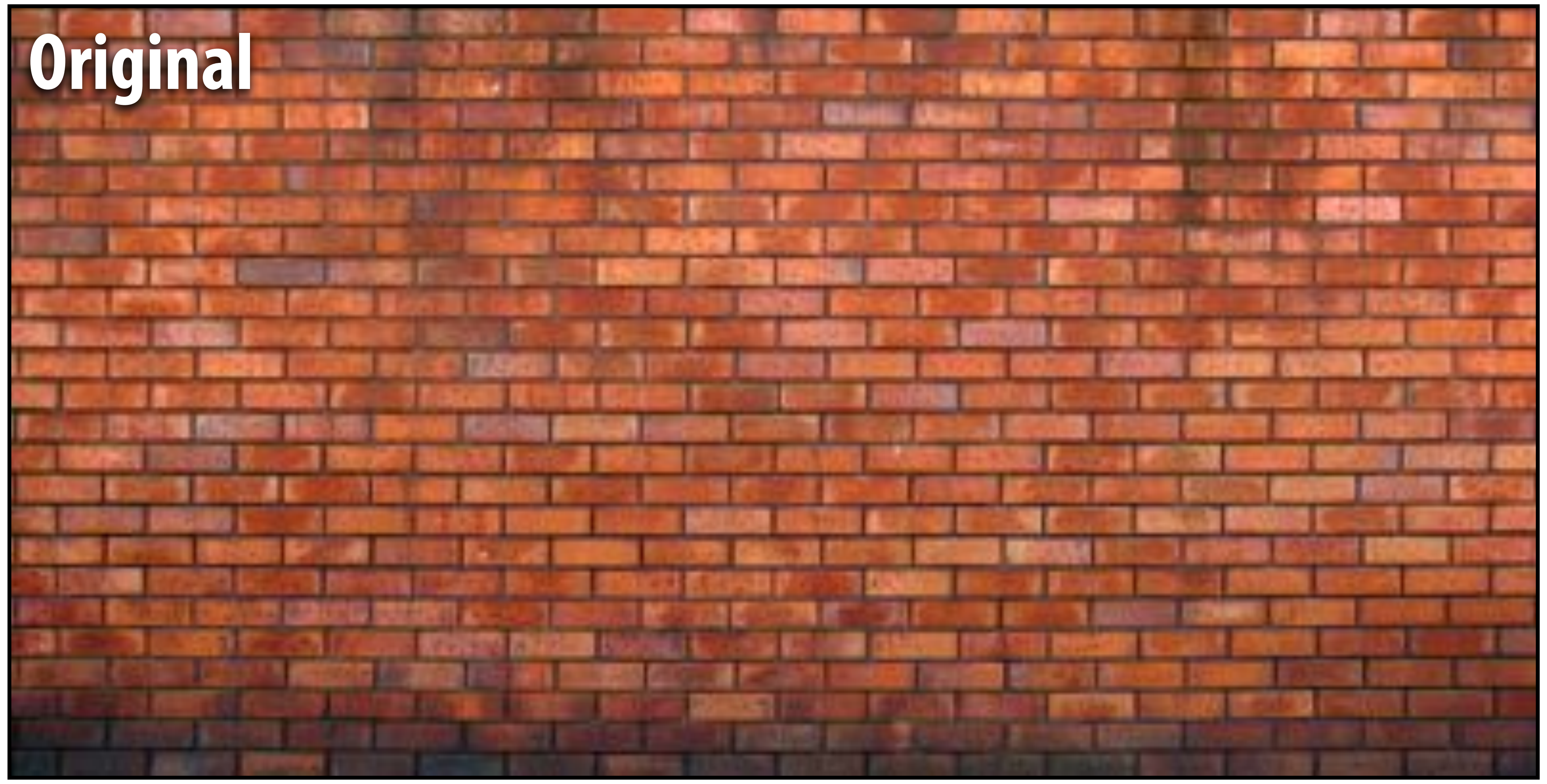


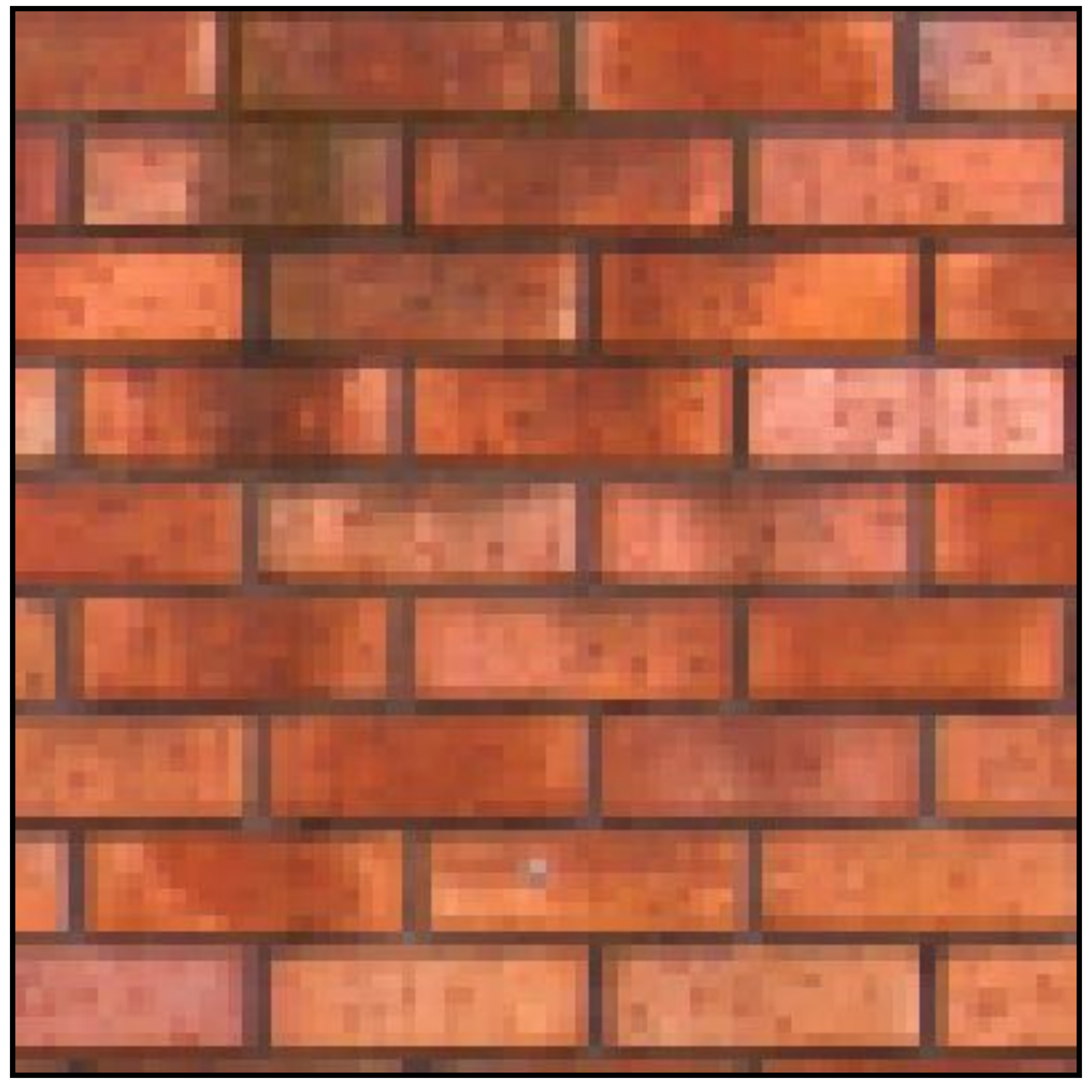


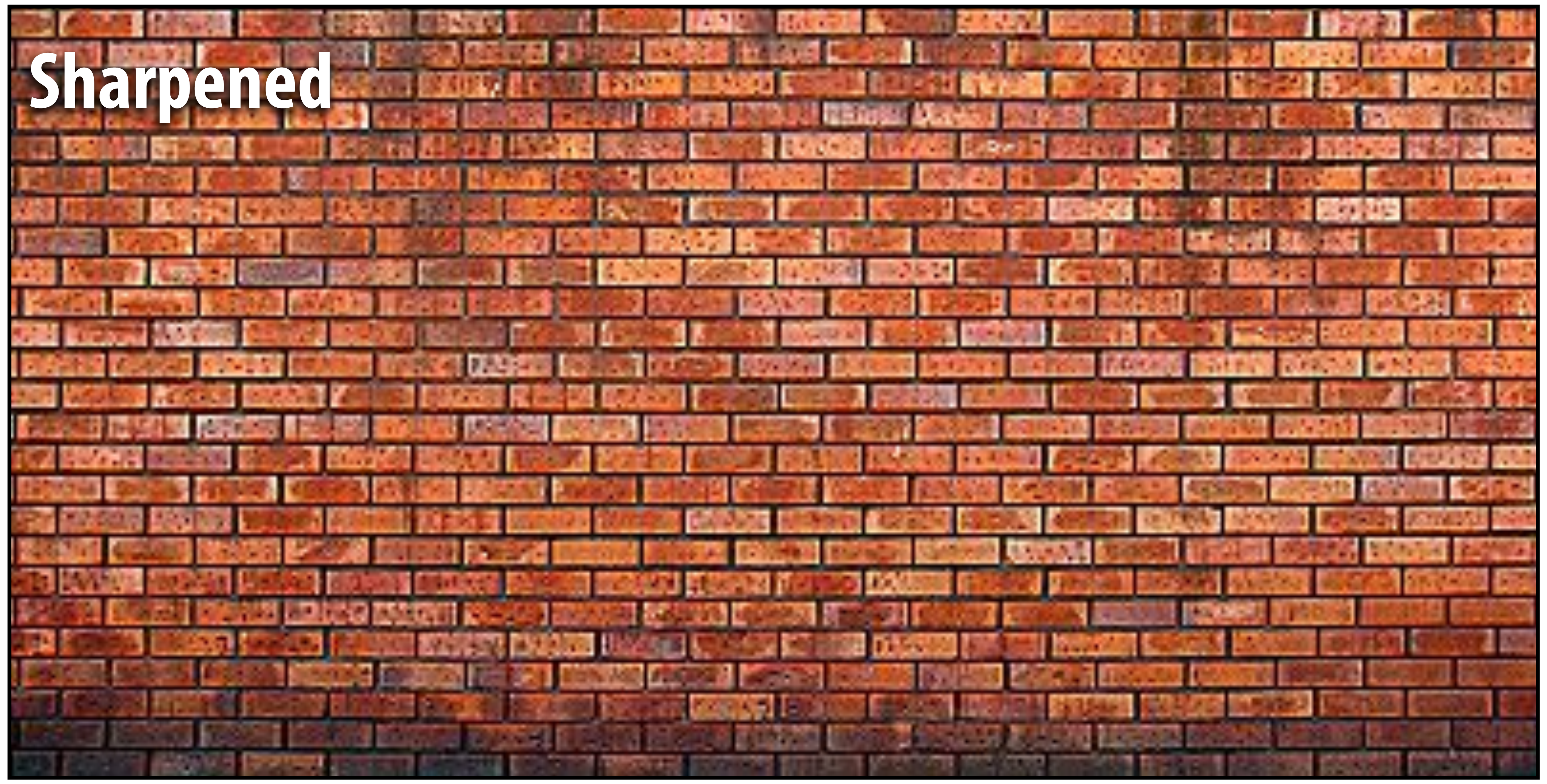


3x3 sharpen filter

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



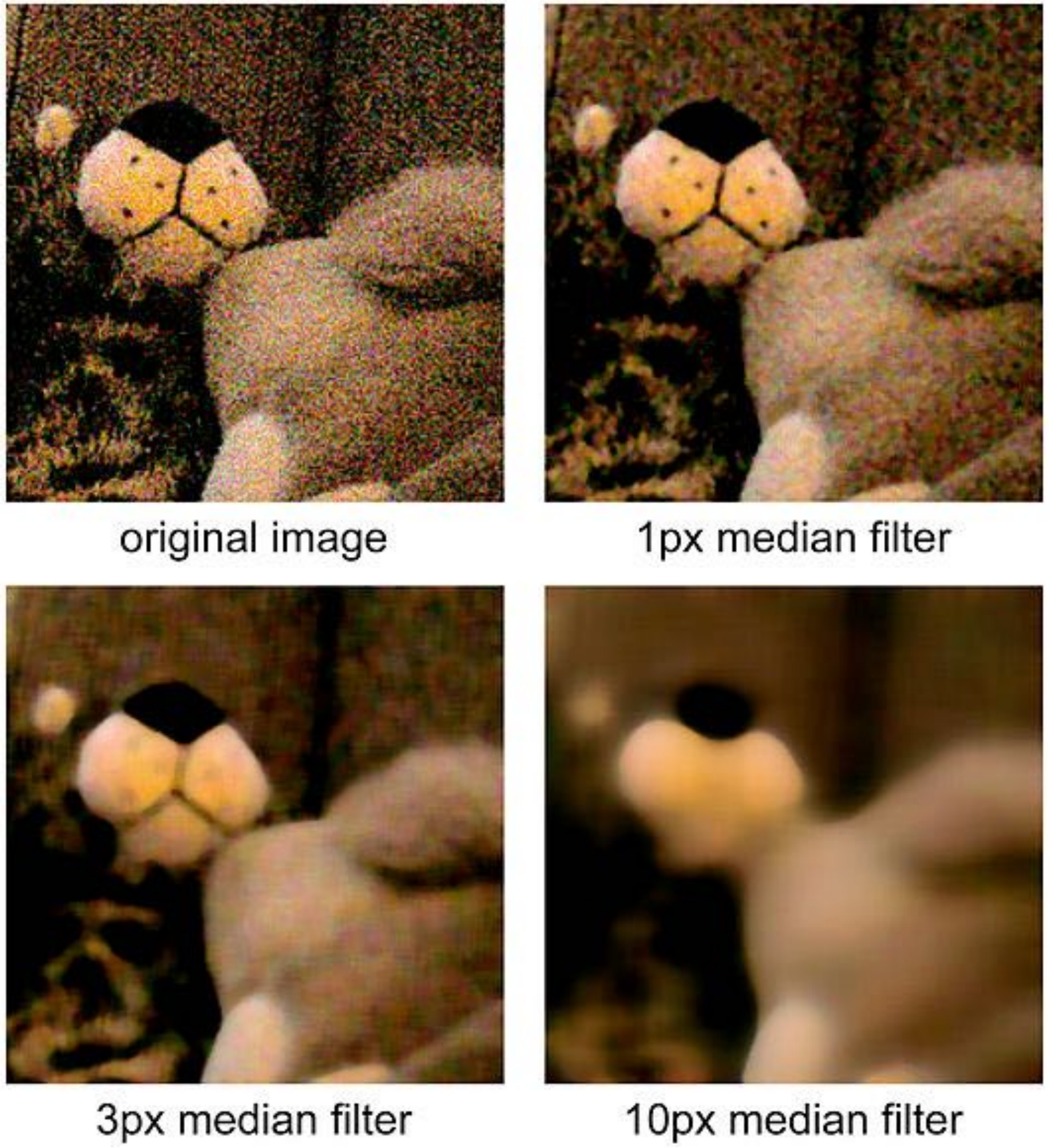




Median filter

- 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: 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
   }
}
```



- 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

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

Gaussian blur kernel: $G_\sigma(i,j)$

Input image

Normalization: $\frac{1}{W_p}$

For all pixels in support region of Gaussian kernel: $\sum_{i,j}$

Re-weight based on difference in input image pixel values

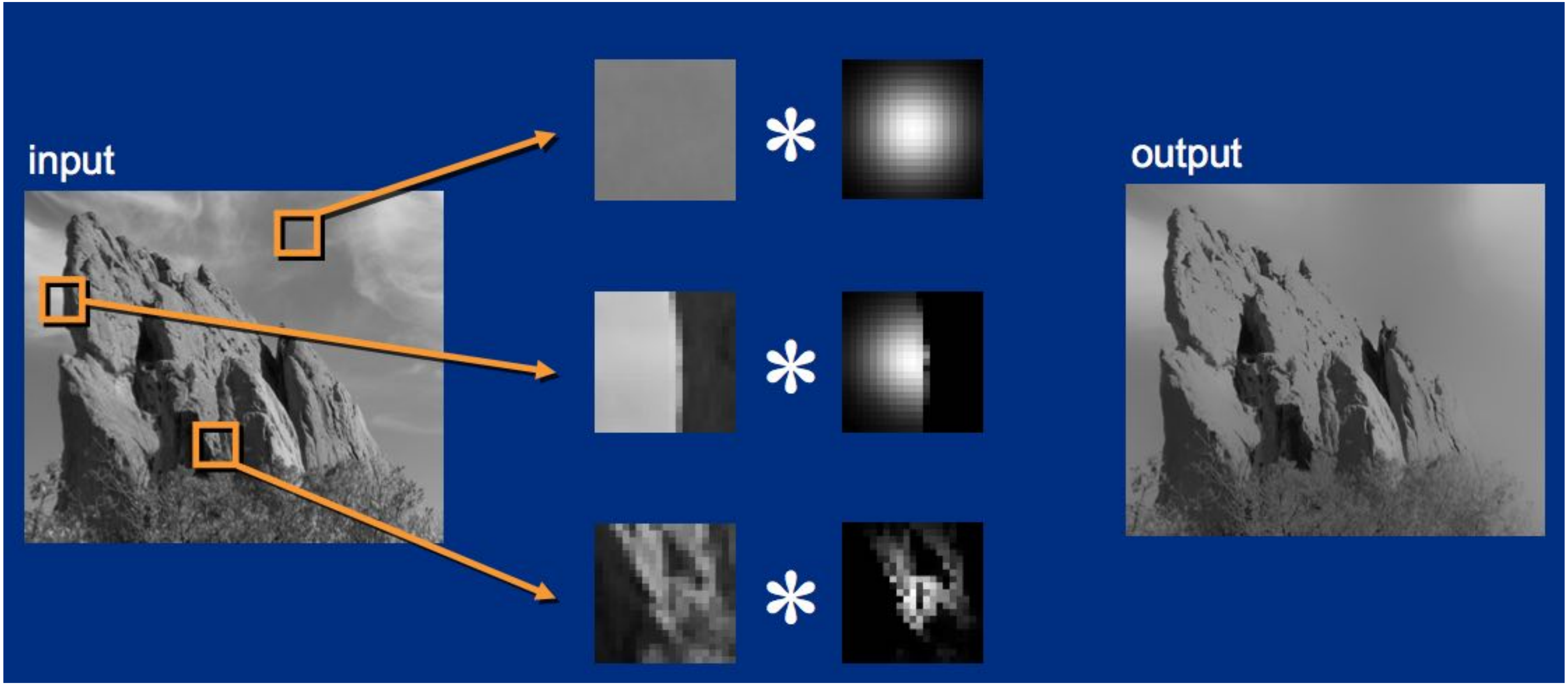
$$W_p = \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|)G_\sigma(i,j)$$

- **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 $\mathit{otherwise}$

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 spatial distance and input image pixel intensity difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

Bilateral filter: kernel depends on image content

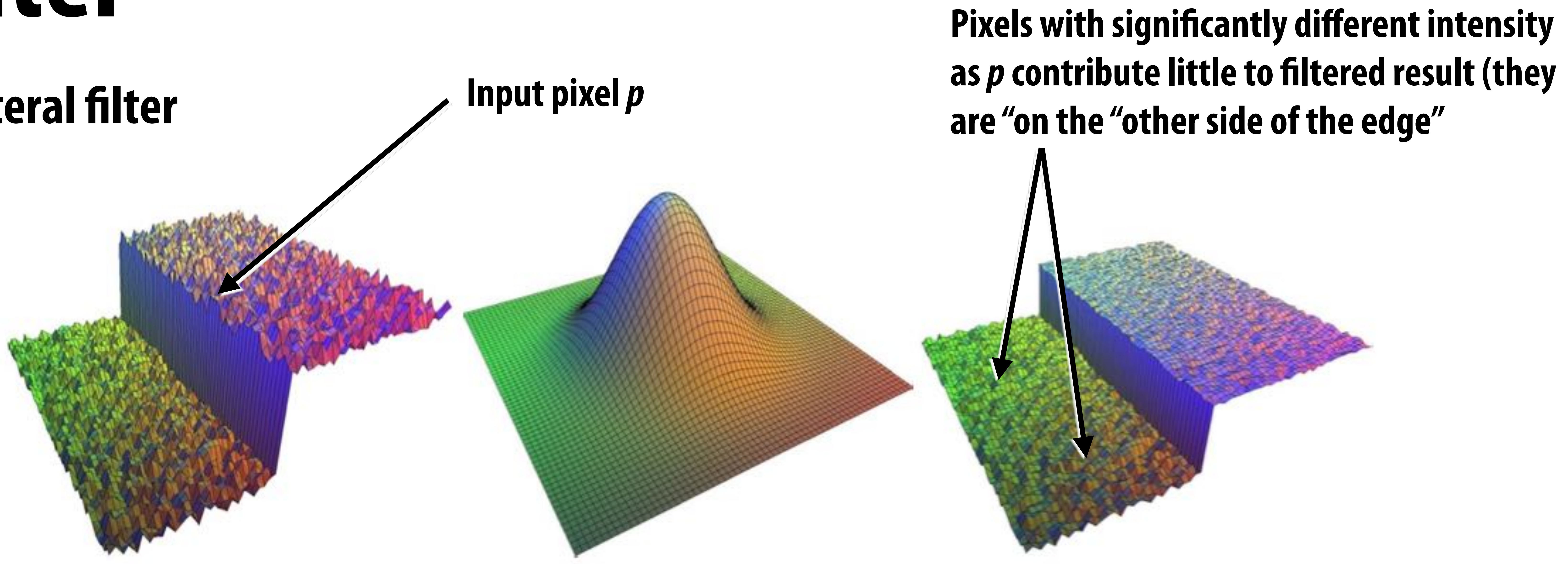


