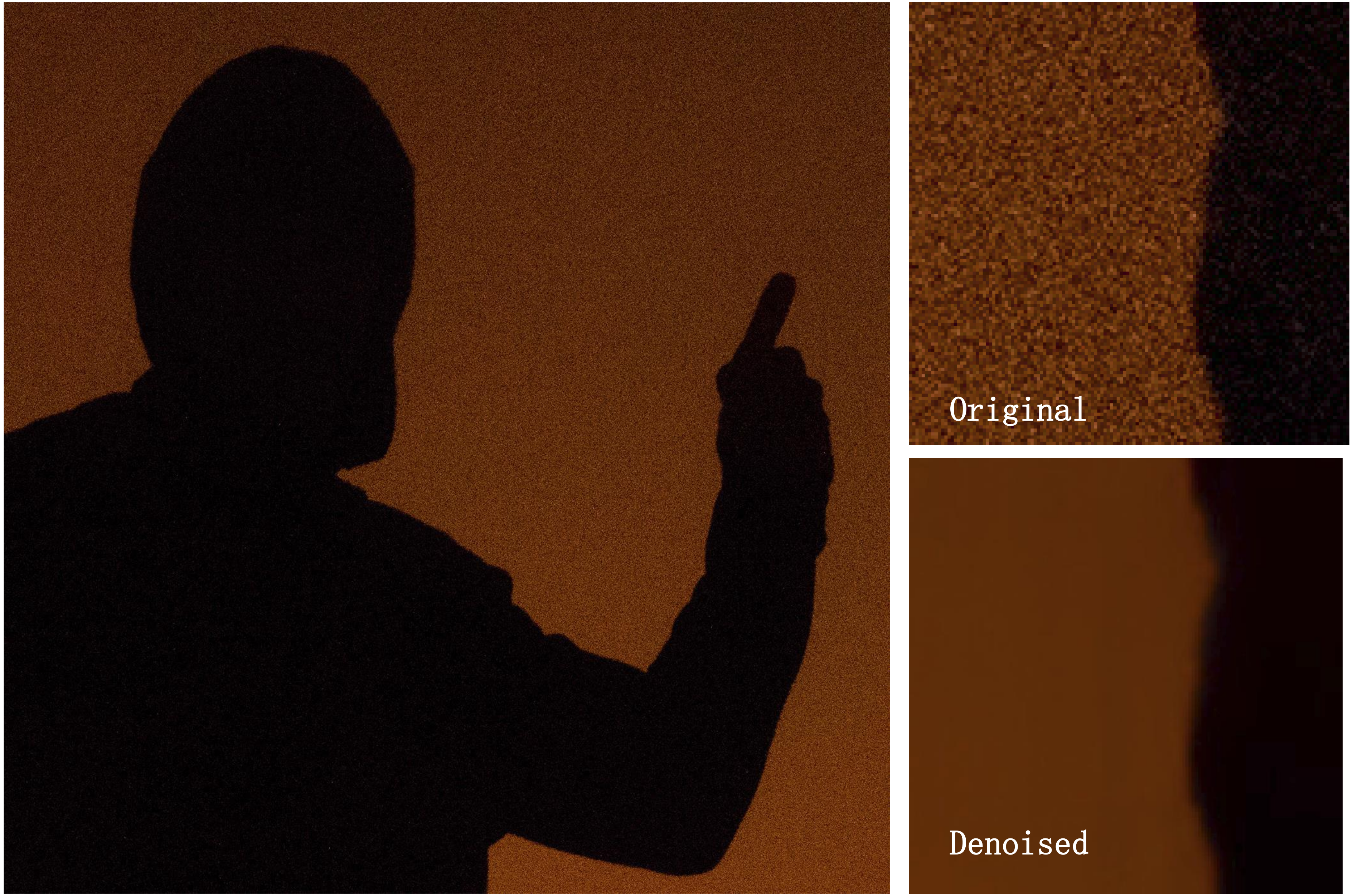
Computer Graphics - Filtering

Junjie Cao @ DLUT

Spring 2016

http://jjcao.github.io/ComputerGraphics/

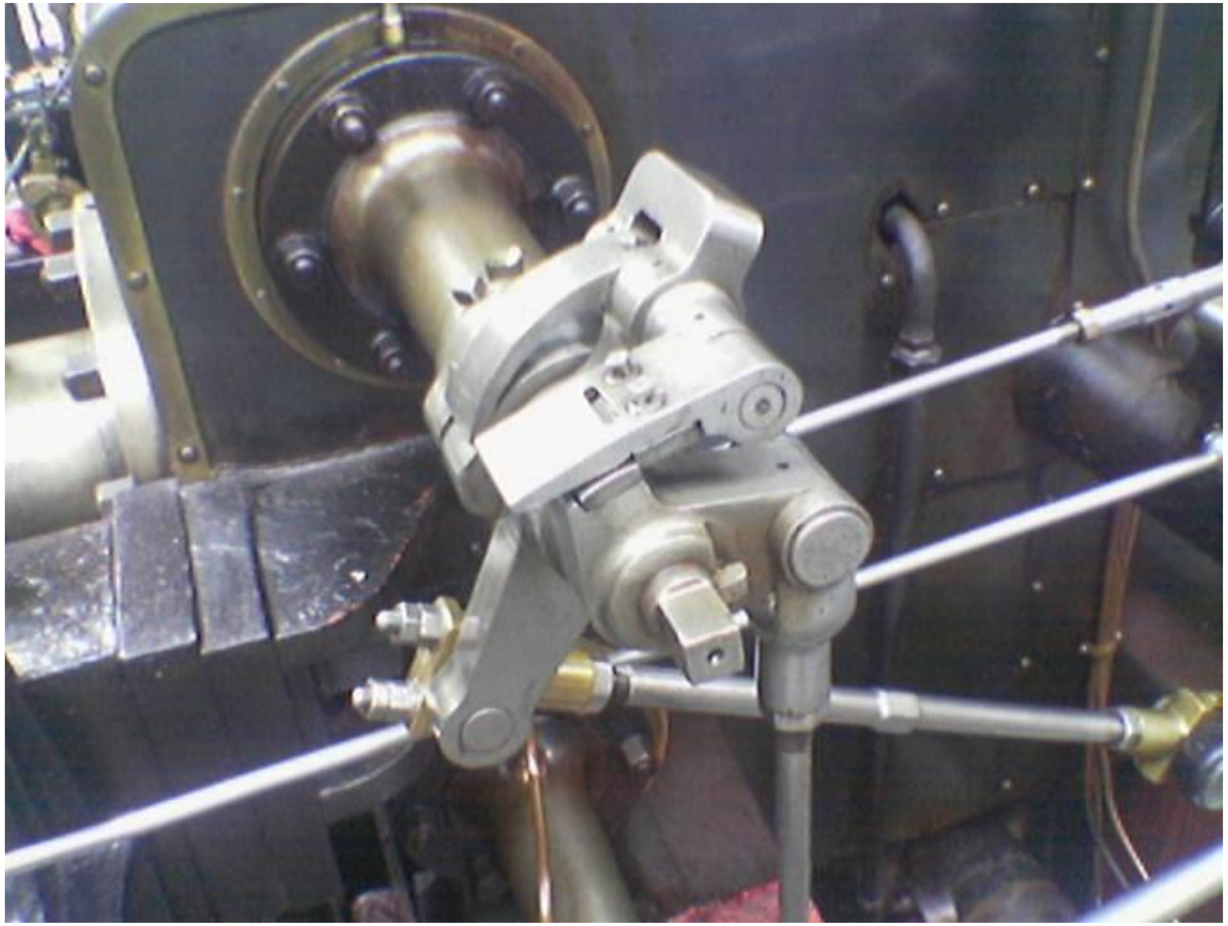
Denoising

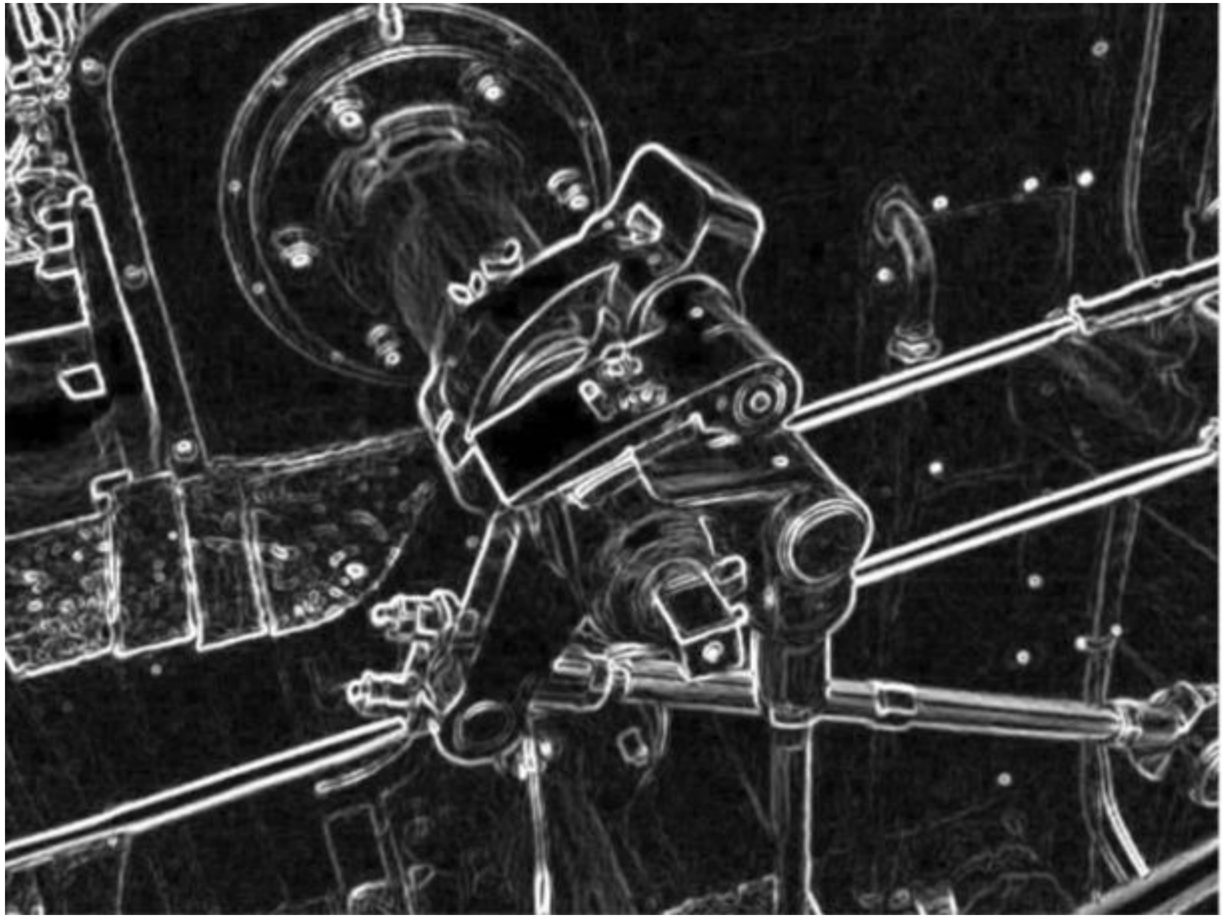


Blur

Sharpen

Edge detection





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

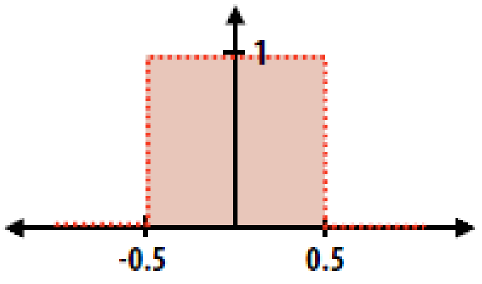
Bilateral filter

Convolution

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

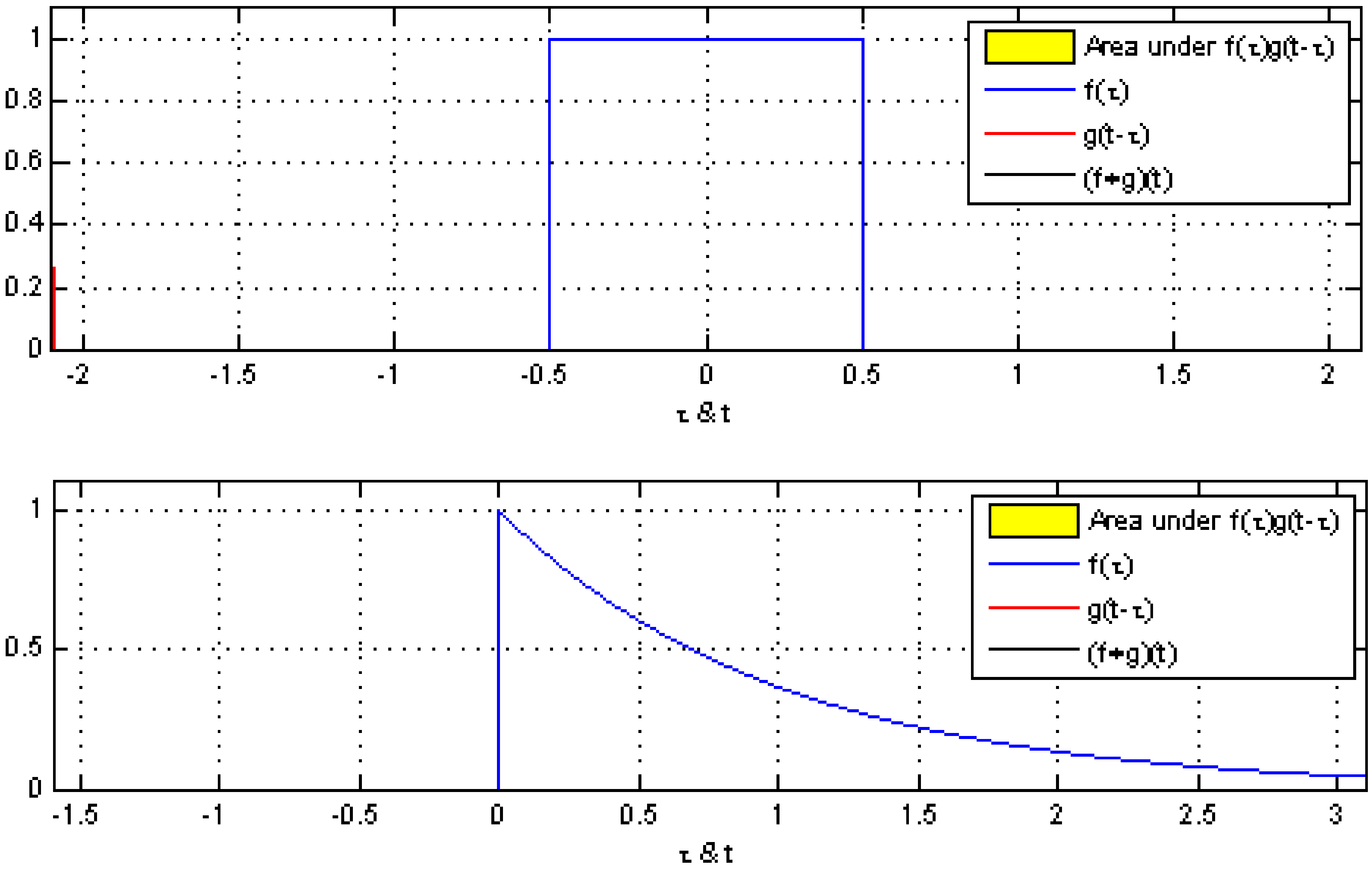
