Lecture 24:

Image Processing

Computer Graphics
CMU 15-462/15-662, Fall 2015

Warm up:
Putting many recent concepts together:
JPEG Compression

JPEG compression: the big ideas

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

Therefore, it's often acceptable for a compression scheme to introduce errors in high-frequency components of the image.

- The human visual system is:
  - less sensitive to high frequency sources of error
  - less sensitive to detail in chromaticity than in luminance

[Credit: Pat Hanrahan]

JPEG: color space conversion and chroma subsampling

- Convert image to Y'CbCr color representation
- Subsample chroma channels (e.g., to 4:2:0 format)

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

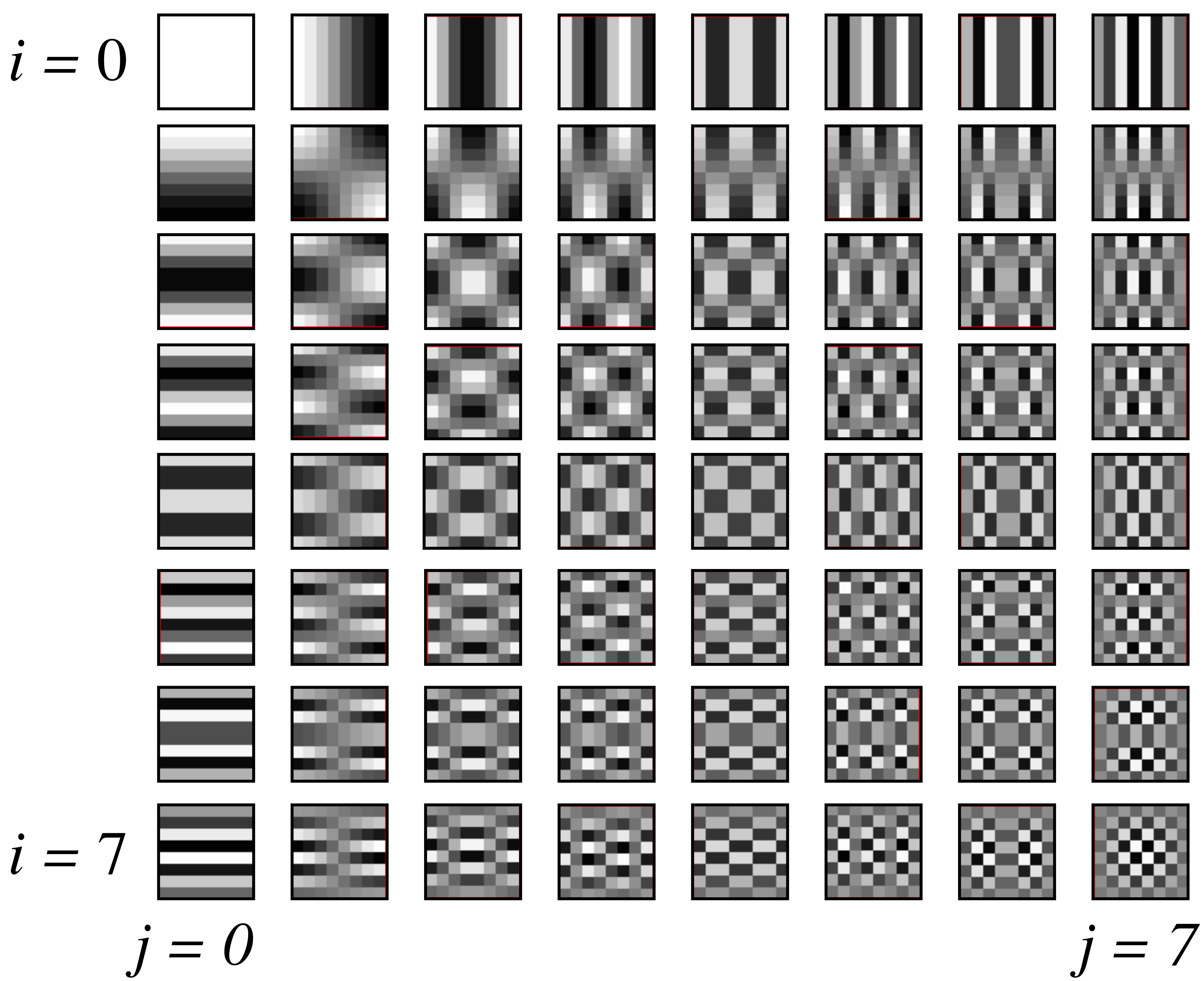
4:2:0 representation:

Store Y' at full resolution

Store Cb, Cr at half resolution in both dimensions

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

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



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

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

Quantization

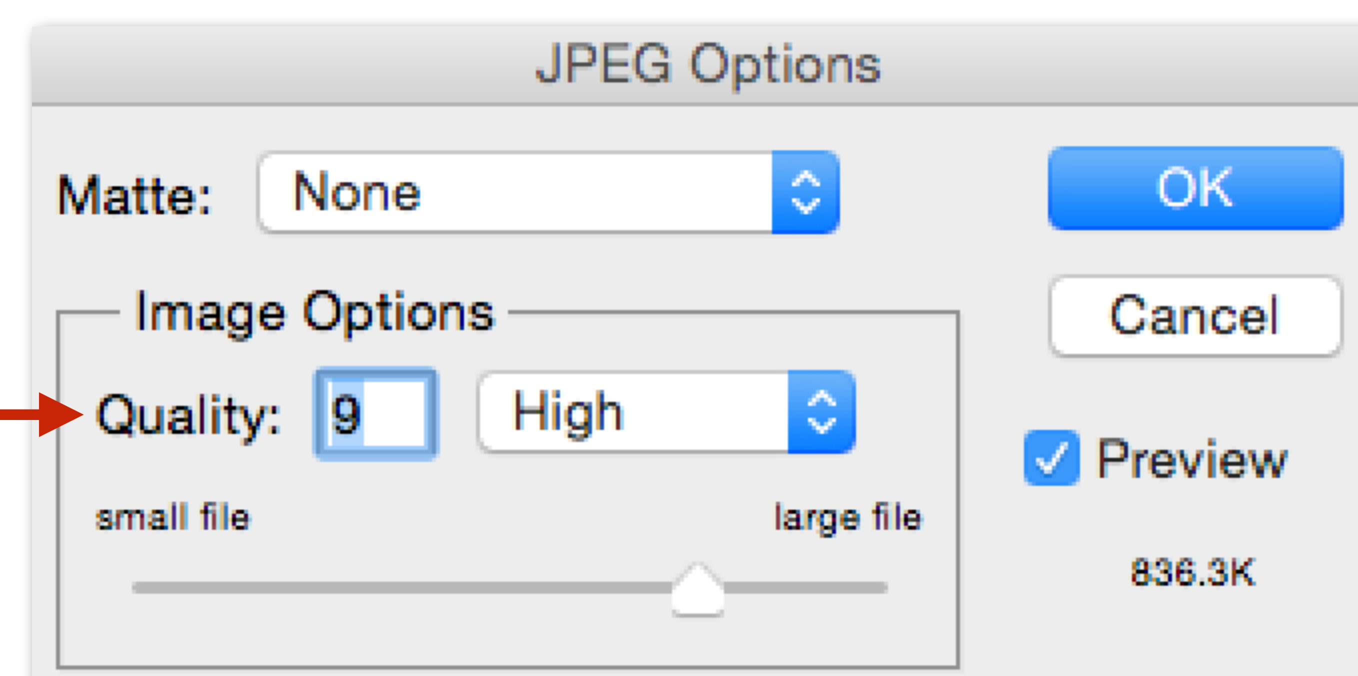
$$
\begin{bmatrix}
-415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\
4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\
-47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\
-49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\
12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\
-8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\
-1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\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

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)

Notice: quantization zeros out many coefficients

[Credit: Pat Hanrahan]

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 6

Quality Level 3

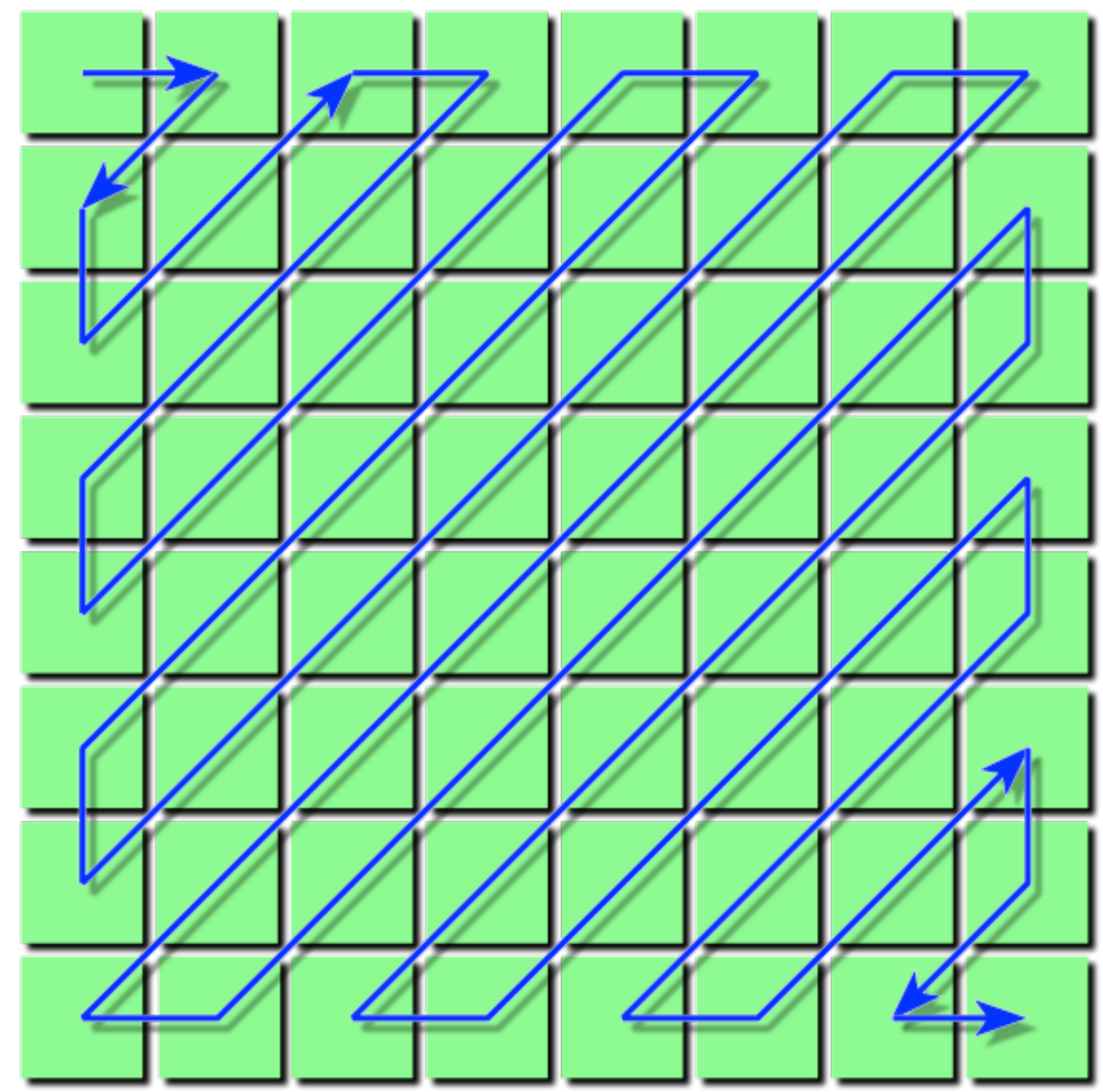
Quality Level 1

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

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

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

- Separation of luminance from chrominance in color representations (e.g, 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

Basic image processing operations

(This section of the lecture will describe how to implement a number of basic operations on images)