See Paris et al. [ECCV 2006] for a fast approximation to the bilateral filter

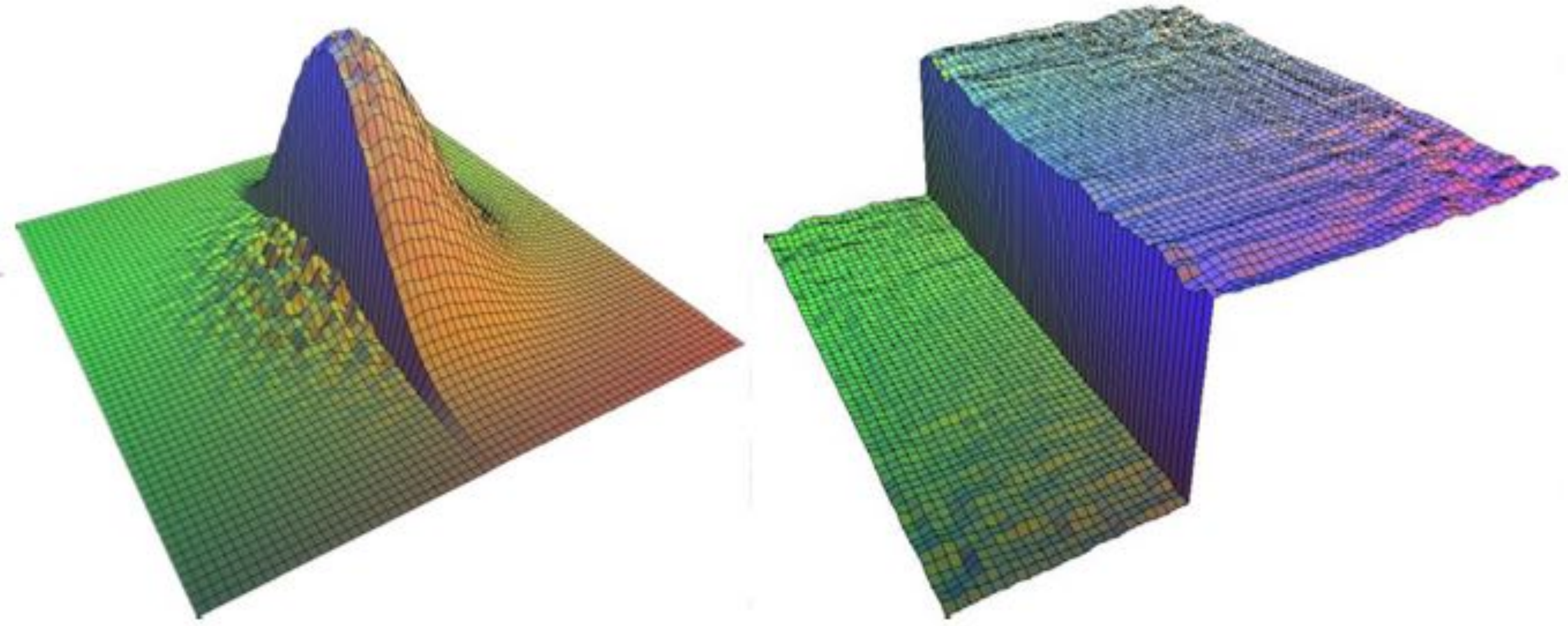
Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

Bilateral filter

- **Visualization of bilateral filter**



Input image | **G(): gaussian about input pixel *p*** | **f(): Influence of support region**



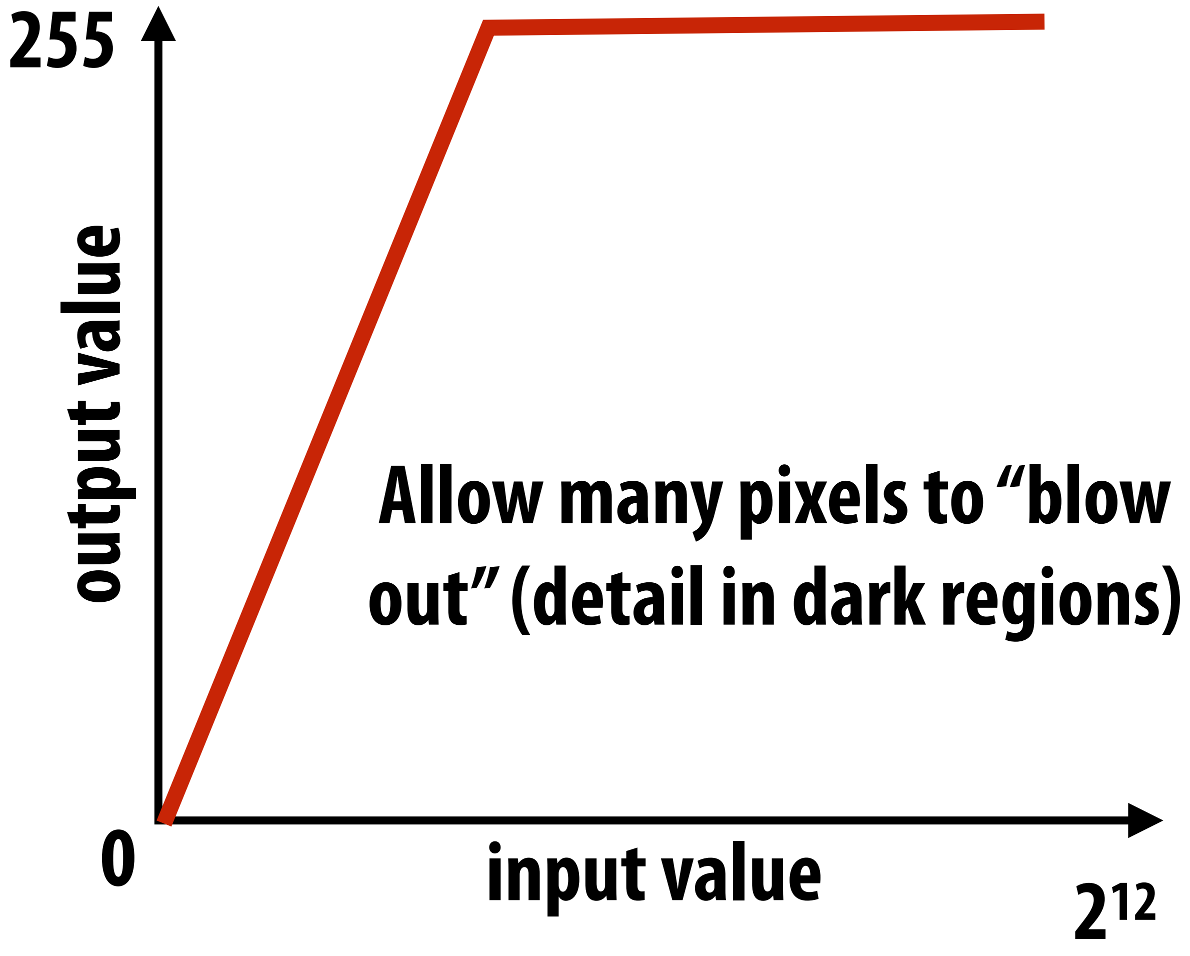
G x f: filter weights for pixel *p* | **Filtered output image**

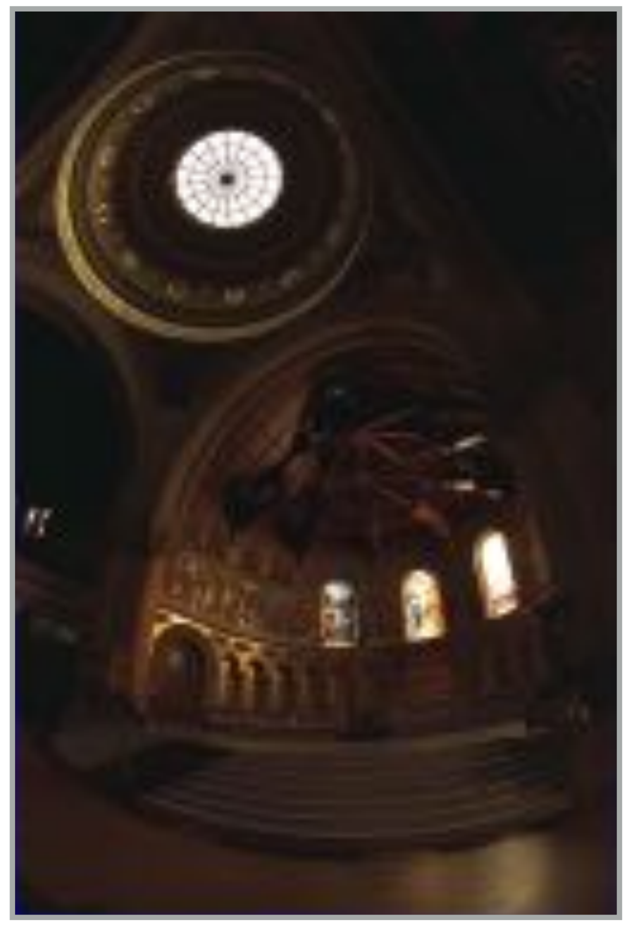
Figure credit: Durand and Dorsey, “Fast Bilateral Filtering for the Display of High-Dynamic-Range Images”, SIGGRAPH 2002

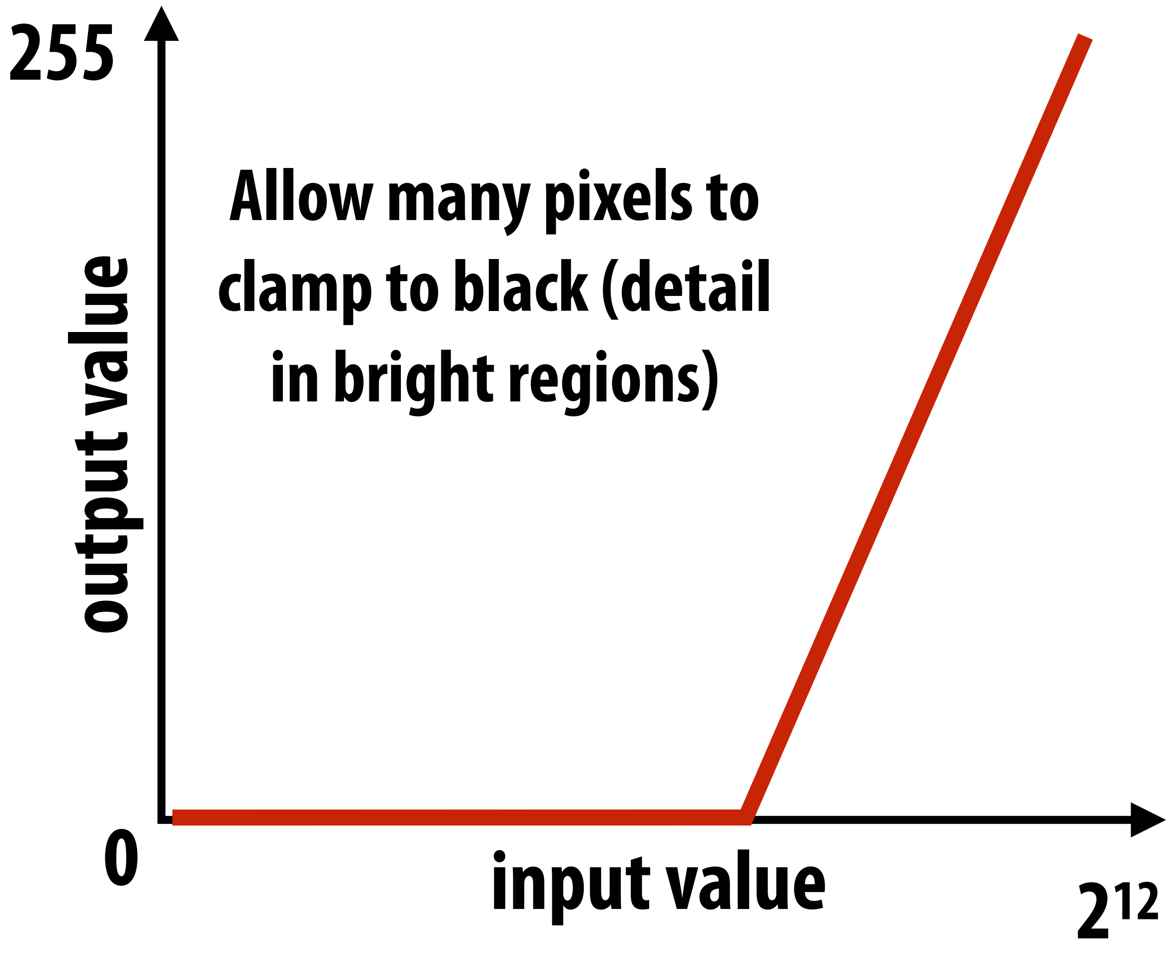
Auto Exposure and Tone Mapping

Global tone mapping

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

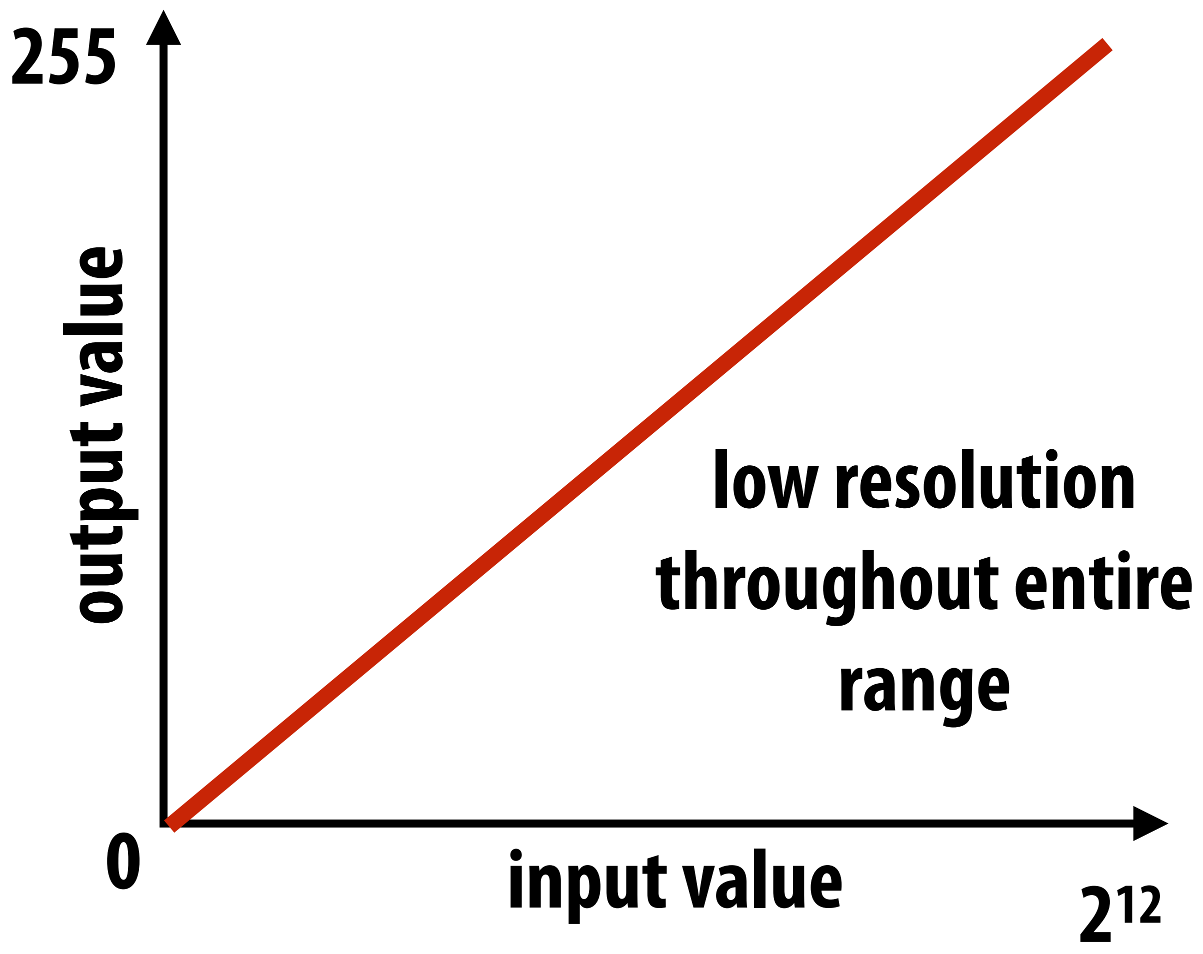


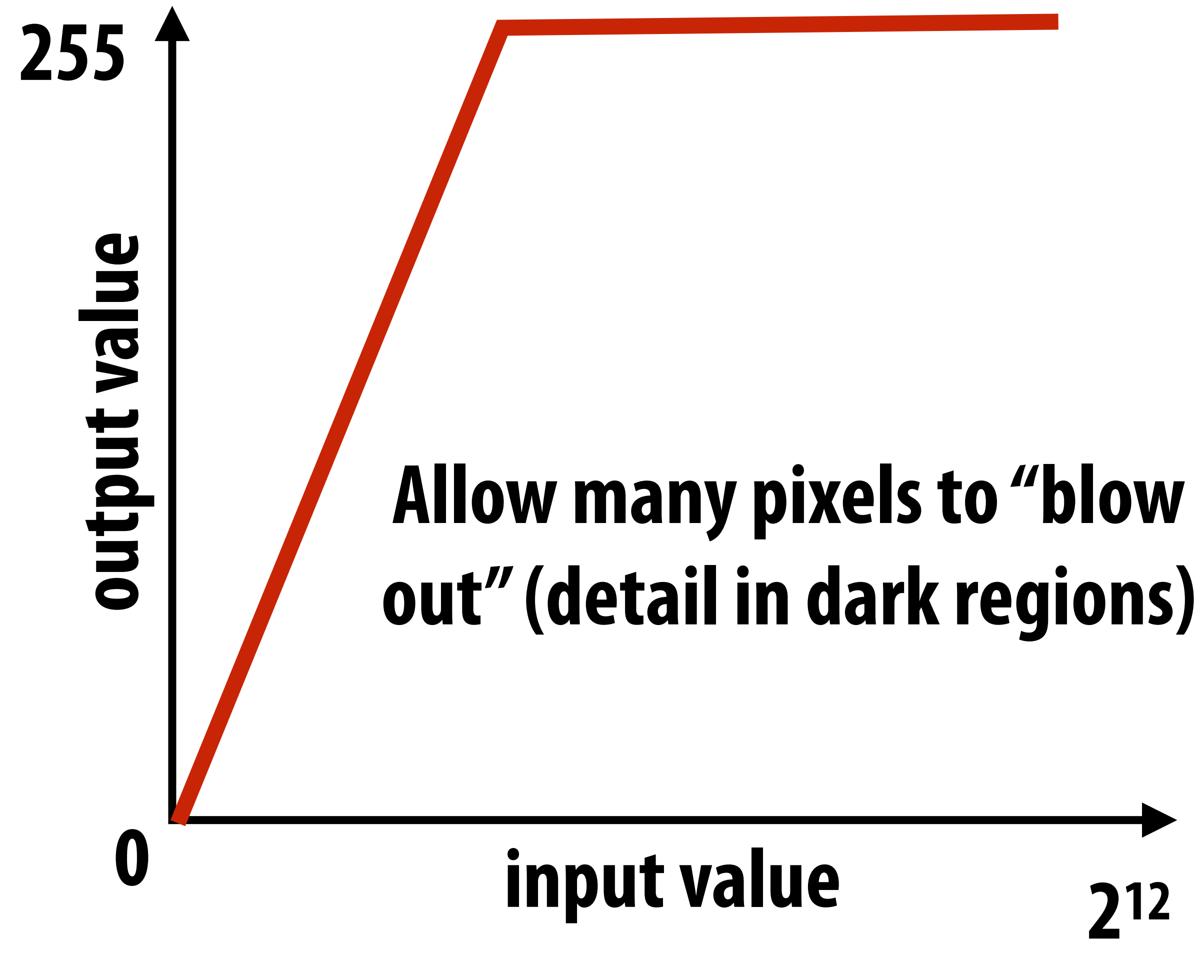


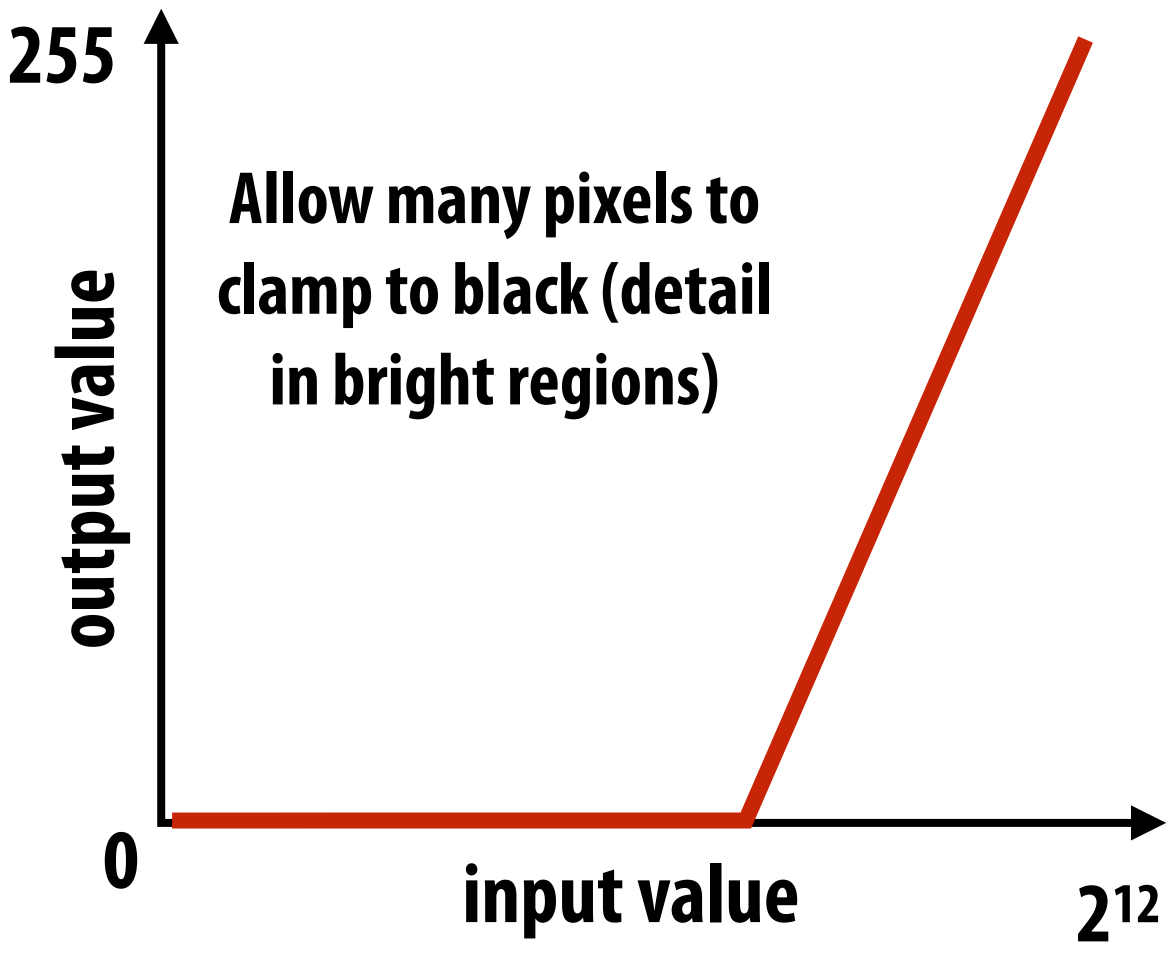


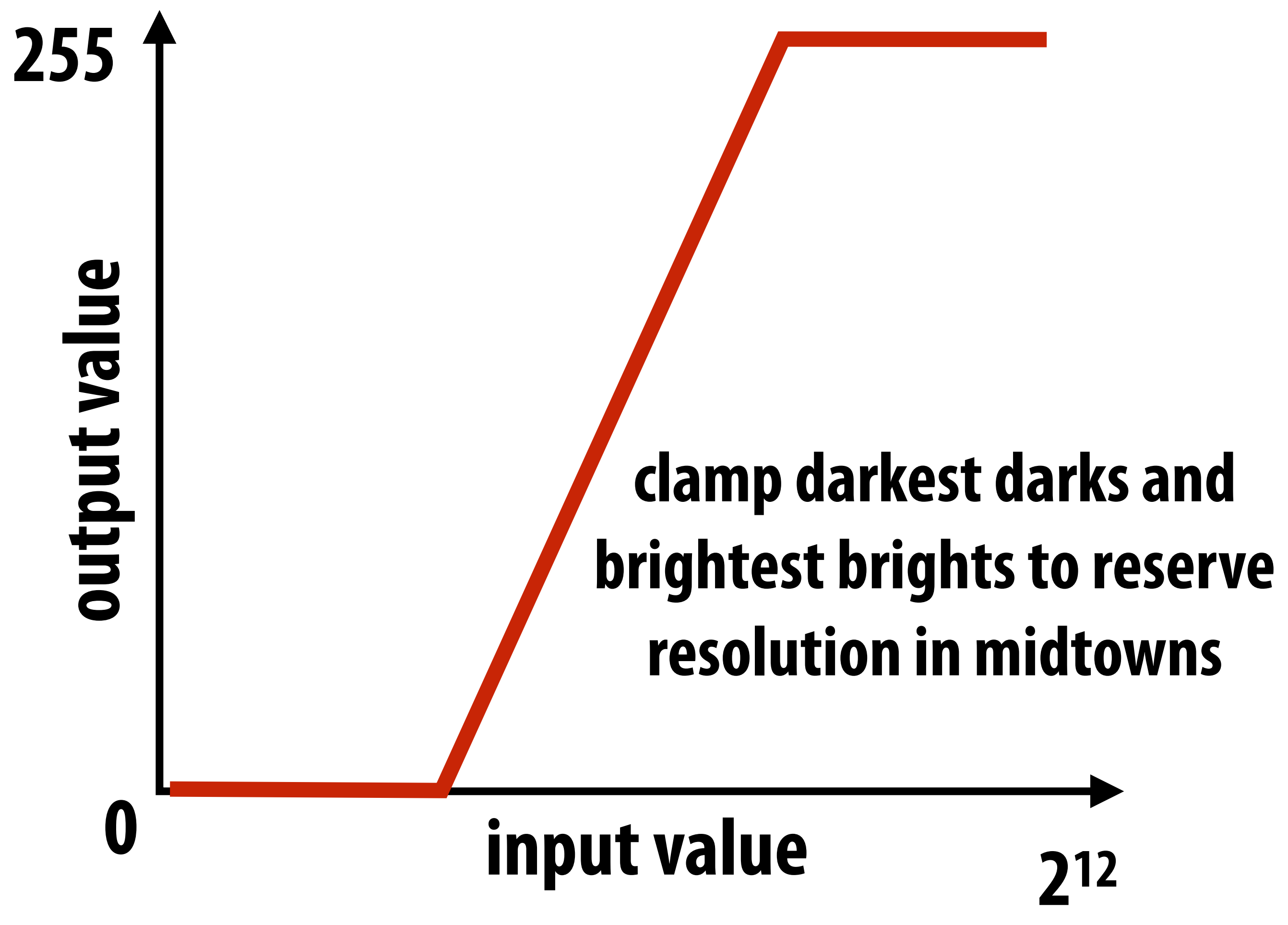
Global tone mapping

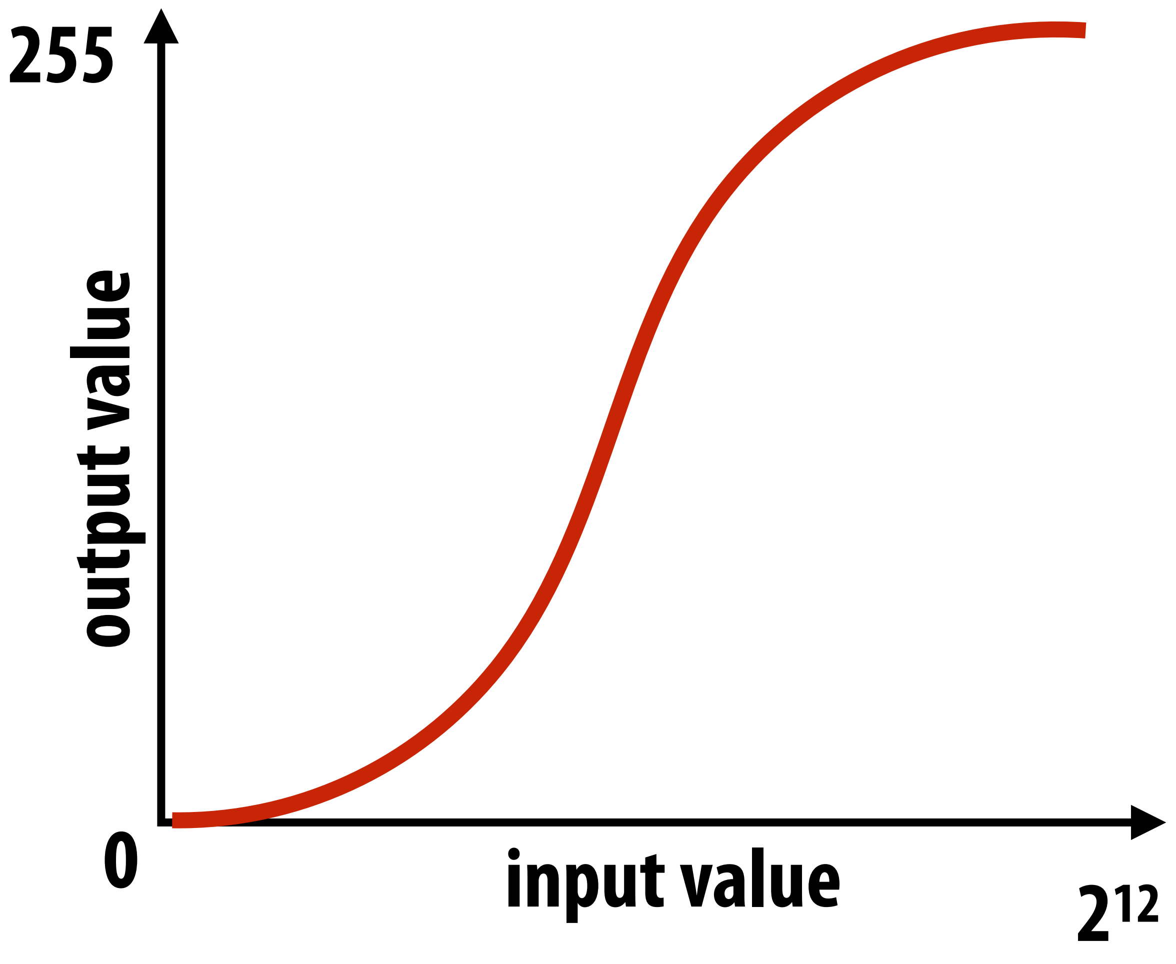
$$out(x,y) = f(in(x,y))$$







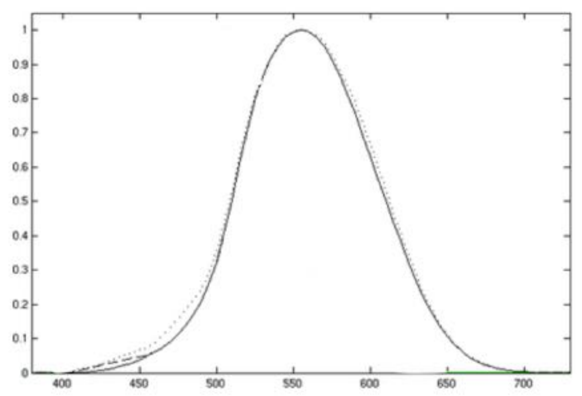




Lightness (<u>perceived</u> brightness) aka luma

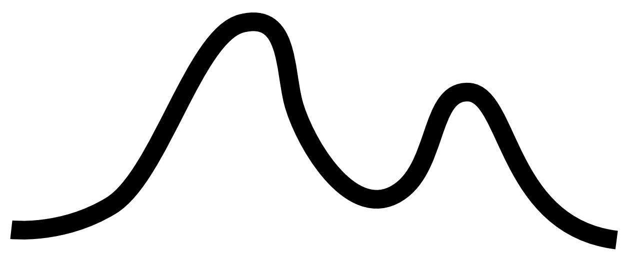
Lightness (L*) $\xleftarrow{?}$ Luminance (Y) $= \int_\lambda$