- **It may be helpful to consider the effect of convolution with the simple unit-area "box" function:**

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



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

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


Area under f(τ)g(t-τ)
f(τ)
g(t-τ)
(f*g)(t)
τ & t
Area under f(τ)g(t-τ)
f(τ)
g(t-τ)
(f*g)(t)
τ & t

Discrete 2D convolution

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

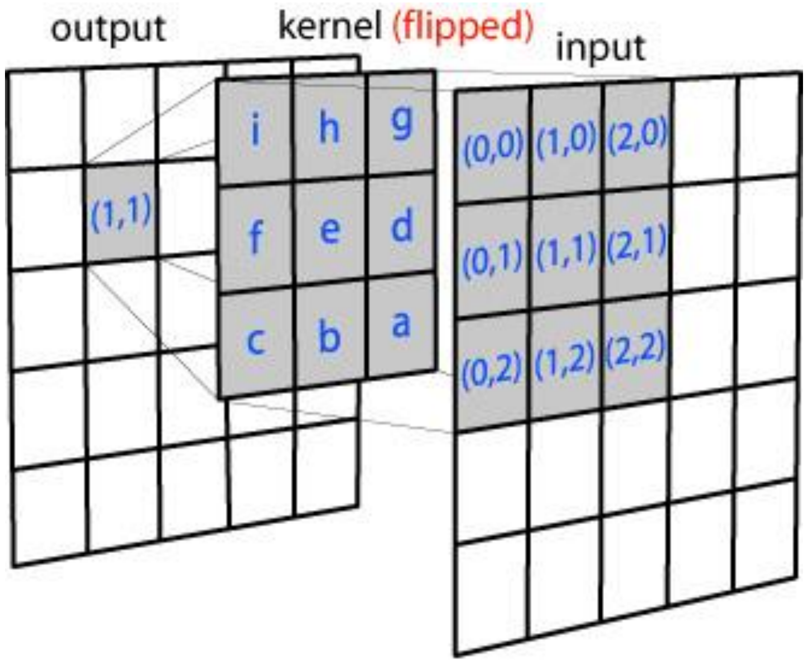
Consider $f(i,j)$ **that is nonzero only when:** $-1 \leq i,j \leq 1$

