Lecture 23:

Image Processing

Computer Graphics and Imaging

UC Berkeley CS184

Case Study: JPEG Compression

JPEG Compression: The Big Ideas

Low-frequency content is predominant in images of the real world

The human visual system is:

- Less sensitive to changes in **chromaticity** than **luminance**
- Less sensitive to high frequency sources of error

Slide credit: Pat Hanrahan

Y’CbCr Color Space

Image credit: Wikipedia

Y’CbCr color space:

- A perceptually- motivated color space akin to **L*a*b*** .
- **Y’** is luma (lightness), **Cb** and **Cr** are chroma channels (blue-yellow and red-green distance from gray)

***Omitting discussion of nonlinear gamma encoding in Y’ channel**

Example Image

Original picture

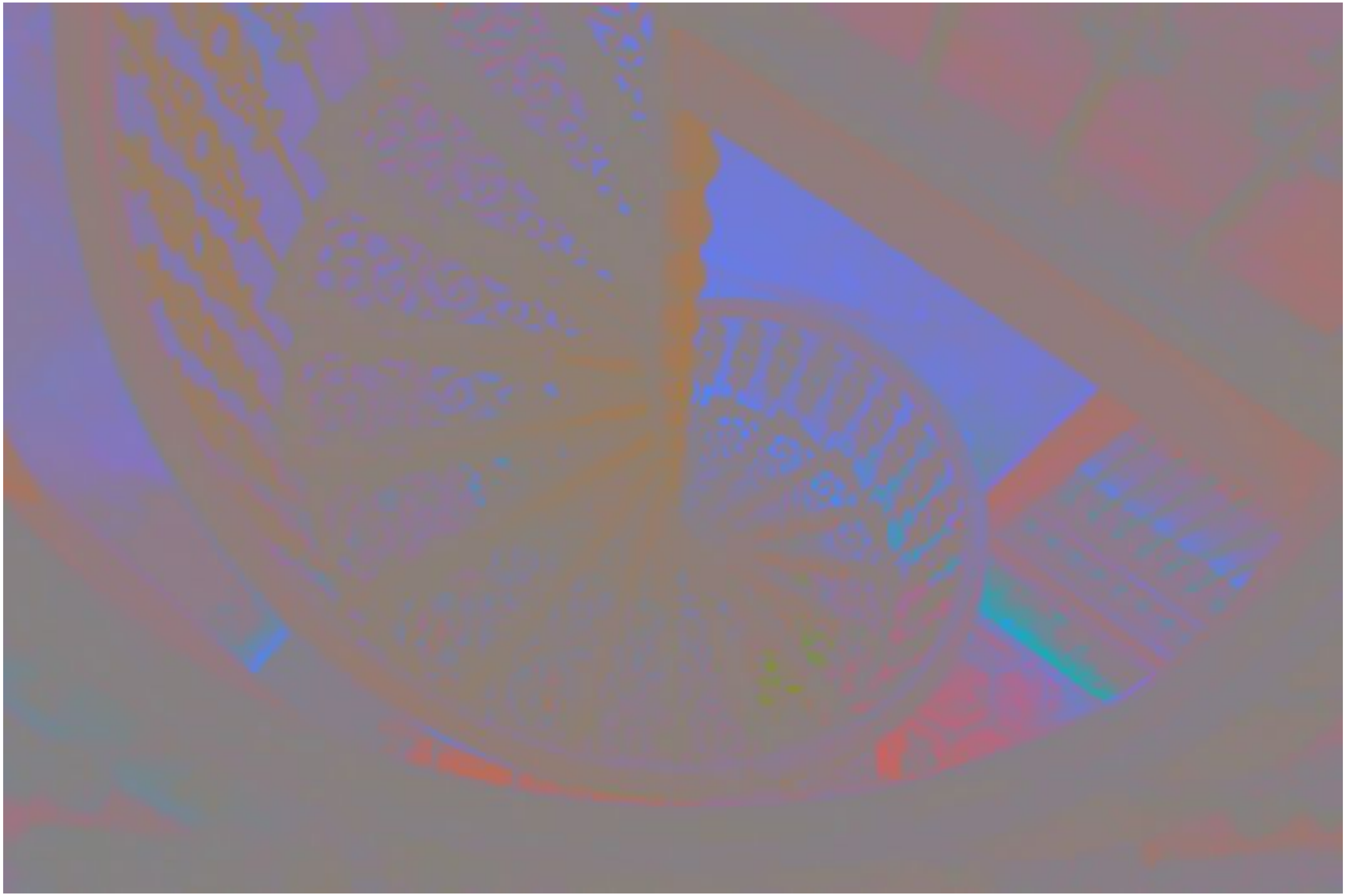
Y’ Only (Luma)

Luma channel

Downsampled Y’

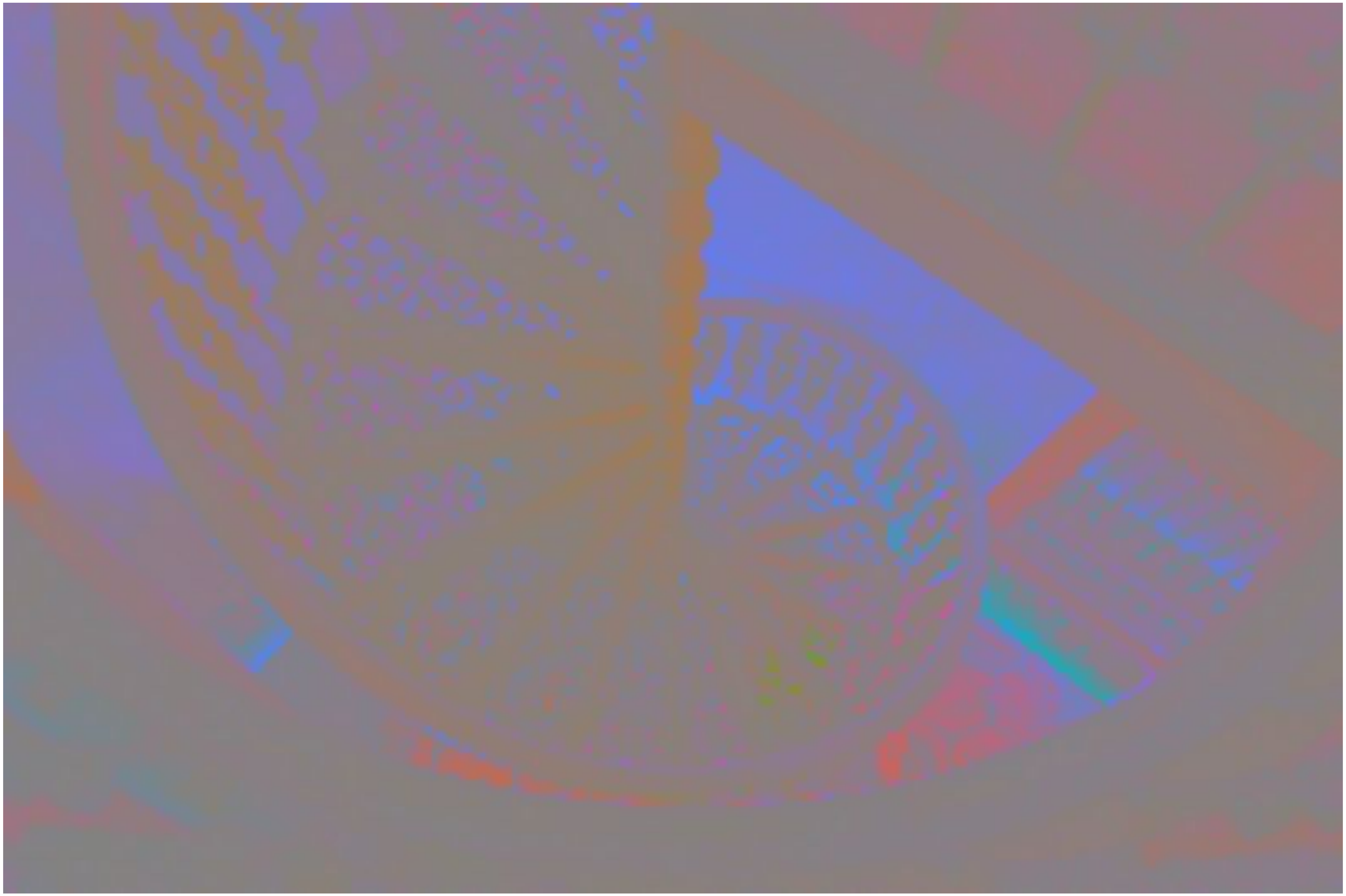
4x4 downsampled luma channel

CbCr Only (Chroma)