(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: $L^* \propto Y^{0.5}$

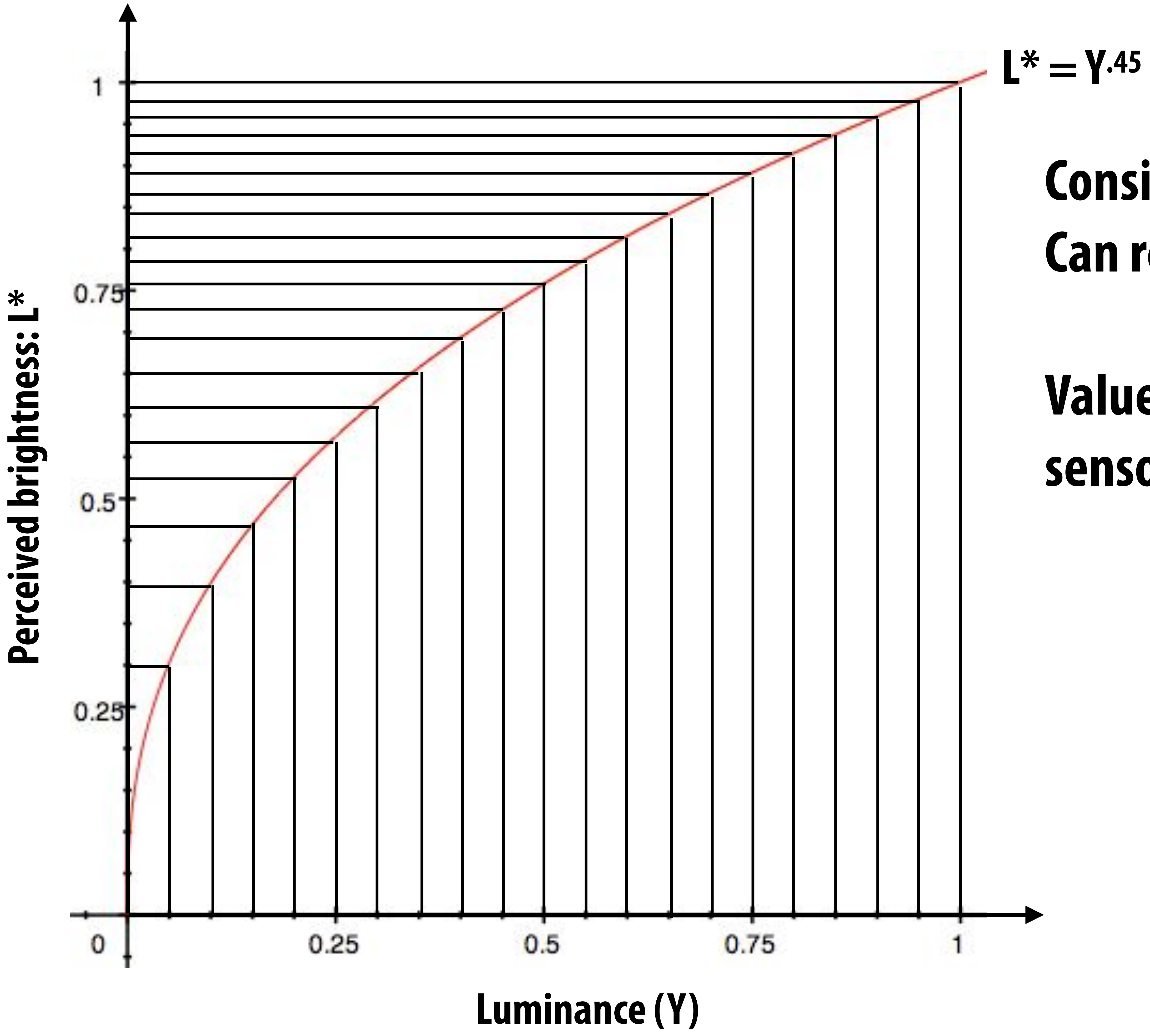
In a dark room, you turn on a light with luminance: Y_1

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)



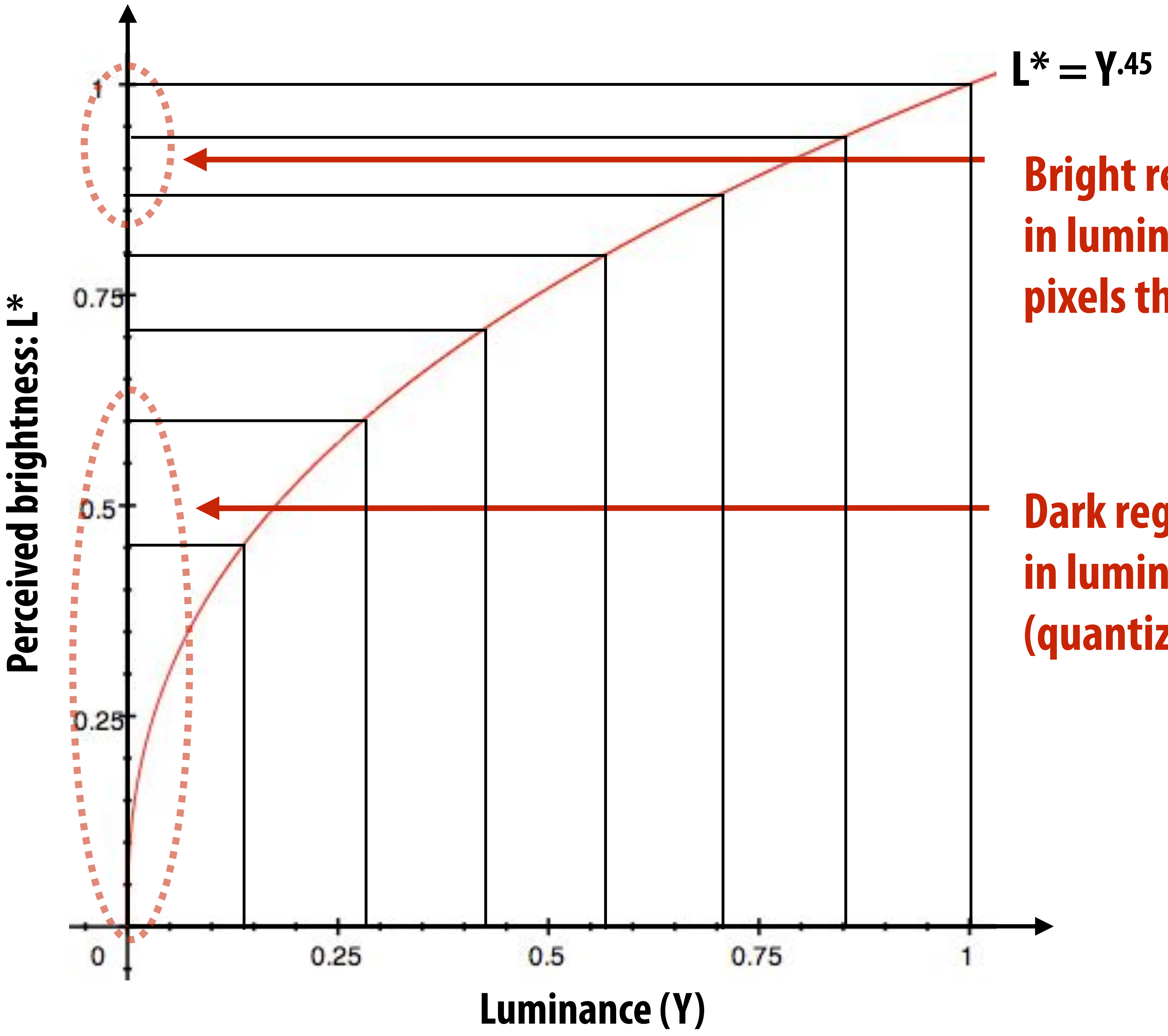
Consider 12-bit sensor pixel:
Can represent 4096 unique luminance values in output image

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



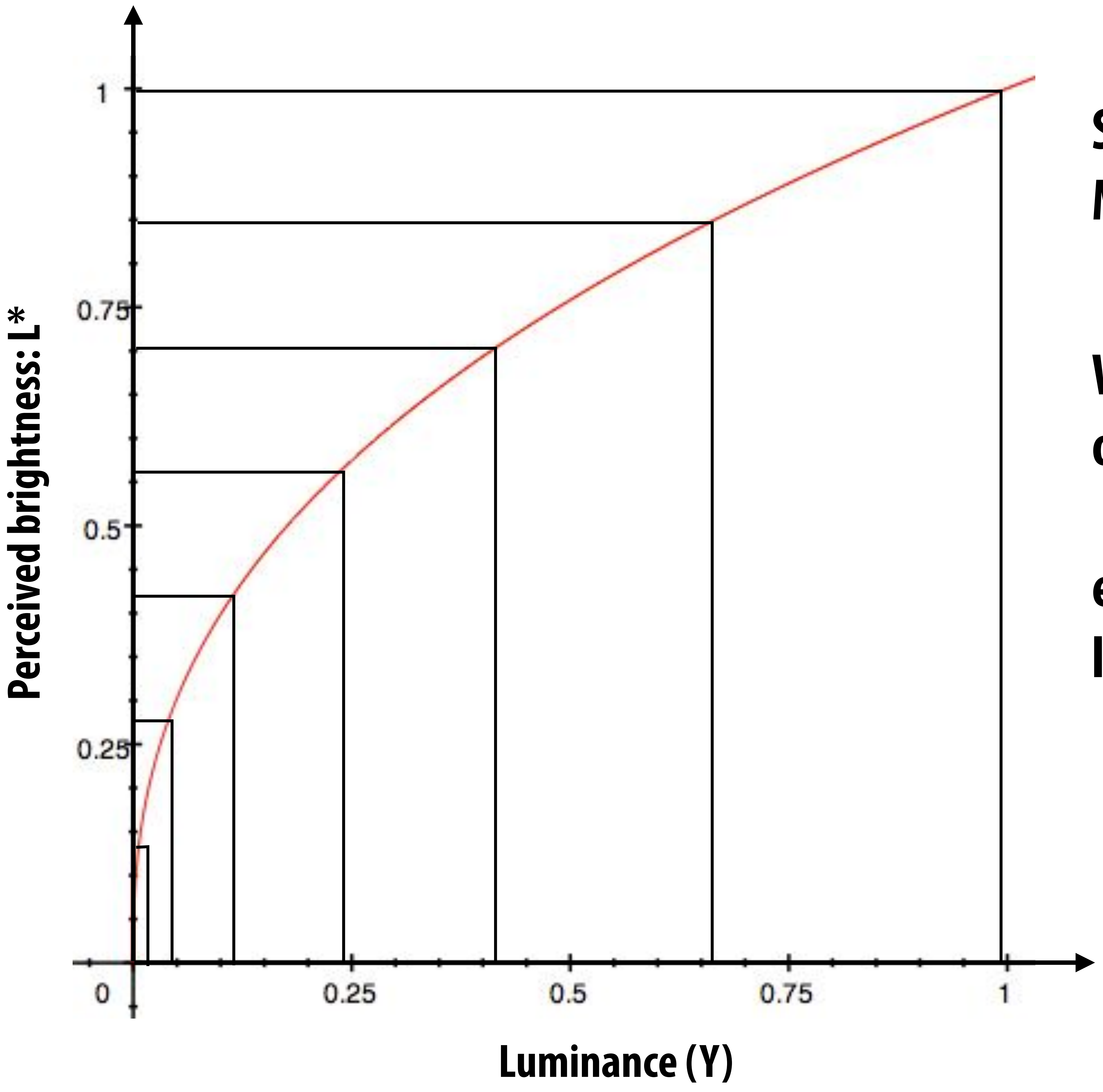
Bright regions of image: perceived difference between pixels that differ by one step in luminance is small! (human may not even be able to perceive difference between pixels that differ by one step in luminance!)

Dark regions of image: perceived difference between pixels that differ by one step in luminance is large!
(quantization error: gradients in luminance will not appear smooth.)

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

Store lightness in 8-bit value, 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.

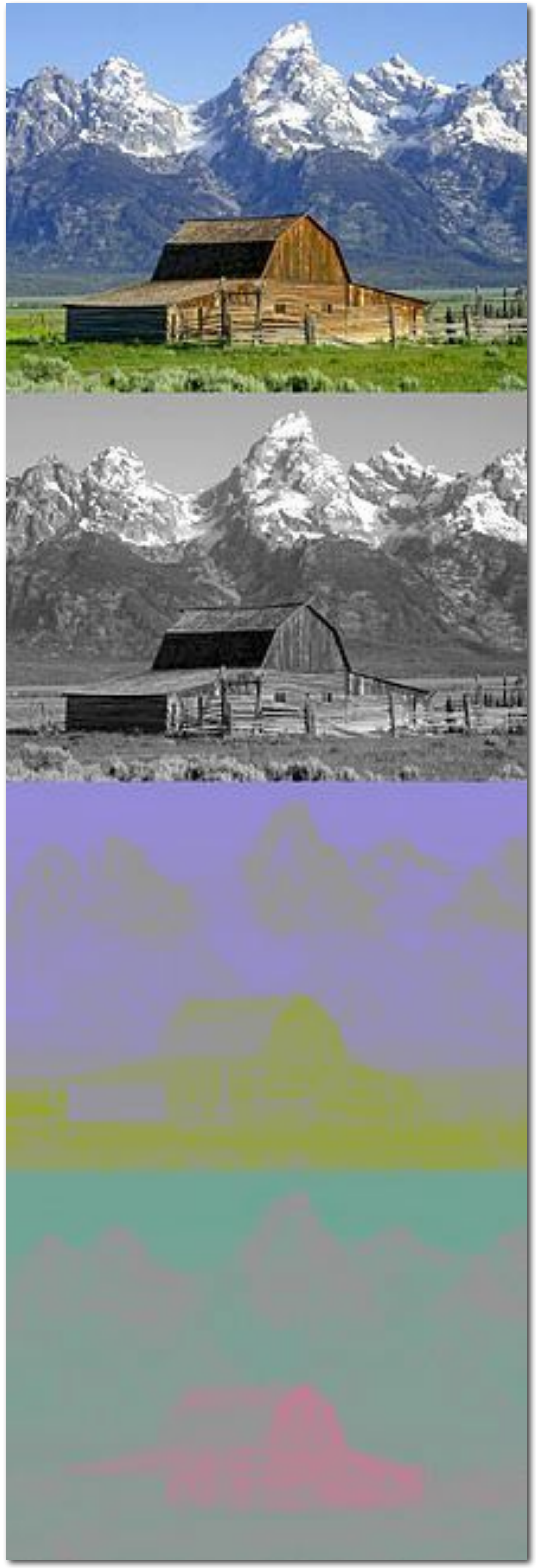
e.g., When adding images should you add pixel values that are encoded as lightness or as luminance?

Y'CbCr color space

Recall: colors are represented as point in 3-space