Then:

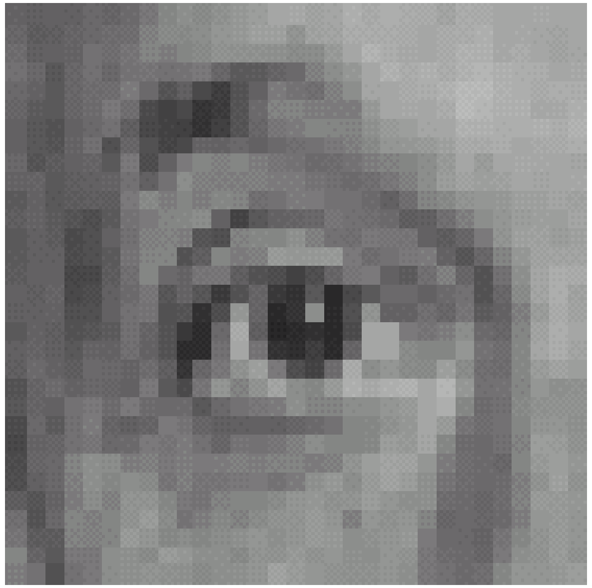
$$(f * g)(x,y) = \sum_{i,j=-1}^{1} f(i,j) I(x-i, y-j)$$

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

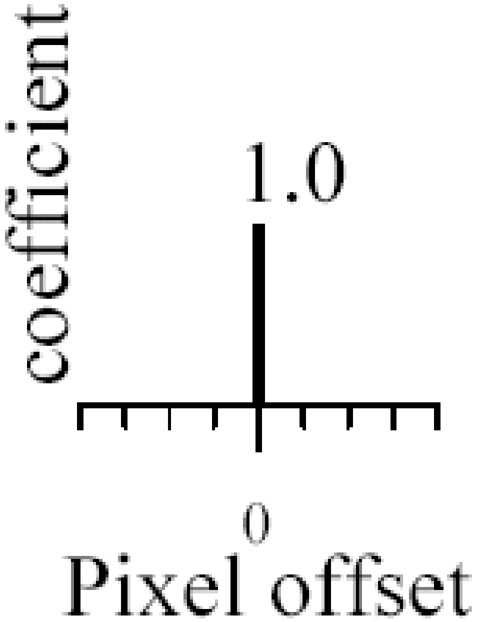
$f(i,j) = \mathbf{F}_{i,j}$ **(often called: "filter weights", "kernel")**



Linear filtering (warm-up slide)

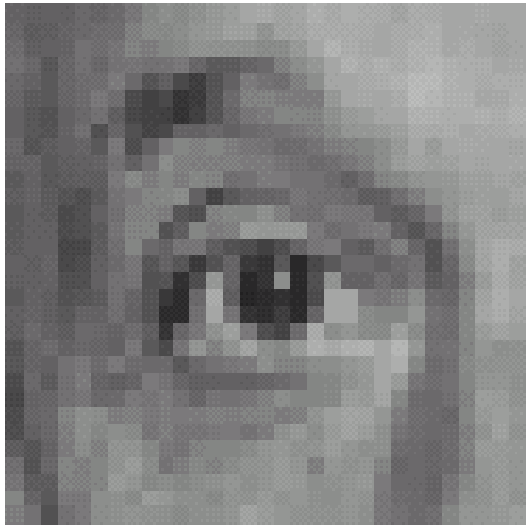


original

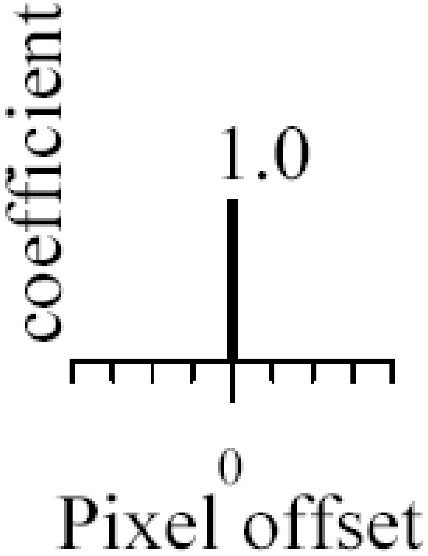


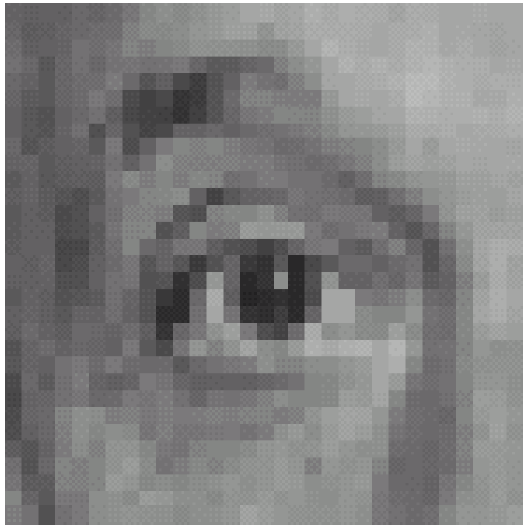
?

Linear filtering (warm-up slide)



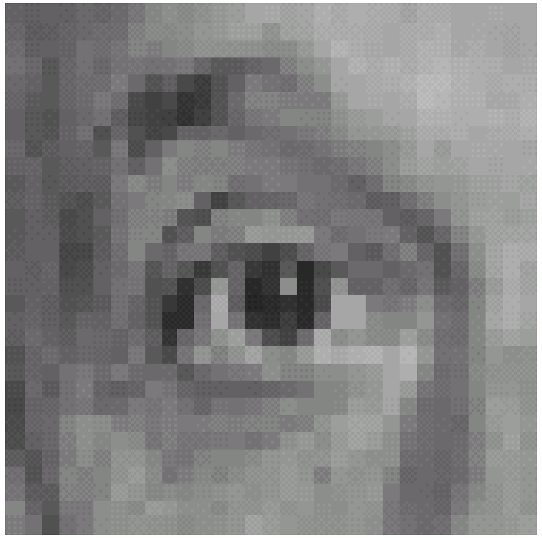
original



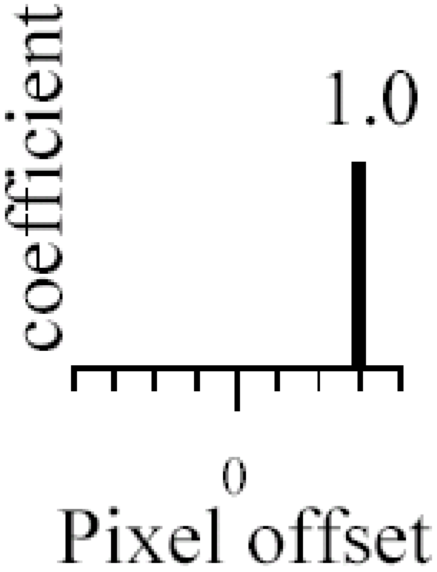


Filtered
(no change)

Linear filtering

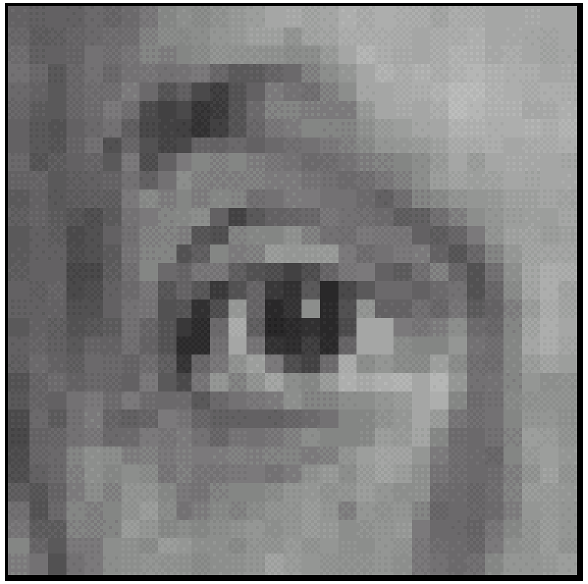


original

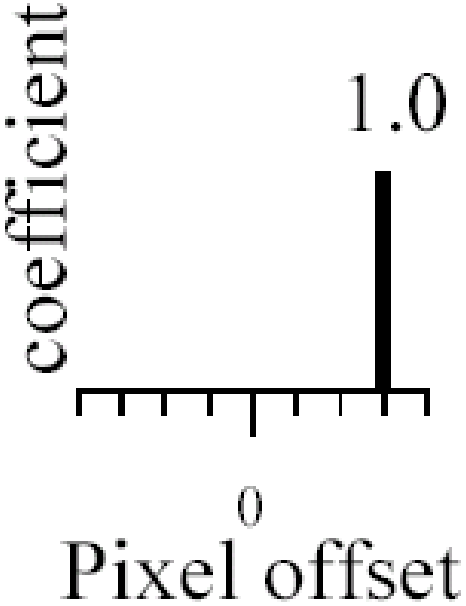


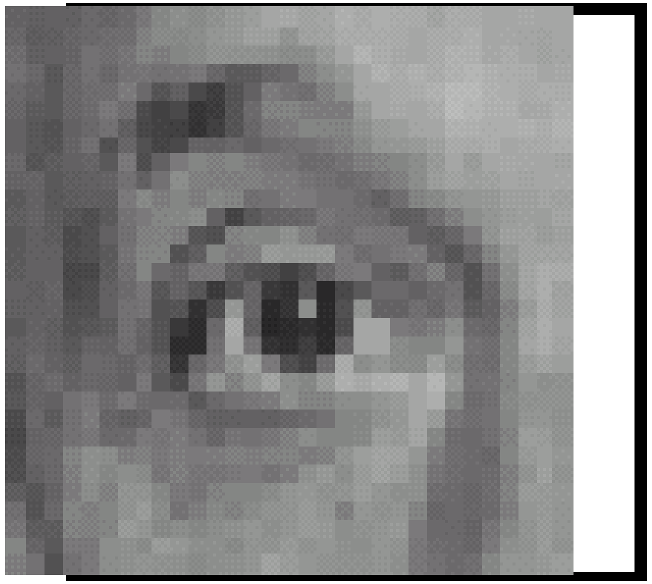
?