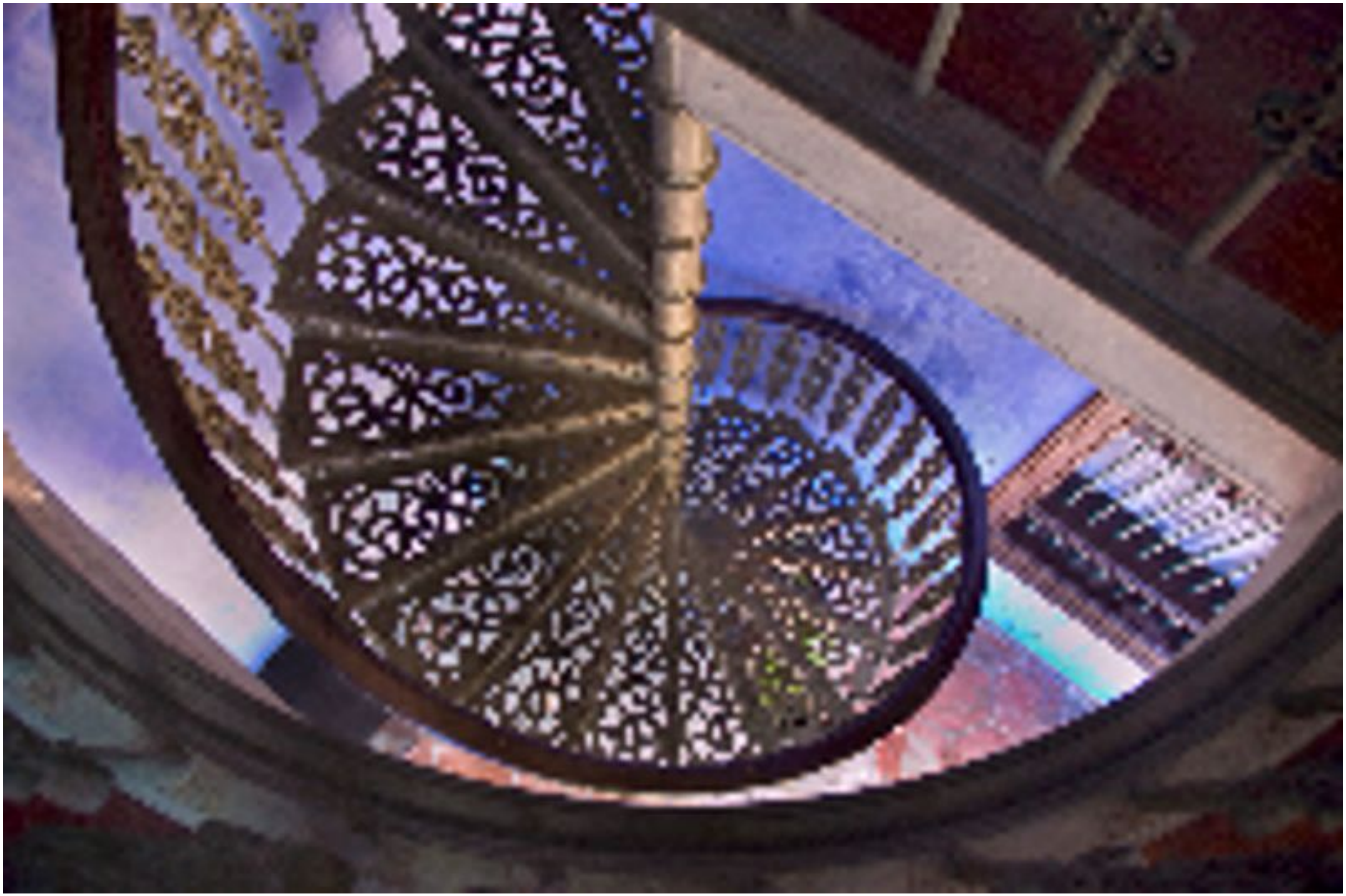
CbCr channels

Downsampled CbCr



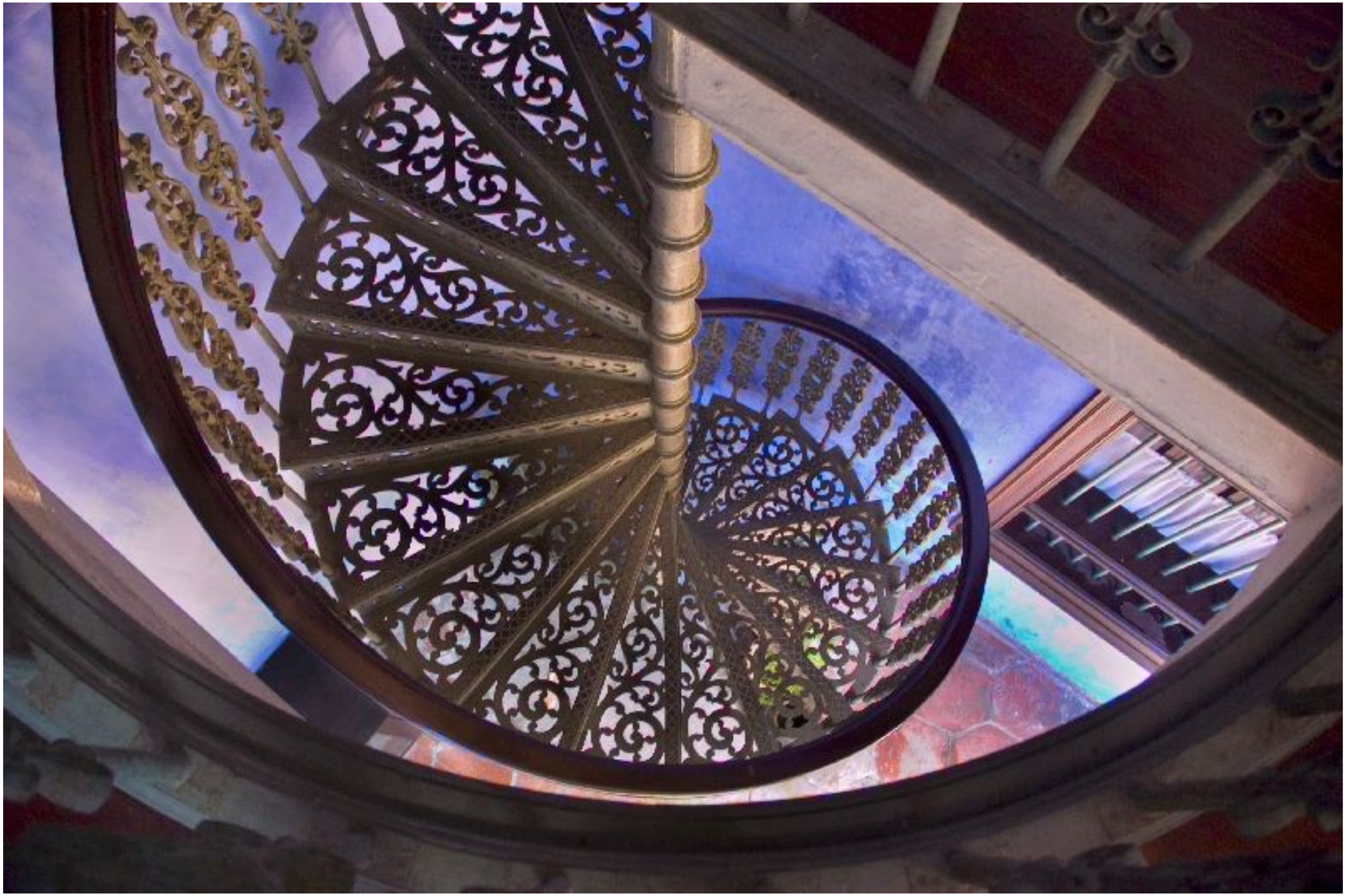
4x4 downsampled CbCr channels

Example: Compression in Y’ Channel



4x4 downsampled Y’, full-resolution CbCr

Example: Compression in CbCr Channels



Full-resolution Y’, 4x4 down sampled CbCr

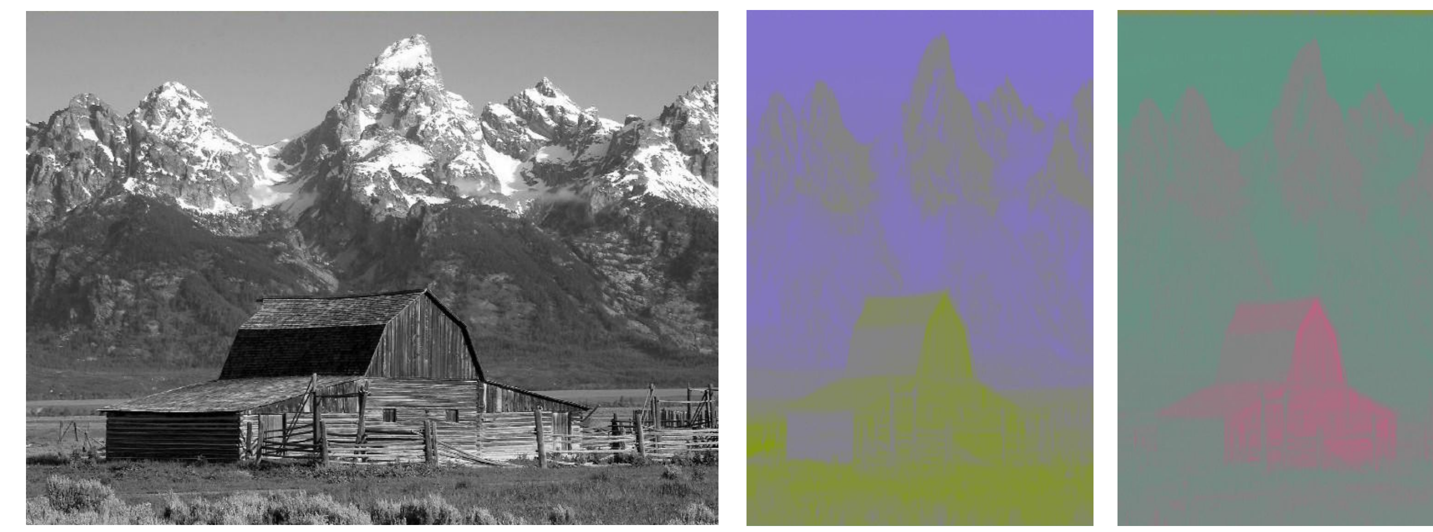
Original Image

JPEG: Chroma Subsampling in Y’CbCr Space

Subsample chroma channels
(e.g. to 4:2:2 or 4:2:0 format)

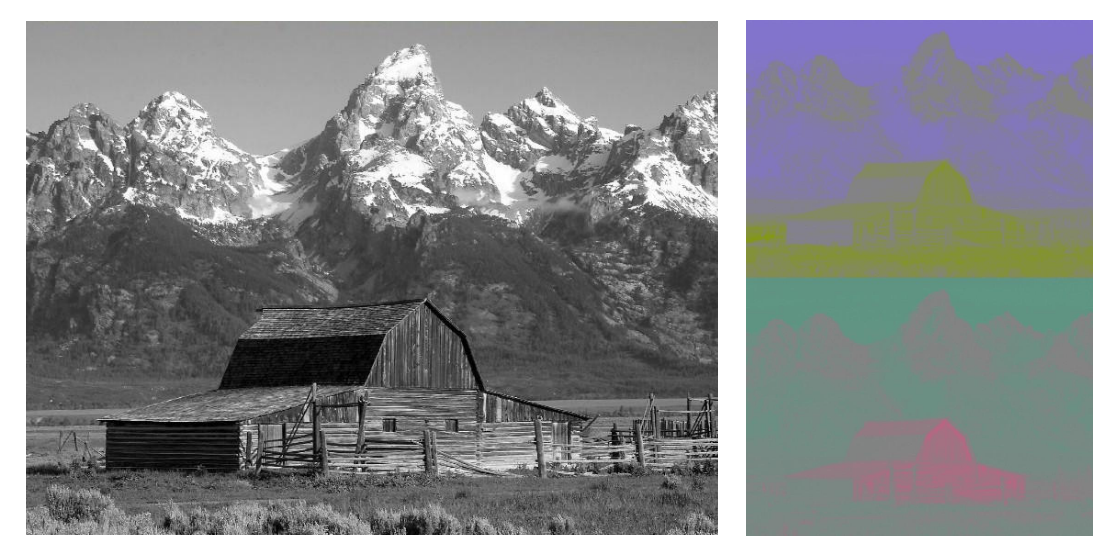
4:2:2 representation: (retain 2/3 values)

- Store Y’ at full resolution
- Store Cb, Cr at half resolution in horizontal dimension



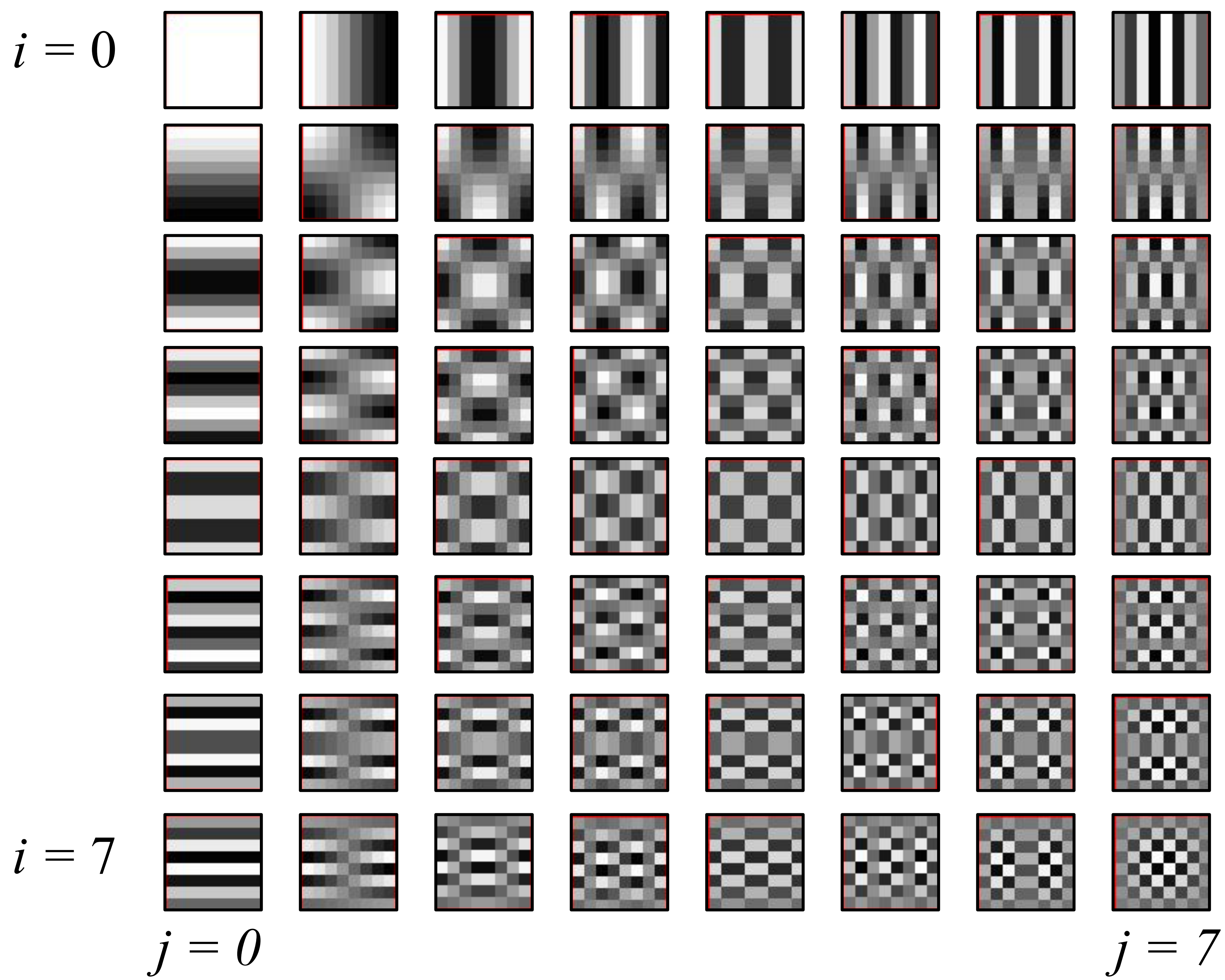
4:2:0 representation: (retain 1/2 values)

- Store Y’ at full resolution
- Store Cb, Cr at half resolution in both dimensions



JPEG: Discrete Cosine Transform (DCT)

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



Apply discrete cosine transform (DCT) to each 8x8 block of image values

DCT computes projection of image onto 64 basis functions: basis[i, j]

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

JPEG Quantization: Prioritize Low Frequencies

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

Quantization Matrix

Result of DCT
(encoded in cosine basis)

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

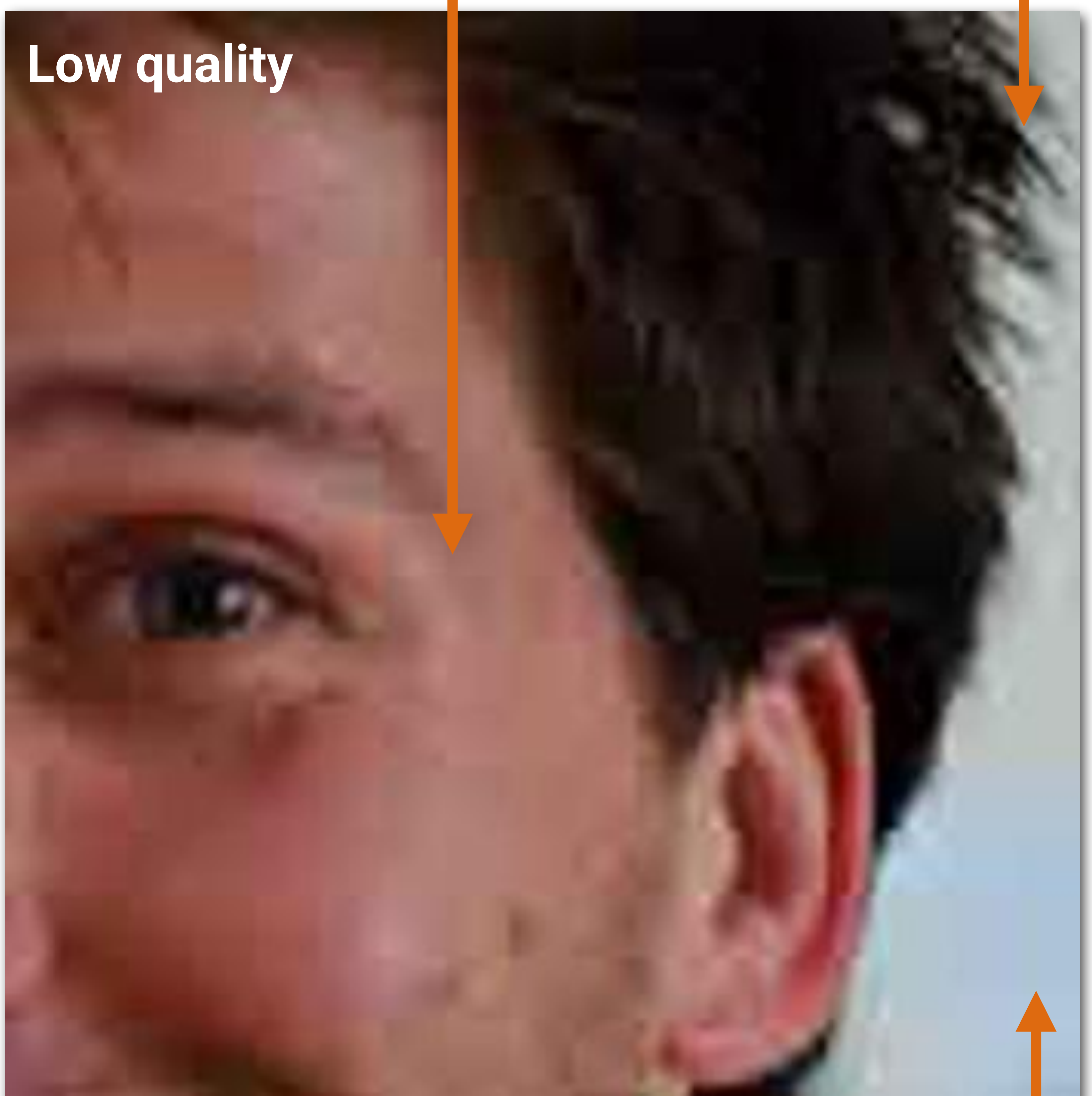
Quantization produces only a few bits per coefficient (small values)
Note: quantization zeros out many coefficients!