RGB is just one possible basis for representing color

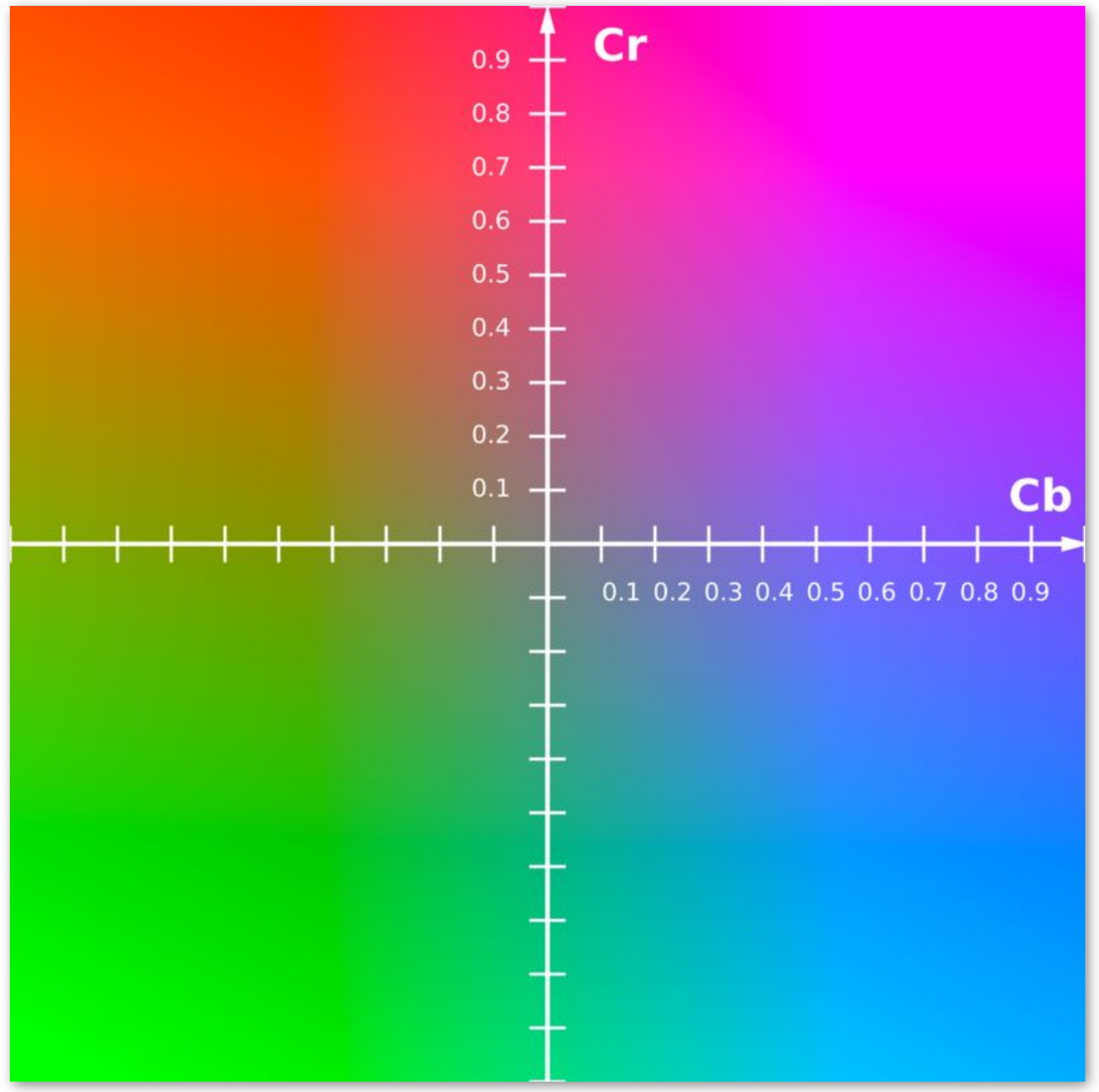
Y'CbCr separates luminance from hue in representation



Y'

Cb

Cr



Y' = luma: perceived luminance

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray

"Gamma corrected" RGB (primed notation indicates perceptual (non-linear) space)

We'll describe what this means this later in the lecture.

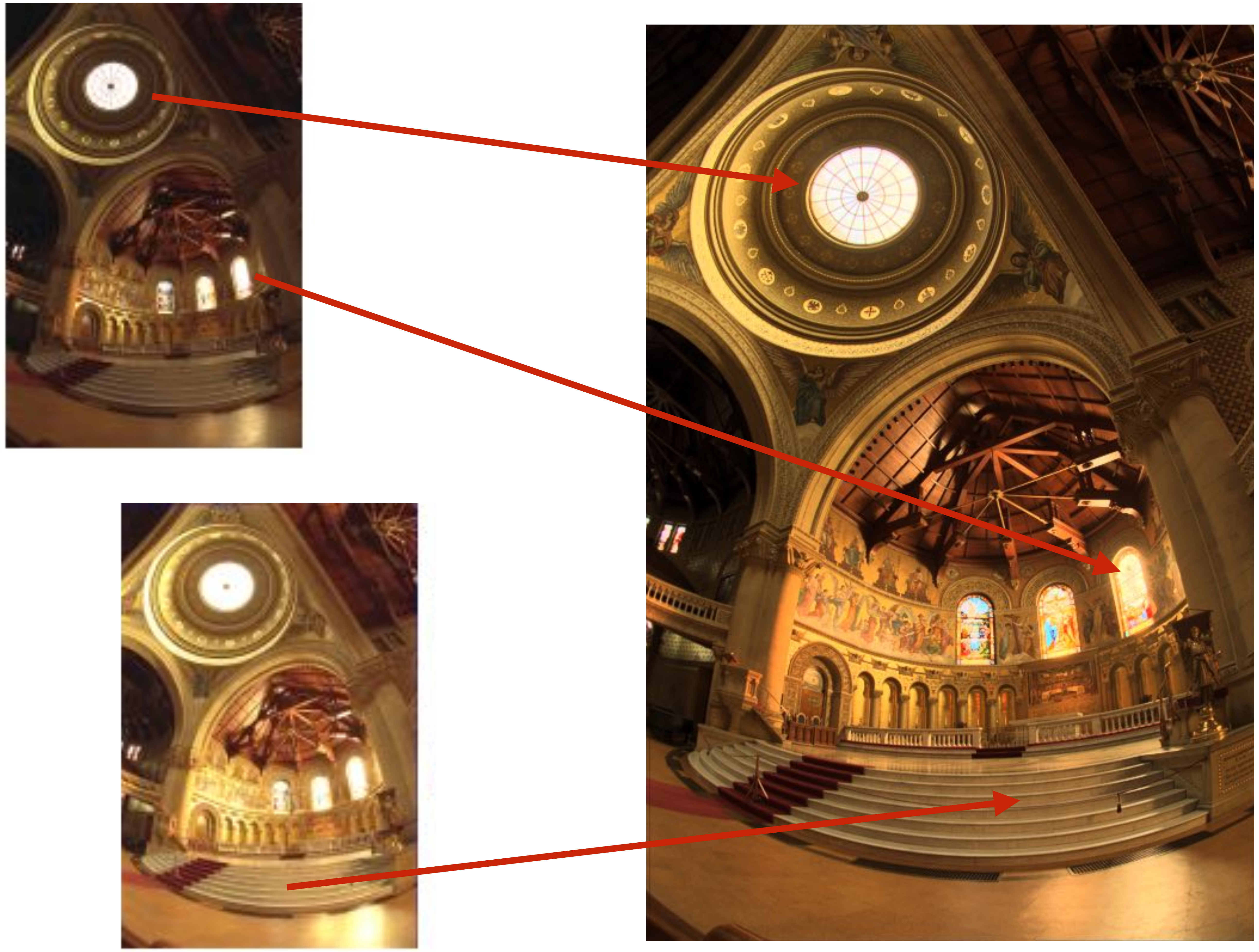
Conversion matrix from R'G'B' to Y'CbCr:

$$
\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}
\end{aligned}
$$

Image credit: Wikipedia

Local tone mapping

- Different regions of the image undergo different tone mapping curves (preserve detail in both dark and bright regions)



Local tone adjustment

Improve picture's aesthetics by locally adjusting contrast, boosting dark regions, decreasing bright regions (no physical basis)

Local tone adjustment

Pixel values

Weights

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

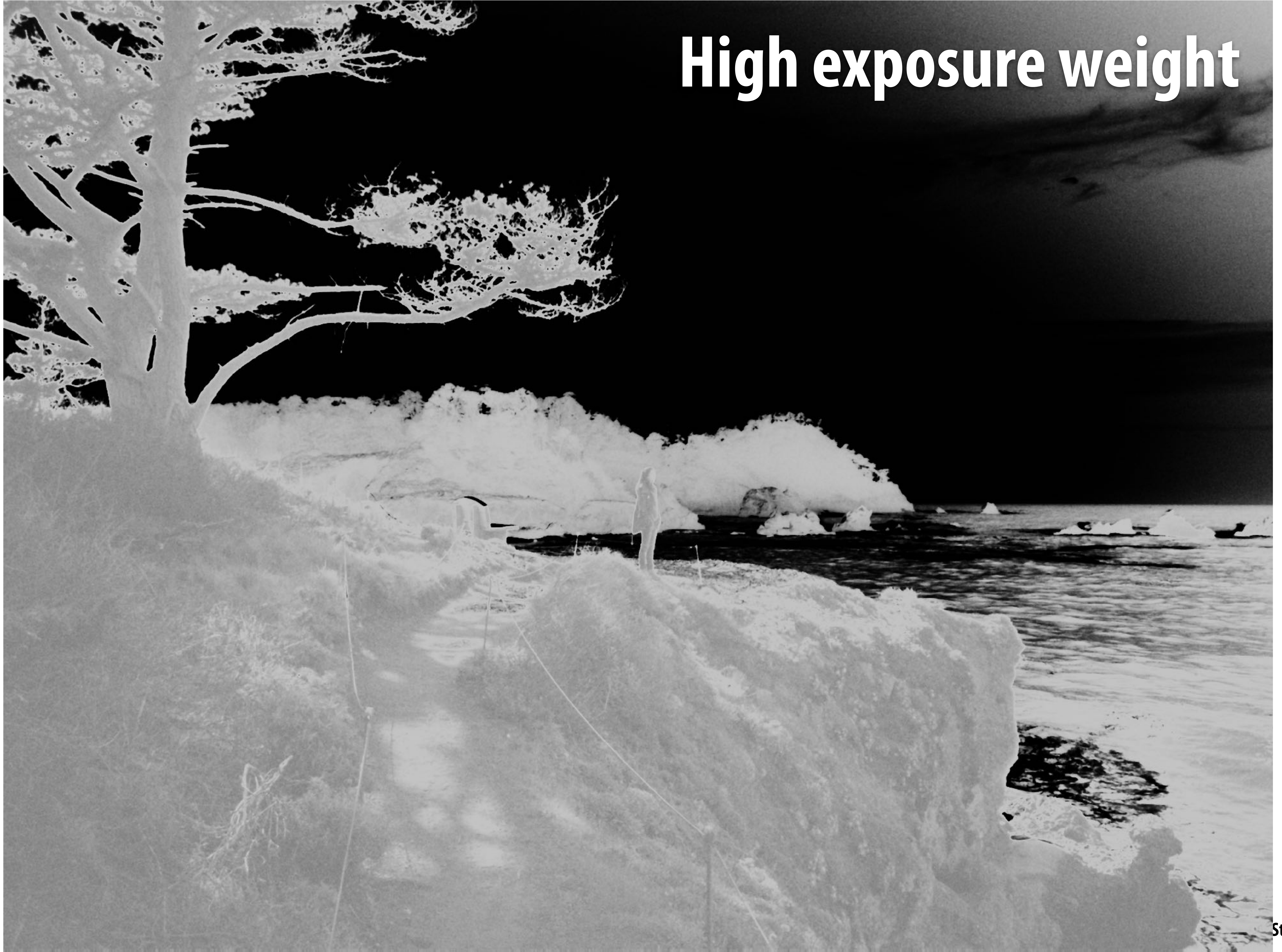
(no physical basis)

Combined image
(unique weights per pixel)

Image credit: Mertens 2007

High exposure image

High exposure weight




Low exposure image

Low exposure weight