shift



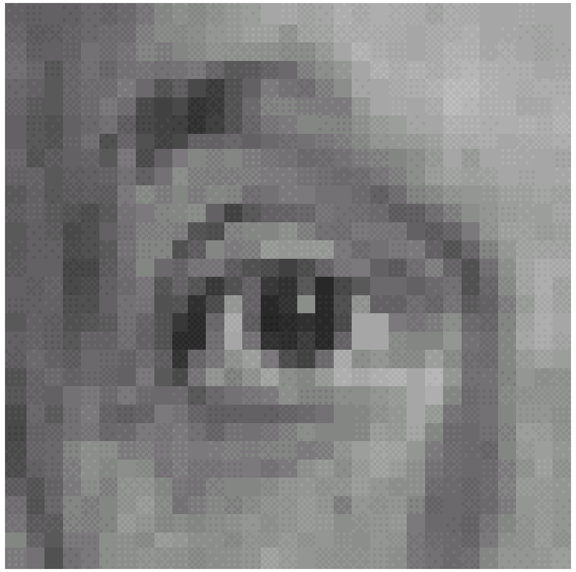
original



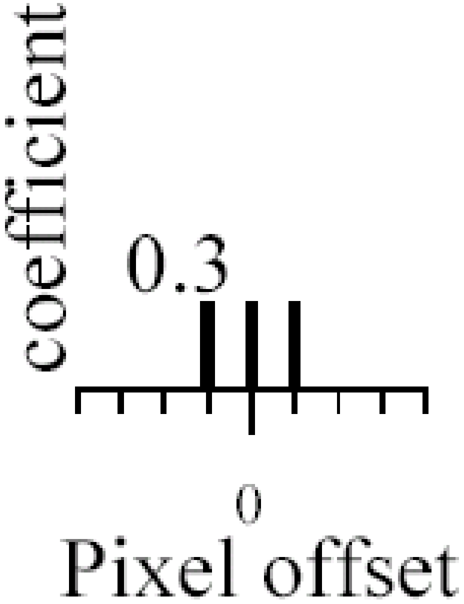


shifted

Linear filtering

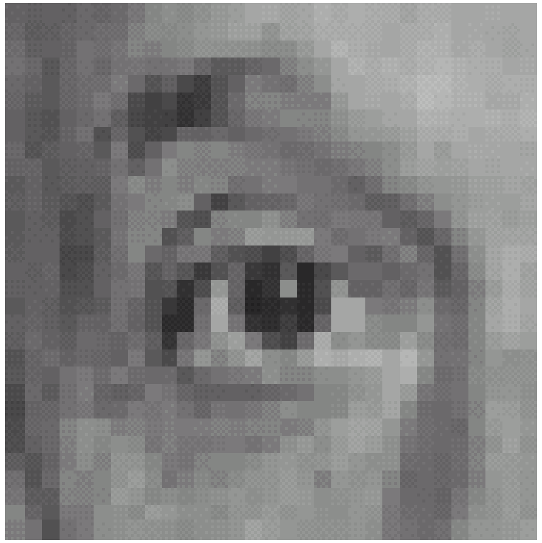


original

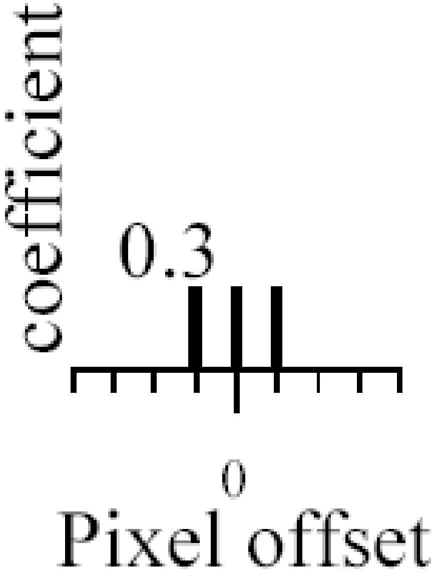


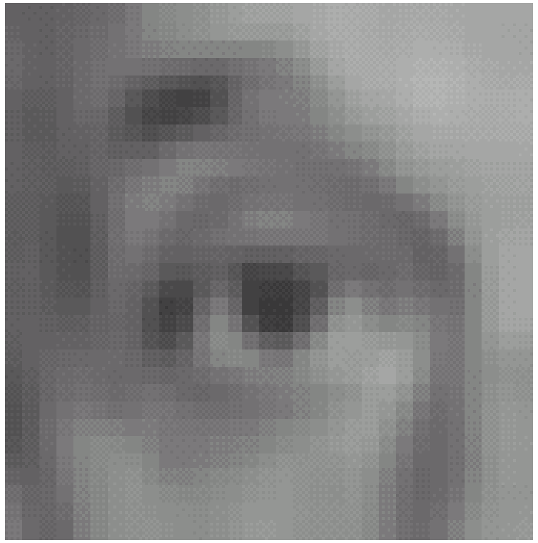
?