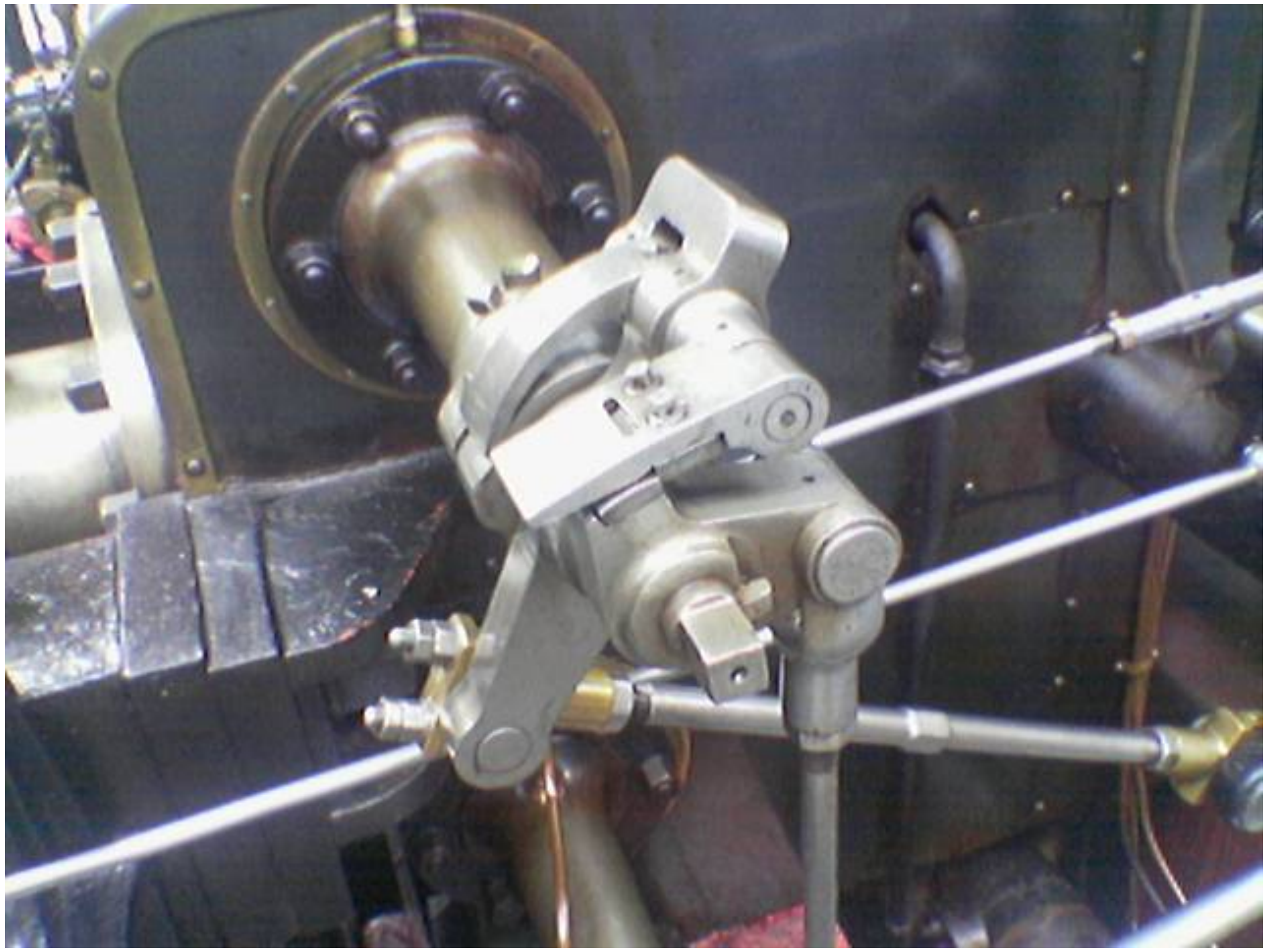
Example image processing operations

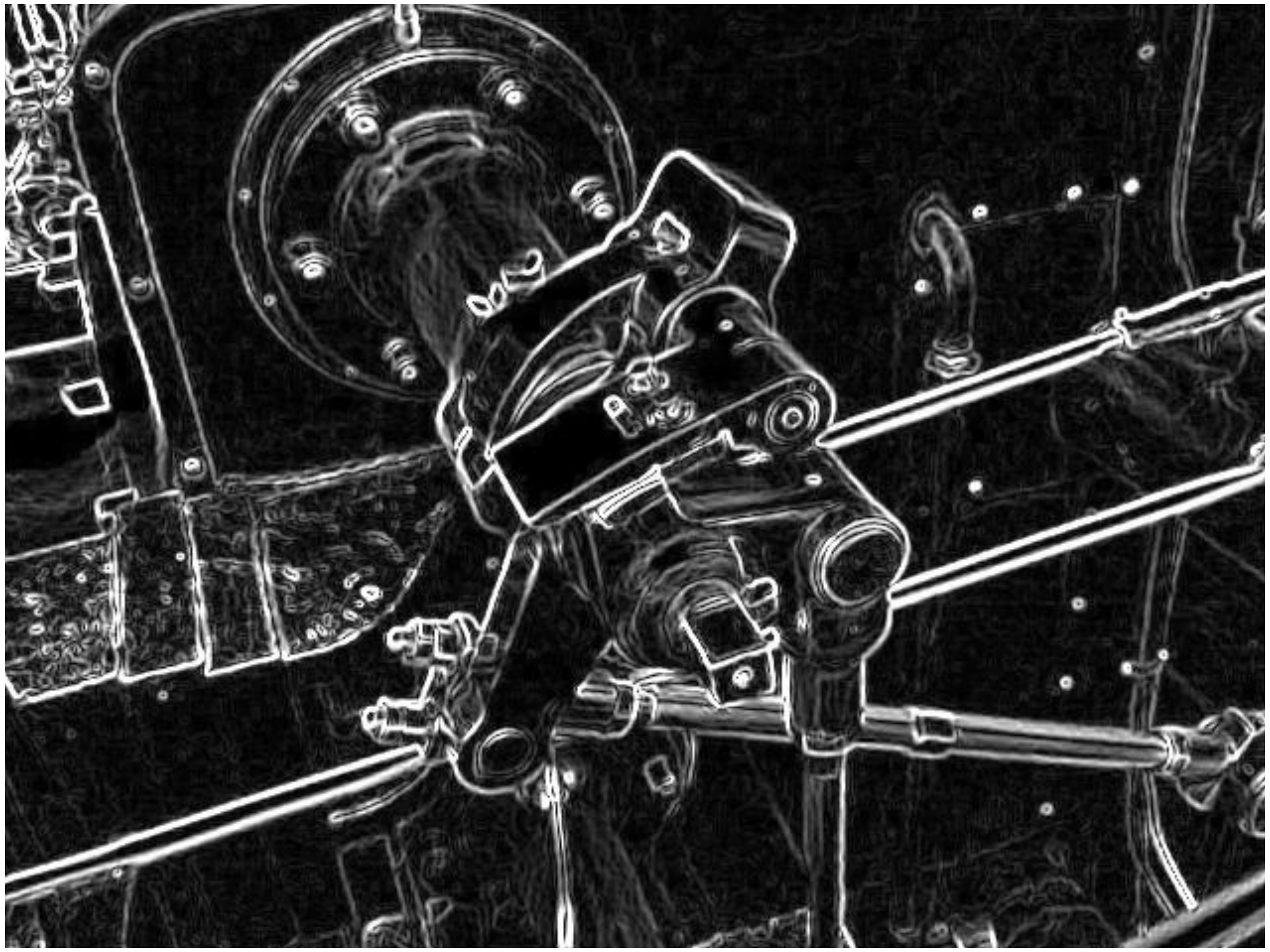
Blur

Example image processing operations

Sharpen

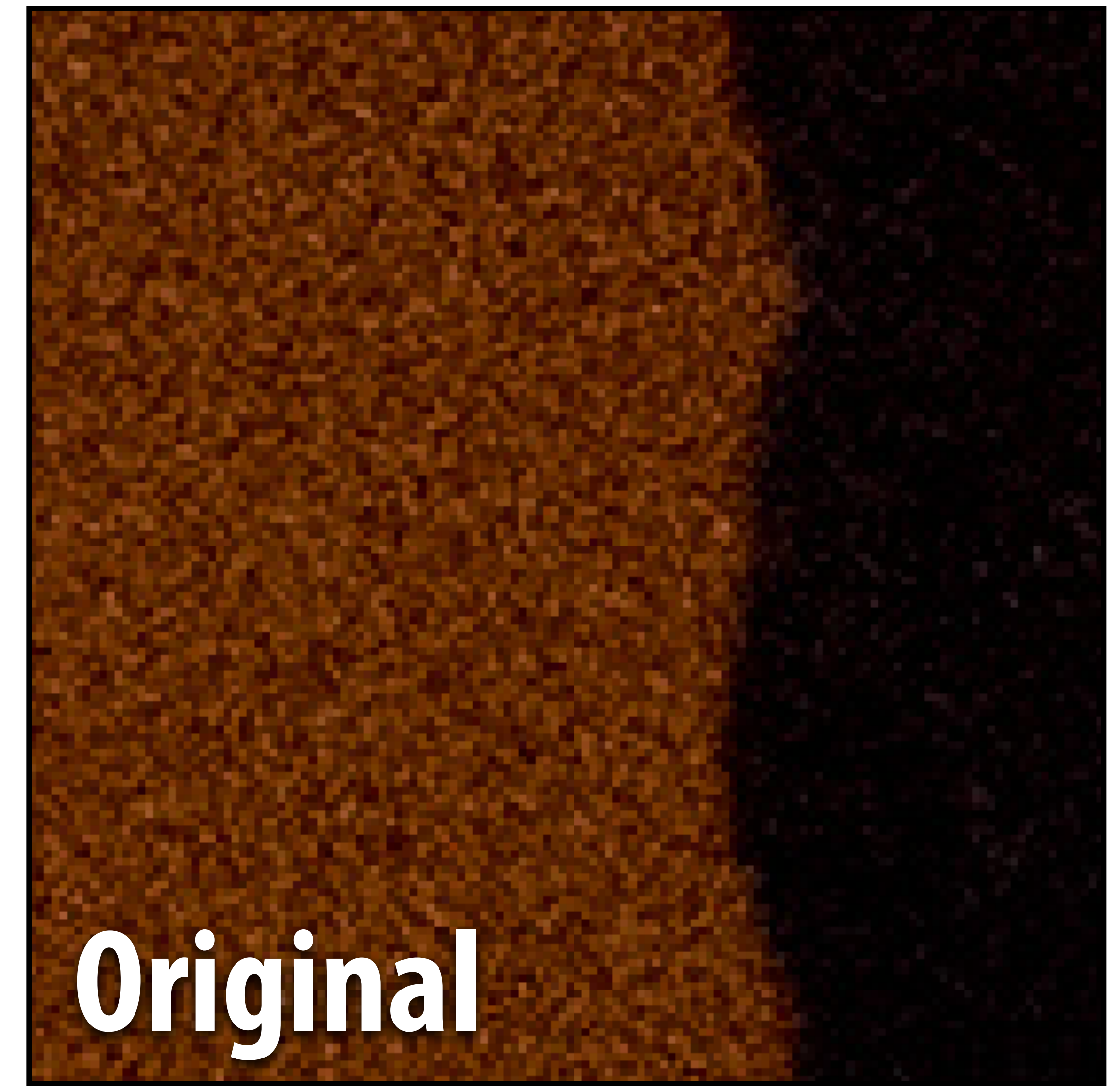
Edge detection

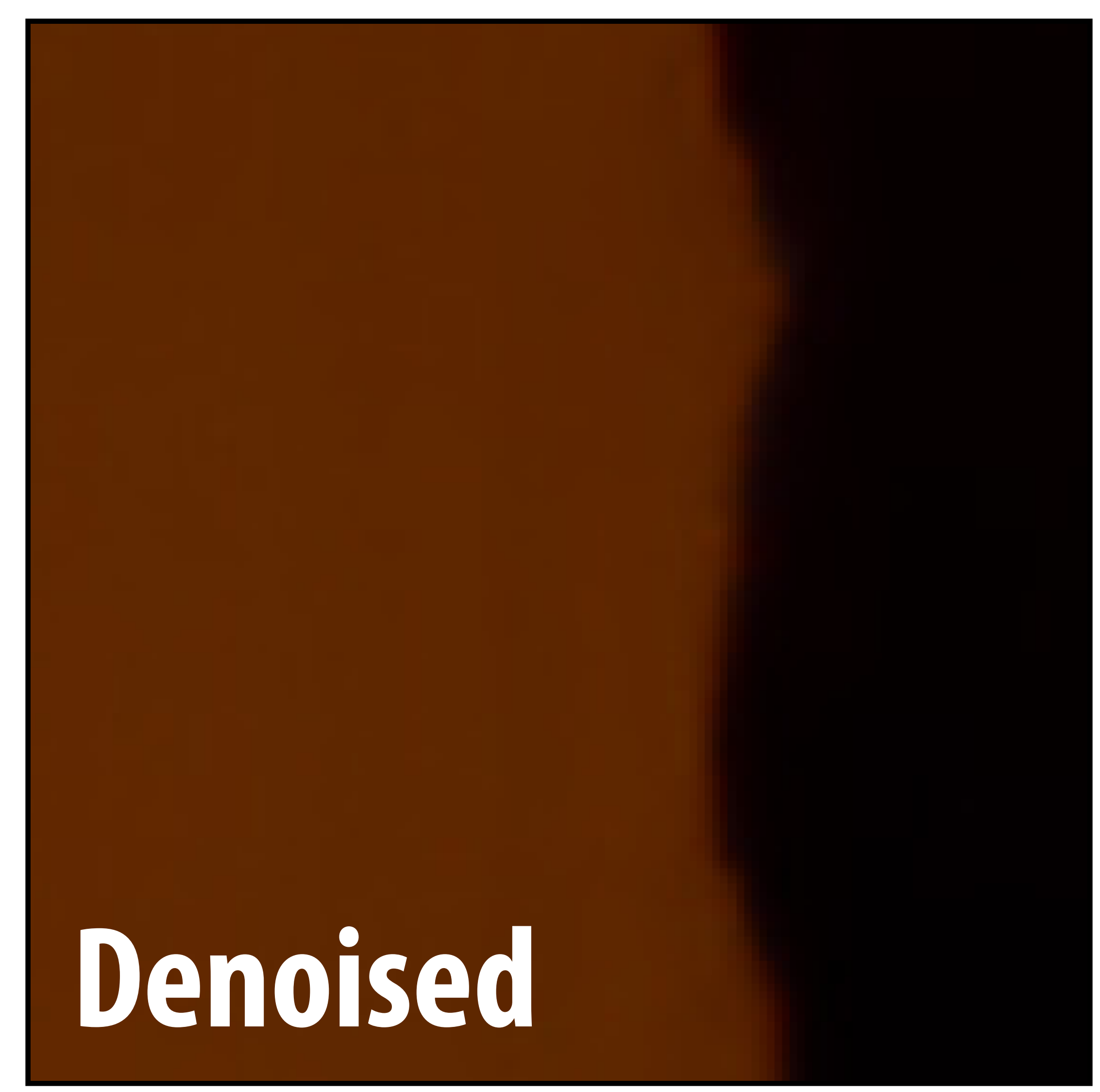




A “smarter” blur (doesn’t blur over edges)

Denoising



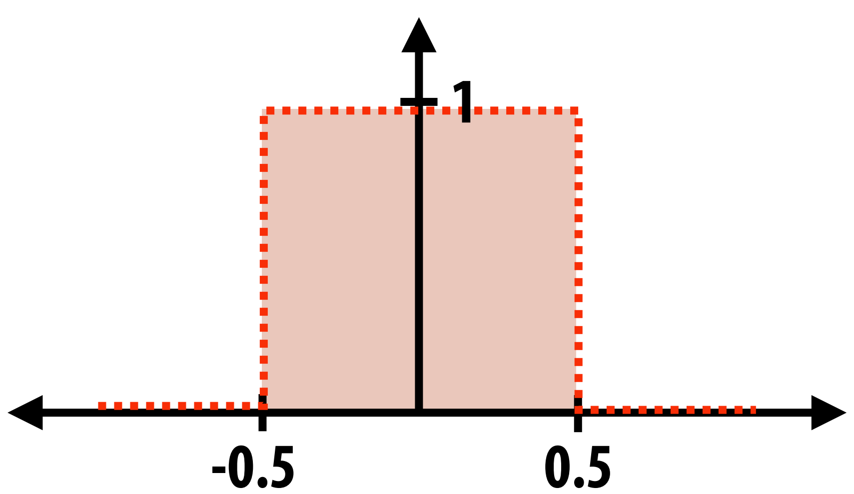


Review: convolution

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

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 "smoothed" version of g

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:

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

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

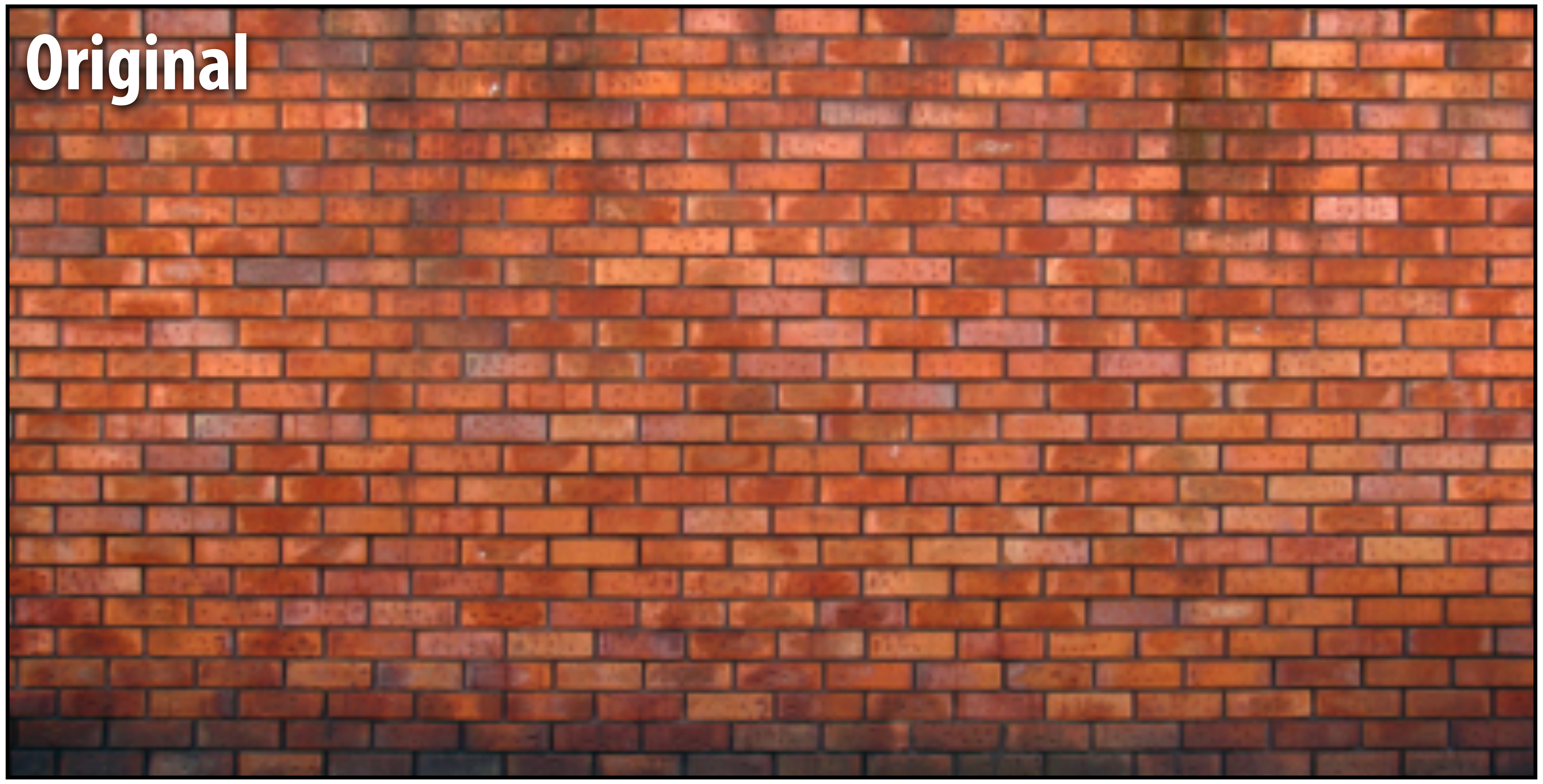
$f(i, j) = \mathbf{F}_{i,j}$ **(often called: "filter weights", "kernel")**