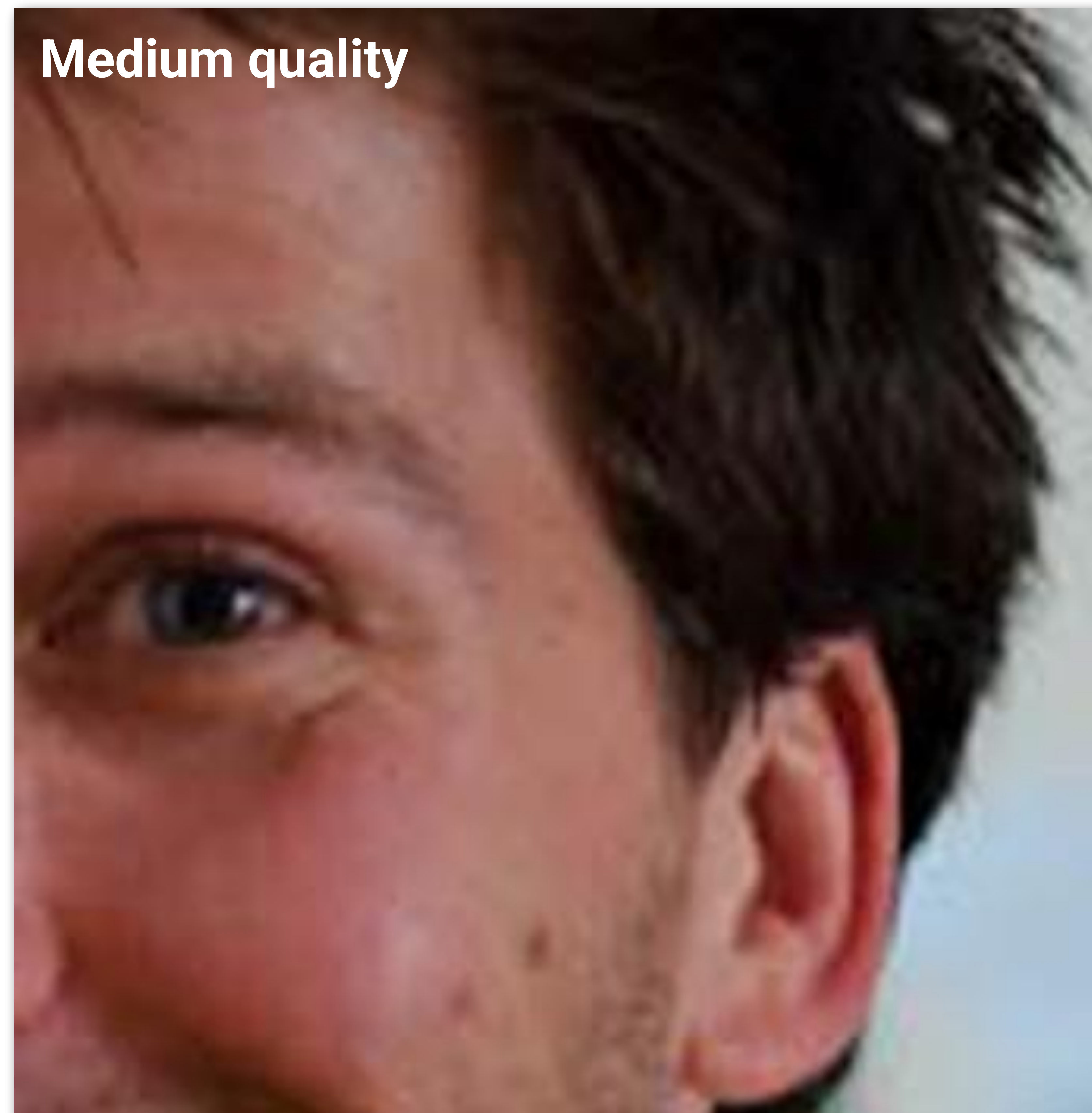
Slide credit: Wikipedia, Pat Hanrahan

JPEG: Compression Artifacts

Noticeable 8x8 pixel block boundaries

Noticeable error near large color gradients





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

JPEG: Compression Artifacts

Original Image

Quality Level 9

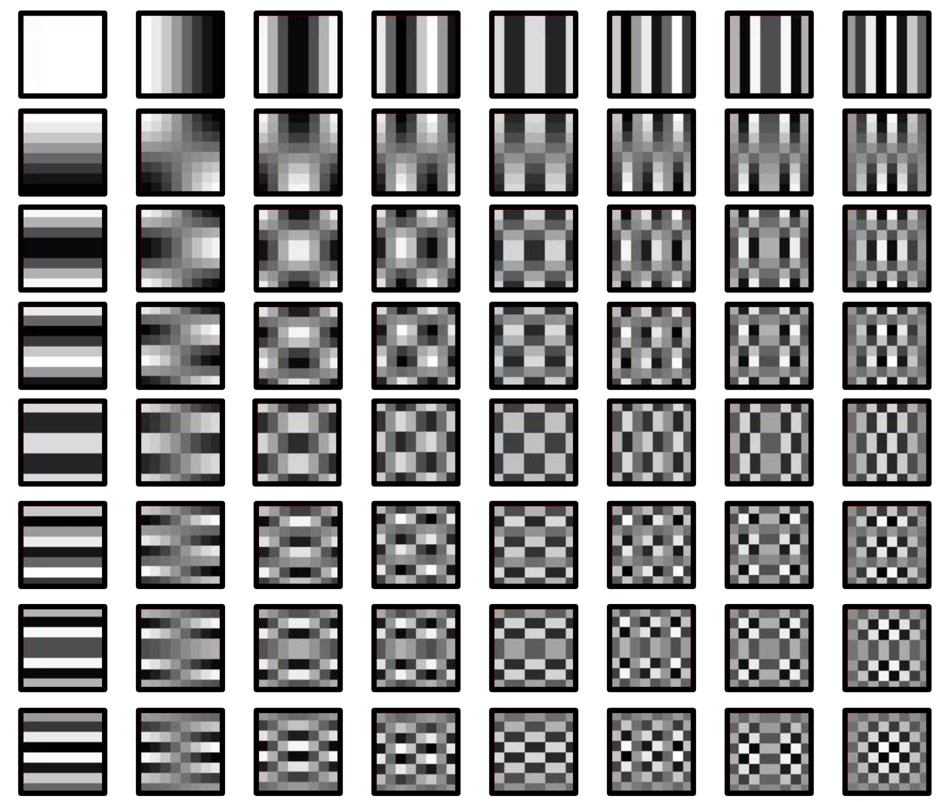
Quality Level 3

Quality Level 1

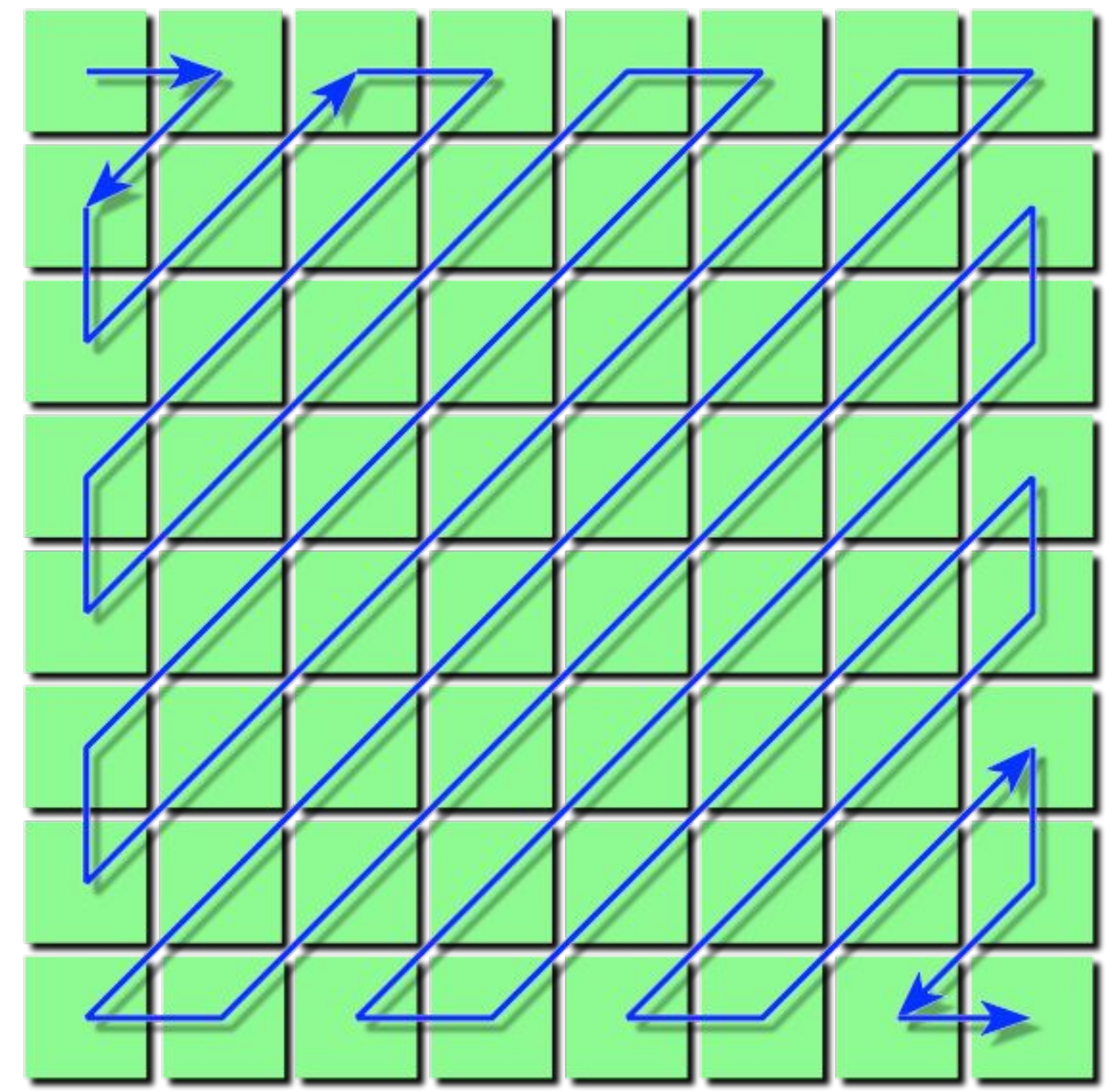
Lossless Compression of Quantized DCT Values

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

Quantized DCT Values



Basis functions



Reordering

Entropy encoding: (lossless)

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

Image credit: Wikipedia

JPEG Compression Summary

1. **Convert image to Y'CbCr color space**
2. **Downsample CbCr (to 4:2:2 or 4:2:0)** (information loss occurs here)
3. **For each color channel (Y', Cb, Cr):**

   For each 8x8 block of values

   Compute DCT

   Quantize results Reorder values (information loss occurs here)

   Run-length encode 0-spans

   Huffman encode non-zero values

Theme: Exploit Perception in Visual Computing

JPEG is an example of exploiting characteristics of human perception to build efficient visual systems

We are perceptually insensitive to color errors:

- Separate luminance from chrominance in color representations (e.g, Y’CbCr) and compress chrominance

We are less perceptually sensitive to high-frequency error

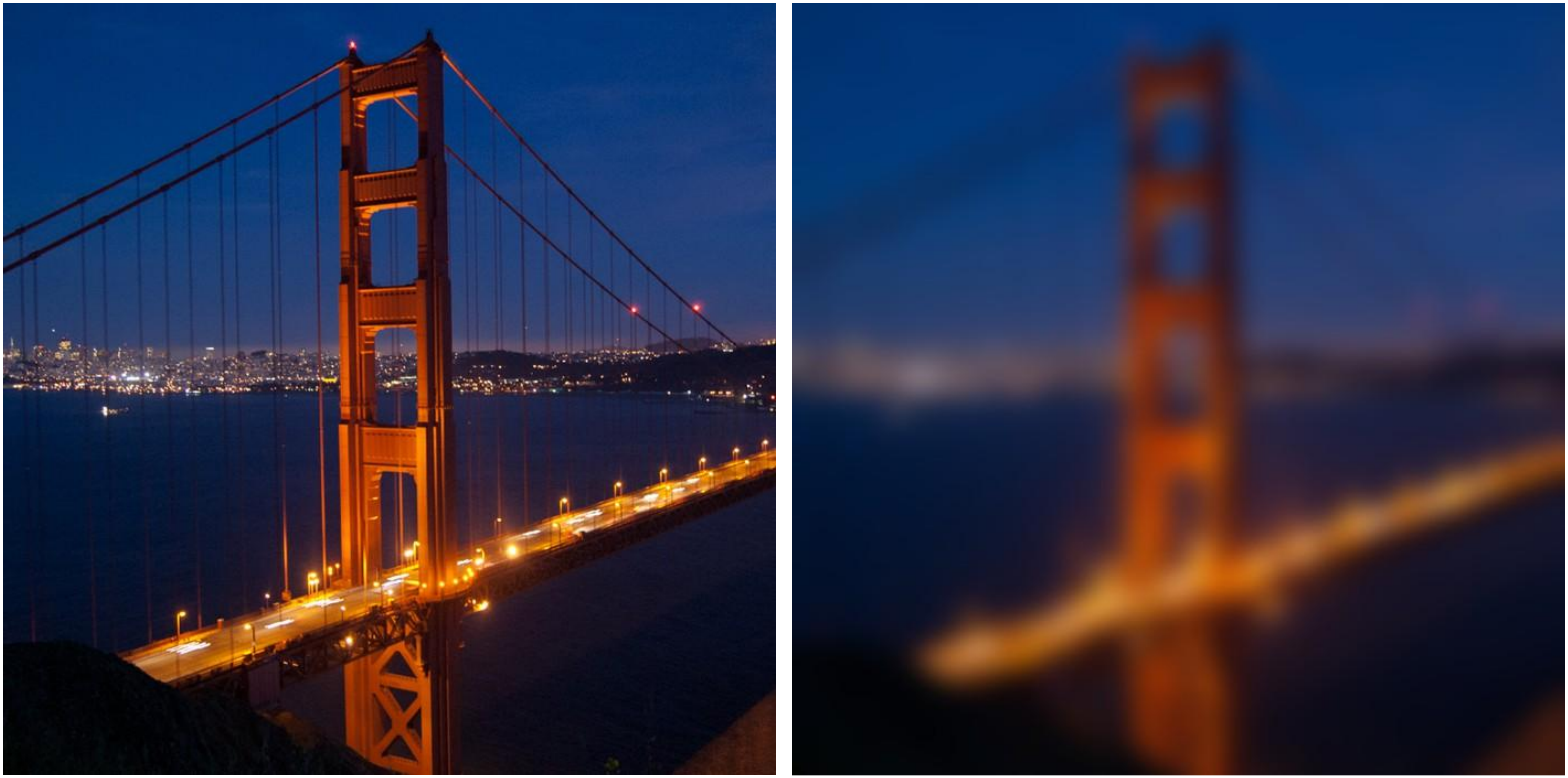
- Use a frequency-based encoding (cosine transform) and compress high-frequency values

We perceive lightness non-linearly (not discussed in this lecture)

- Encode pixel values non-linearly to match perceived brightness using **gamma curve**