Blurring



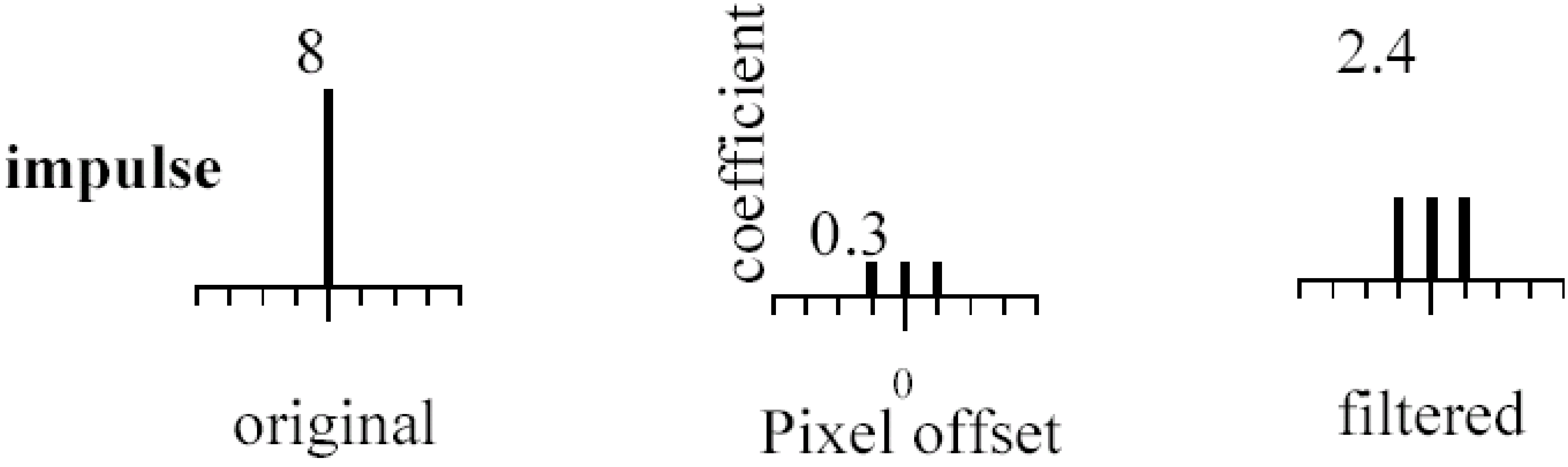
original



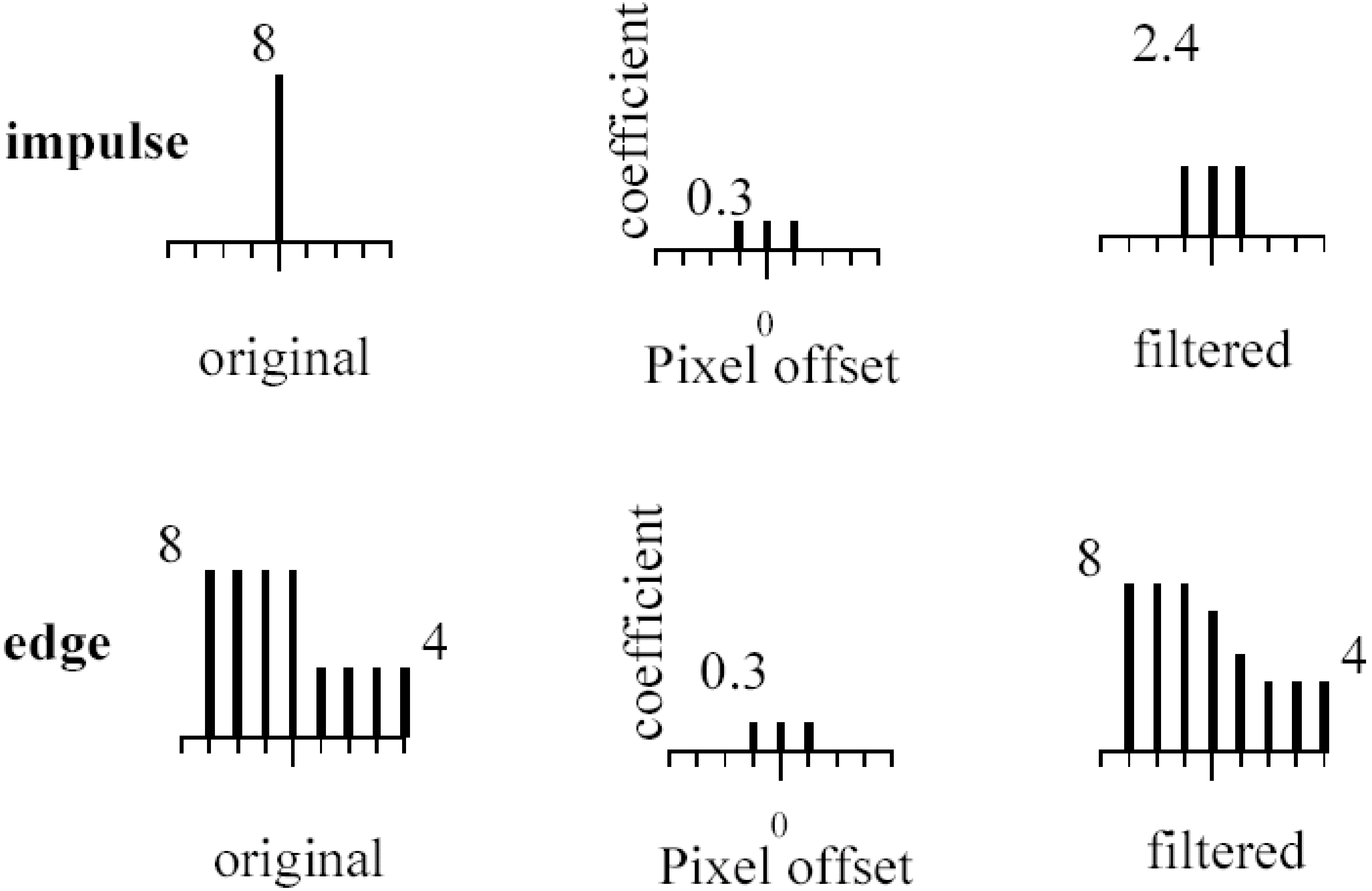


Blurred (filter applied in both dimensions).

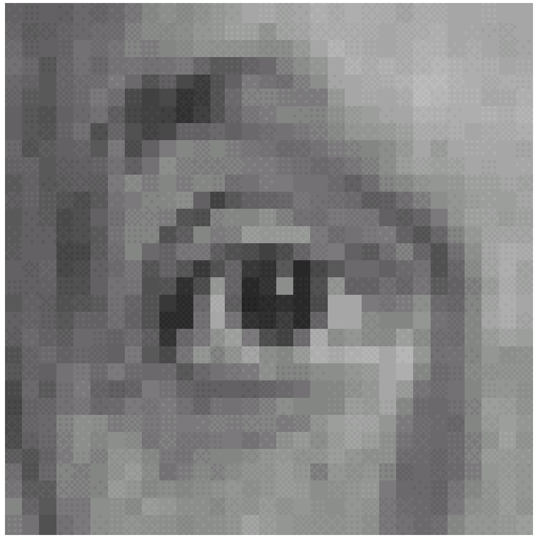
Blur examples



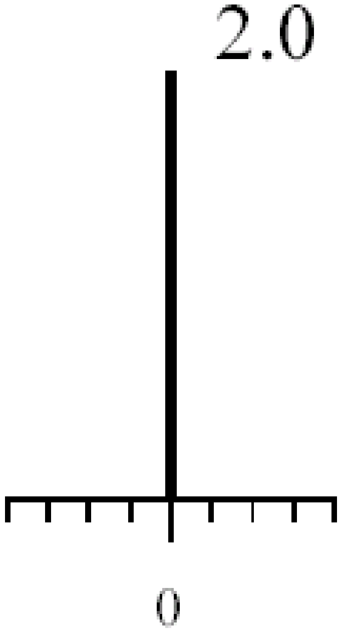
Blur examples



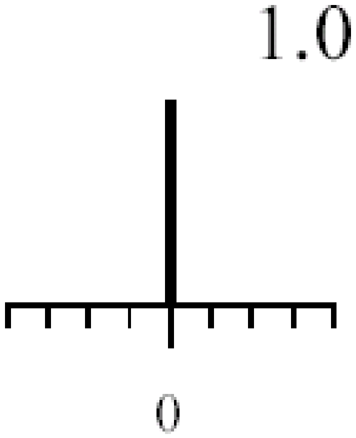
Linear filtering (warm-up slide)



original

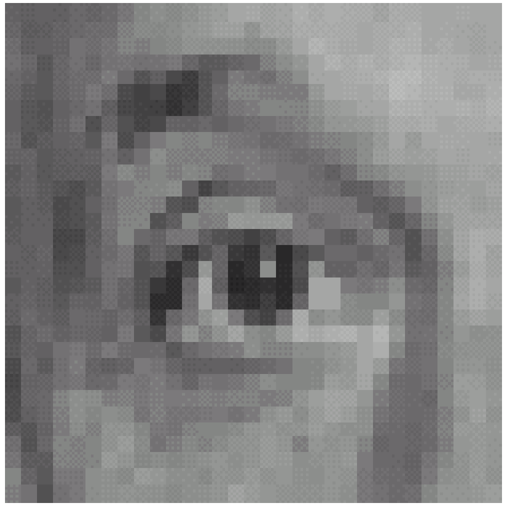


−

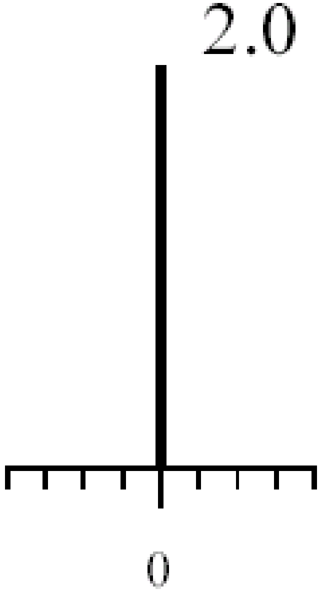


?

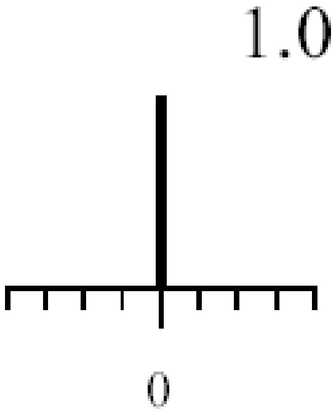
Linear filtering (no change)