Combined result

Combined result

Local tone mapping was performed on lightness (luma).
Now I added back in chrominance channels.

Challenge of merging images

Four exposures (weights not shown)

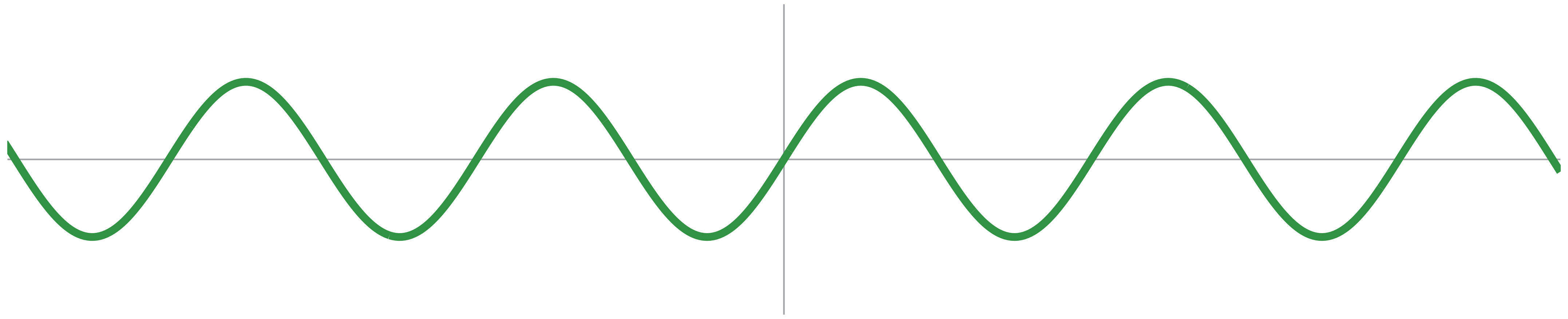
Merged result (based on weight masks)
Notice heavy "banding" since absolute intensity
of different exposures is different

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

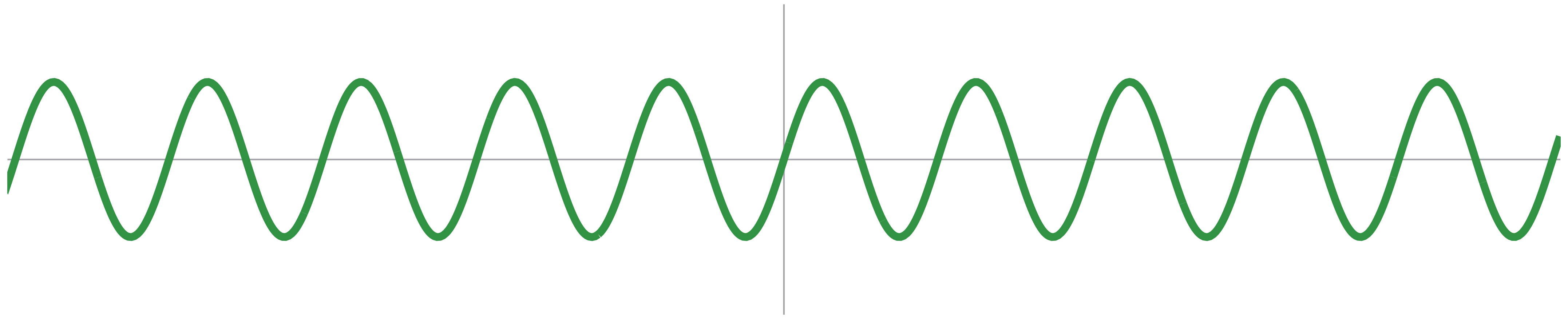
Review:
Frequency interpretation of images

Representing sound as a superposition of frequencies

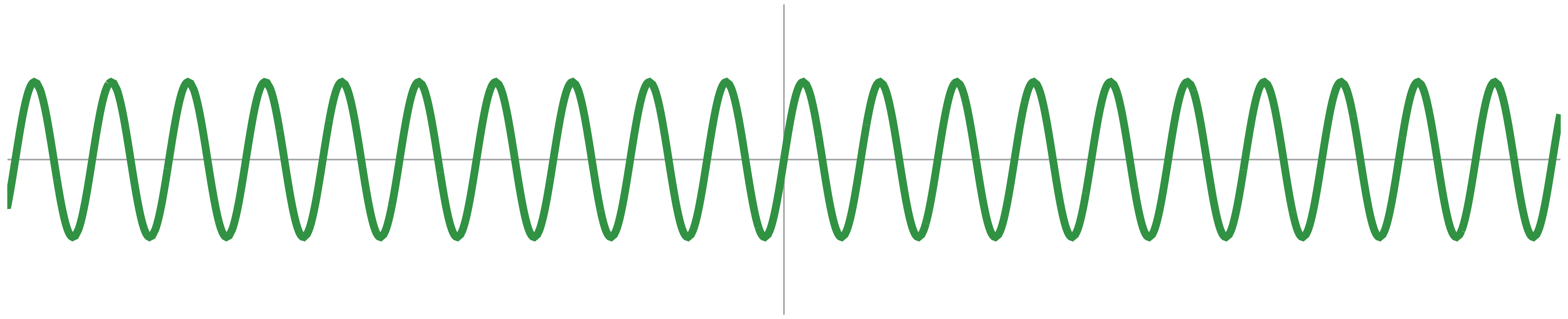
$f_1(x) = sin(\pi x)$



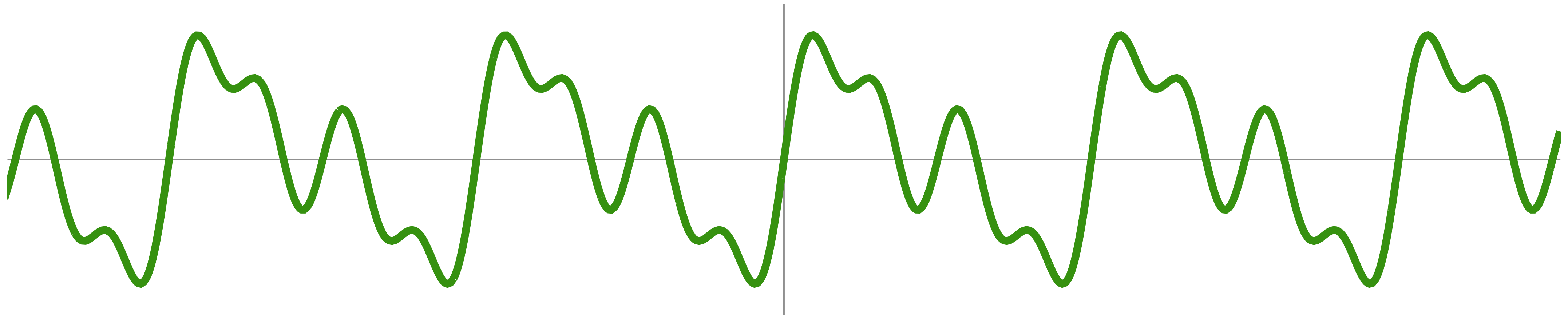
$f_2(x) = sin(2\pi x)$



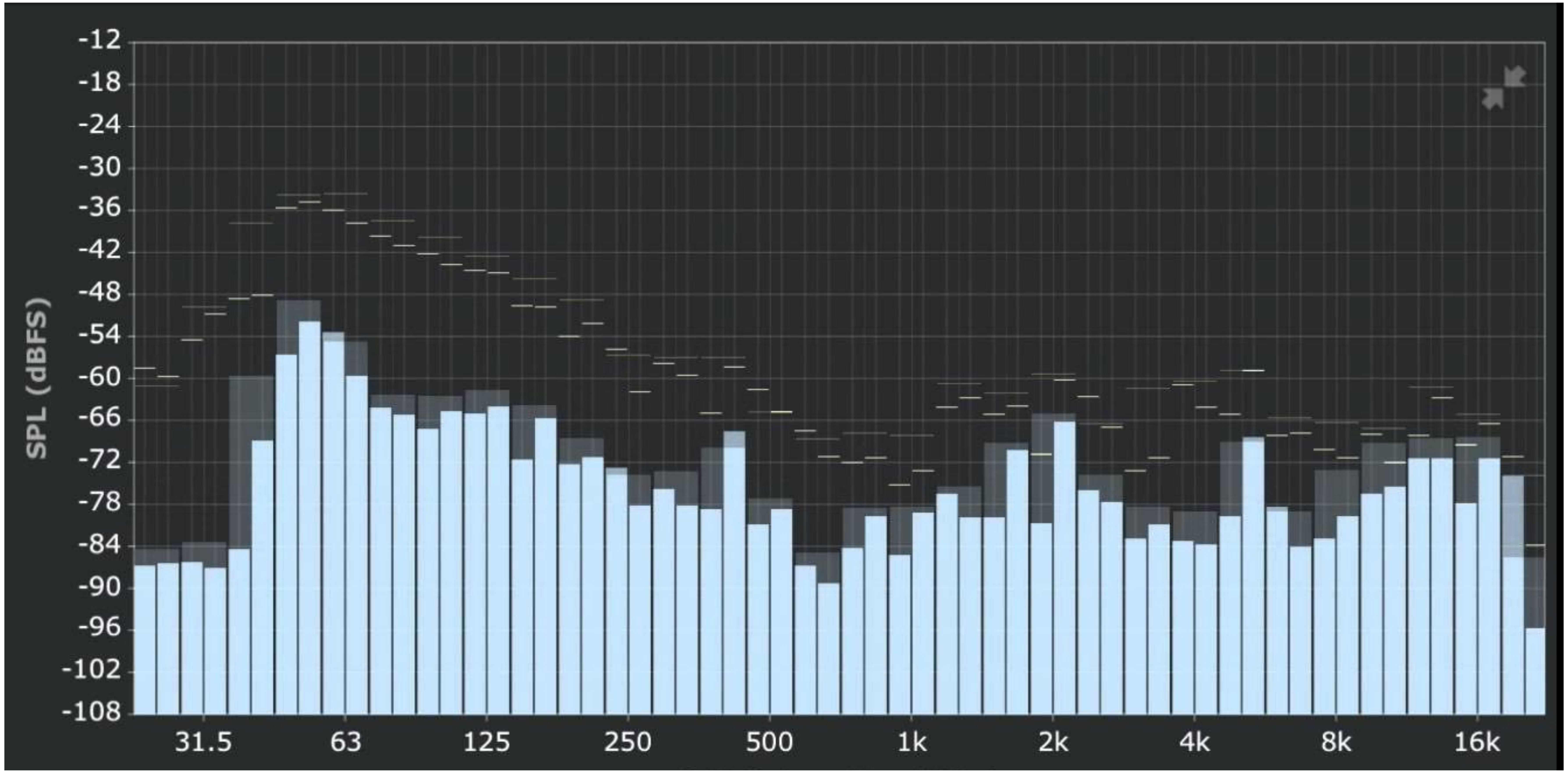
$f_4(x) = sin(4\pi x)$



$f(x) = f_1(x) + 0.75\, f_2(x) + 0.5\, f_4(x)$



Audio spectrum analyzer: representing sound as a sum of its constituent frequencies



Intensity of low-frequencies (bass)

Intensity of high frequencies

Image credit: ONYX Apps

Fourier transform

- **Convert representation of signal from spatial/temporal domain to frequency domain by projecting signal into its component frequencies**

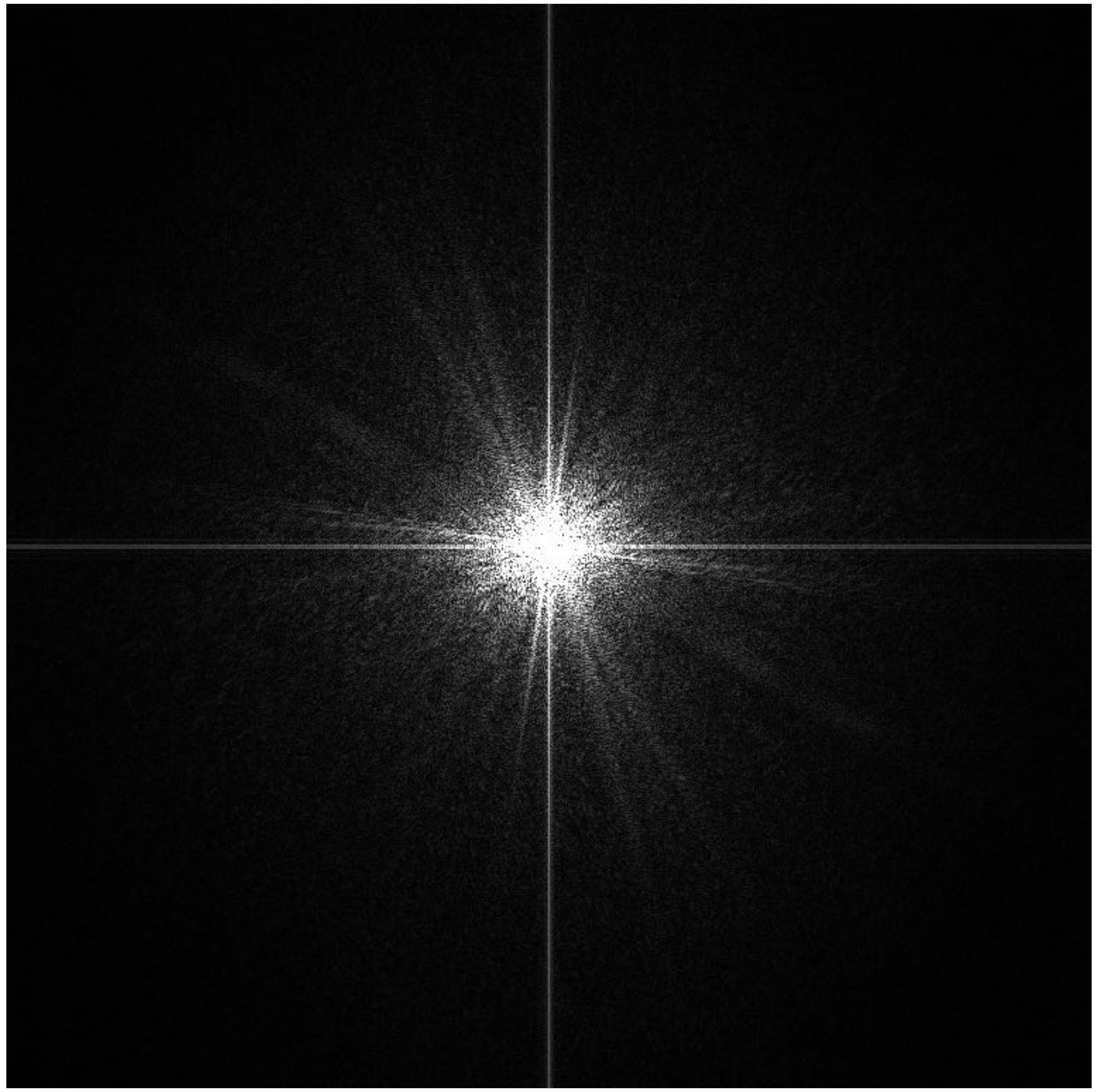
$$
\begin{aligned}