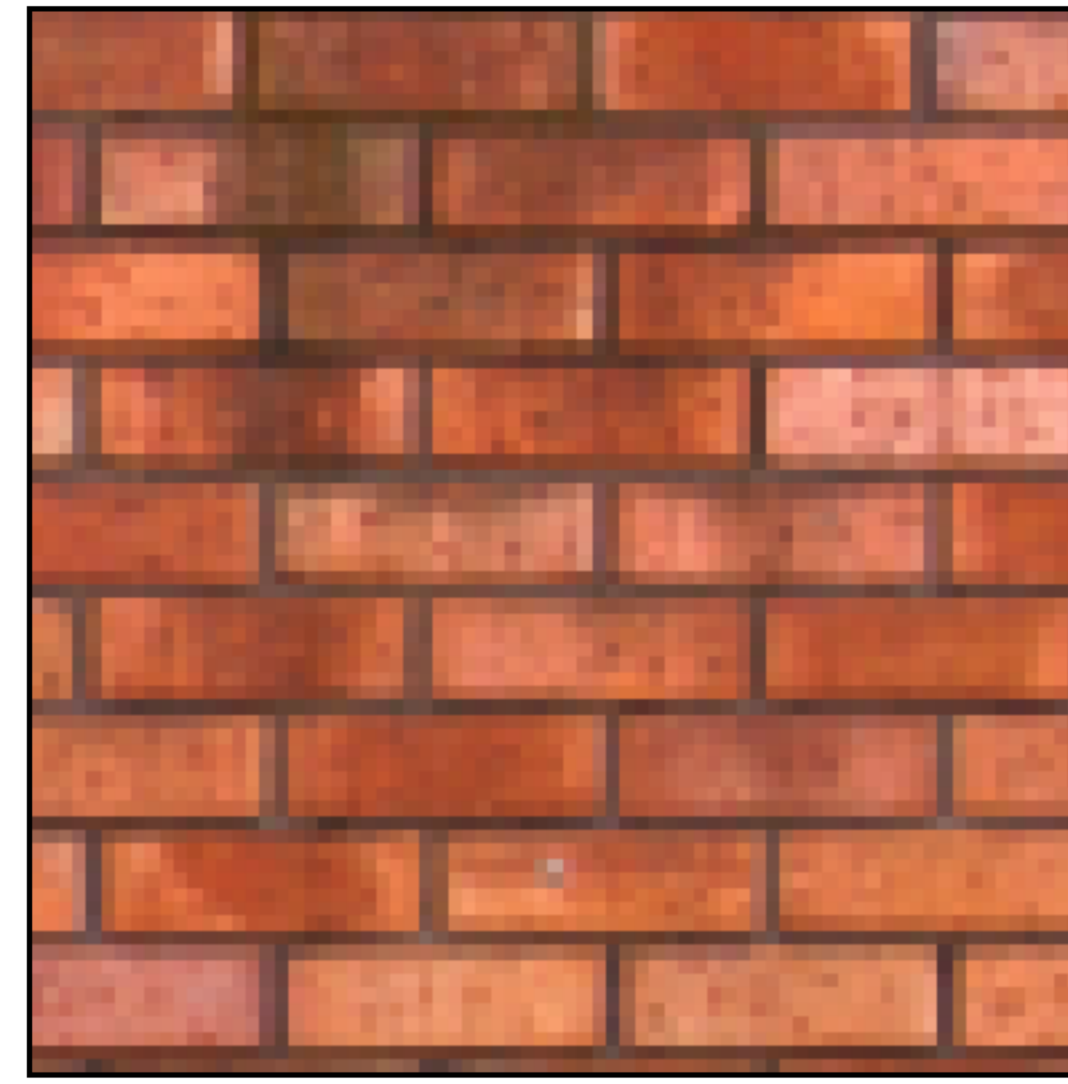
Simple 3x3 box blur

```
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

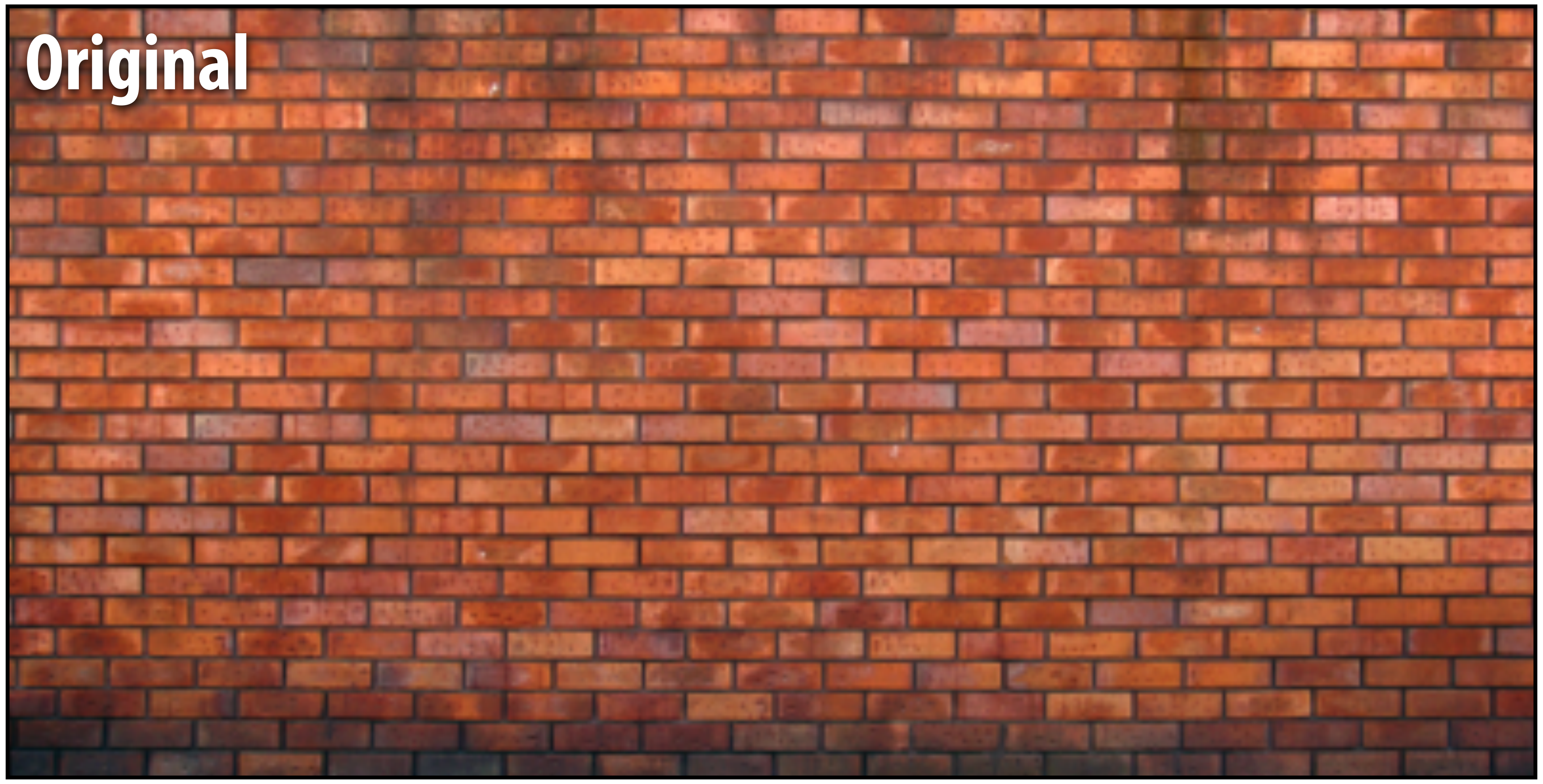
- **Obtain filter coefficients from sampling 2D Gaussian**

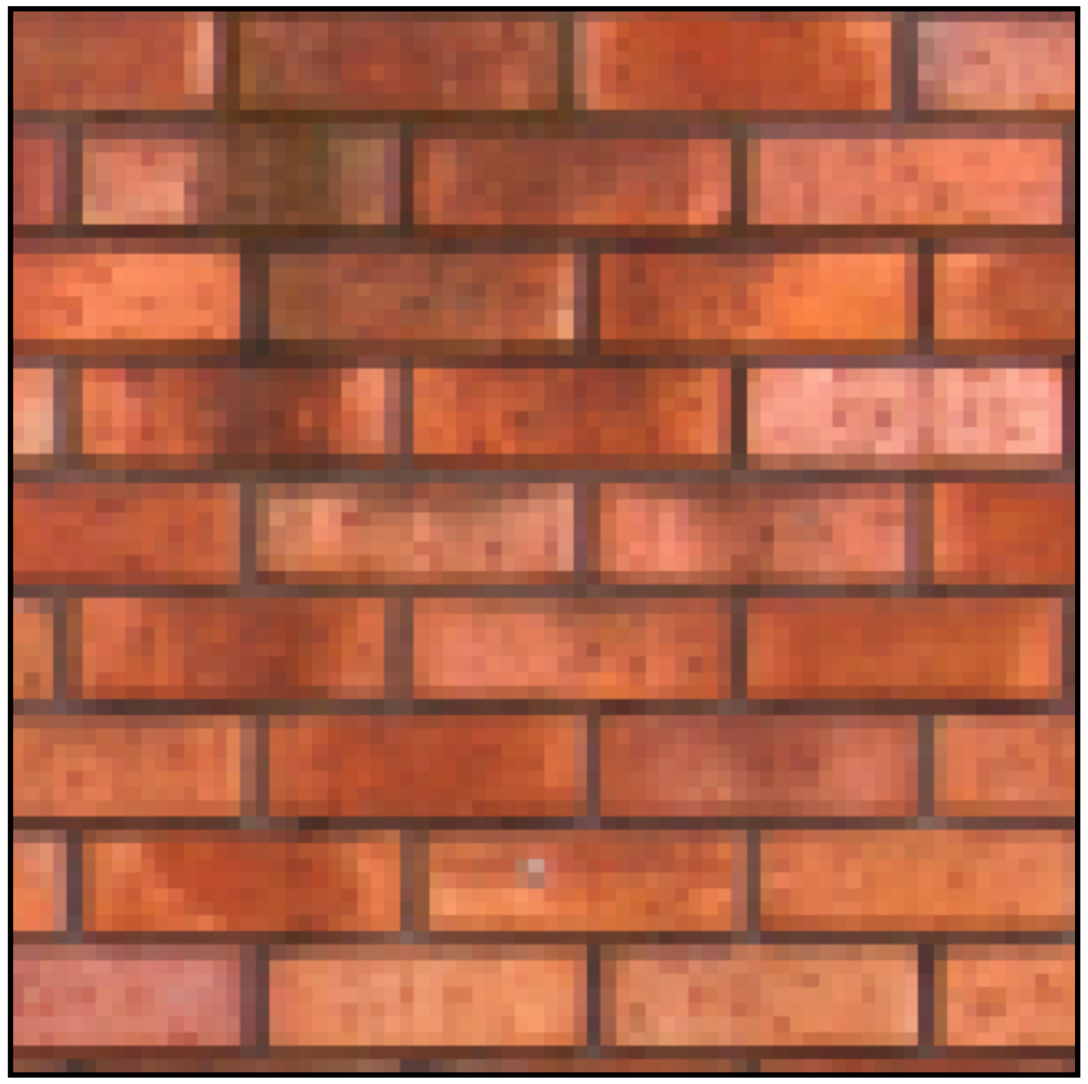
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

- **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



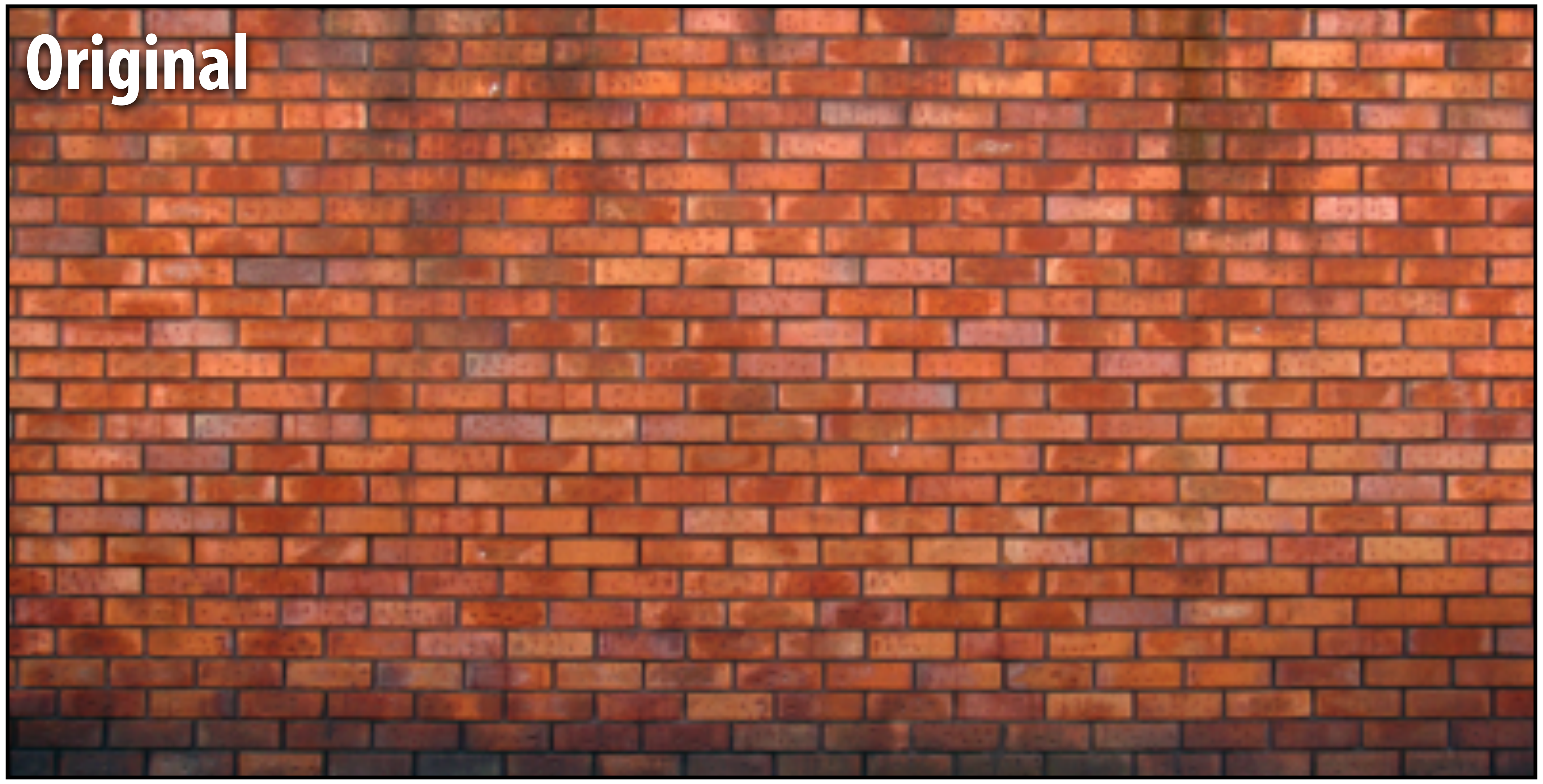


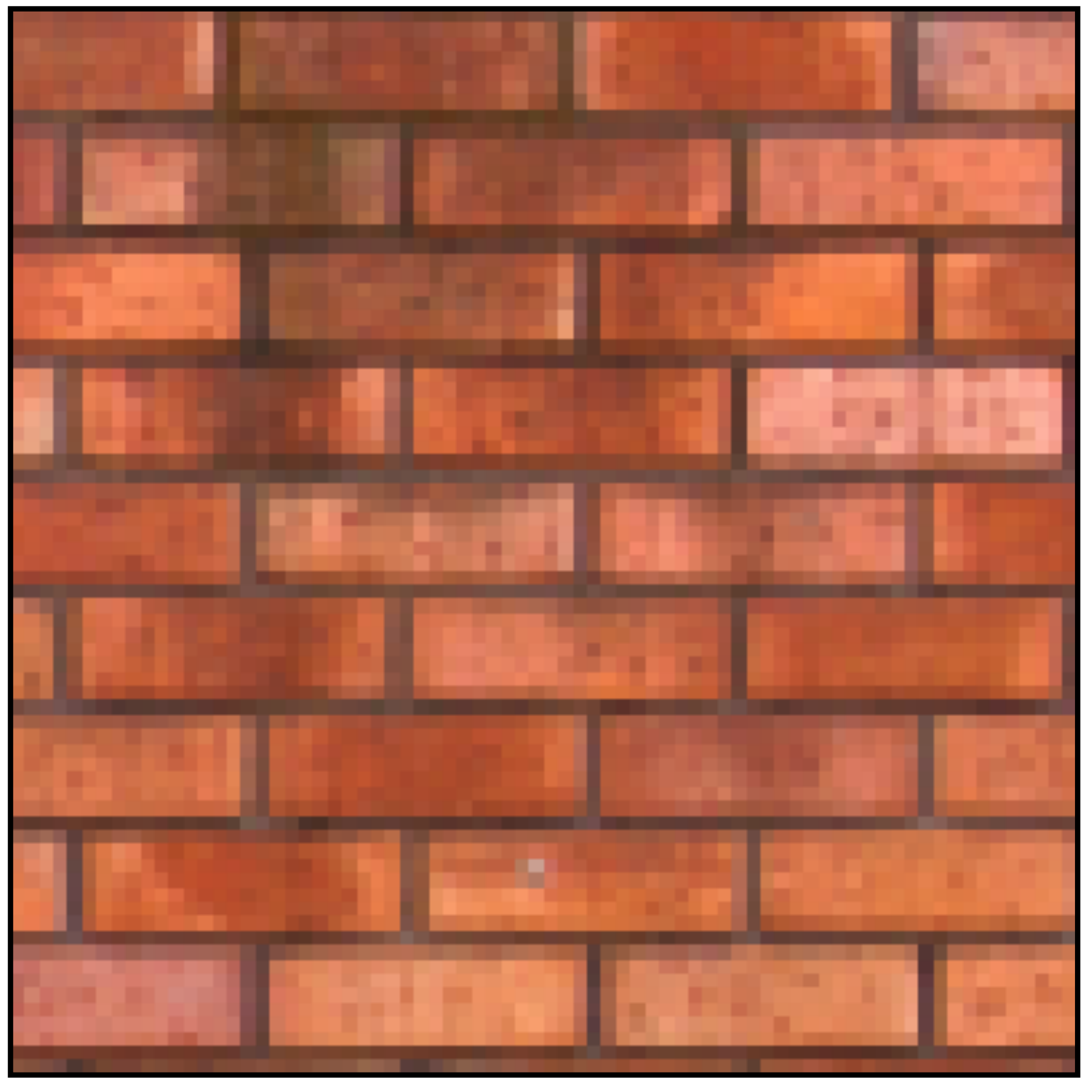
What does convolution with this filter do?

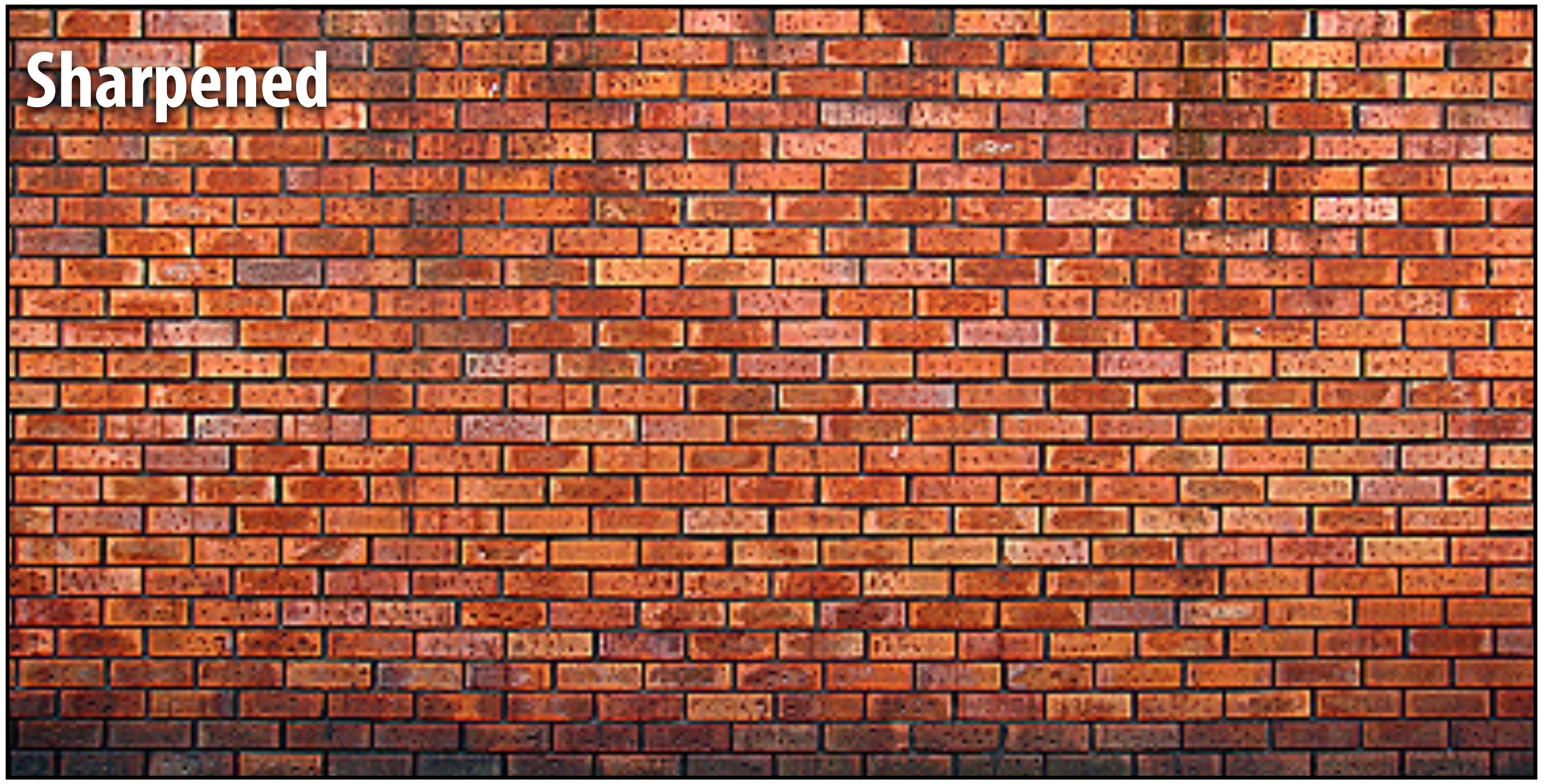
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Sharpens image!

3x3 sharpen filter







What does convolution with these filters do?

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

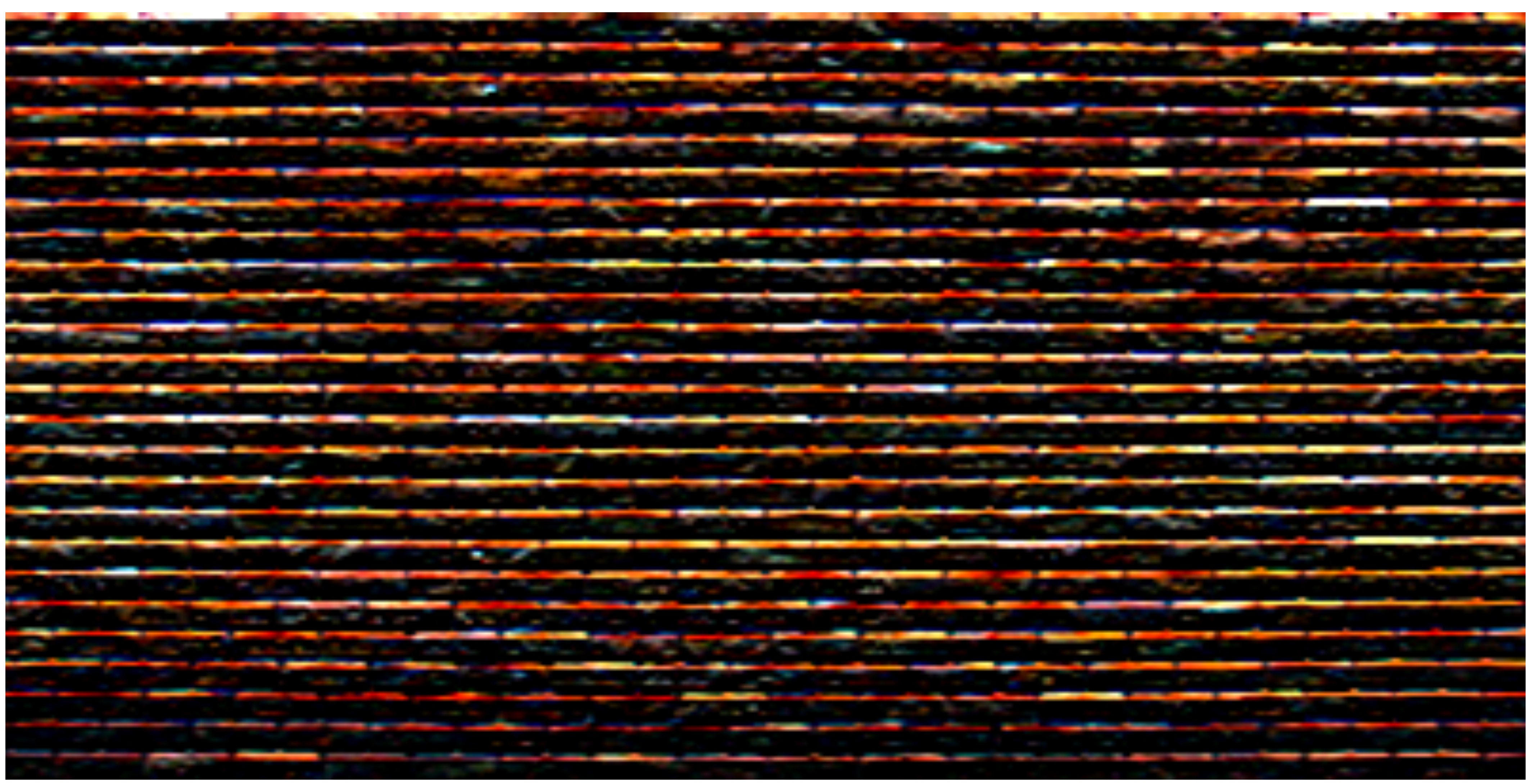
Extracts horizontal gradients

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

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**

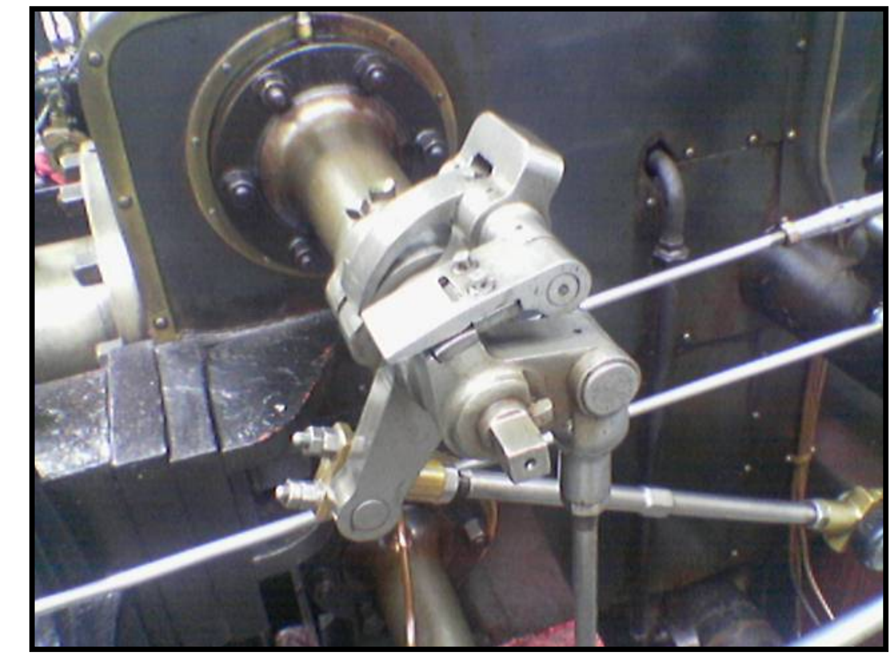
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

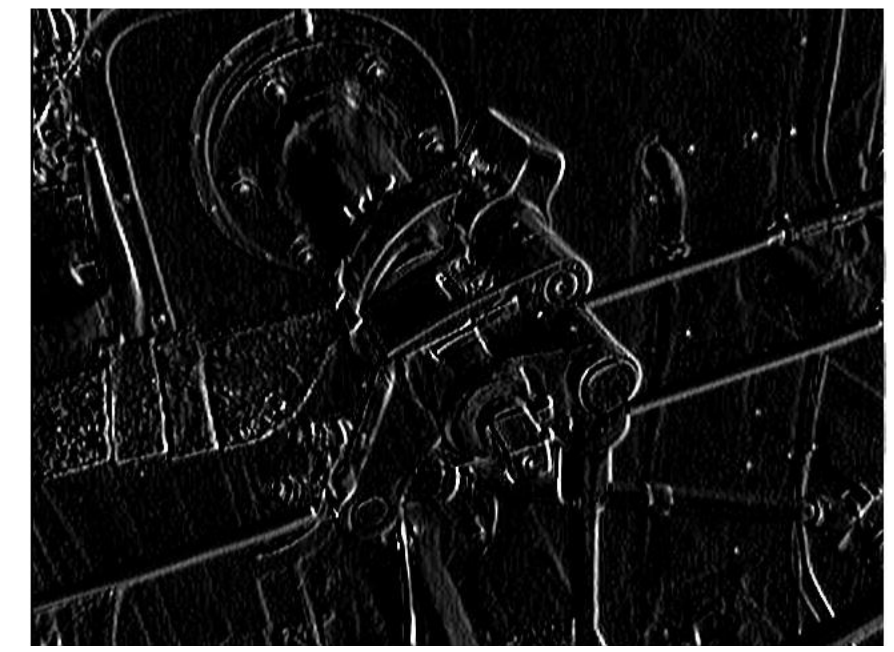
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