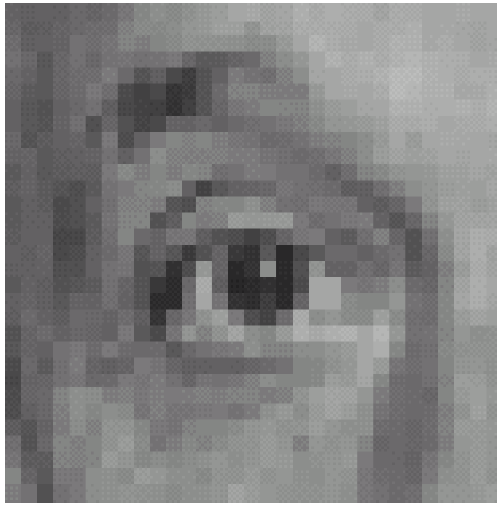


original



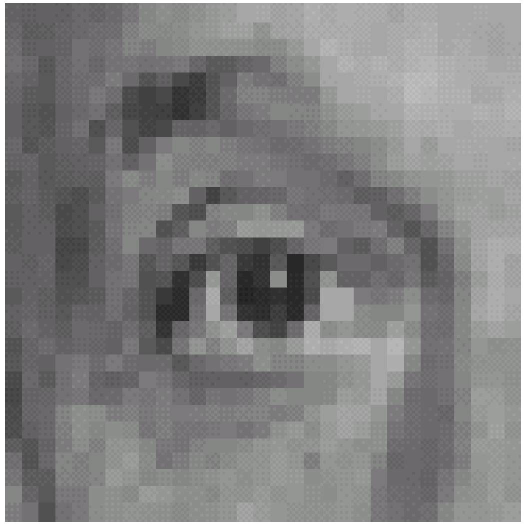
−



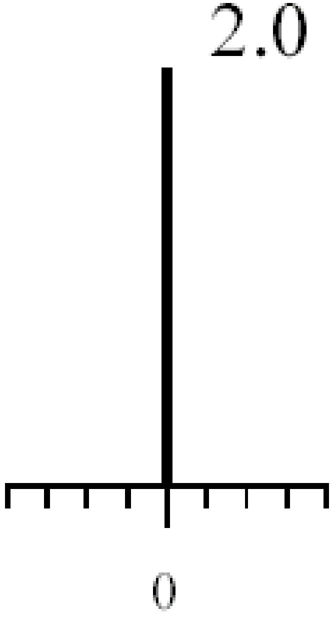


Filtered
(no change)

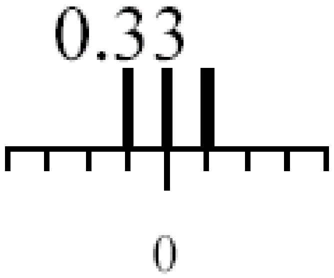
Linear filtering



original

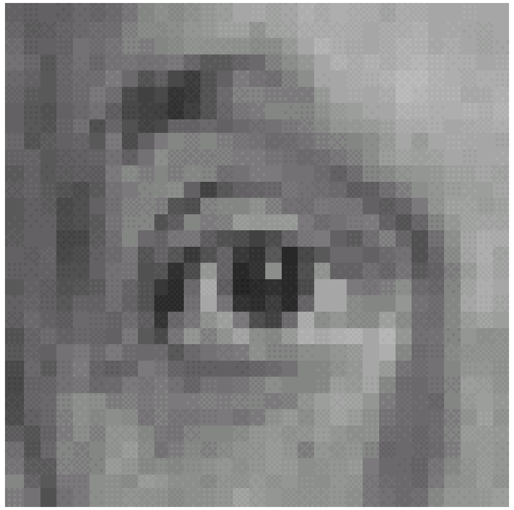


−

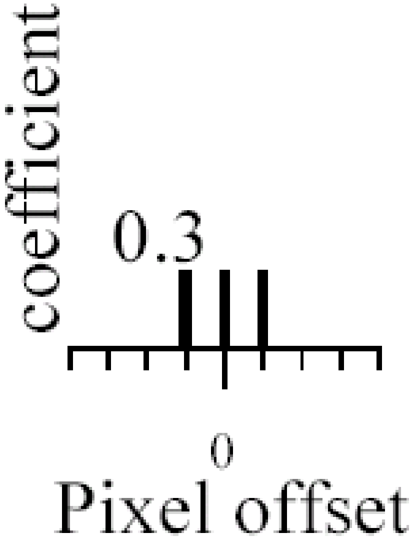


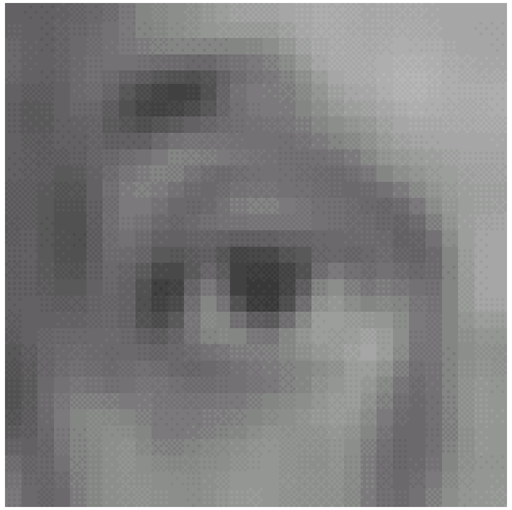
?

(remember blurring)



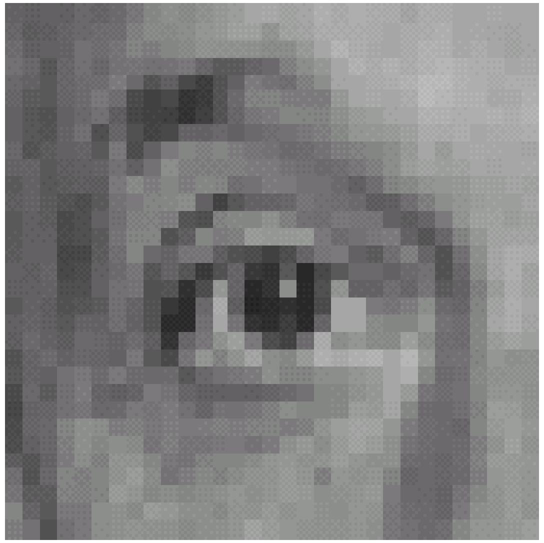
original



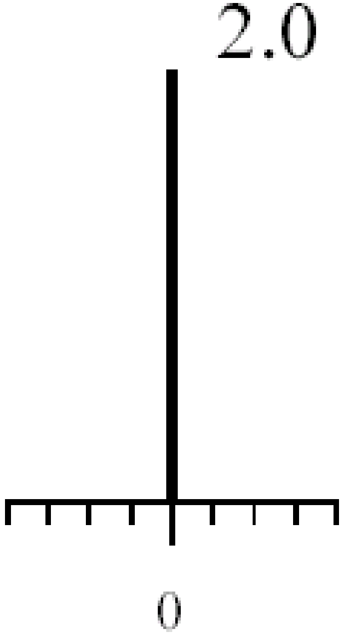


Blurred (filter applied in both dimensions).