Image Processing Operations

Example Image Processing Operations

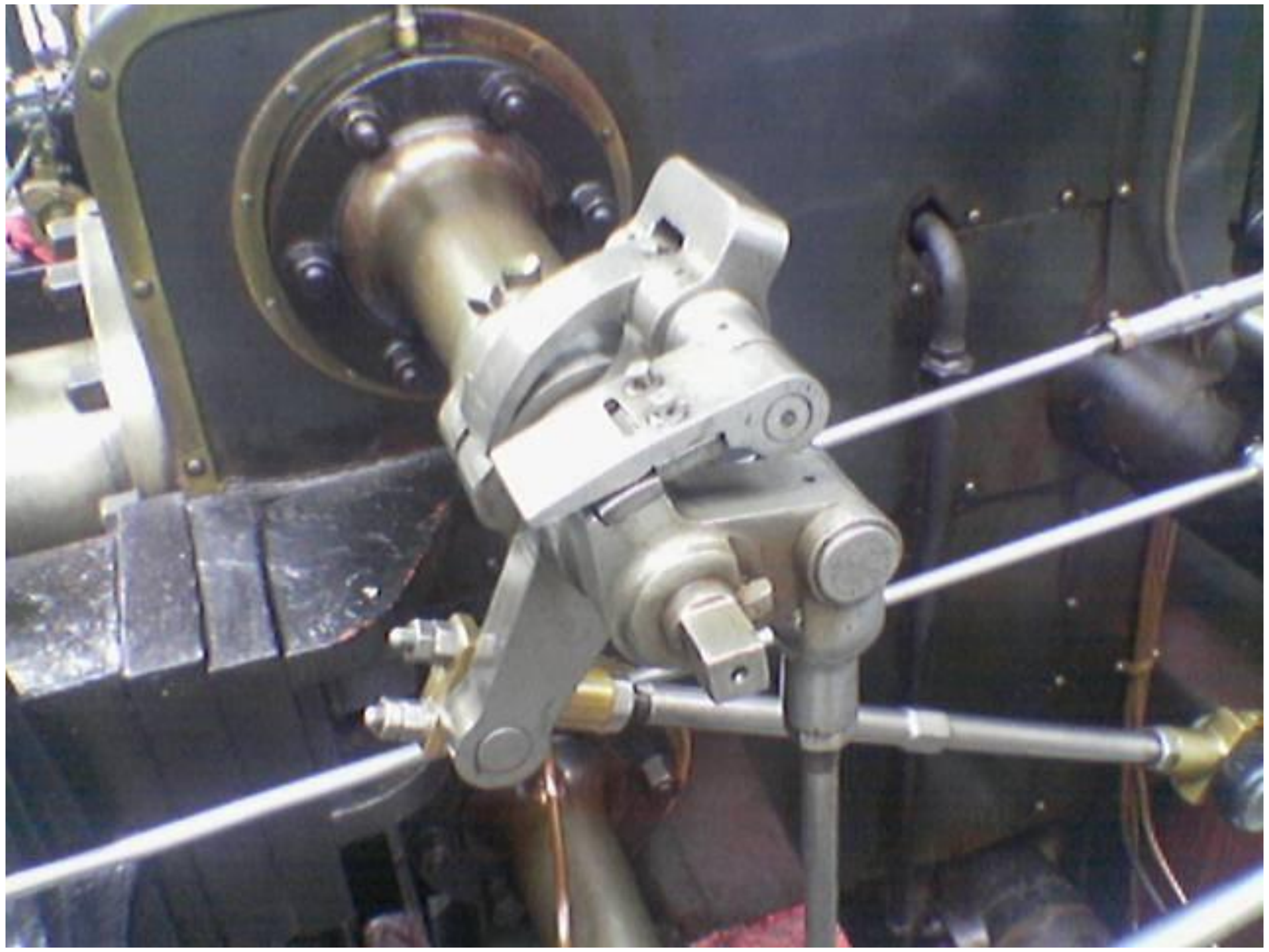


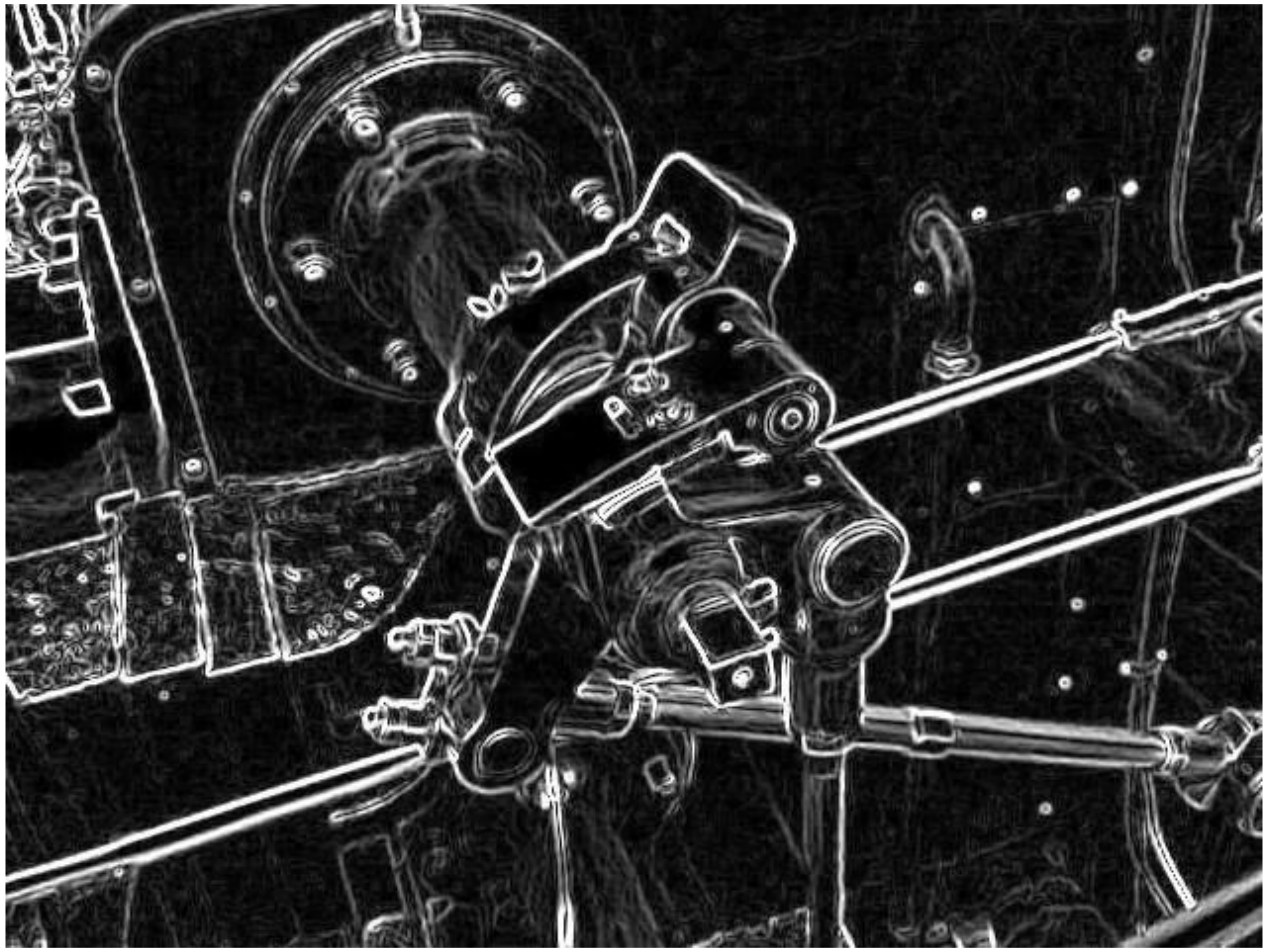
Blur

Example Image Processing Operations

Sharpen

Example Image Processing Operations



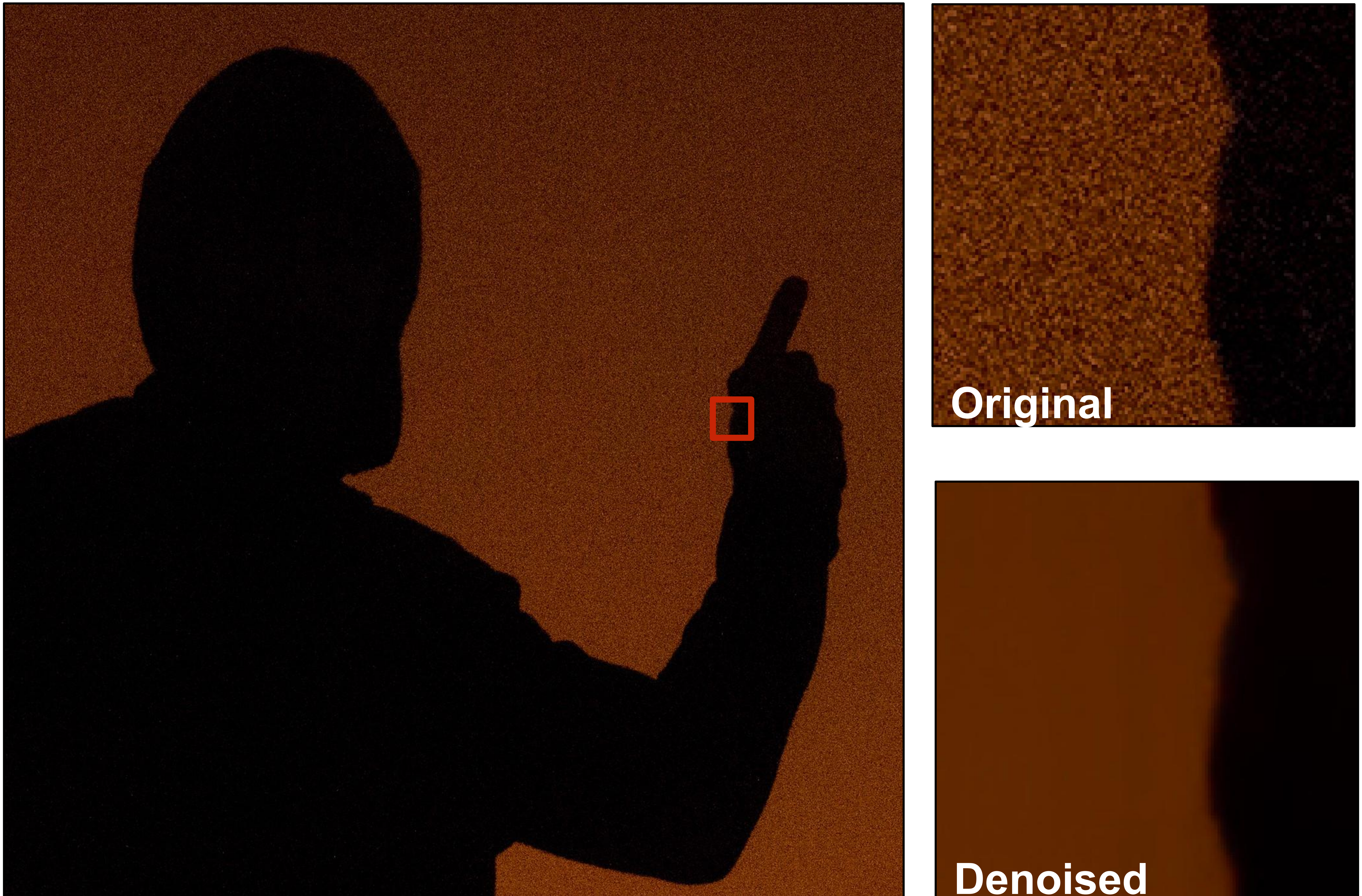


Edge Detection

Example Image Processing Operations

A "Smart" Blur (Preserves Crisp Edges)

Denoising



Review: Convolution

$$\underbrace{(f * g)(x)}_{\text{output signal}} = \int_{-\infty}^{\infty} \underbrace{f(y)}_{\text{filter}}\,\underbrace{g(x-y)}_{\text{input signal}}\,dy$$

Discrete 2D Convolution

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

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

Then:

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

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

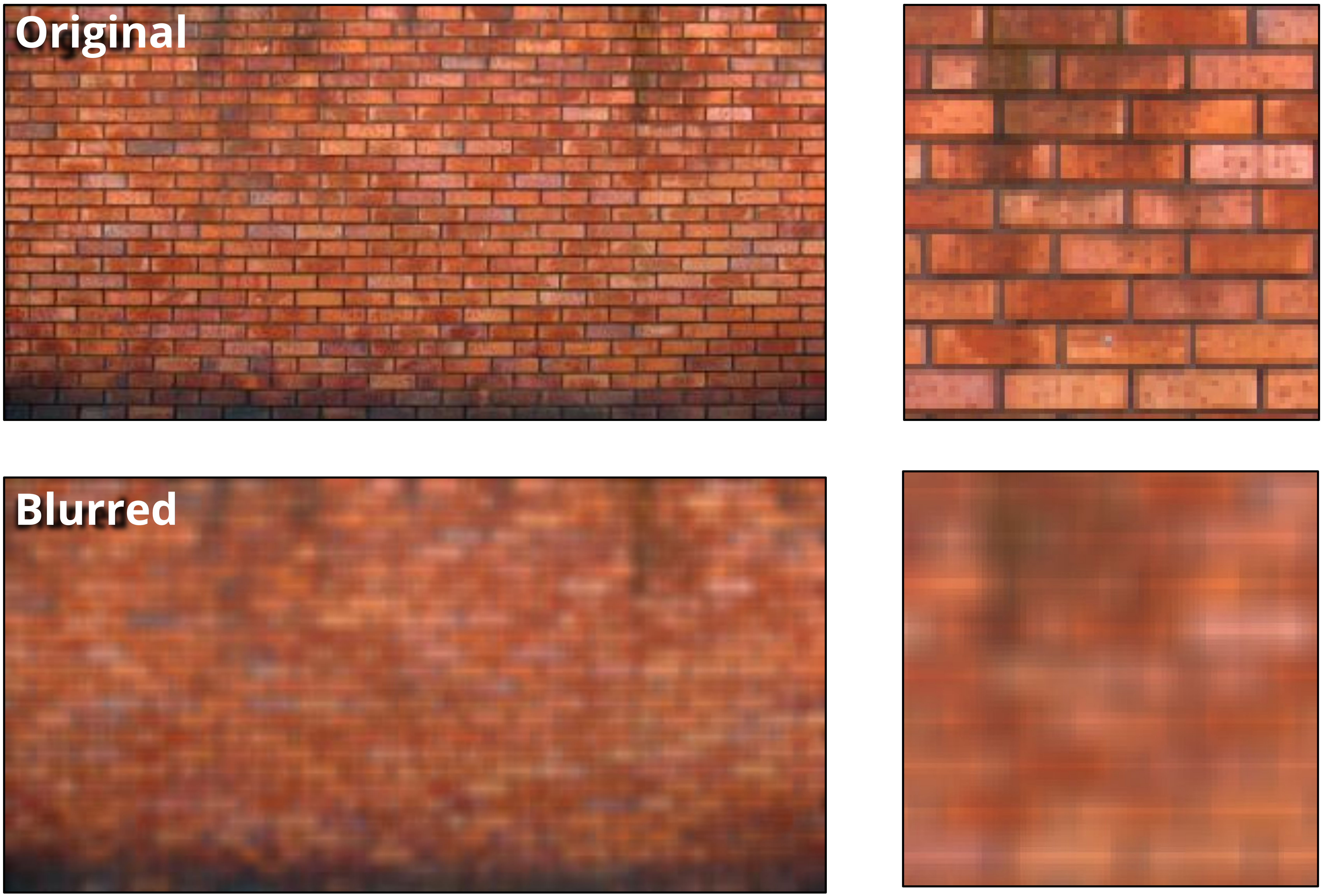
These values are often called "filter weights" or the "kernel".