- **Find pixels with large gradients**

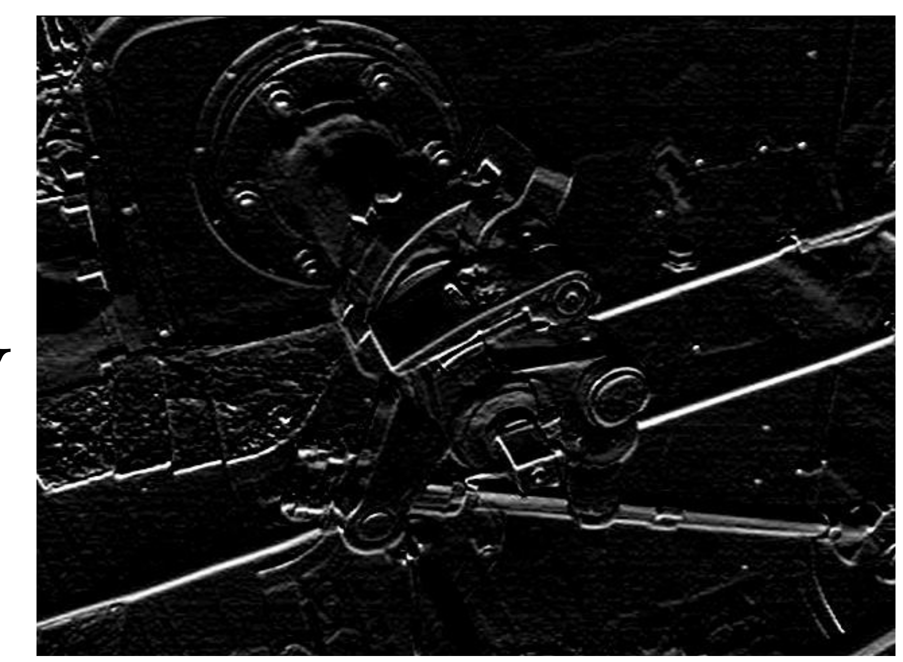
$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

Pixel-wise operation on images

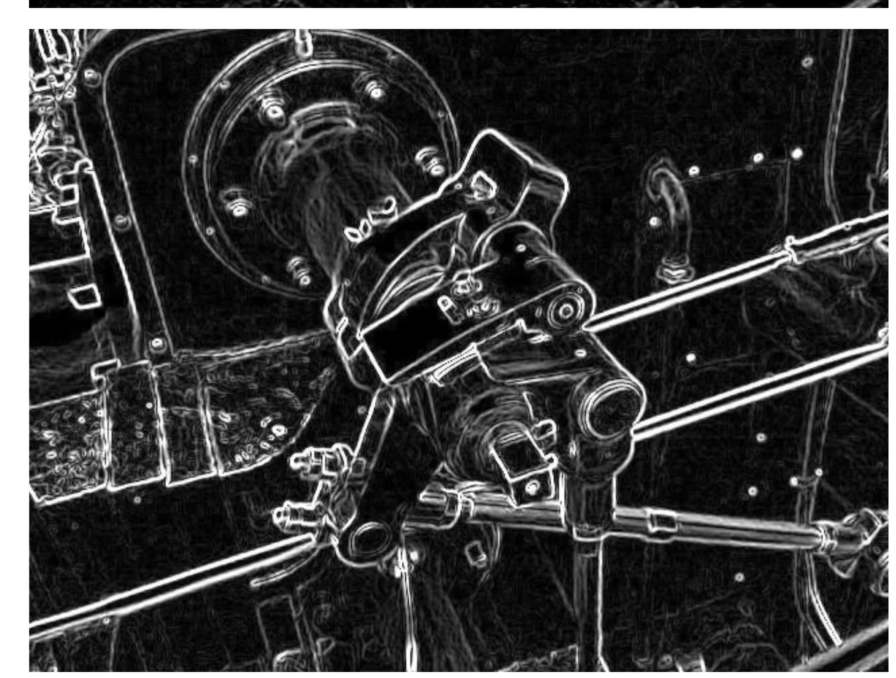




G_x



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 is the 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 = 6 x WIDTH x HEIGHT

For NxN filter: 2N x WIDTH x HEIGHT

Extra cost of this approach?

Storage!

Challenge: can you achieve this work complexity without incurring this cost?

Data-dependent filter (not a convolution)

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

for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      float min_value = min( min(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                             min(input[j*WIDTH + i-1], input[j*WIDTH + i+1]) );
      float max_value = max( max(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                             max(input[j*WIDTH + i-1], input[j*WIDTH + i+1]) );
      output[j*WIDTH + i] = clamp(min_value, max_value, input[j*WIDTH + i]);
    }
}
```