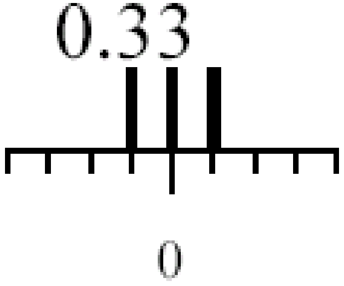
Sharpening

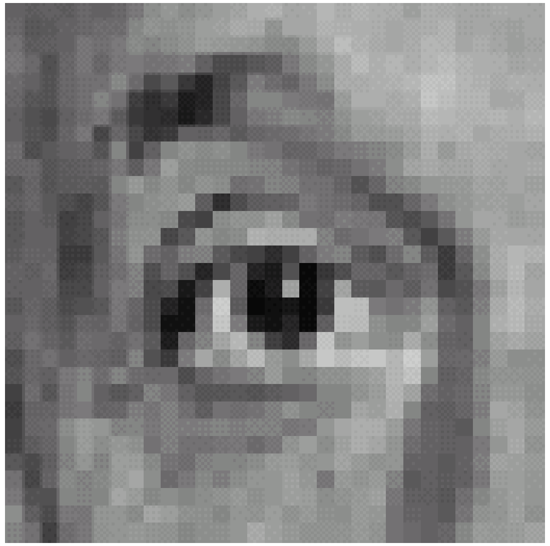


original



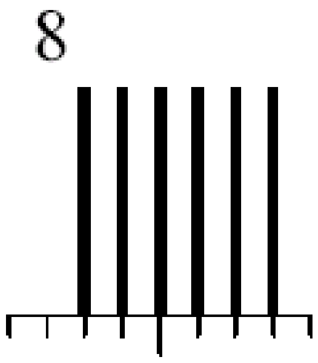
−



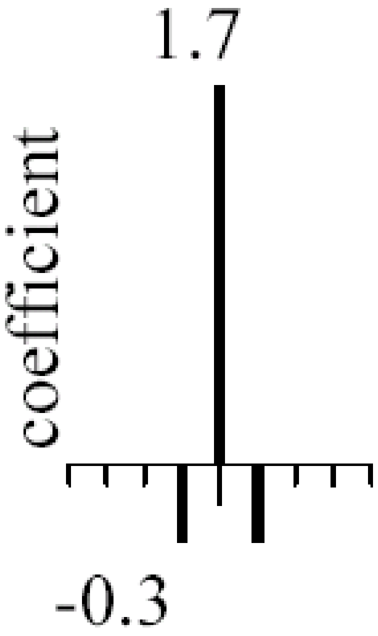


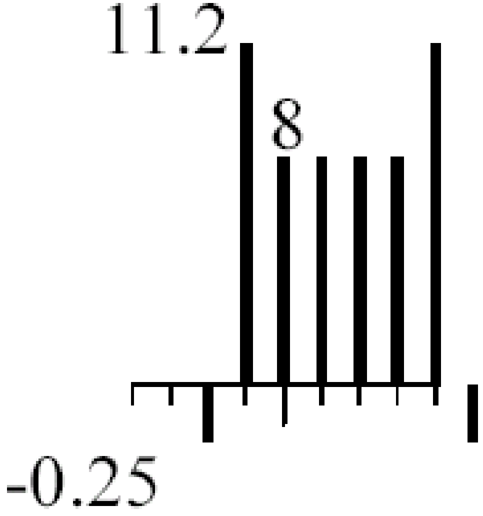
Sharpened
original

Sharpening example



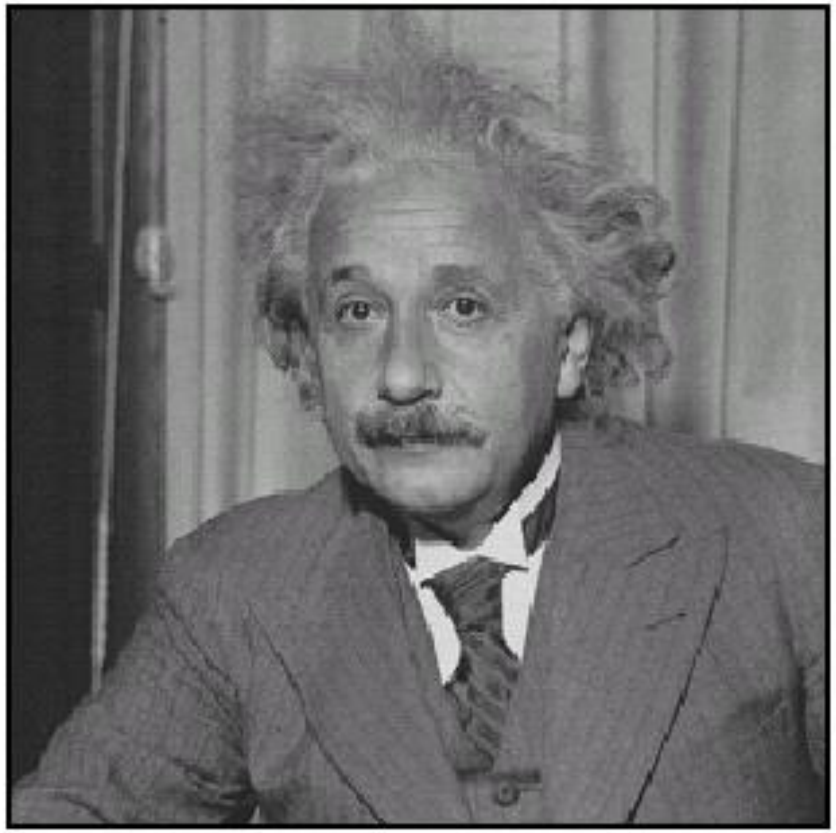
original



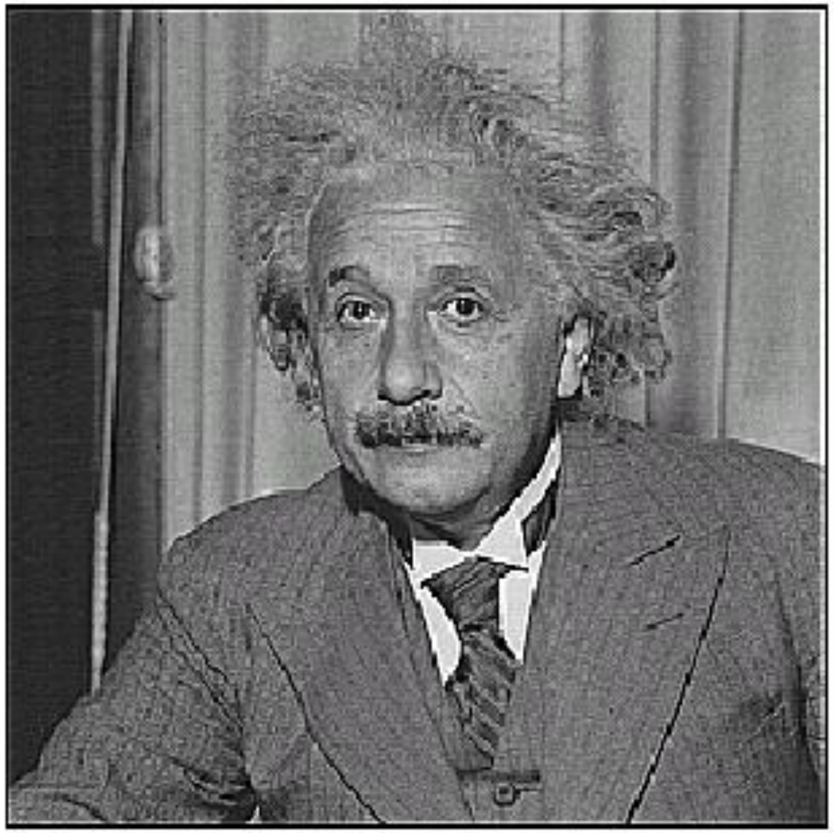


Sharpened
(differences are
accentuated; constant
areas are left untouched).

Sharpening



before



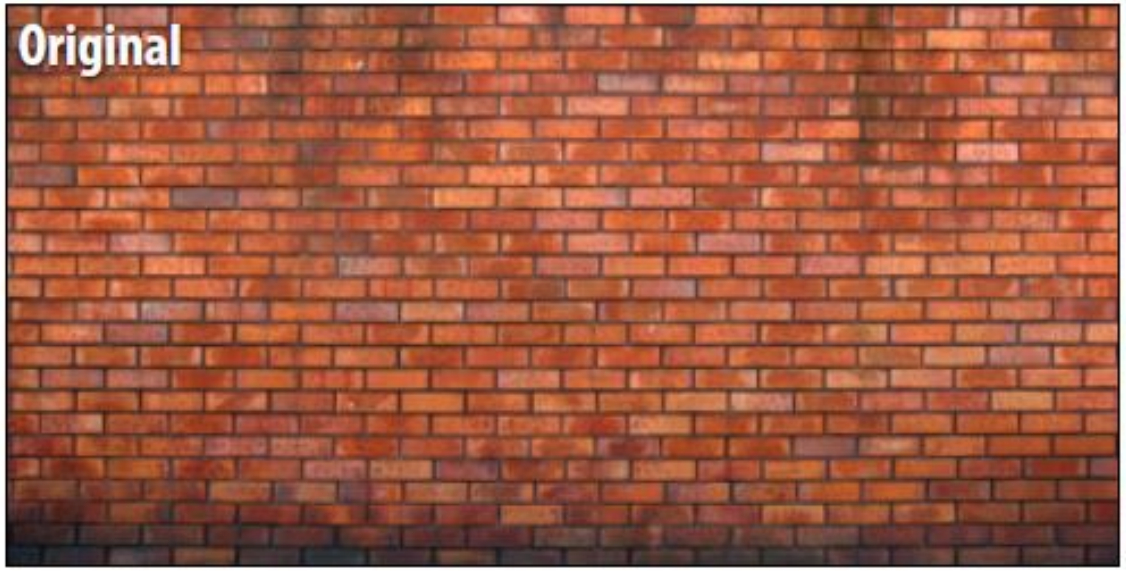
after

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

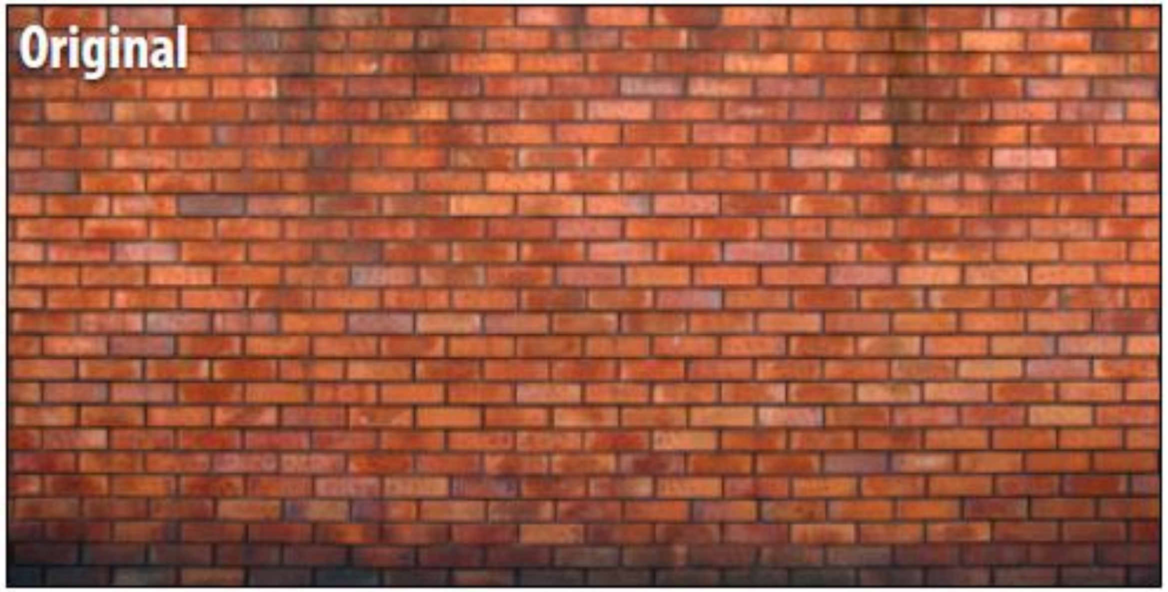
- **Obtain filter coefficients from sampling 2D Gaussian**