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;
  }
}
```

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

7x7 Box Blur



Gaussian Blur

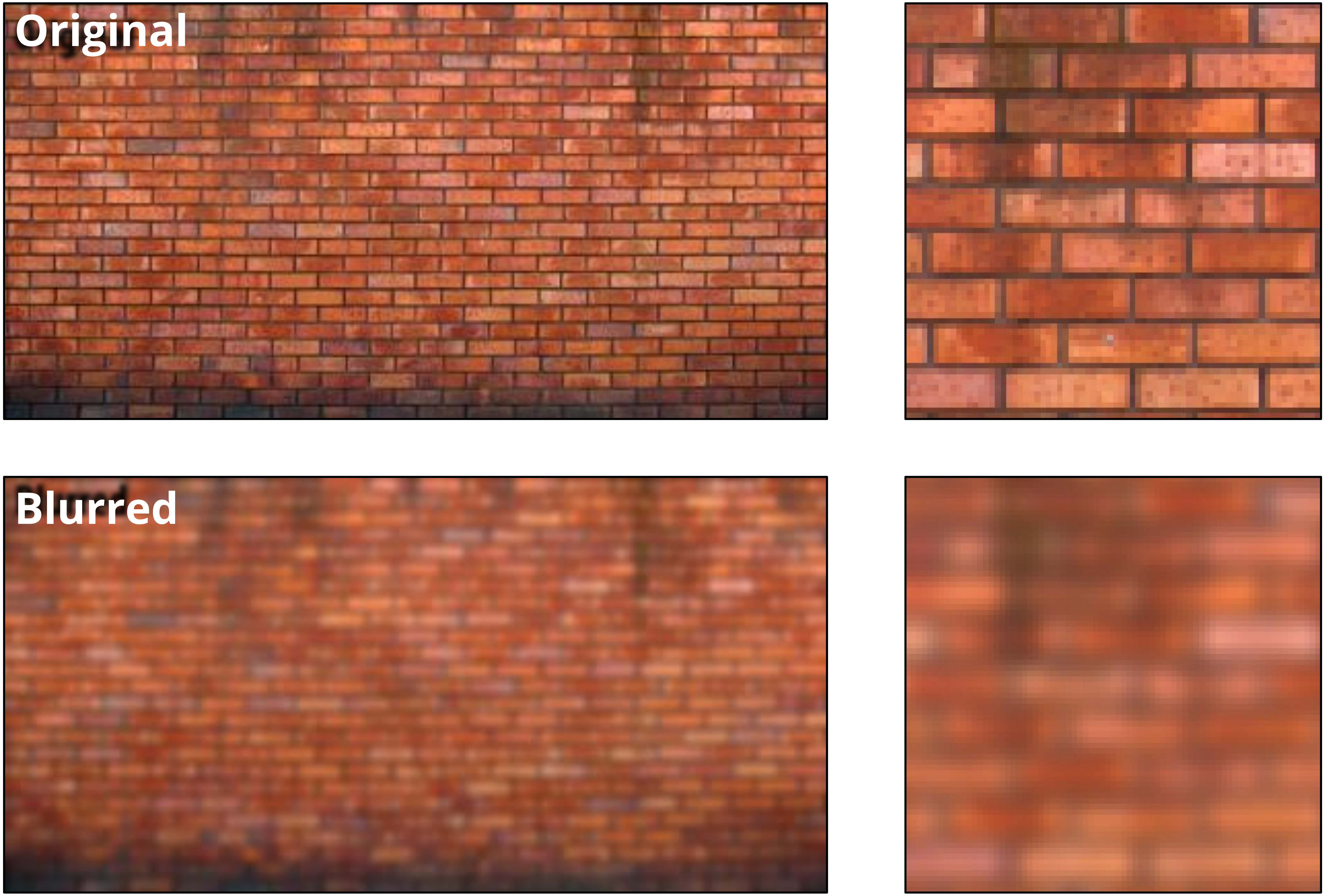
Obtain filter coefficients from sampling 2D Gaussian

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

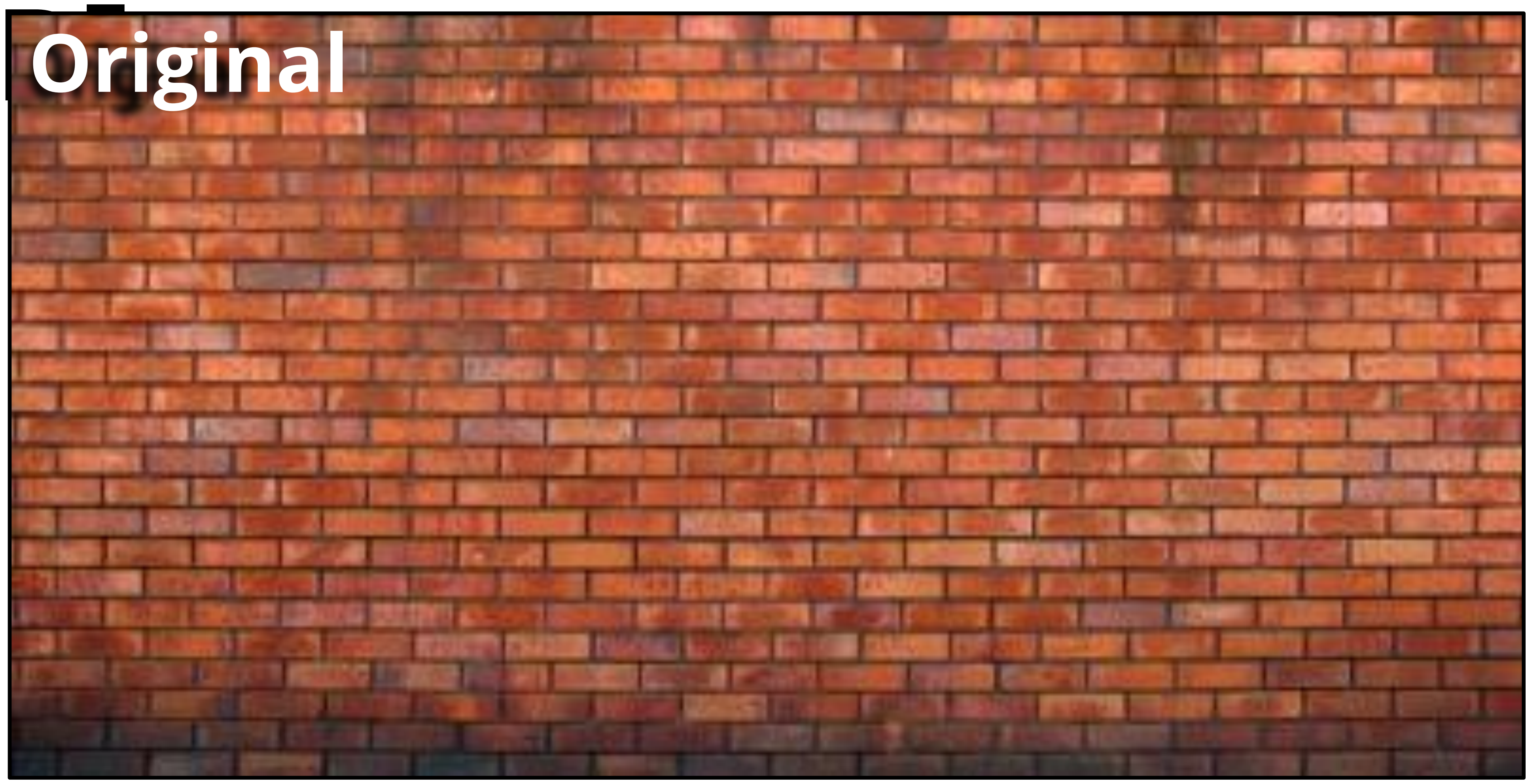
- Produces weighted sum of neighboring pixels (contribution falls off with distance)
  - Truncate filter beyond certain distance

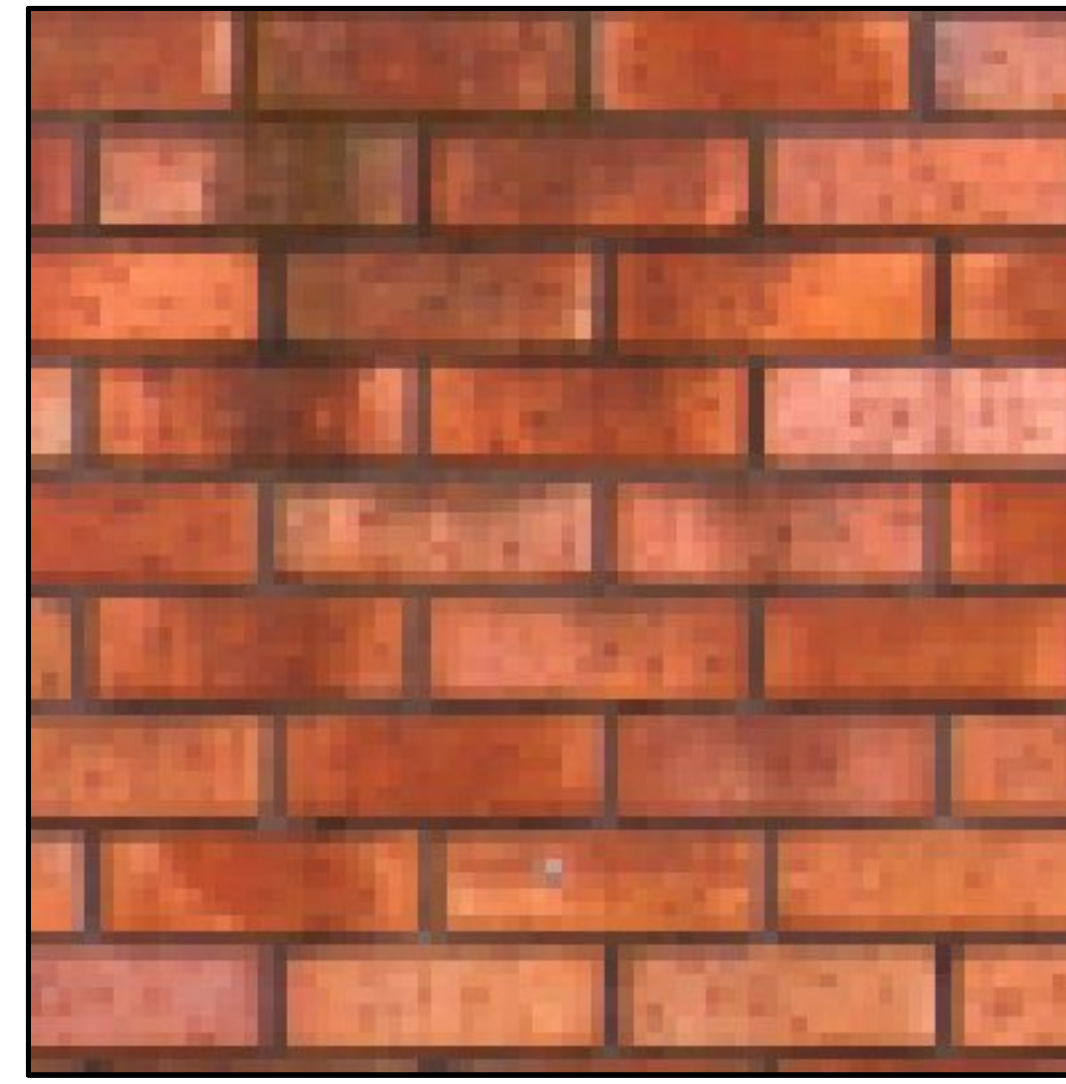
$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

7x7 Gaussian Blur



Compare: 7x7 Box

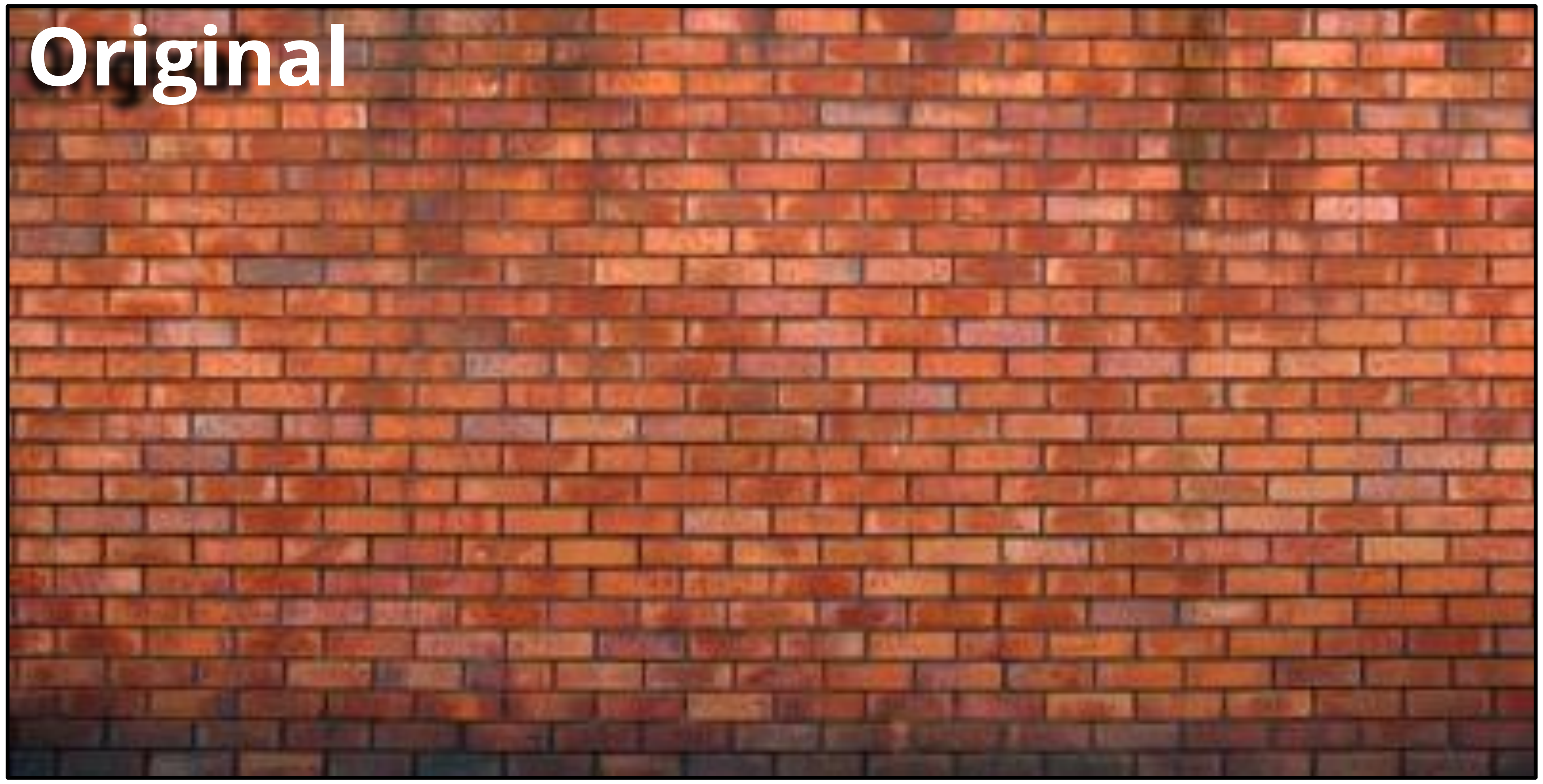


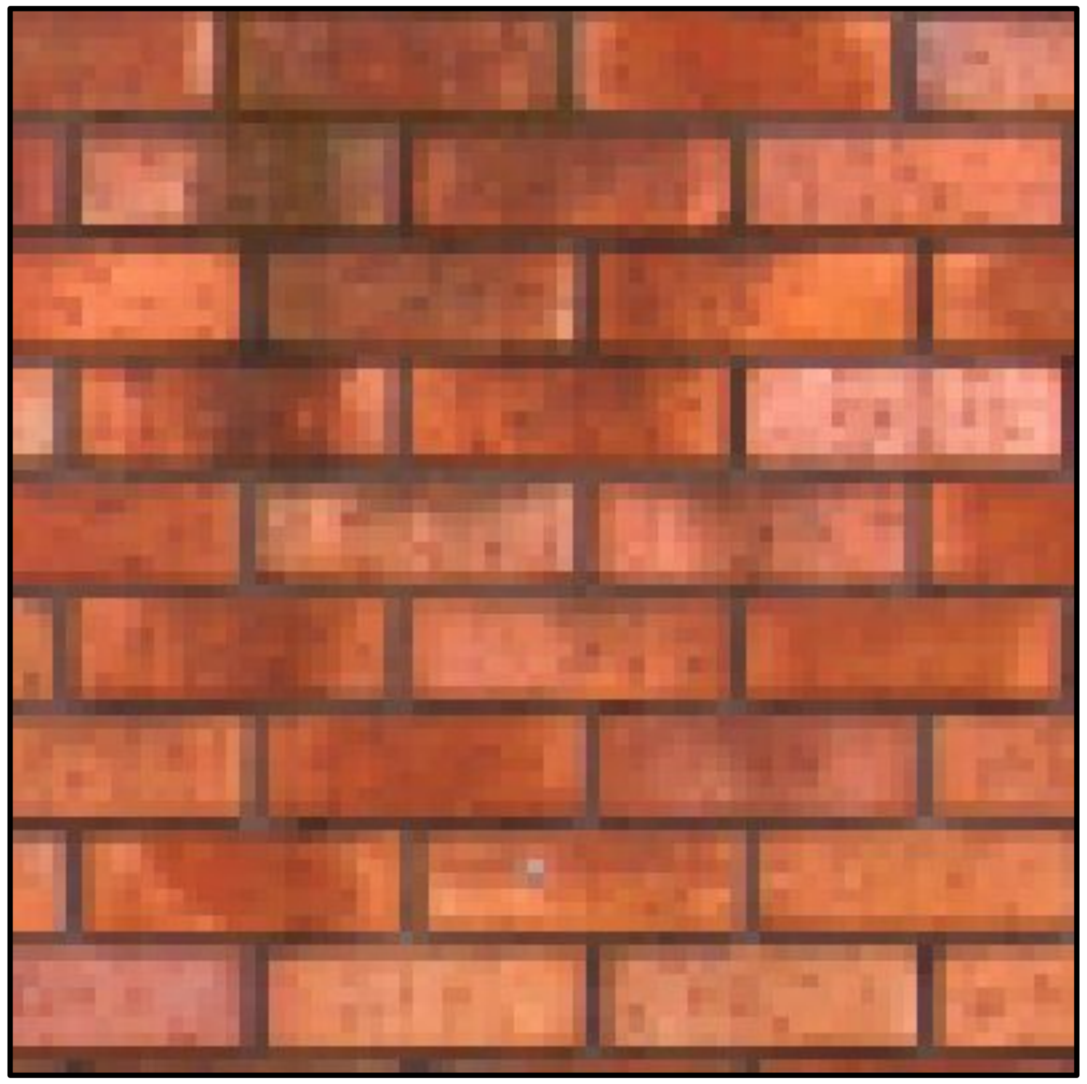


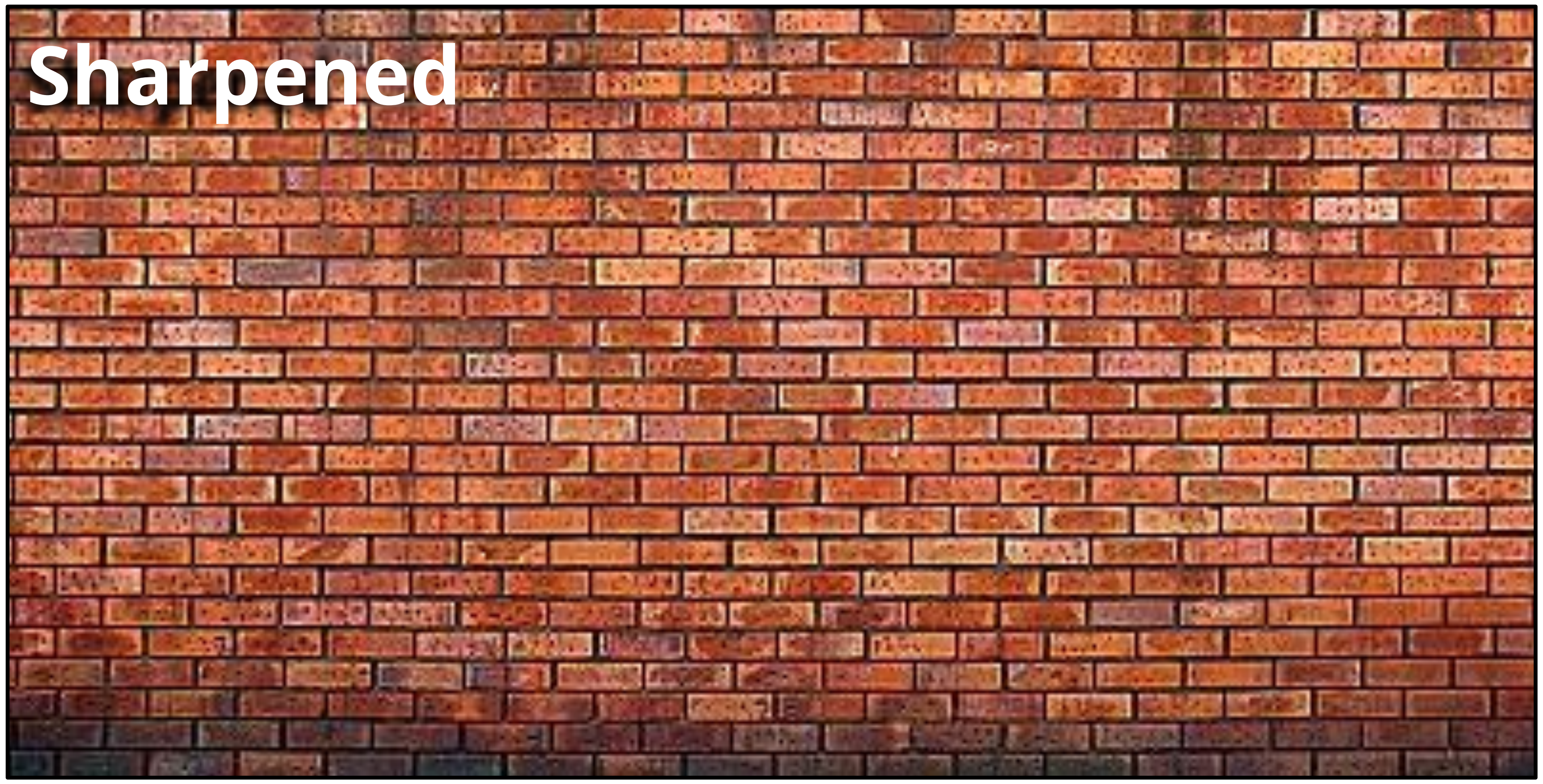
What Does this Convolution Filter Do?

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

3x3 Sharpen Filter







What Do these Convolution Filters Do?

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

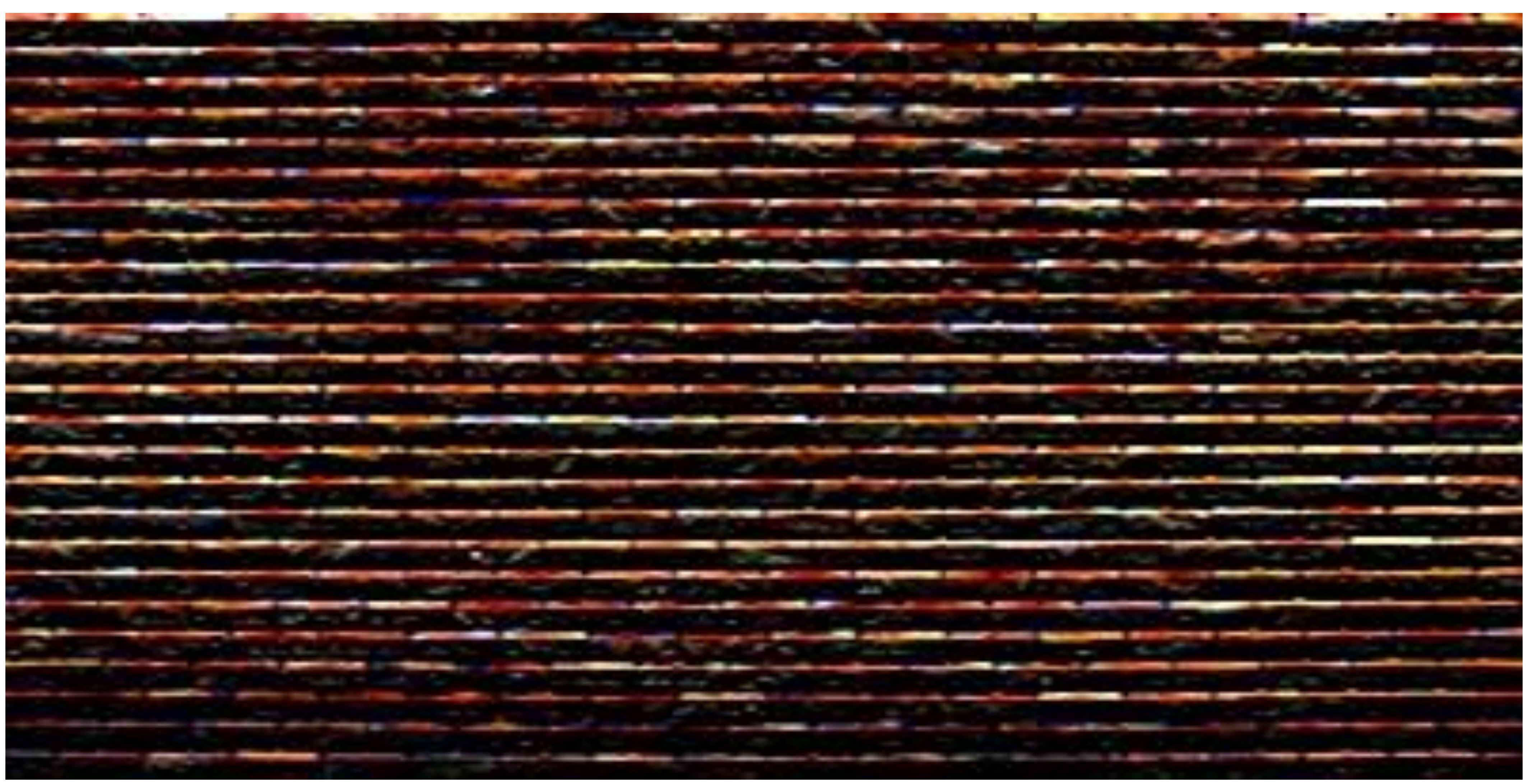
Extracts horizontal gradients

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

Extracts vertical gradients

Gradient Detection Filters

Horizontal gradients



Vertical gradients

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

Sobel Edge Detection

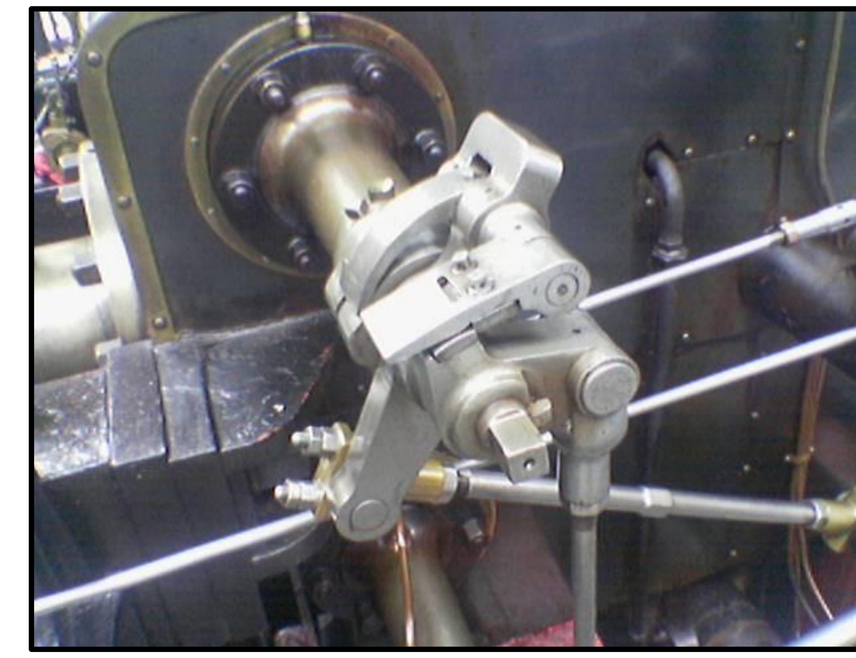
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

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

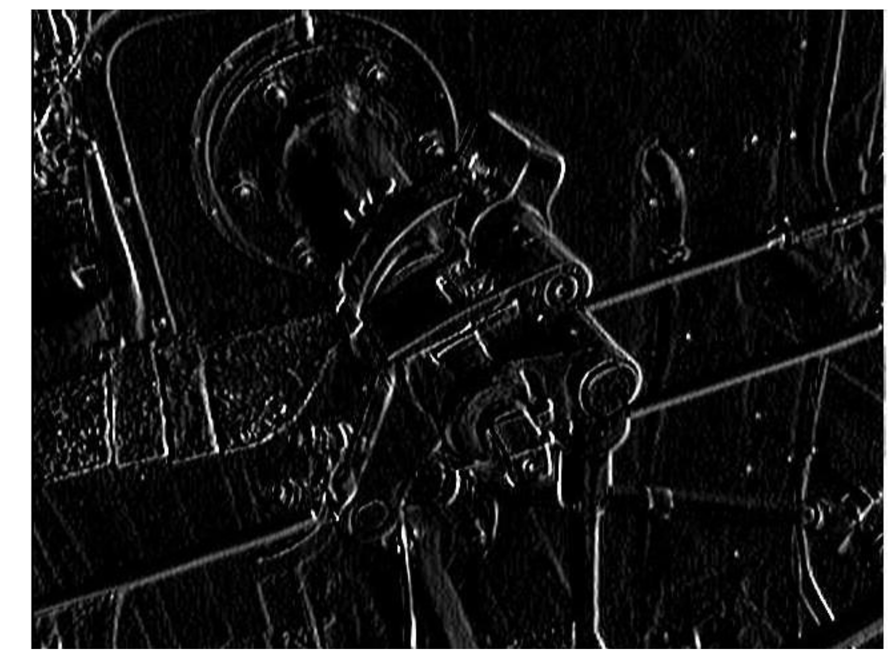
- **Find pixels with large gradients**

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

Pixel-wise operation on images

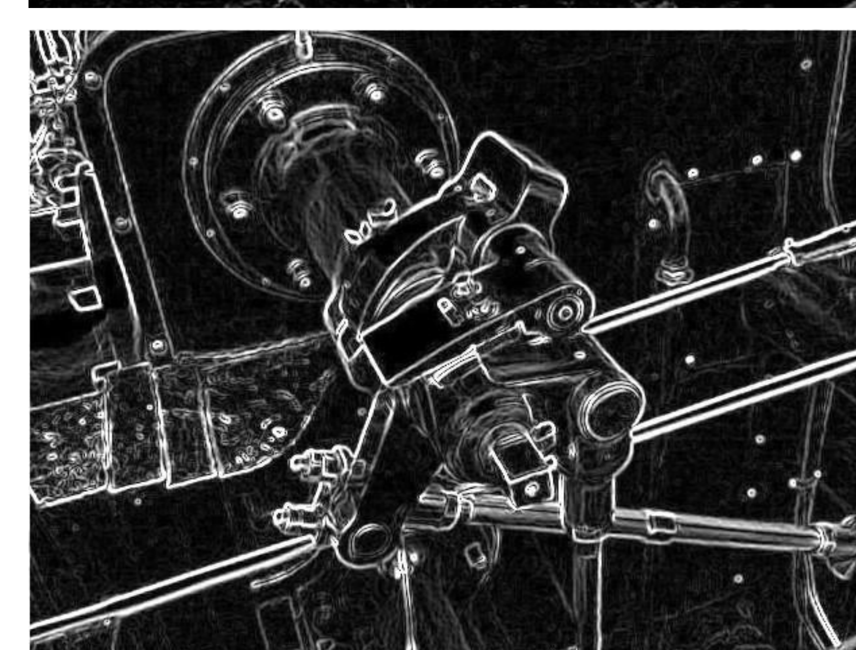


G_x



G_y

G



Tunis

Algorithmic Cost of Convolution Based Image Processing

Cost of Convolution with N x N Filter?

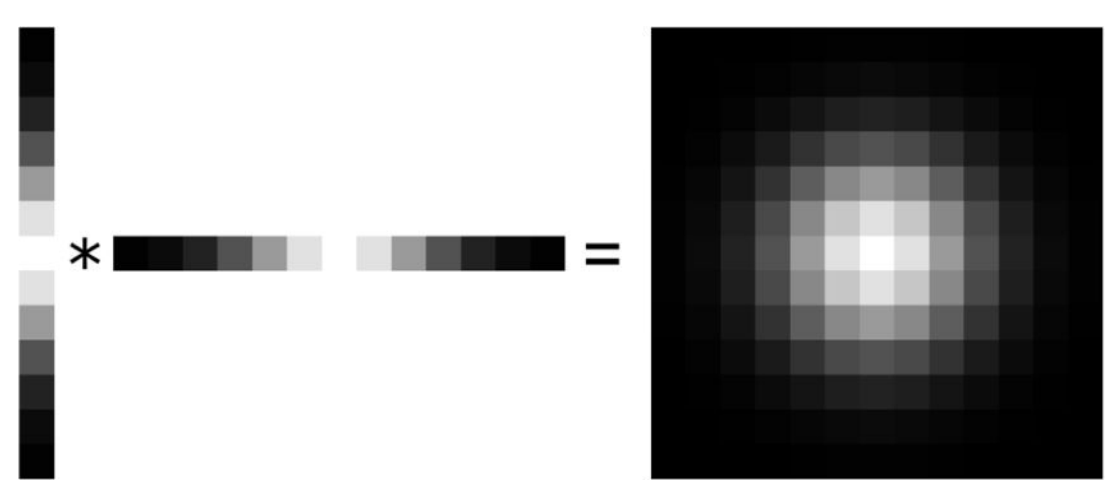
```
float  input[(WIDTH+2) * (HEIGHT+2)];
float  output[WIDTH * HEIGHT];