This filter clamps pixels to the min/max of its cardinal neighbors (e.g., hot-pixel suppression)

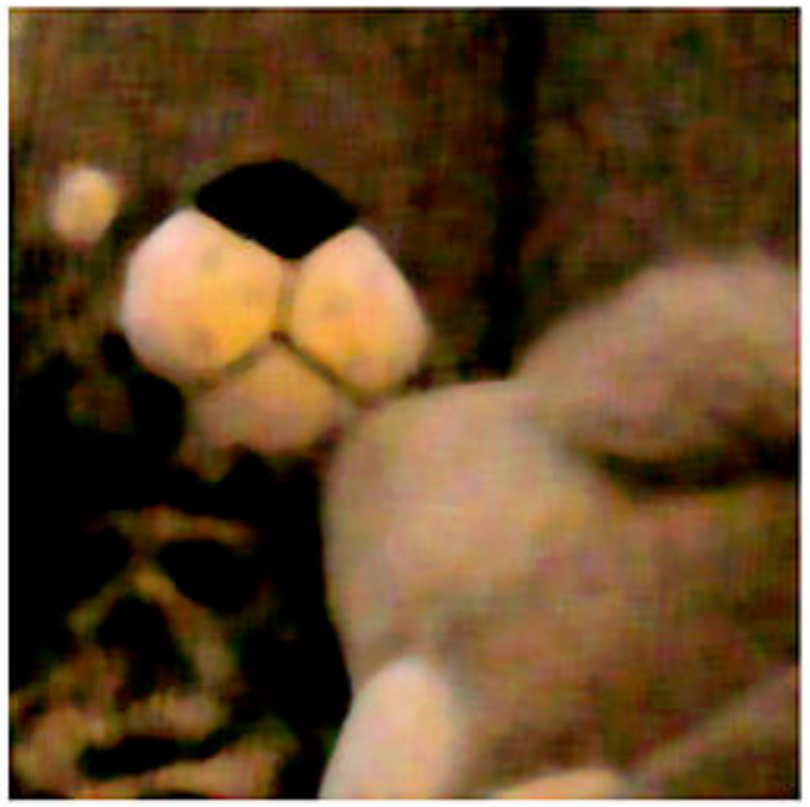
Median filter

- **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, 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

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

For all pixels in support region of Gaussian kernel

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

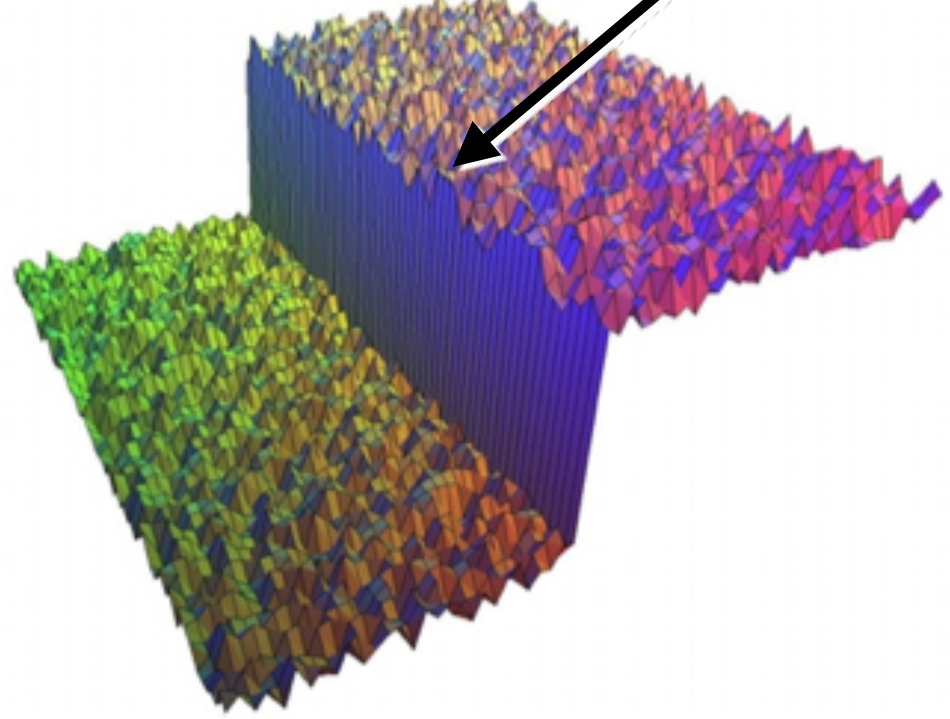
But weight is combination of <u>spatial distance</u> and <u>input image pixel intensity</u> difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

- **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* $x >$ *threshold*, 1 *otherwise*

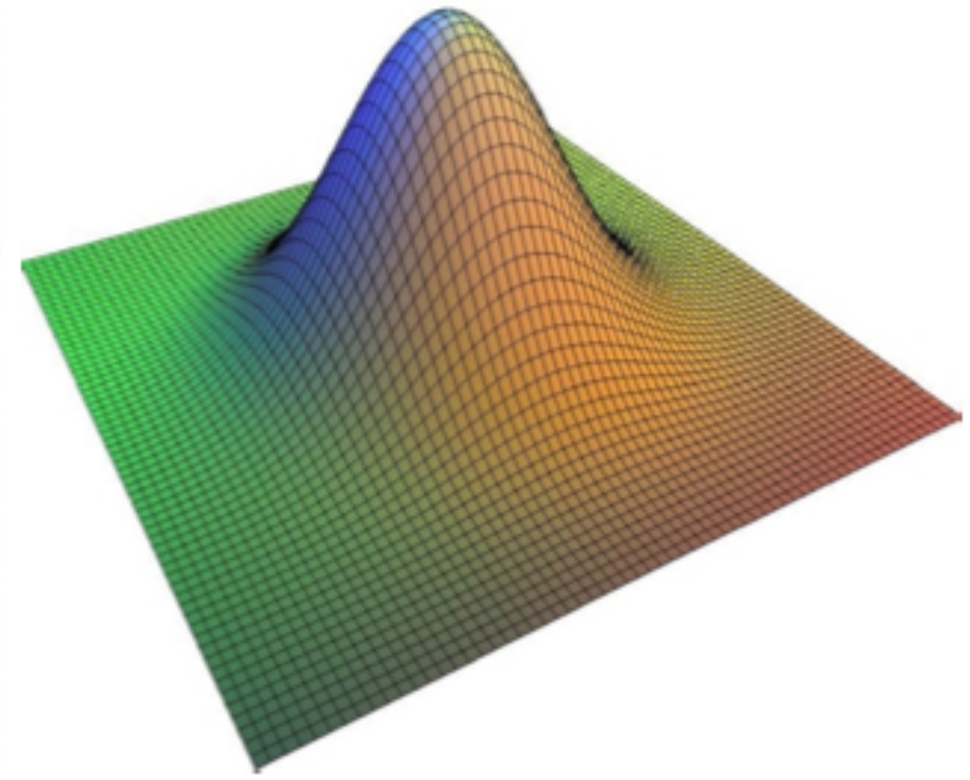
Bilateral filter

Input pixel *p*

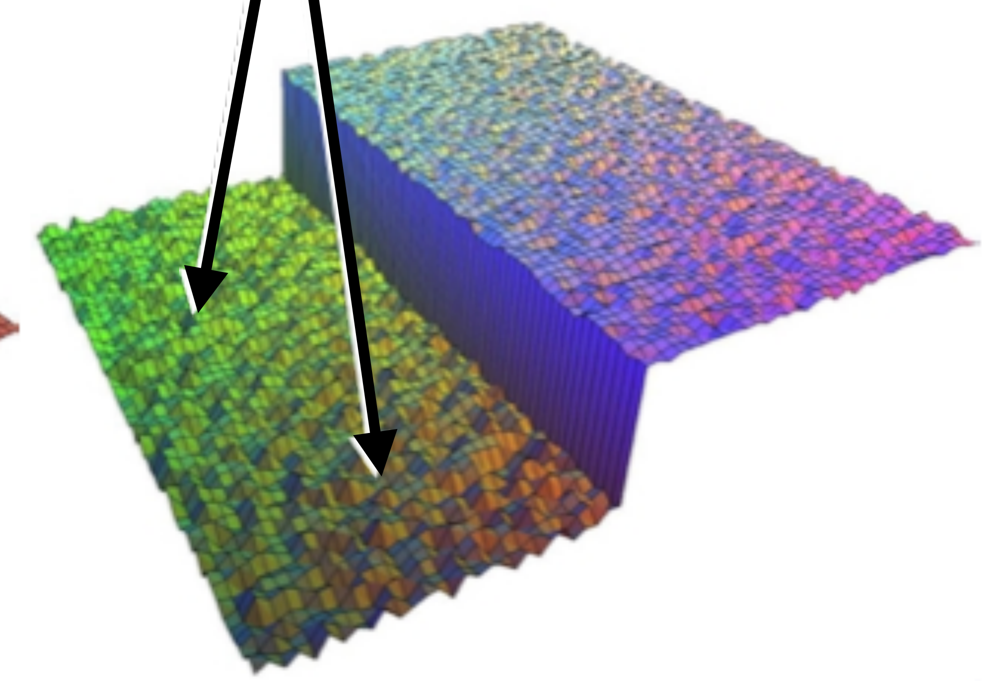
Pixels with significantly different intensity as *p* contribute little to filtered result (they are "on the "other side of the edge"