f(\xi) &= \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \\
&= \int_{-\infty}^{\infty} f(x)(\cos(2\pi\xi x) - i\sin(2\pi\xi x)) dx
\end{aligned}
$$

- **2D form:**

$$
f(u,v) = \int \int f(x,y) e^{-2\pi i(ux+vy)} dx dy
$$

Visualizing the frequency content of images

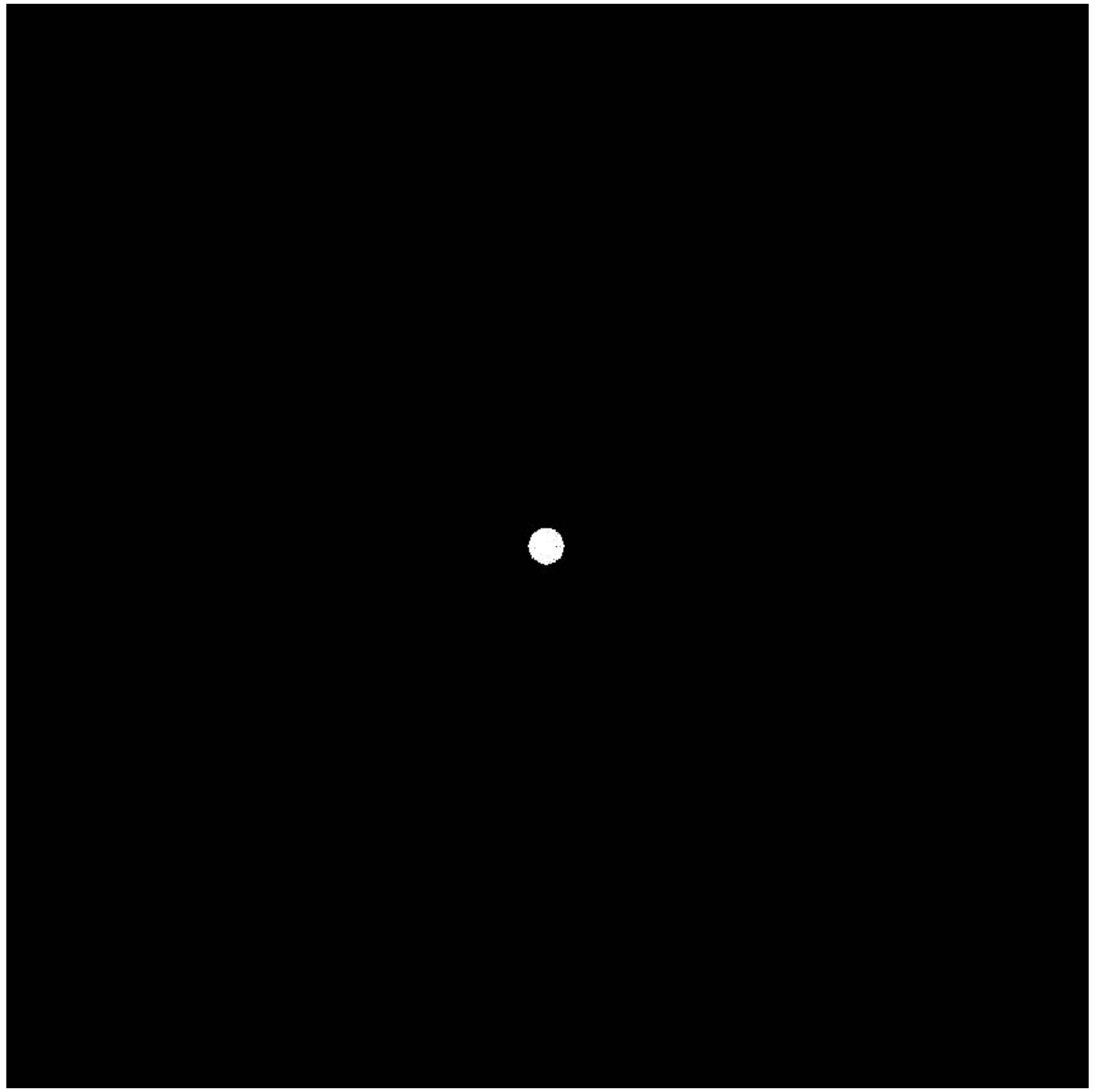
Spatial domain result



Spectrum

Low frequencies only (smooth gradients)

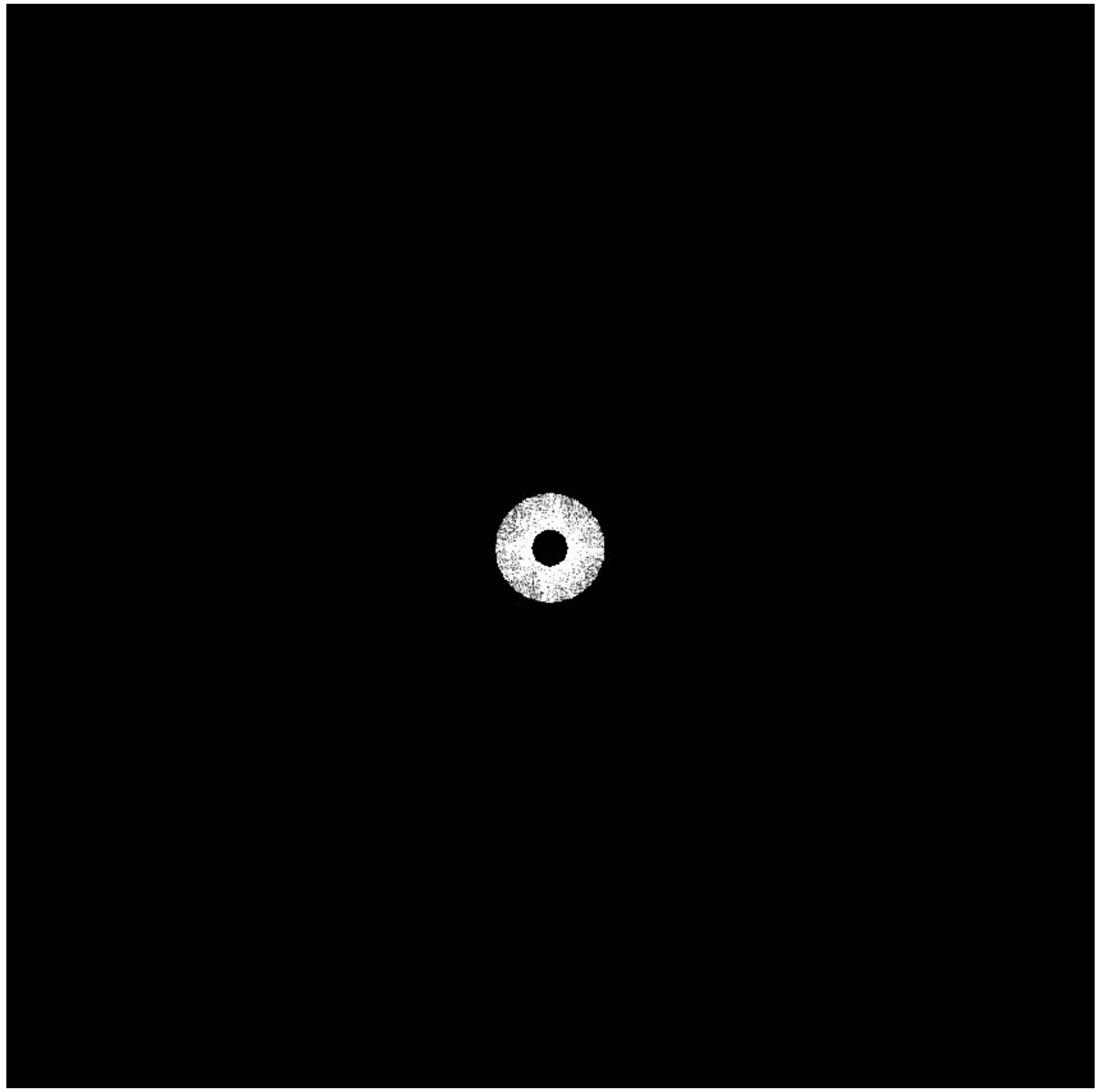
Spatial domain result



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

Mid-range frequencies

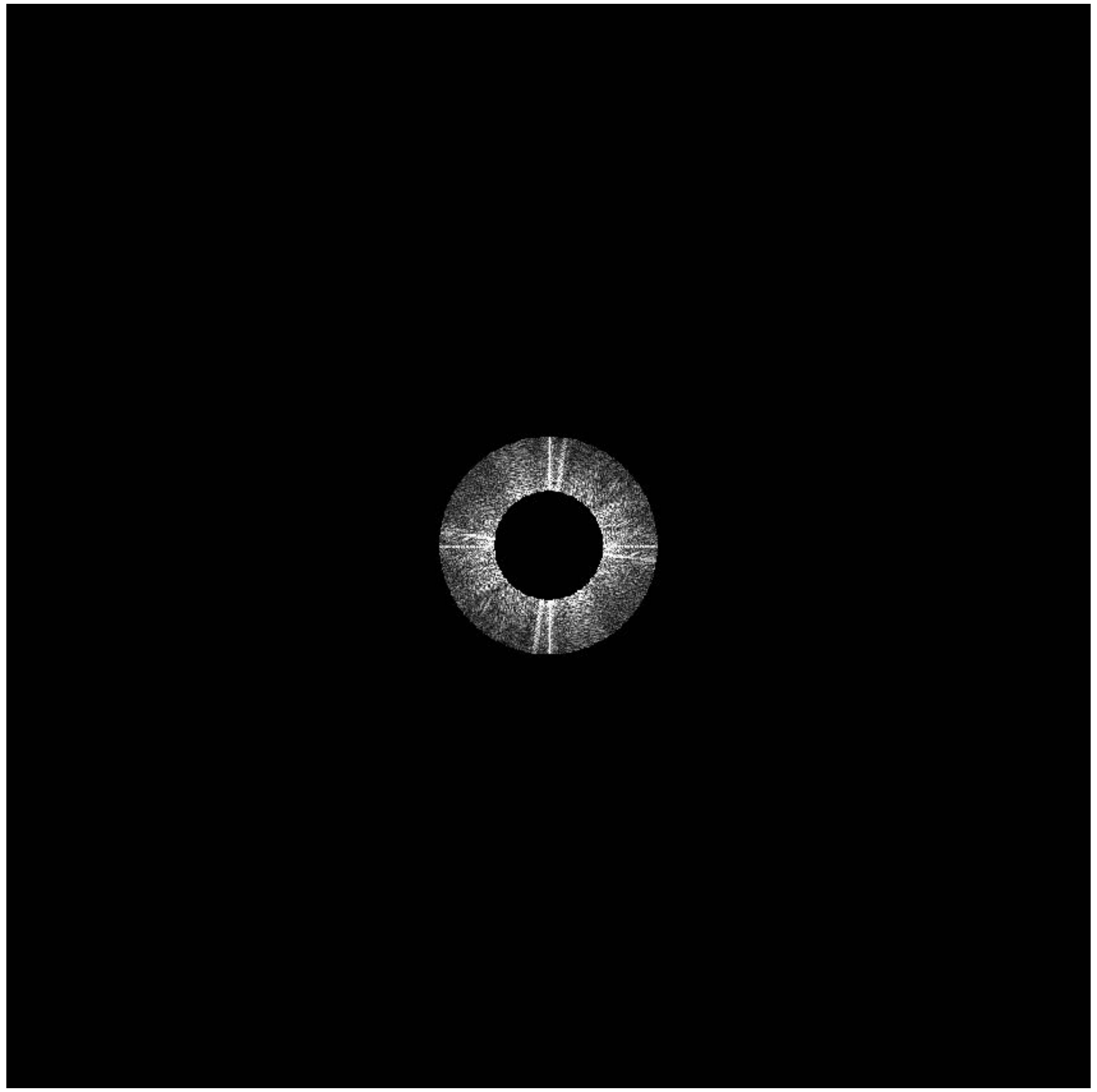
Spatial domain result



Spectrum (after band-pass filter)

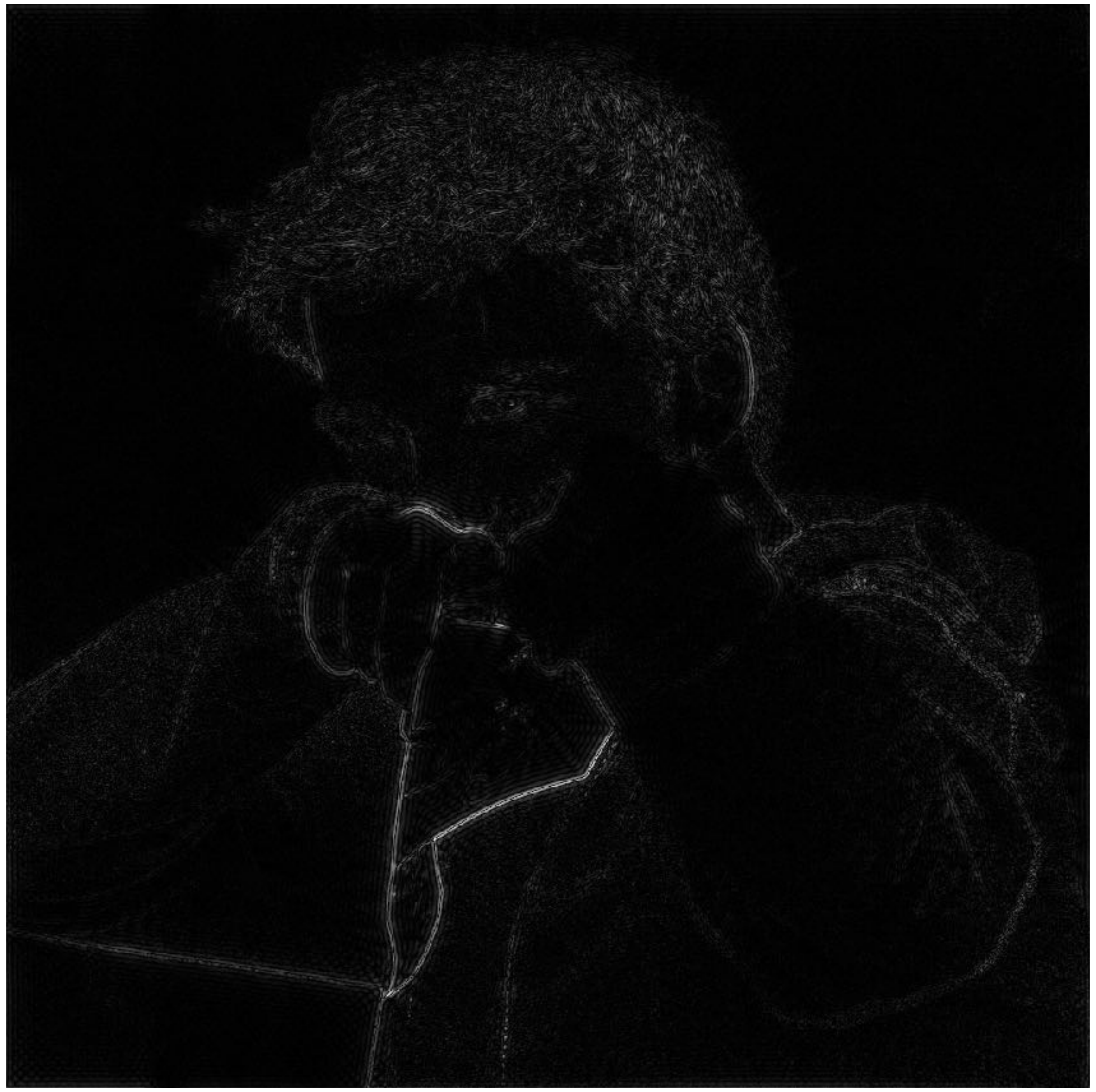
Mid-range frequencies

Spatial domain result

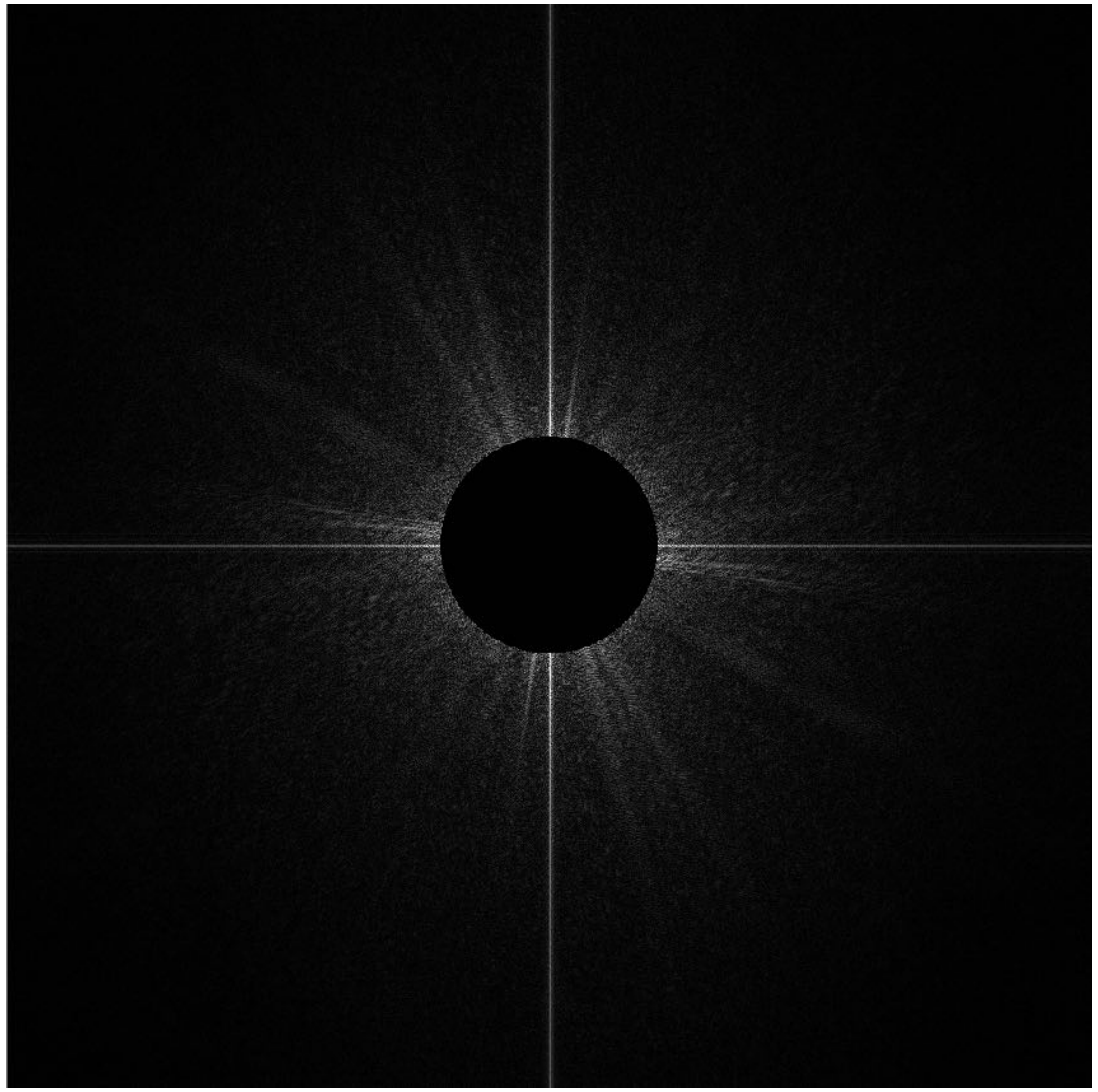


Spectrum (after band-pass filter)

High frequencies (edges)

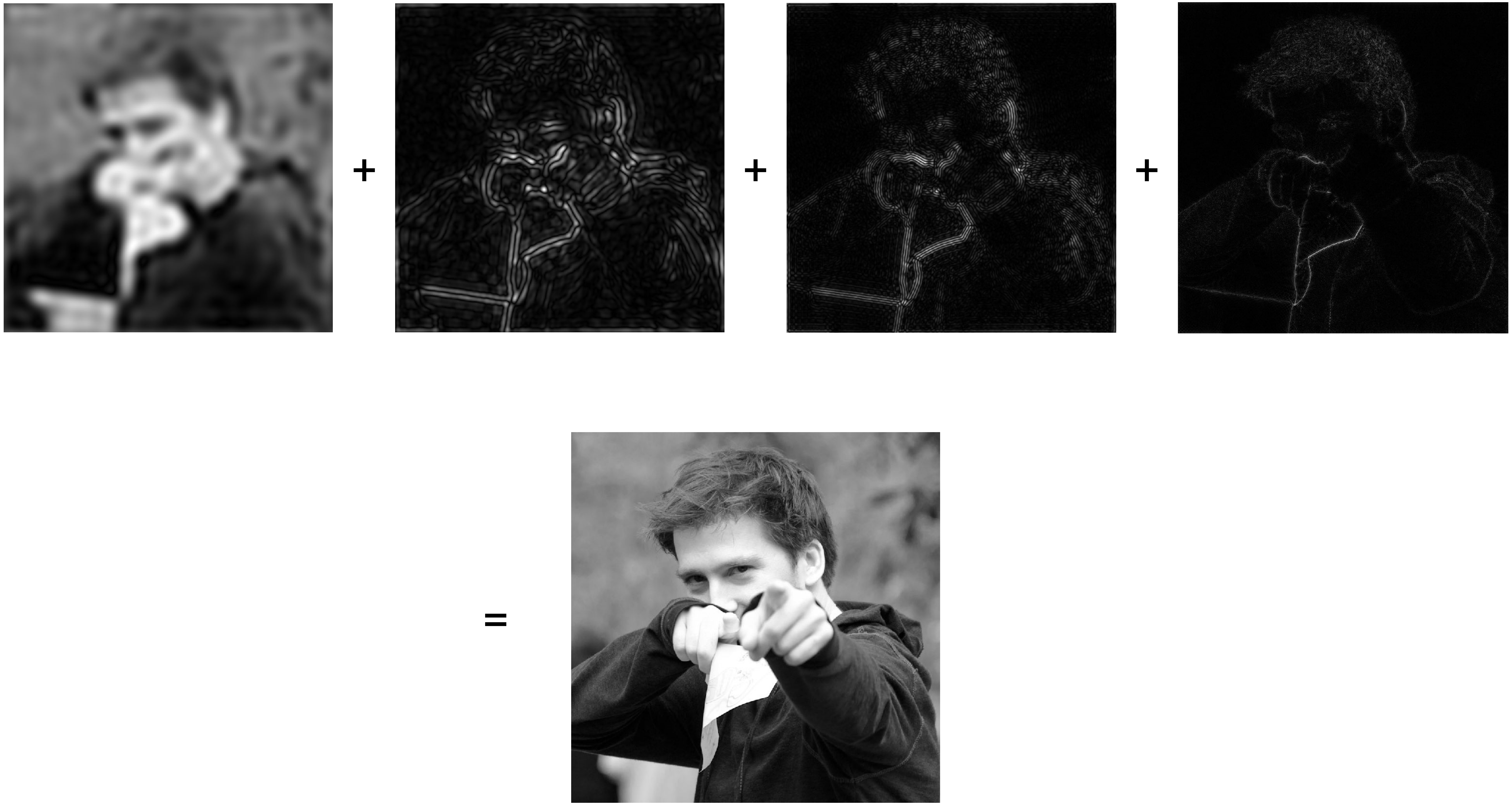


Spatial domain result
(strongest edges)



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

An image as a sum of its frequency components

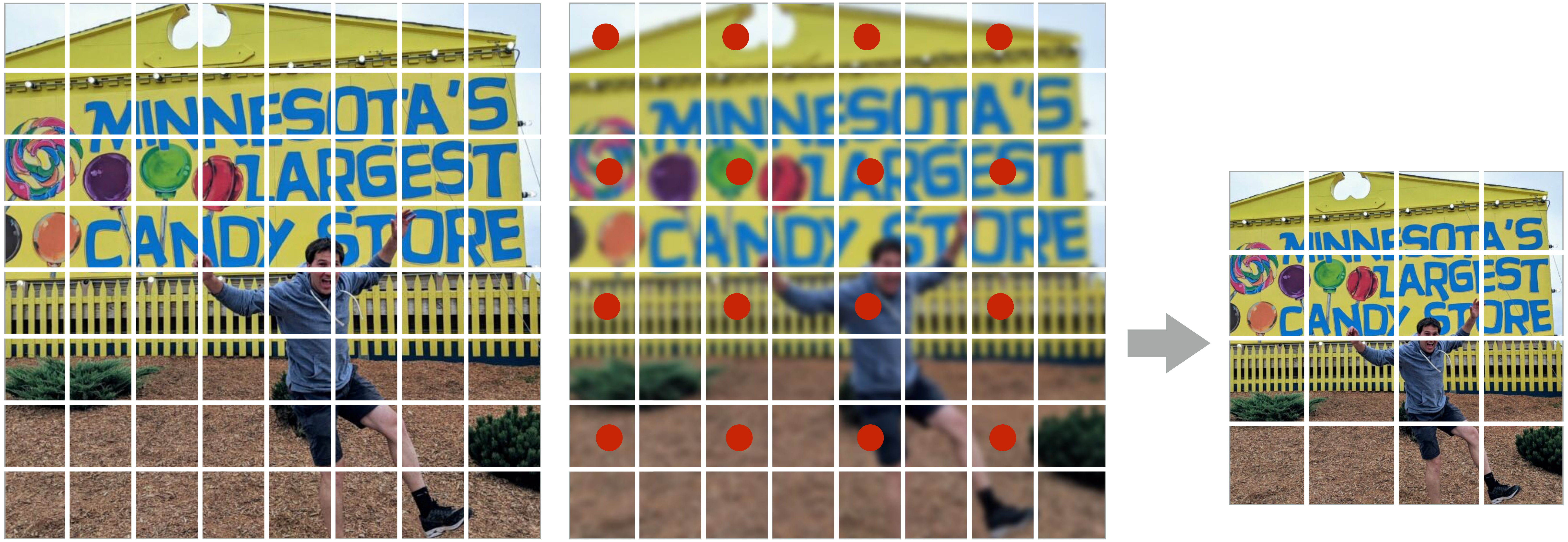


But what if we wish to localize image edits both in space and in frequency?

(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;
  }
}
```