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(intjj=0;    jj<3; jj++)
        for(intii=0;  ii<3; ii++)
          tmp  += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj * 3 + ii];
          output[j*WIDTH  + i]= tmp;
  }
}
```

Separable Filters

A filter is separable if it's the product of two other filters

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

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



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

Fast 2D Box Blur via Two 1D Convolutions

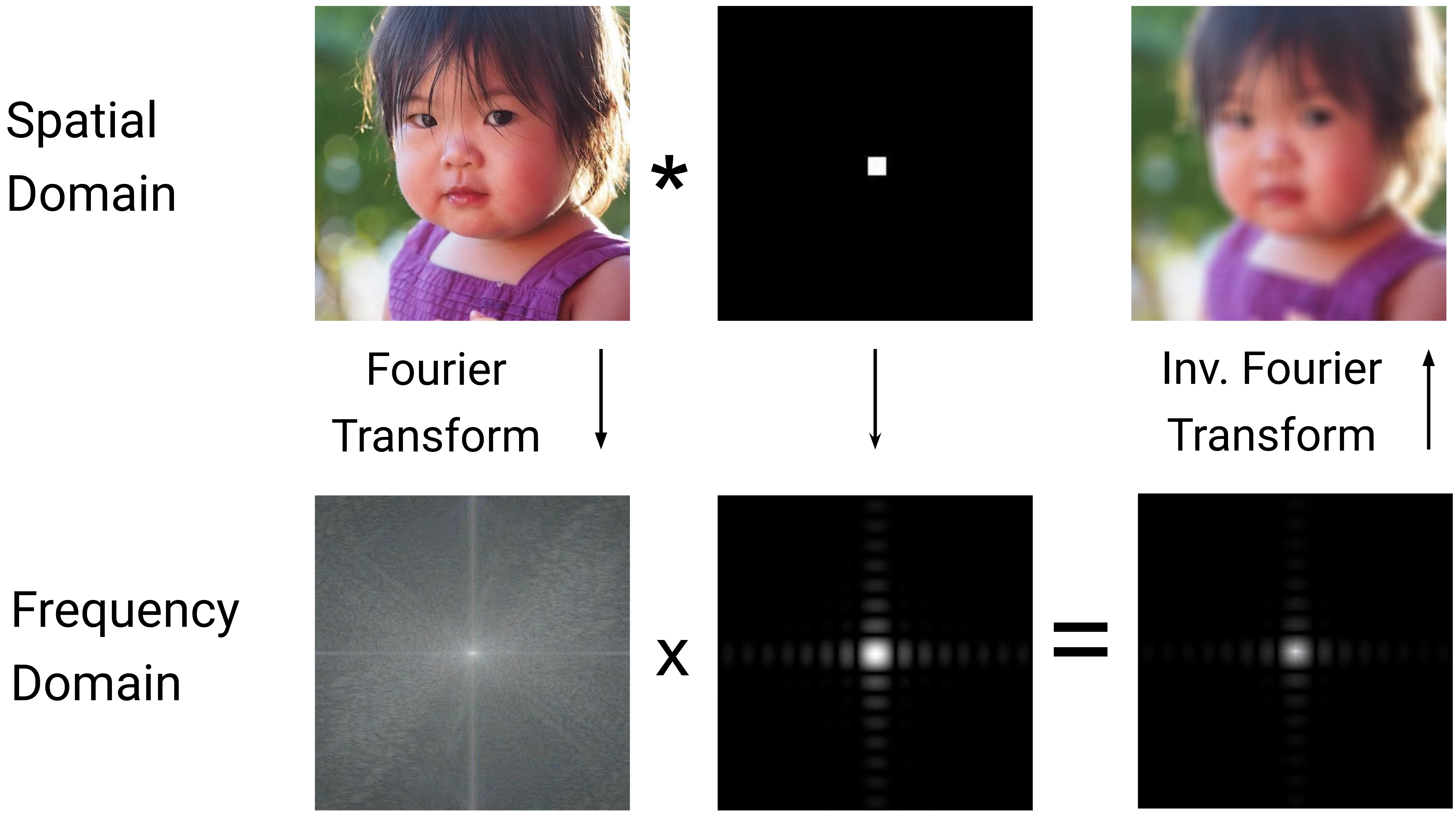
```
int WIDTH = 1024
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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 = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