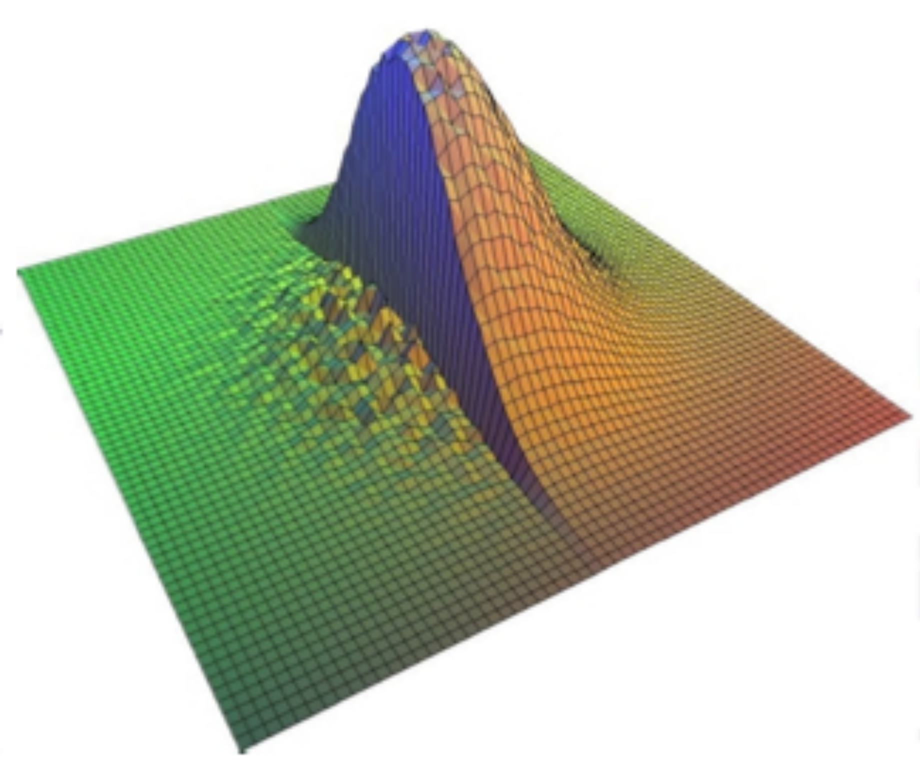
Input image



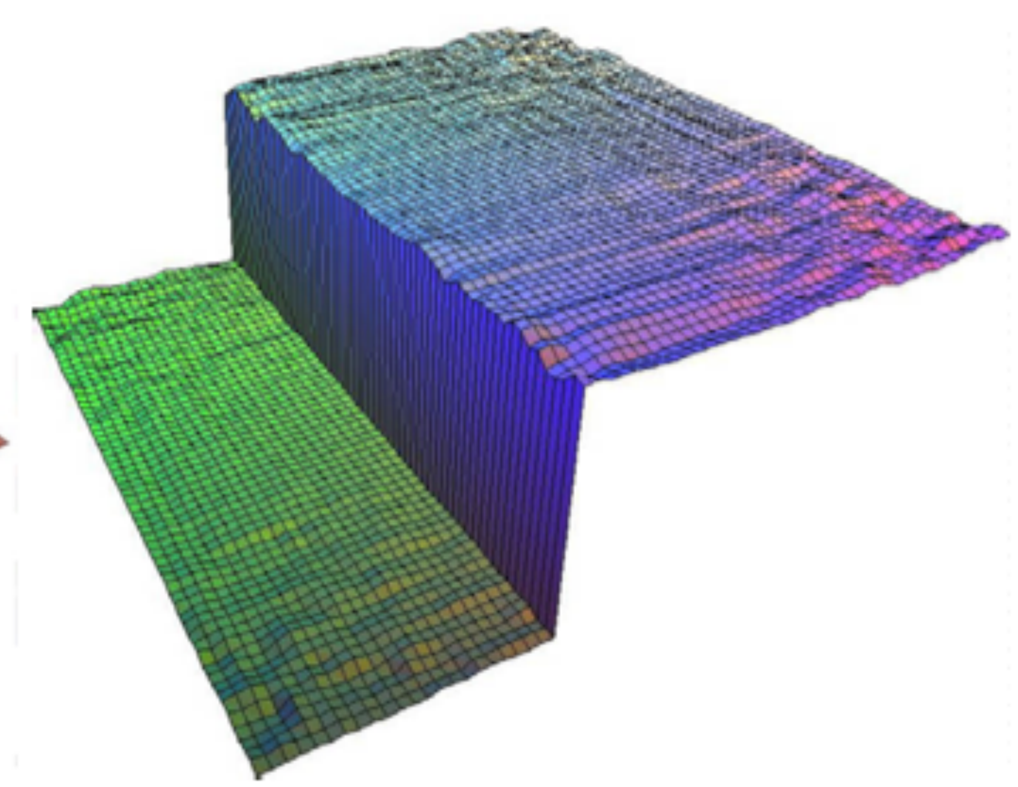
G(): gaussian about input pixel *p*



f(): Influence of support region



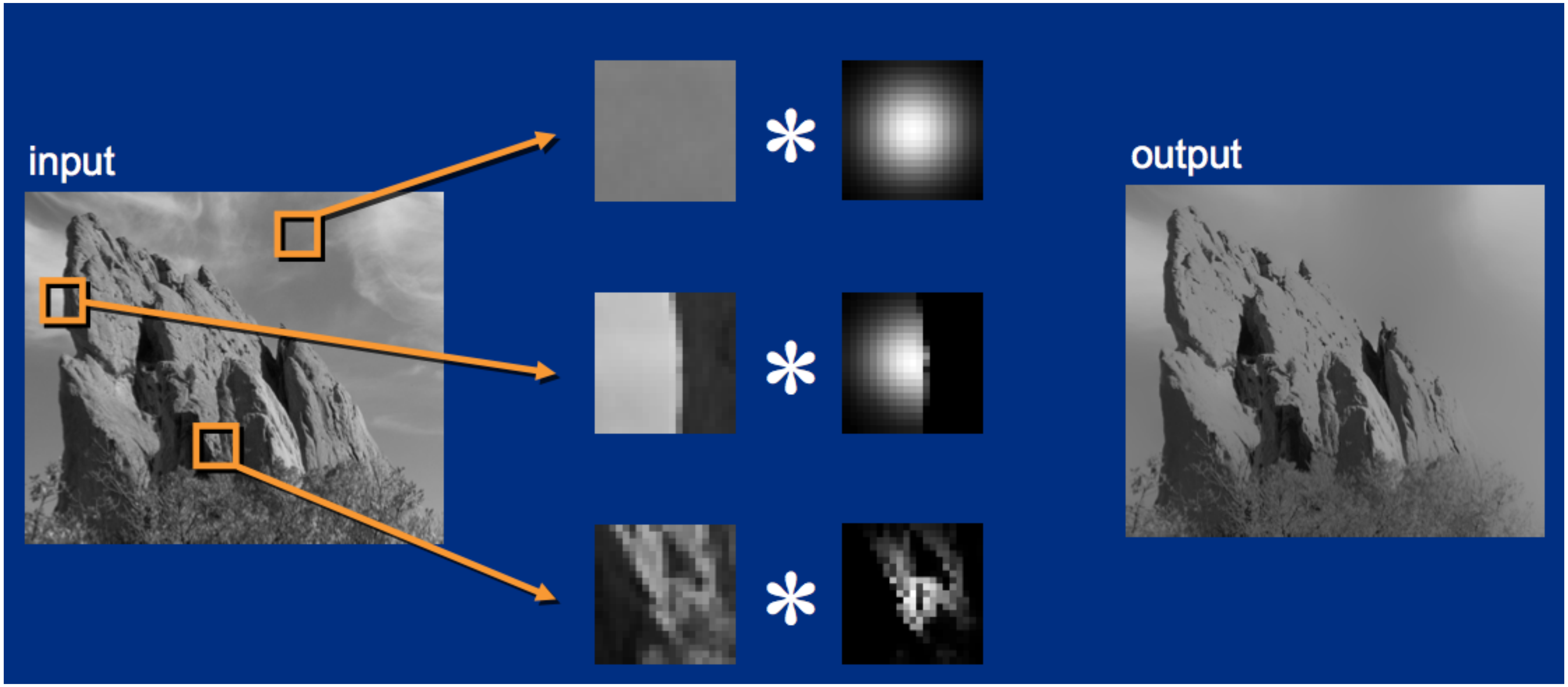
G x f: filter weights for pixel *p*



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



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

Question: describe a type of edge the bilateral filter will not respect (it will blur across these edges)

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”

(main idea: pixel patterns in another part of the image are hints for how to improve image in the current region)

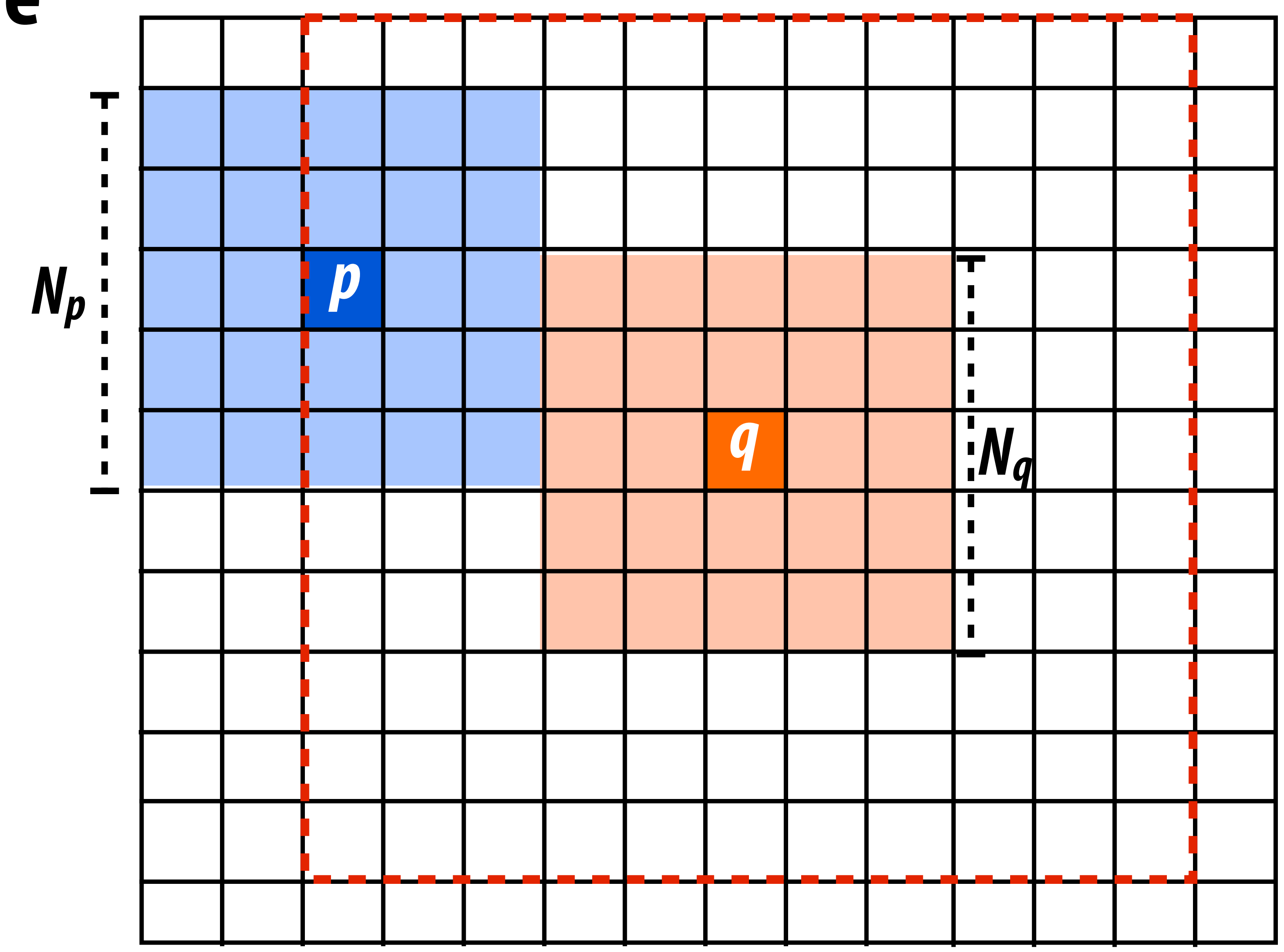
Denoising using non-local means

- **Main idea: replace pixel with average value of nearby pixels that have a similar surrounding region.**
  - **Assumption: images have repeating structure**

$$\mathrm{NL}[I](\mathrm{p}) = \sum_{\mathrm{q} \in \mathrm{S}(\mathrm{p})} w(\mathrm{p},\mathrm{q}) I(q)$$

All points in search region about p

$$w(\mathrm{p},\mathrm{q}) = \frac{1}{C_p} e^{-\frac{\|N_p - N_q\|^2}{h^2}}$$

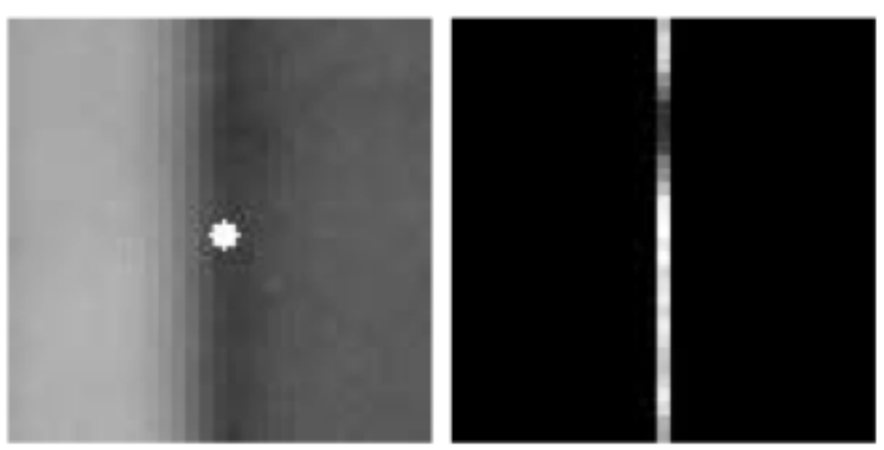


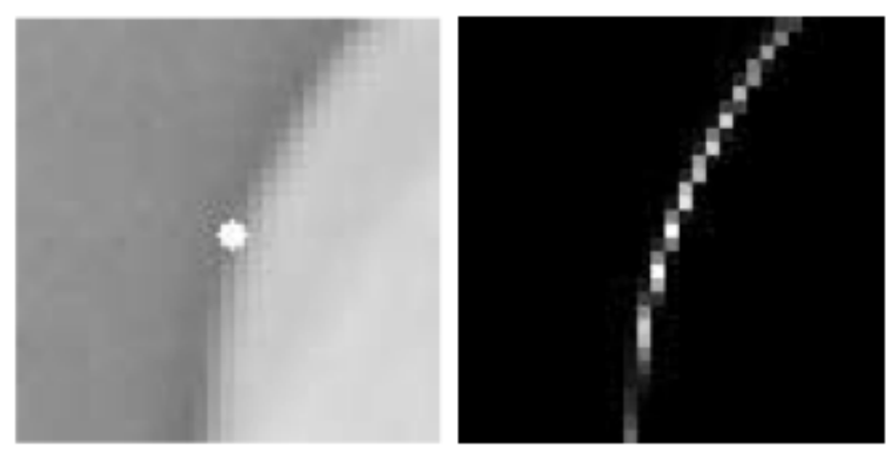
  - N_p and P_q are vectors of pixel values in square window around pixels *p* and *q* (highlighted regions in figure)
  - L^2 difference between N_p and P_q = "similarity" of surrounding regions
  - *Cp* is just a normalization constant to ensure weights sum to one for pixel *p*.
  - *S* is the search region around *p* (given by dotted red line in figure)