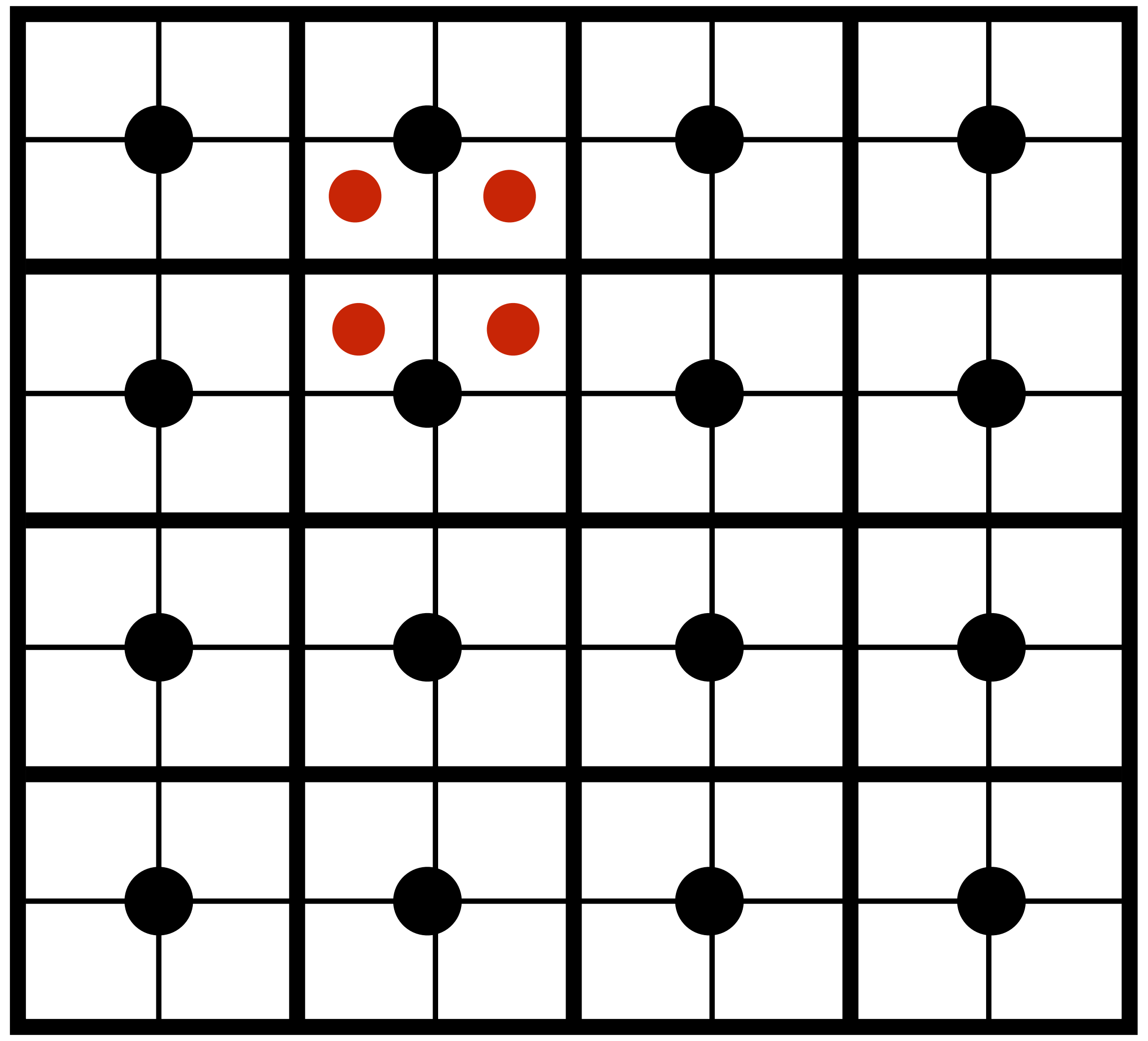
Upsample

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 = image

G_1 = down(G_0)