Extra cost of this approach?

Recall: Convolution Theorem



Efficiency?

When is it faster to implement a filter by convolution in the **spatial domain**?

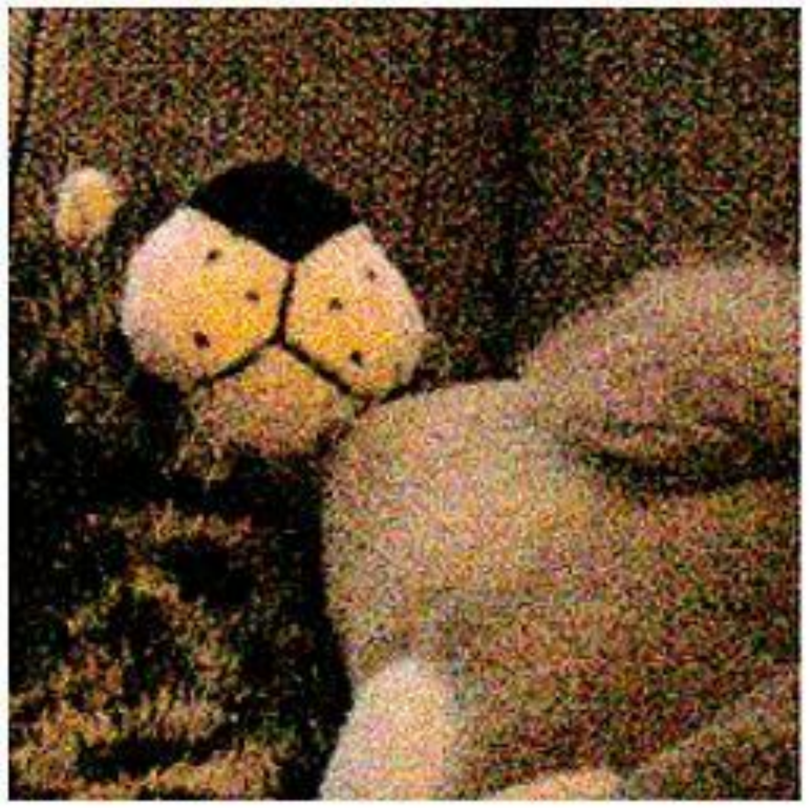
When is it faster to implement a filter by multiplication in the **frequency domain**?

Data-Dependent Filters

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

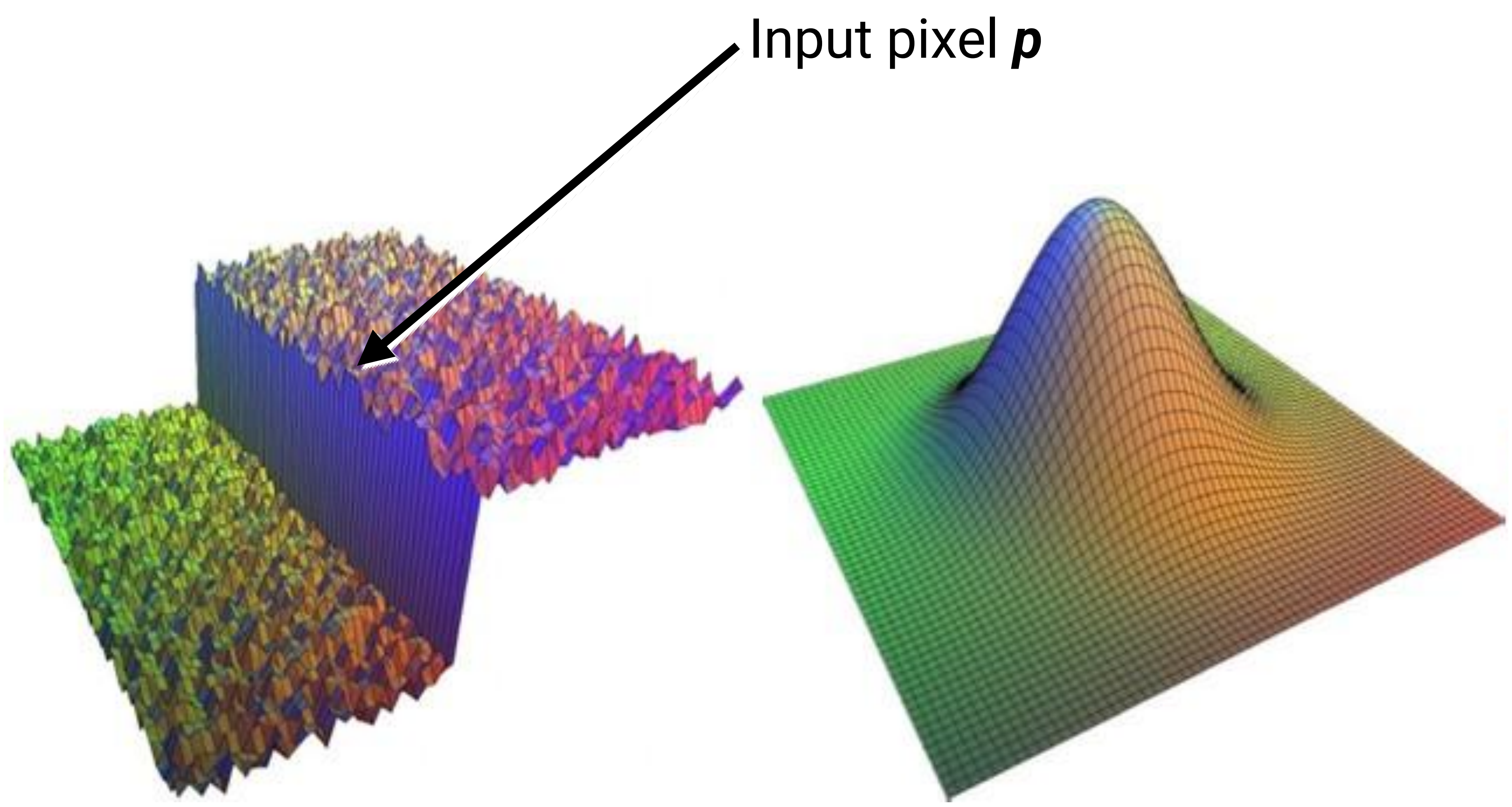
Bilateral Filter

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

Bilateral Filter

$$\mathrm{BF}[I](x,y) = \underbrace{\sum_{i,j}}_{\substack{\text{For all pixels in support}\\ \text{region of Gaussian kernel}}} f(\| \underbrace{I(x-i,y-j) - I(x,y)}_{\substack{\text{Re-weight based on difference}\\ \text{in input image pixel values}}} \|) \overbrace{G(i,j)}^{\text{Gaussian blur kernel}} \overbrace{I(x-i,y-j)}^{\text{Input image}}$$

Bilateral Filter



Input image

G(): gaussian about input pixel *p*

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

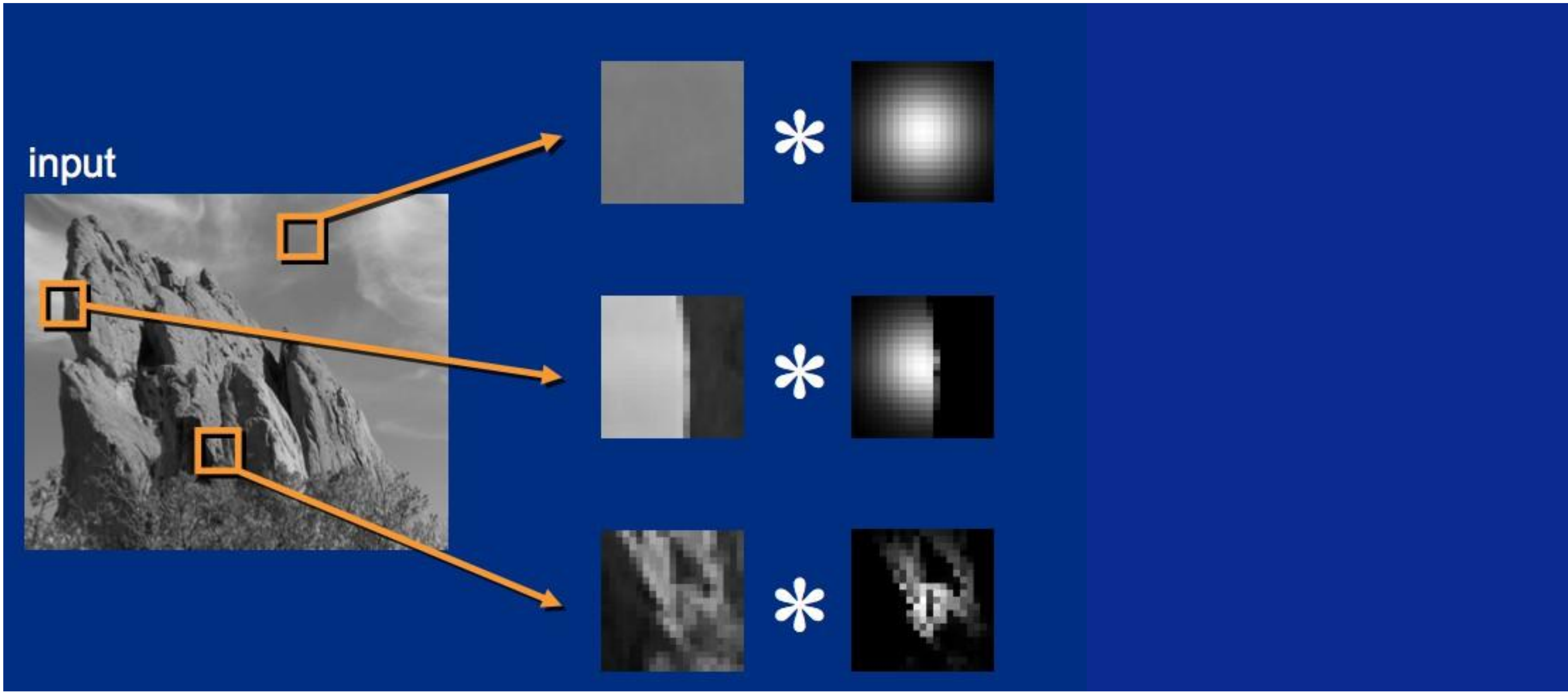


Figure credit: SIGGRAPH 2008 Course: “A Gentle Introduction to Bilateral Filtering and its Applications” Paris et al.