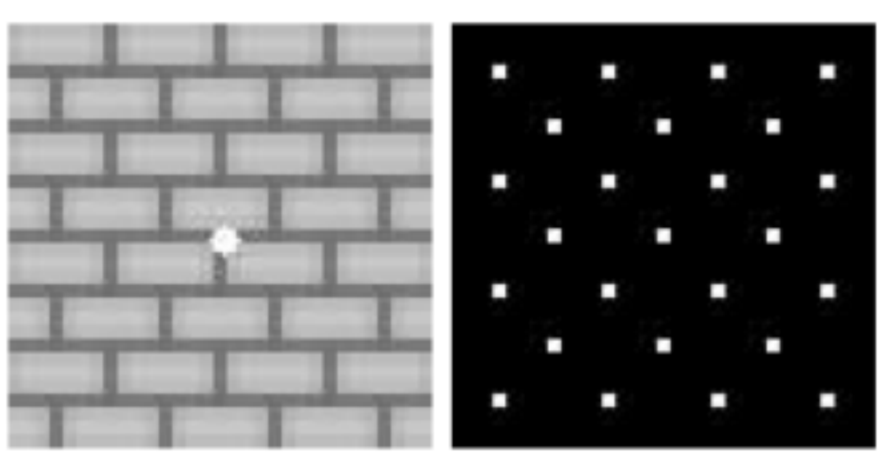
Denoising using non-local means

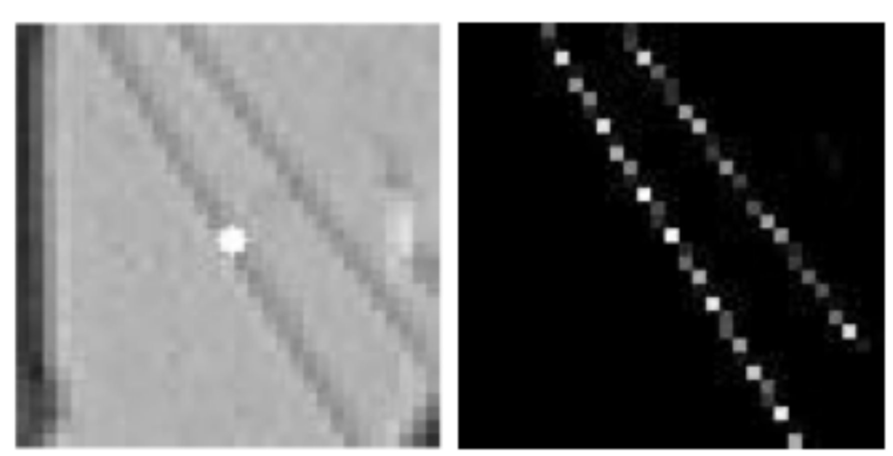
- Large weight for input pixels that have similar neighborhood as *p*
  - Intuition: filtered result is the average of pixels "like" this one
  - In example below-right: *q1* and *q2* have high weight, *q3* has low weight

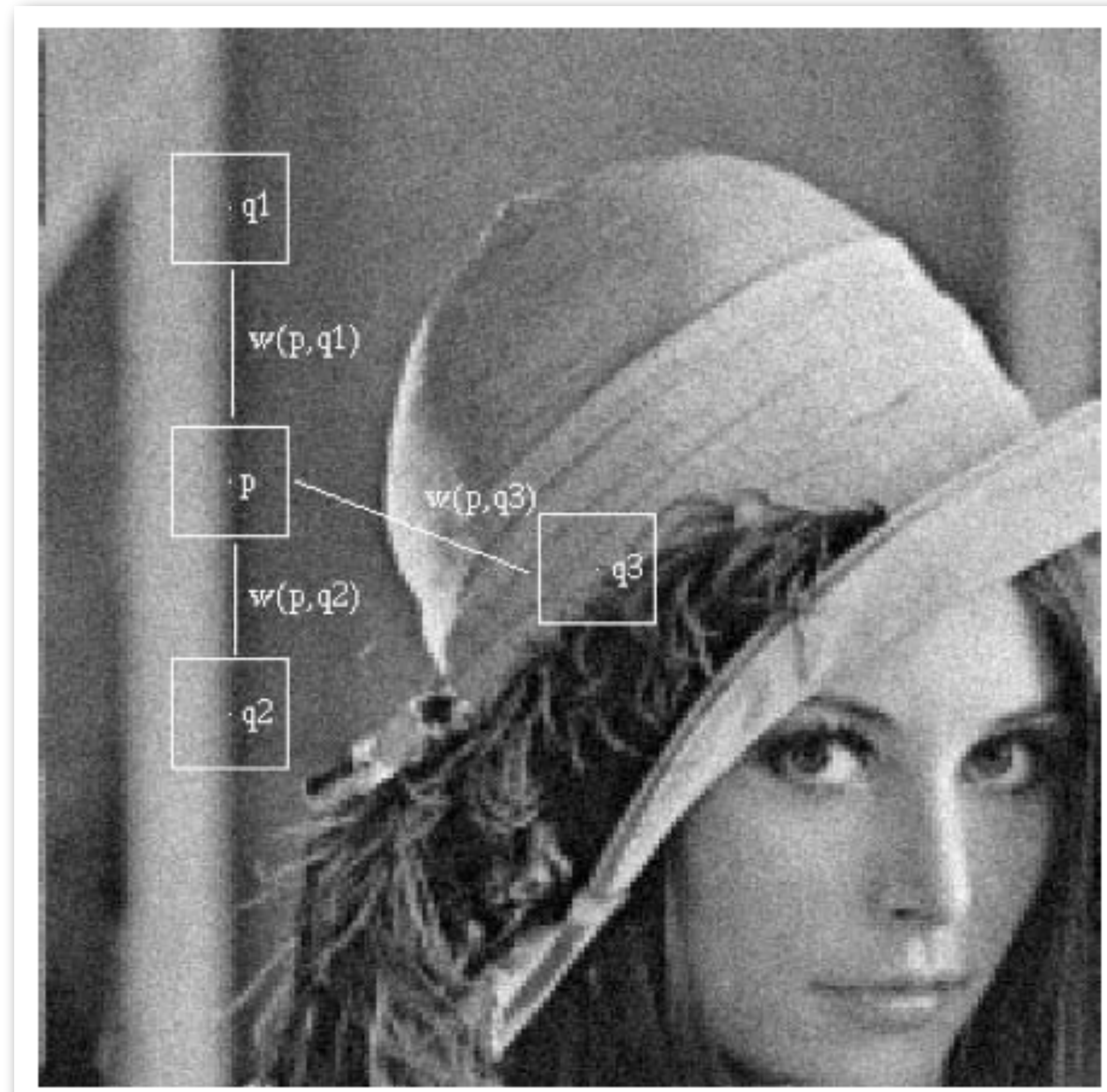
In each image pair below:
- Image at left shows the pixel *p* to denoise.
- Image at right shows weights of pixels in 21x21-pixel kernel support window surrounding *p*.


(A)


(B)


(C)


(D)



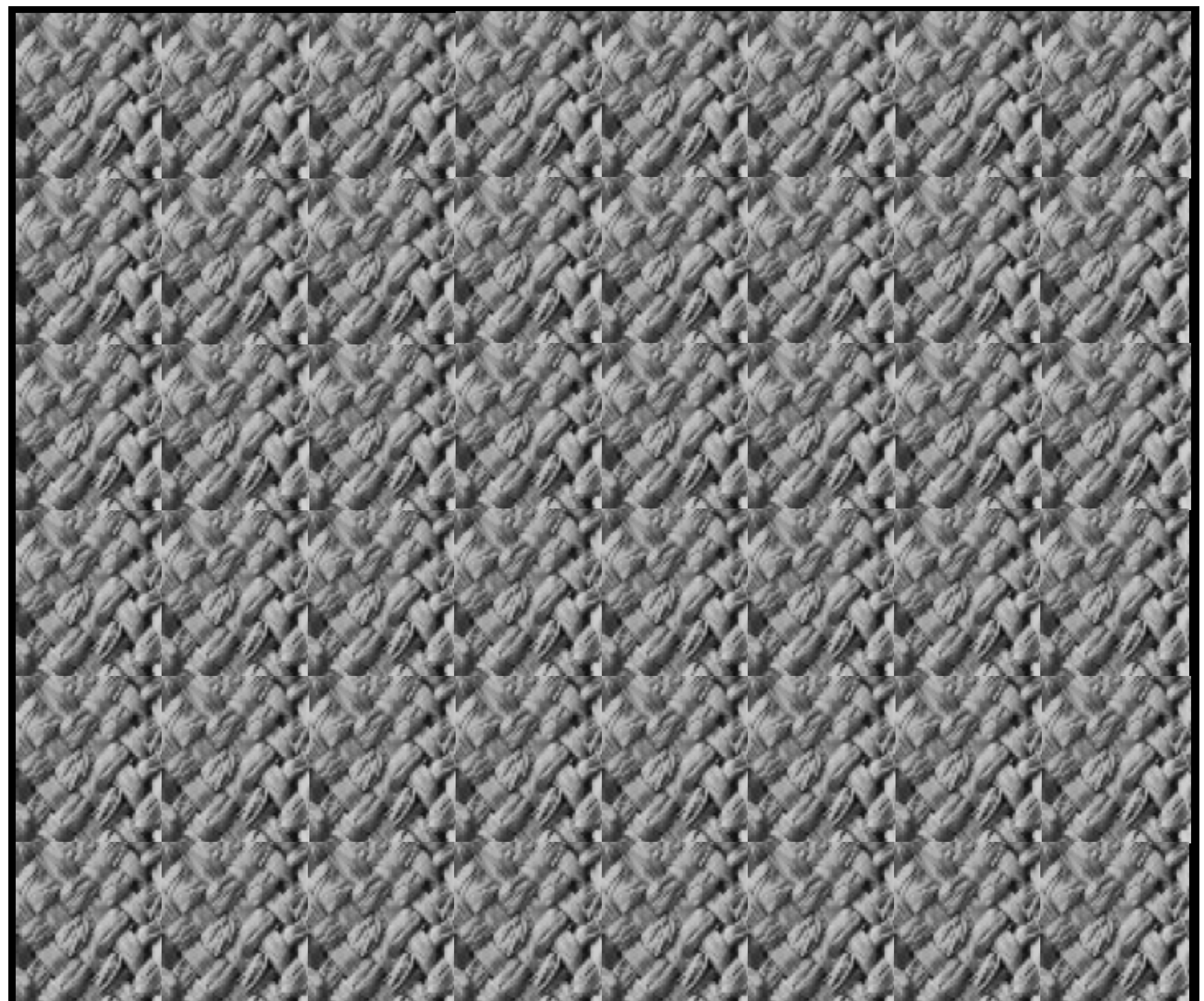
Buades et al. CVPR 2005

Texture synthesis

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

Source texture
(low resolution)

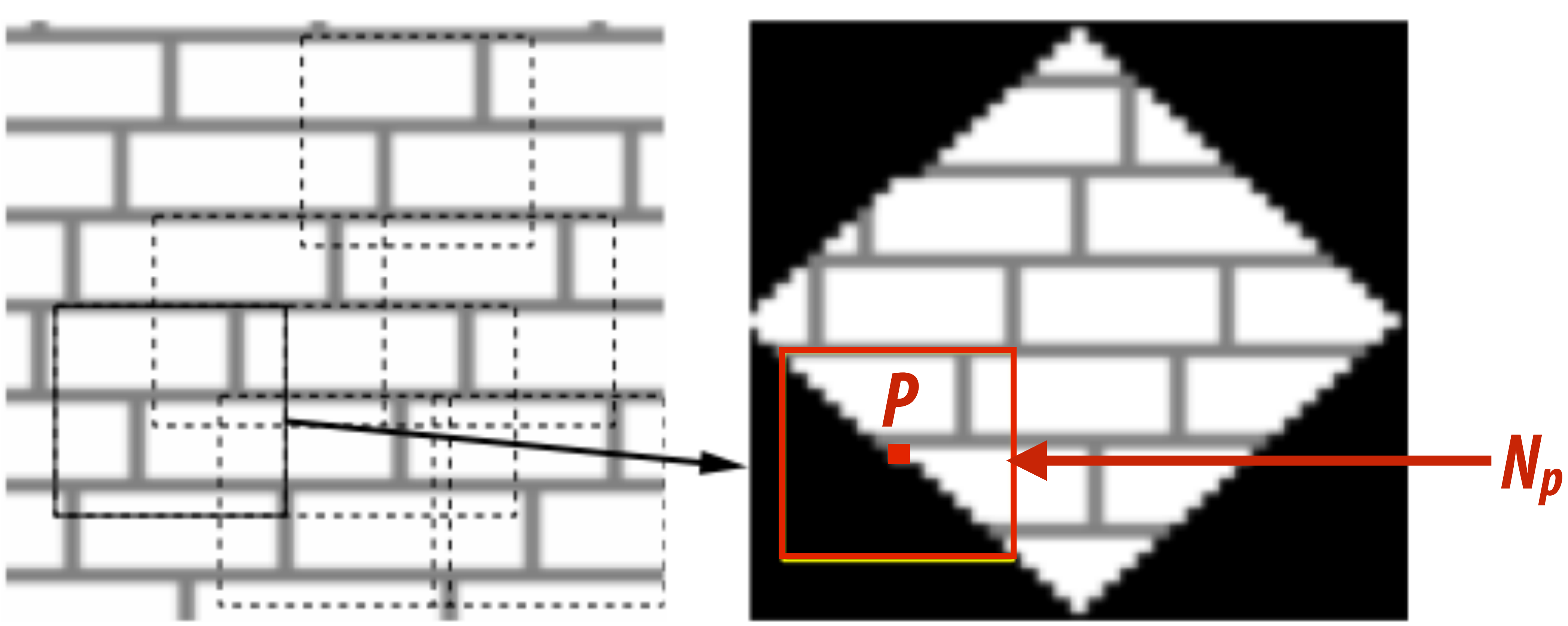
High-resolution texture generated by naively tiling low-resolution texture



Algorithm: non-parametric texture synthesis

Main idea: given the NxN neighborhood N_p around unknown pixel p, want a probability distribution function for possible values of p, given N_p: $P(p=X \mid N_p)$