G_2 = 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

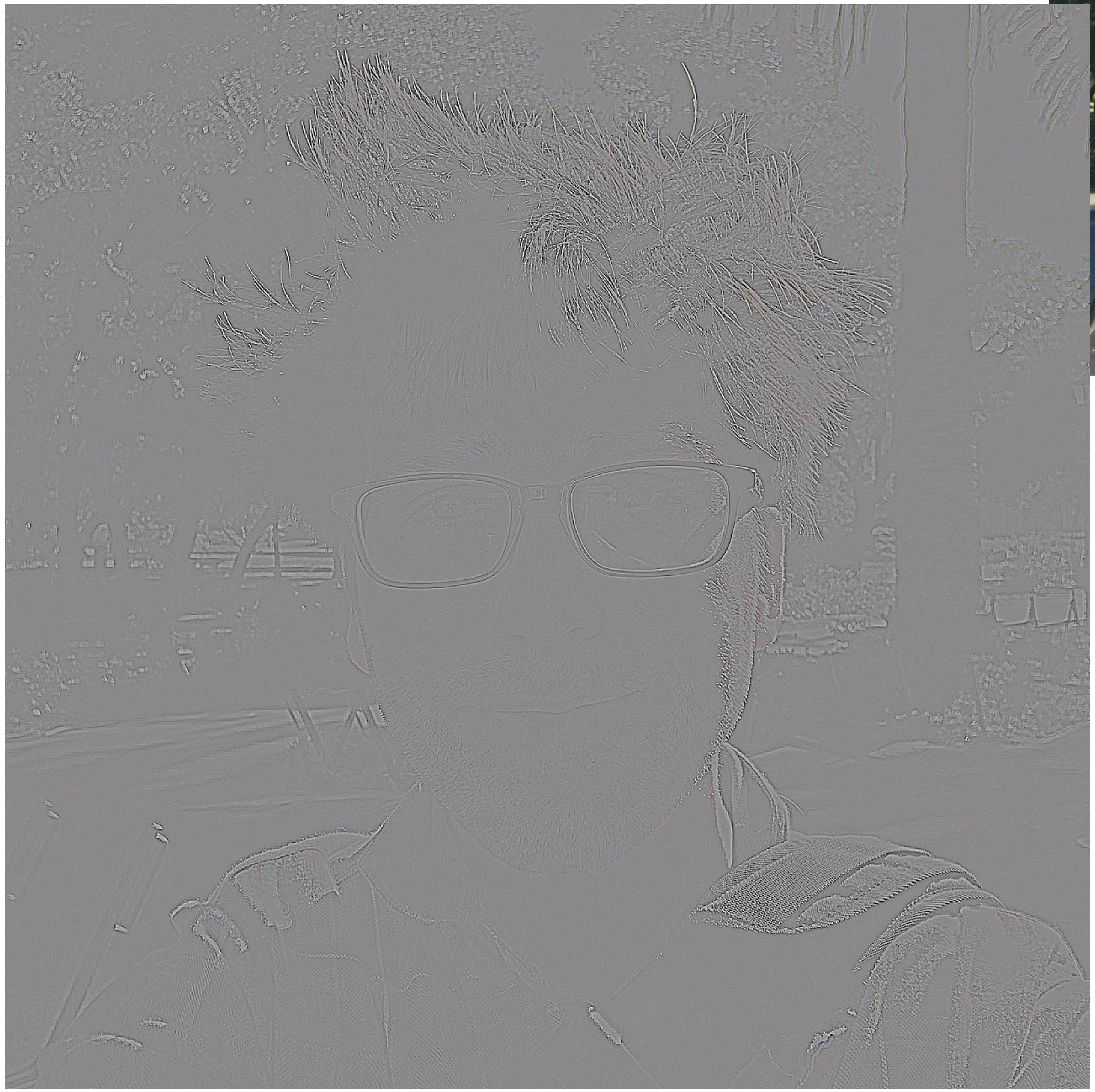
Gaussian pyramid

G_5

Laplacian pyramid

G_0

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



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

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

[Burt and Adelson 83]

Laplacian pyramid

$L_0 = G_0 - up(G_1)$

$L_1 = G_1 - up(G_2)$

Laplacian pyramid

$L_0 = G_0 - up(G_1)$

$L_1 = G_1 - up(G_2)$

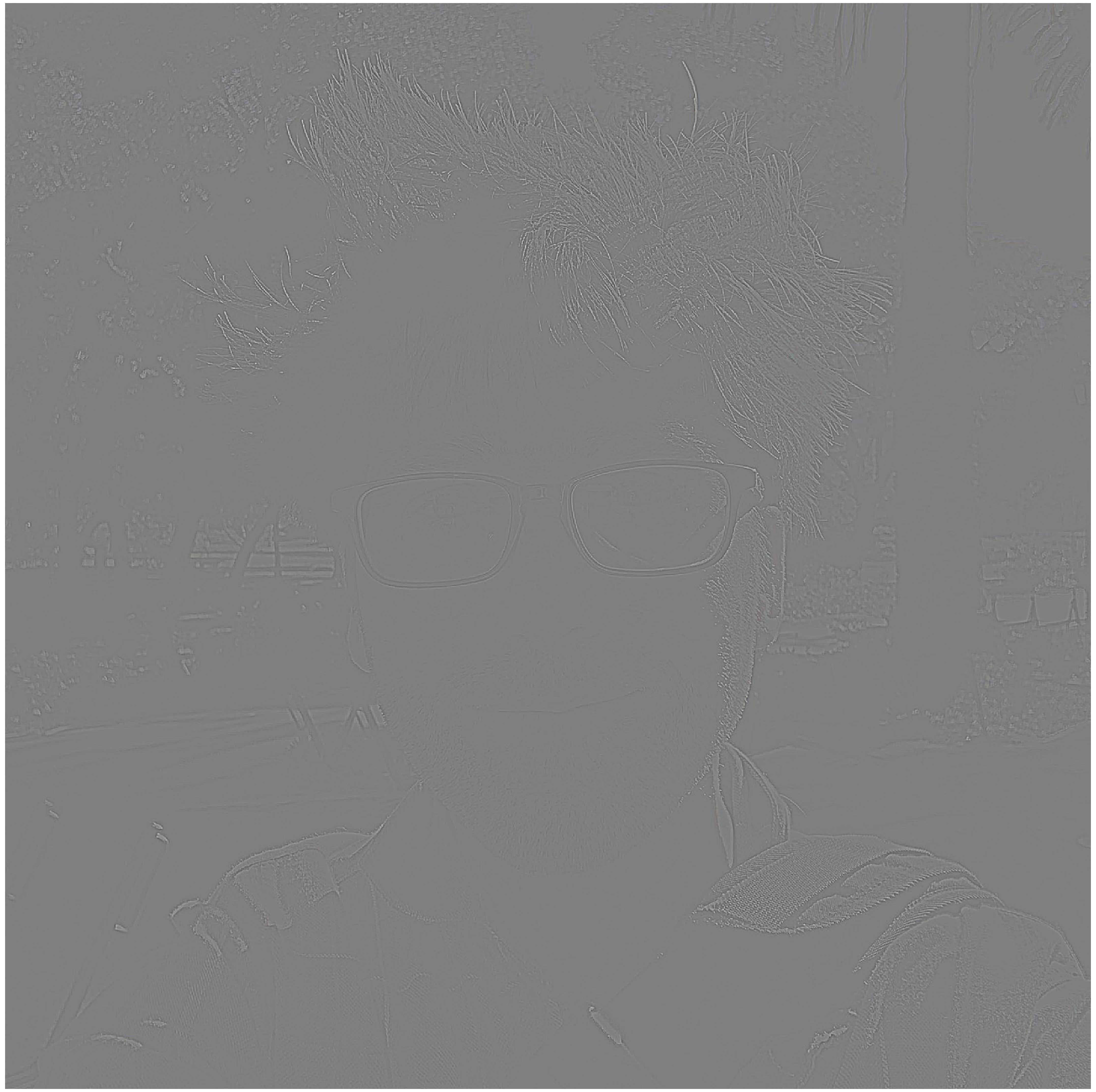
$L_2 = G_2 - up(G_3)$

$L_3 = G_3 - up(G_4)$

$L_4 = G_4$

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

$L_2 = G_2 - up(G_3)$

Laplacian pyramid

$L_3 = G_3 - up(G_4)$

Laplacian pyramid

$L_4 = G_4 - up(G_5)$

Laplacian pyramid

$L_5 = G_5$

Summary

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

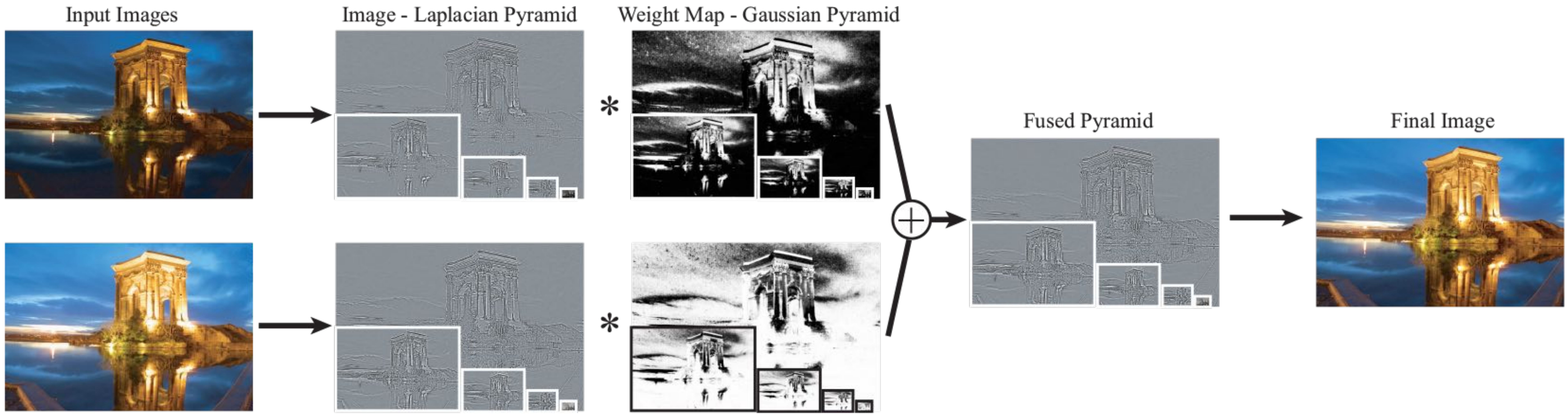
- **$G_i(x,y)$ — frequencies up to limit given by *i***

- **$L_i(x,y)$ — 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**

Use of Laplacian pyramid in tone mapping

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



Challenges of merging images

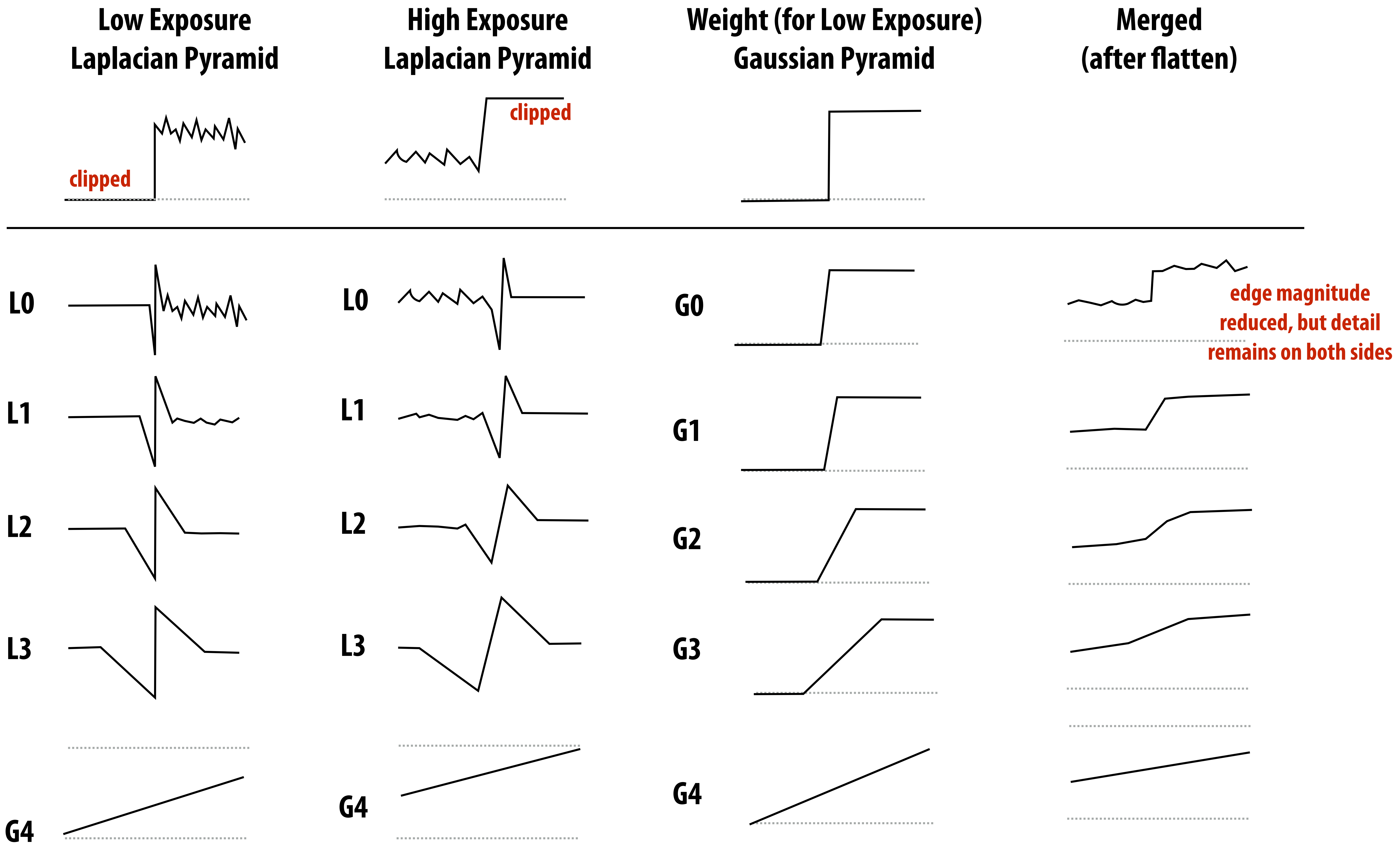
Four exposures (weights not shown)

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

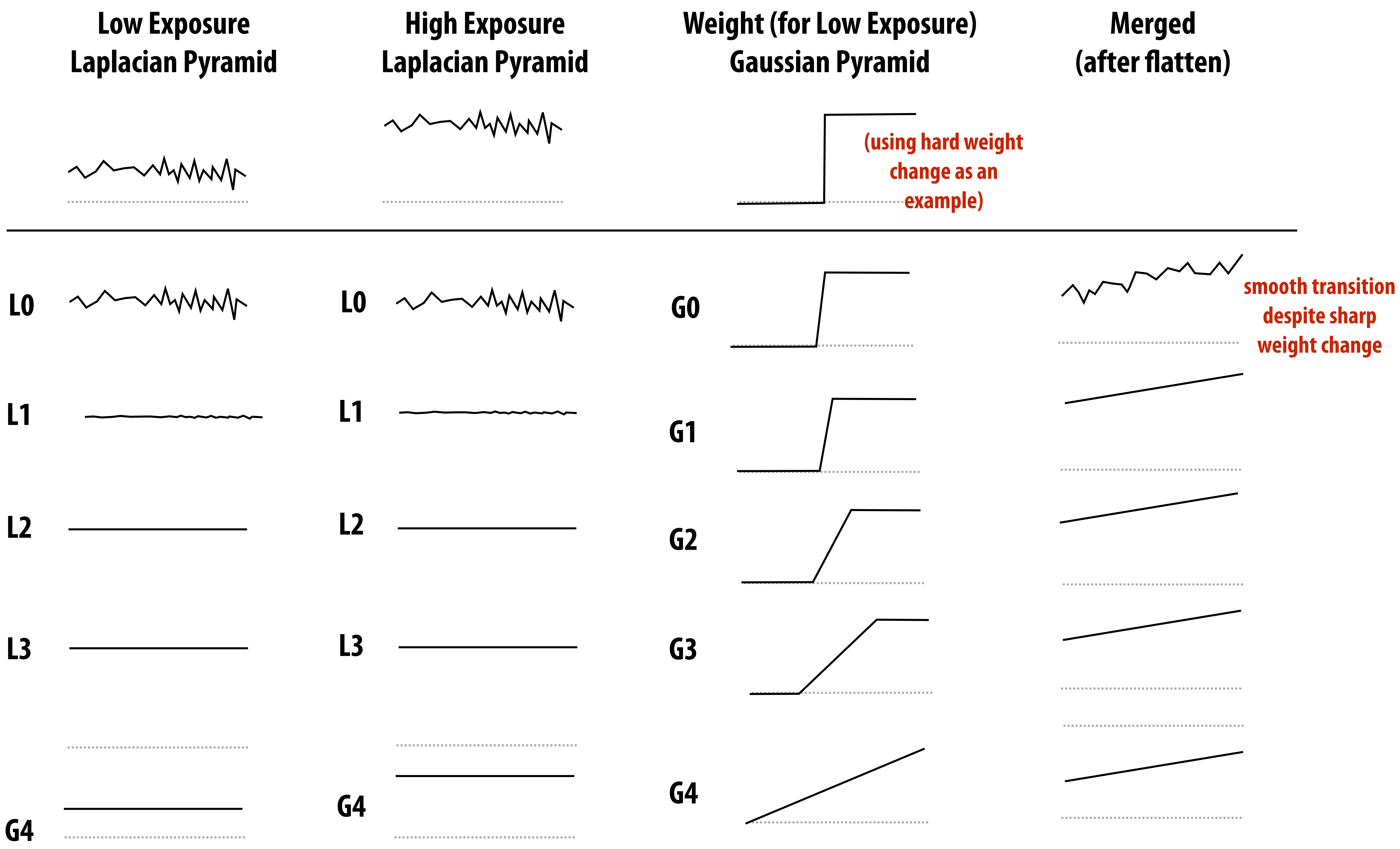
Merged result
(based on multi-resolution pyramid merge)

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

Consider low and high exposures of an edge



Consider low and high exposures of flat image region



Summary: simplified image processing pipeline

- Correct pixel defects
- Align and merge (to create high signal to noise ration RAW image)
- Correct for sensor bias (using measurements of optically black pixels)
- Vignetting compensation
- White balance
- Demosaic

(10-12 bits per pixel)
1 intensity value per pixel
Pixel values linear in energy

- Denoise
- Gamma Correction (non-linear mapping)
- Local tone mapping

3 x (10-12) bits per pixel
RGB intensity per pixel
Pixel values linear in energy

- Final adjustments sharpen, fix chromatic aberrations, hue adjust, etc.

3x8-bits per pixel
Pixel values perceptually linear