Data-Driven Image Processing:

Image Manipulation by Example

Texture Synthesis

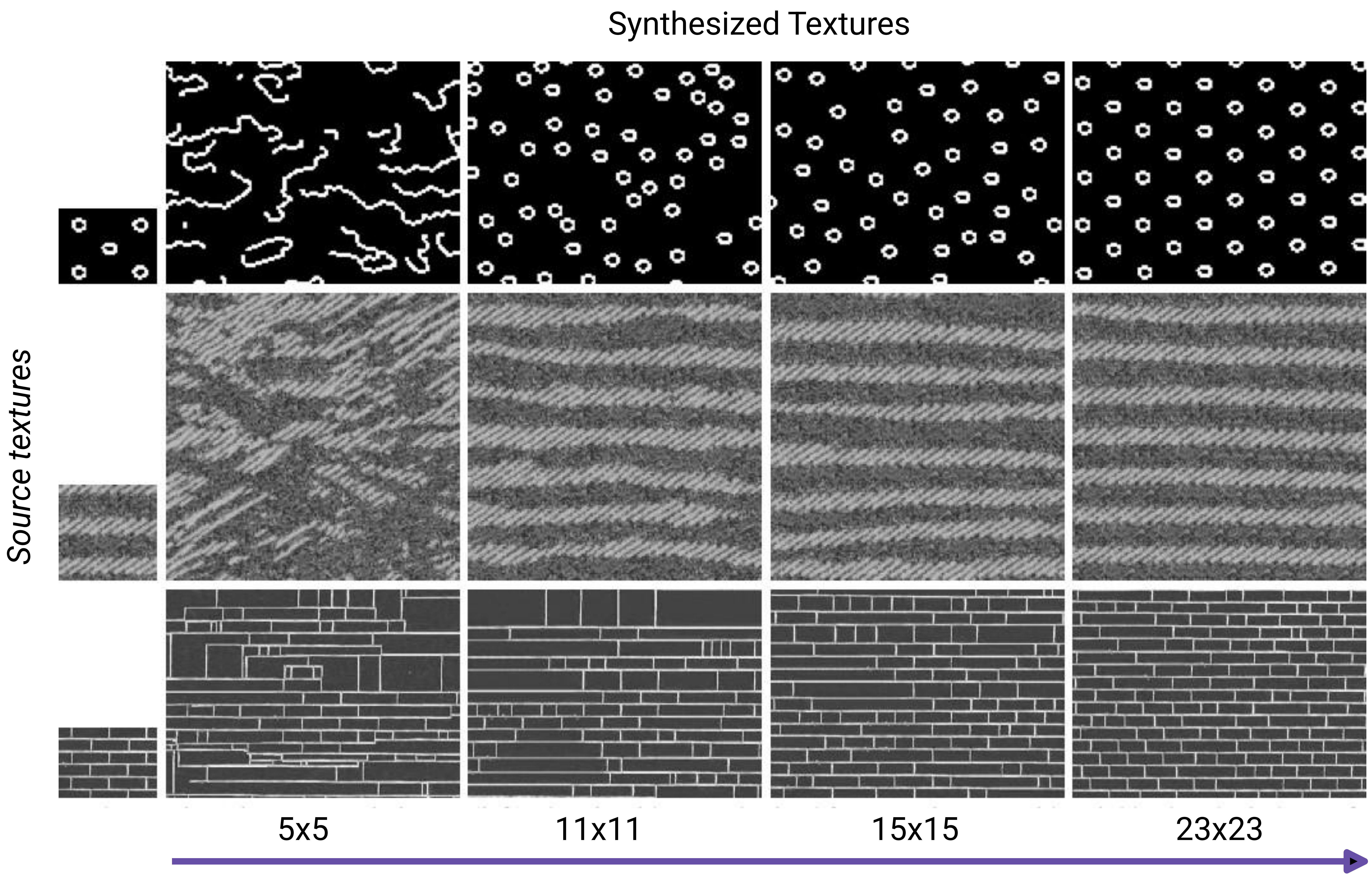
Input: low-resolution texture image

Desired output: high-resolution texture that appears “like” the input

Algorithm: Non-Parametric Texture Synthesis

Main idea: For a given pixel p, find a probability distribution function for possible values of p, based on its neighboring pixels.

Non-Parametric Texture Synthesis



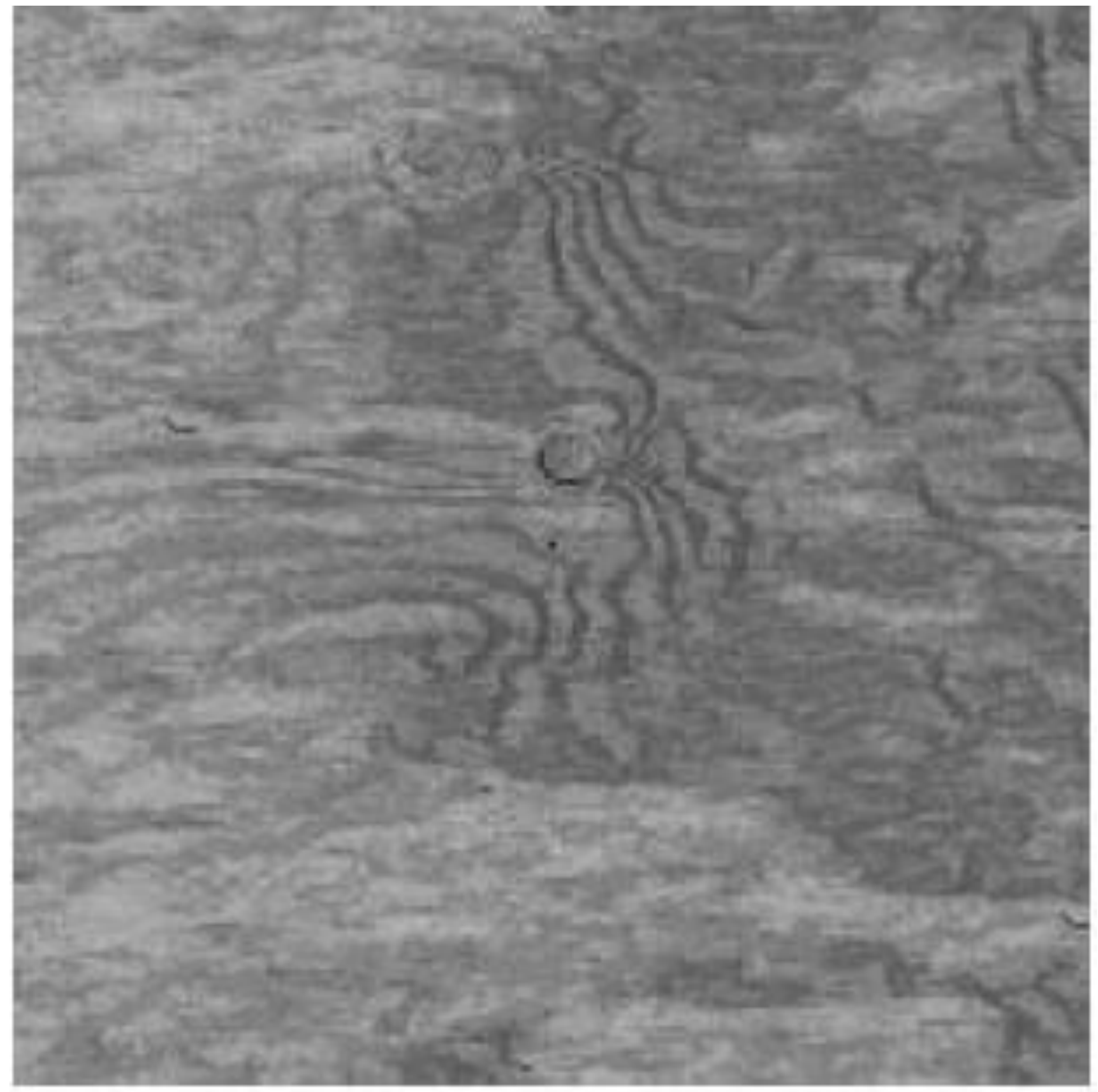
➢ Increasing size of neighborhood search window: $w(p)$

[Efros and Leung 99]

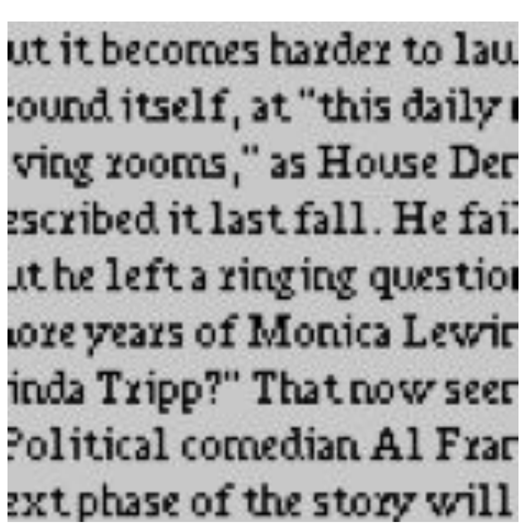
Allows for Better Texture Synthesis

Synthesized Textures

Source Textures



Source Textures

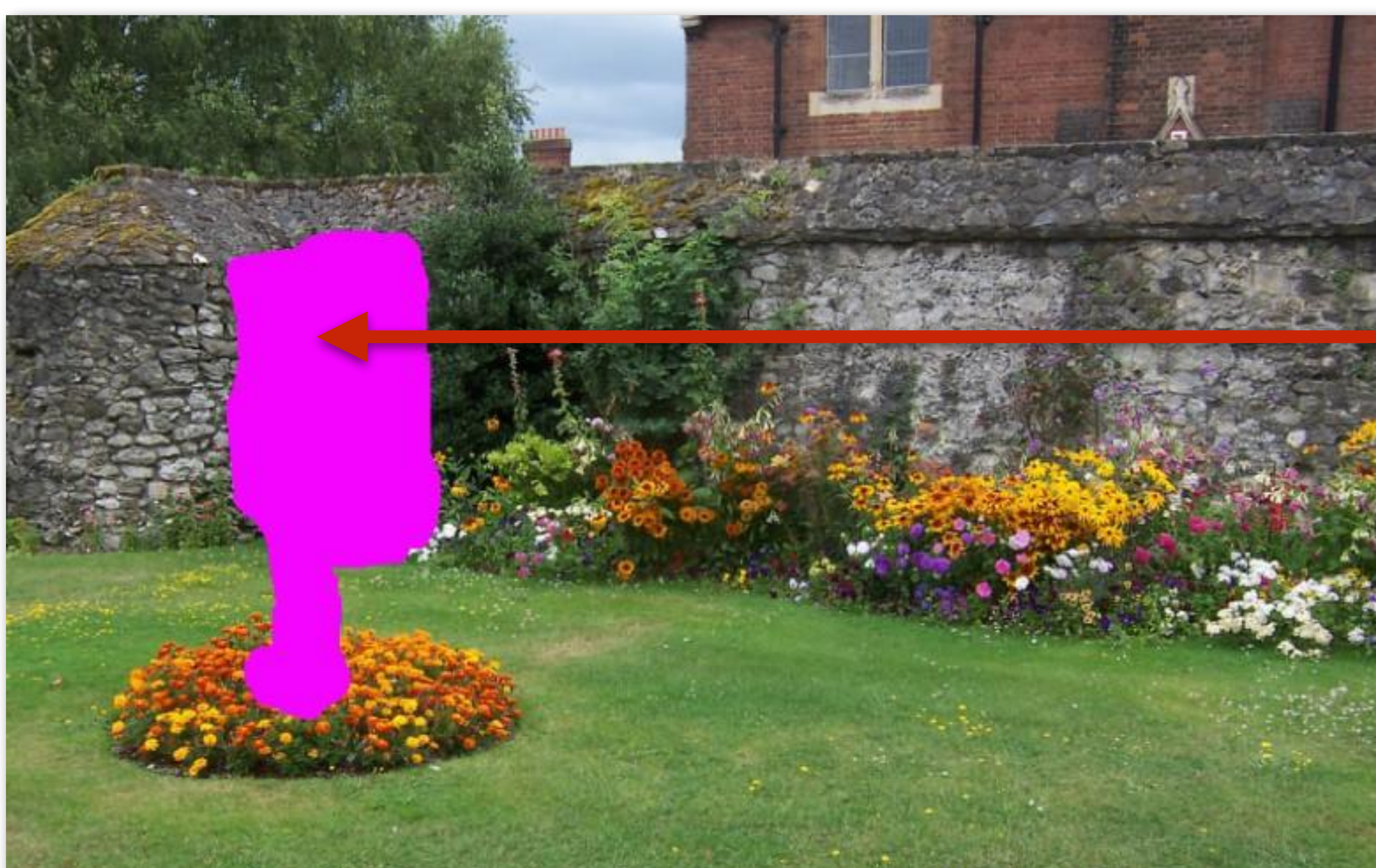


[Efros and Leung 99]

Image Completion Example

Original Image

Completion Result



Masked Region

Goal: fill in masked region with "plausible" pixel values.

See **PatchMatch** algorithm [Barnes 2009] for a fast randomized algorithm for finding similar patches.

Image credit: [Barnes et al. 2009]

Things to Remember

- JPEG as an example of exploiting perception in visual systems
- Chroma subsampling and DCT Image processing via convolution
- Vary operations by changing filter kernel weights
- Fast separable filter implementation: multiple 1D filters
- Data-dependent image processing techniques
    - Bilateral filtering, Efros-Leung texture synthesis

Acknowledgments

Many thanks to Kayvon Fatahalian for this lecture!