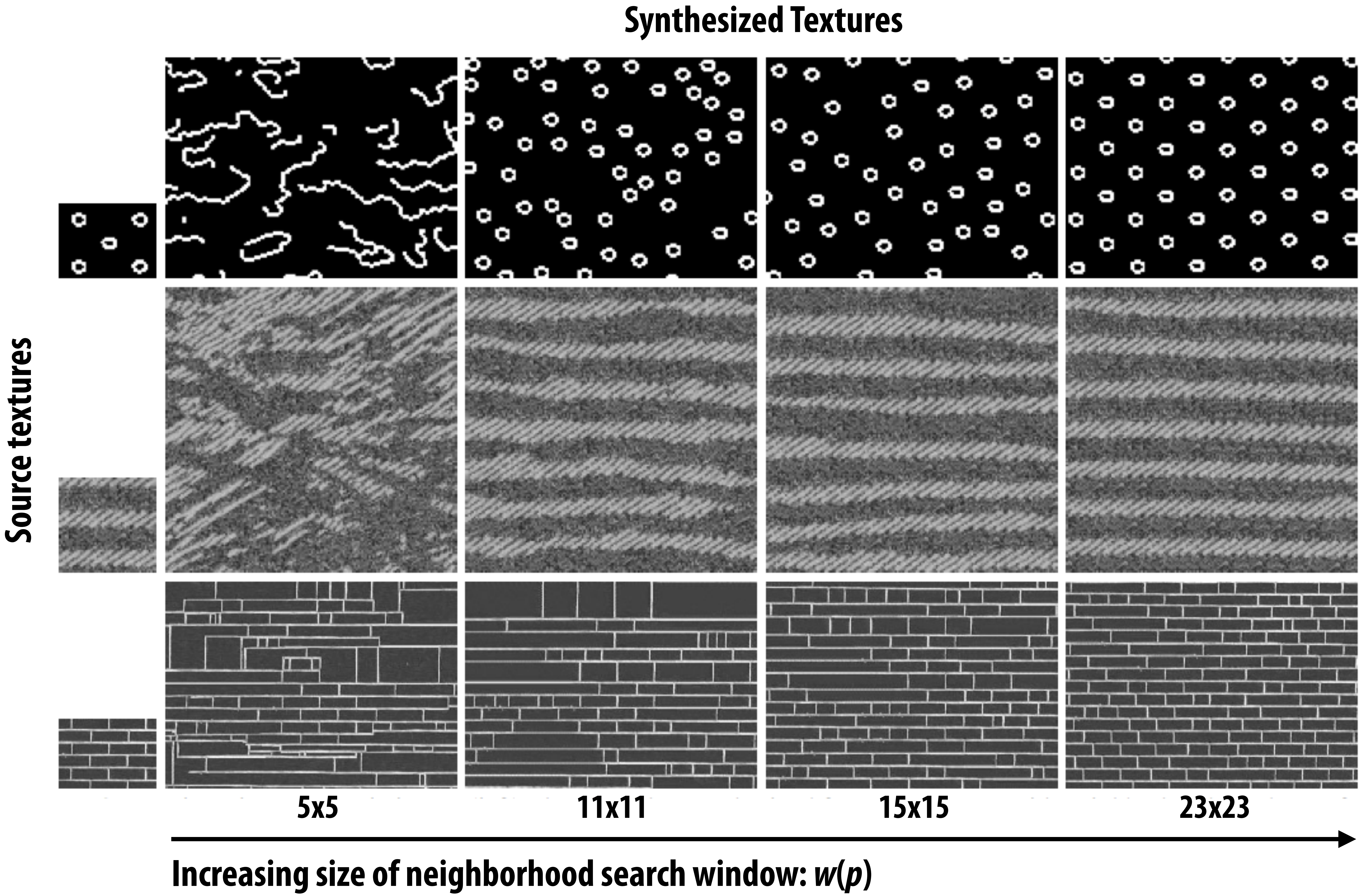
```
For each pixel p to synthesize:
    1.  Find other NxN patches (Nq) in the image that are most similar to Np
    2.  Center pixels of the closest patches are candidates for p
    3.  Randomly sample from candidates weighted by distance d(Np,Nq)
```



[Efros and Leung 99]

Non-parametric texture synthesis

[Efros and Leung 99]

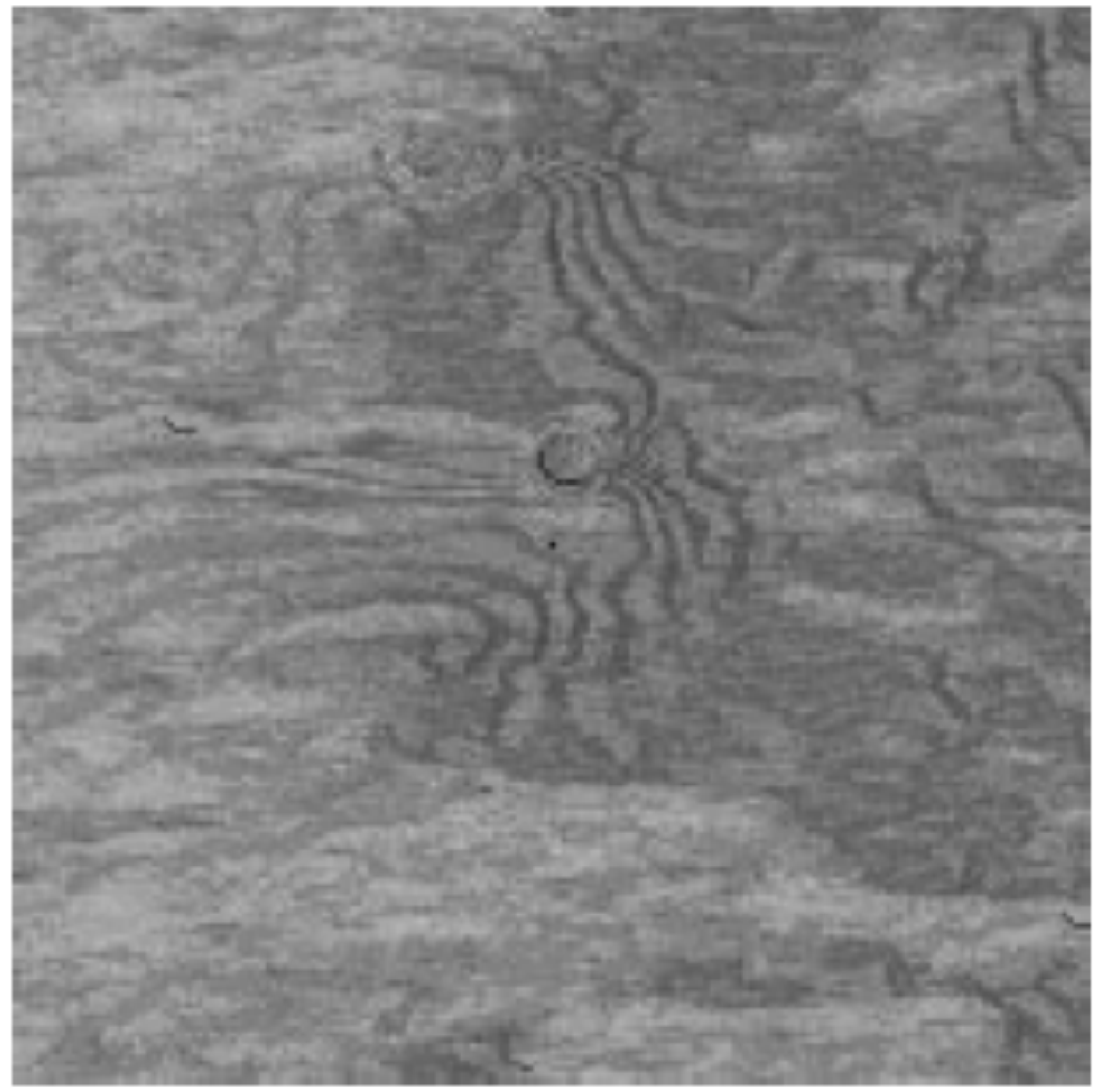


More texture synthesis examples

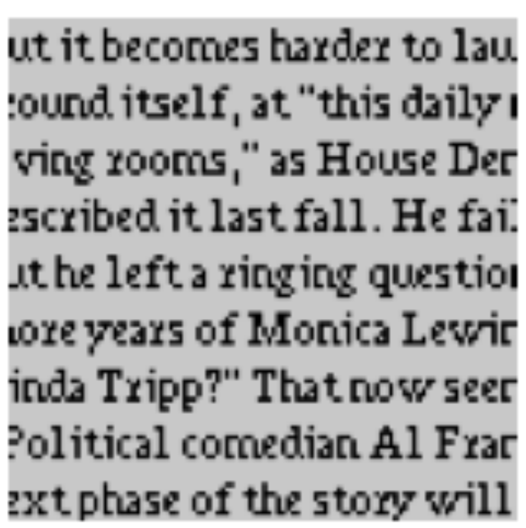
[Efros and Leung 99]

Source textures

Synthesized Textures



Naive tiling solution



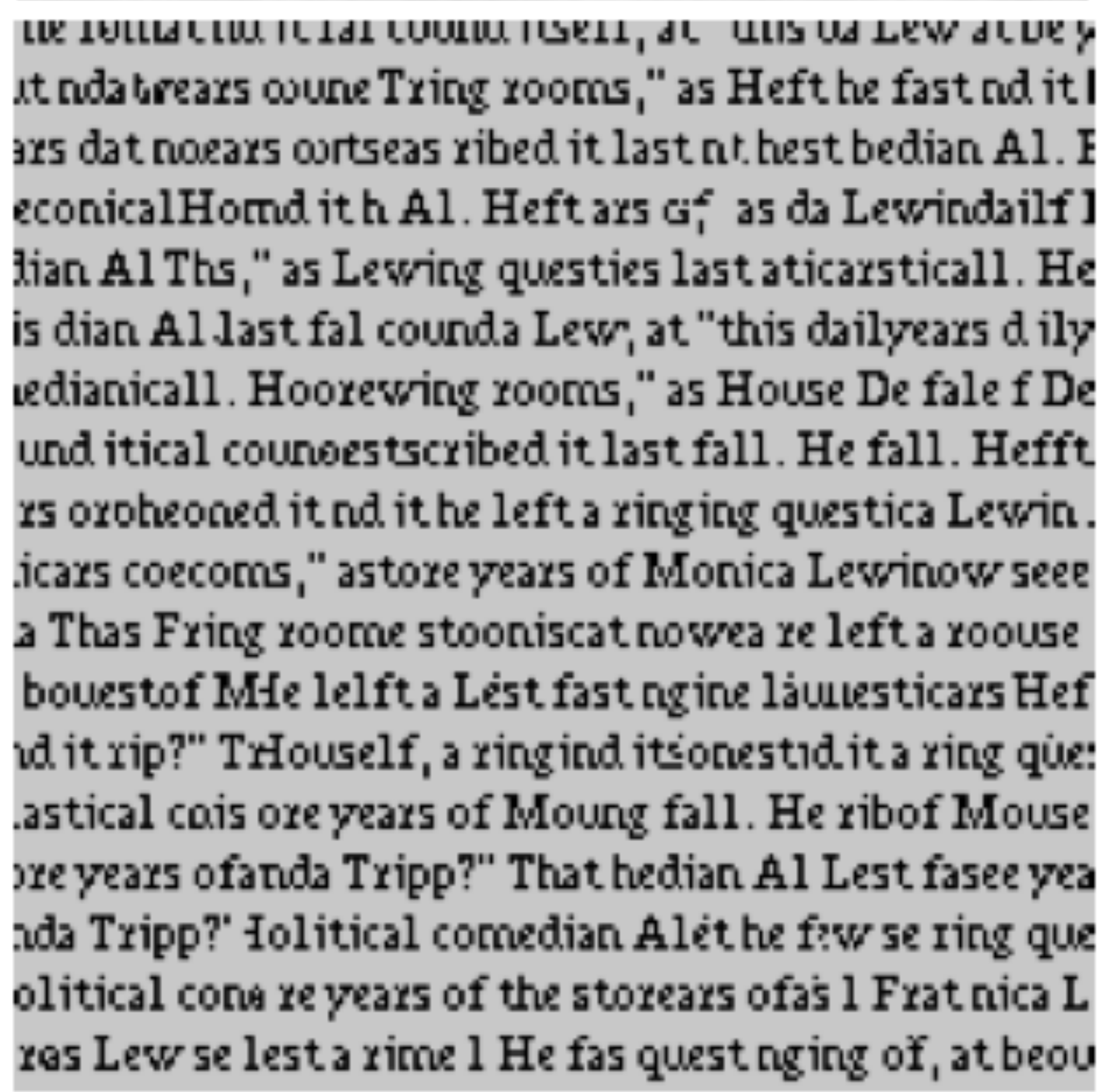
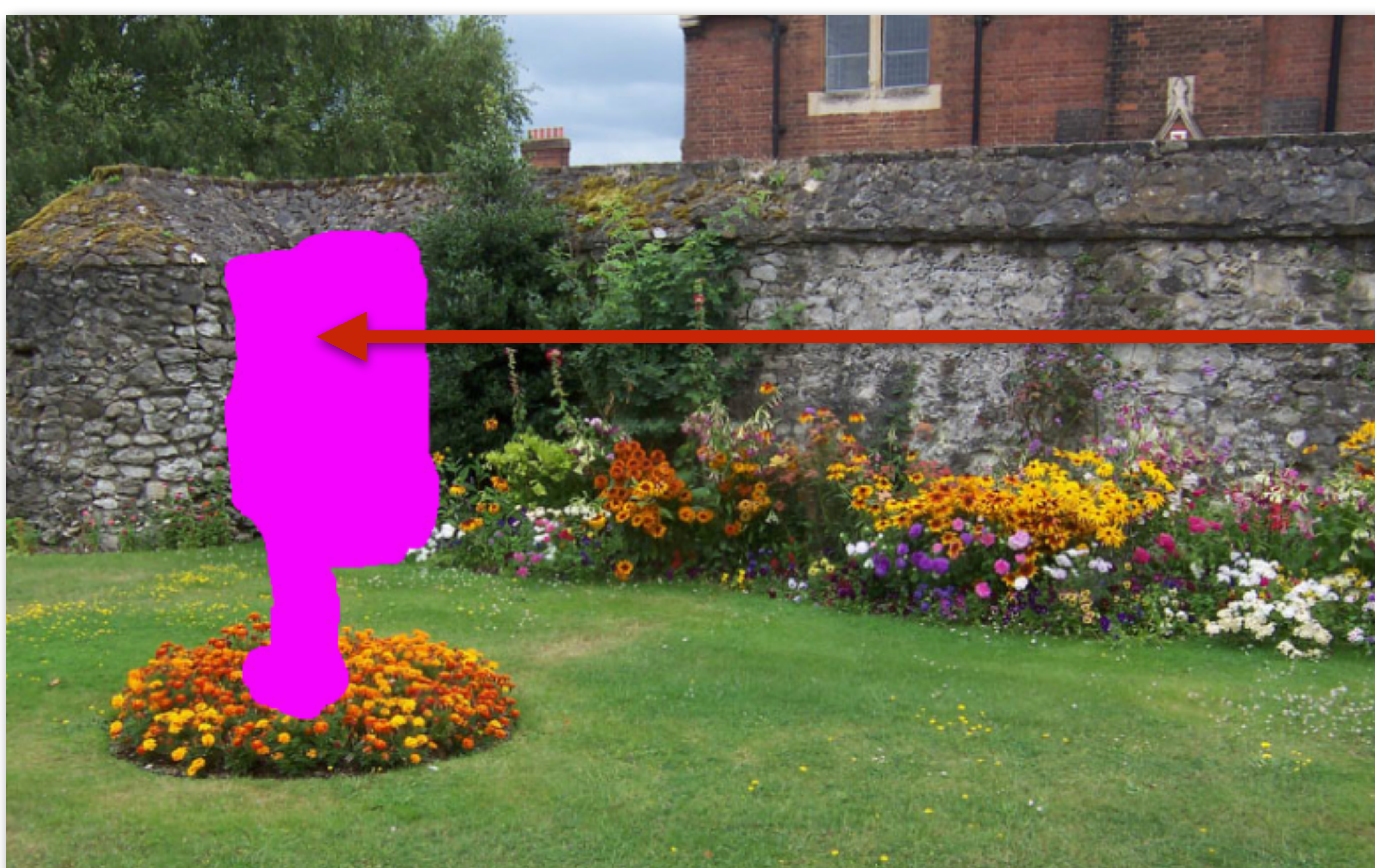


Image completion example

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]

Image processing summary

- Image processing via convolution
  - Different operations specified by changing weights of convolution kernel
  - Separable filters lend themselves to efficiency implementation as multiple 1D filters

- Data-driven image processing techniques
  - Key idea: use examples from other places in the image as priors to determine how to manipulate image

- To learn more: consider 15-463/663: "Computational Photography"