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

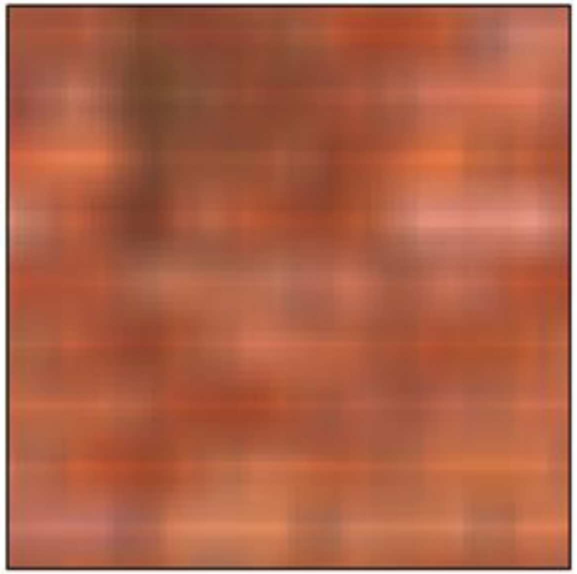
- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **Truncate filter beyond certain distance**

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

7*7 Gaussian blur



7x7 box blur

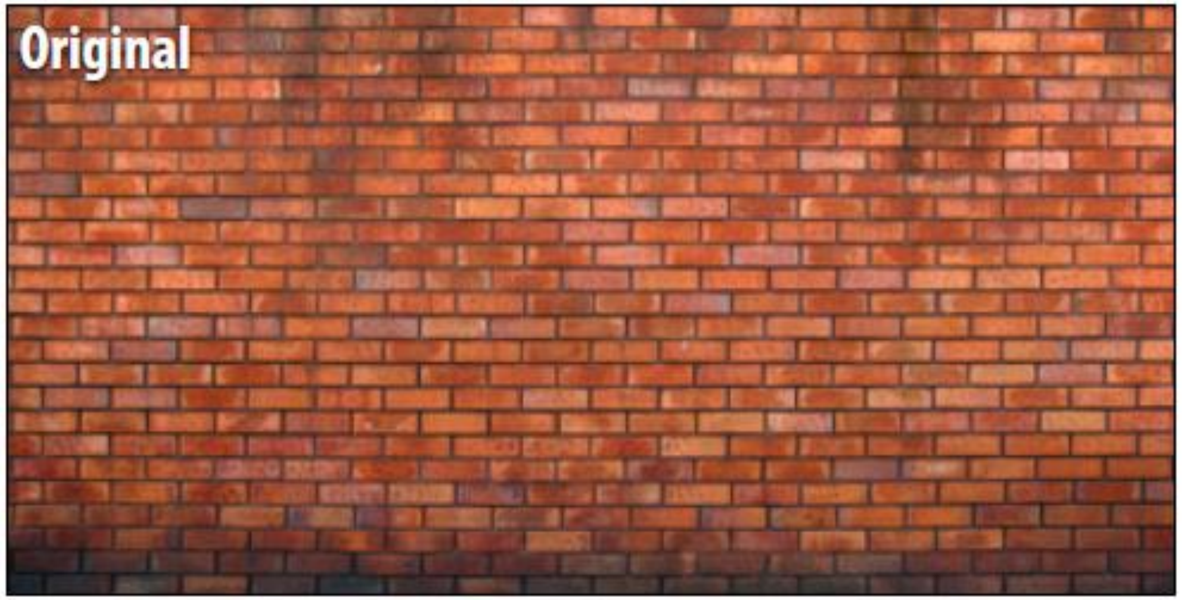


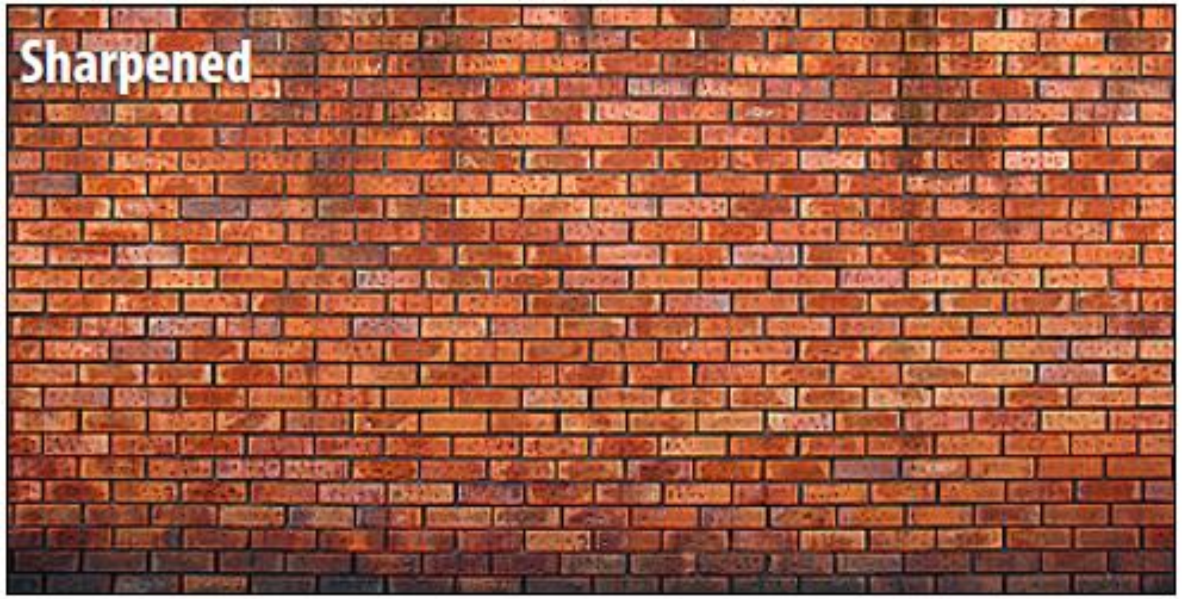
What does convolution with this filter do?

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

Sharpens image!

3x3 sharpen filter





What does convolution with these filters do?

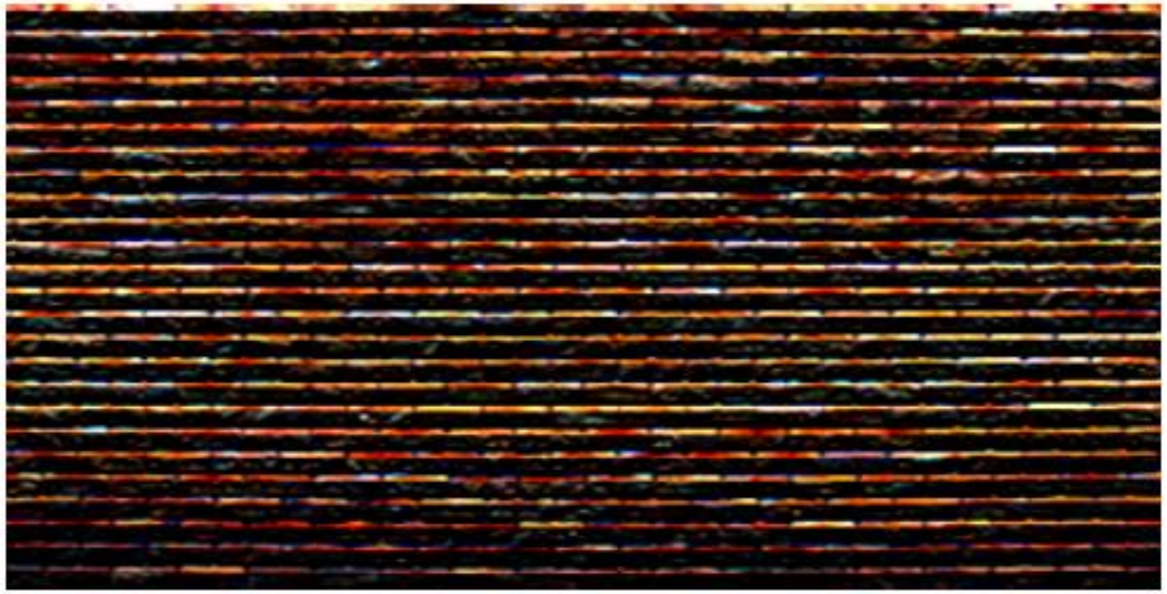
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

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)

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)

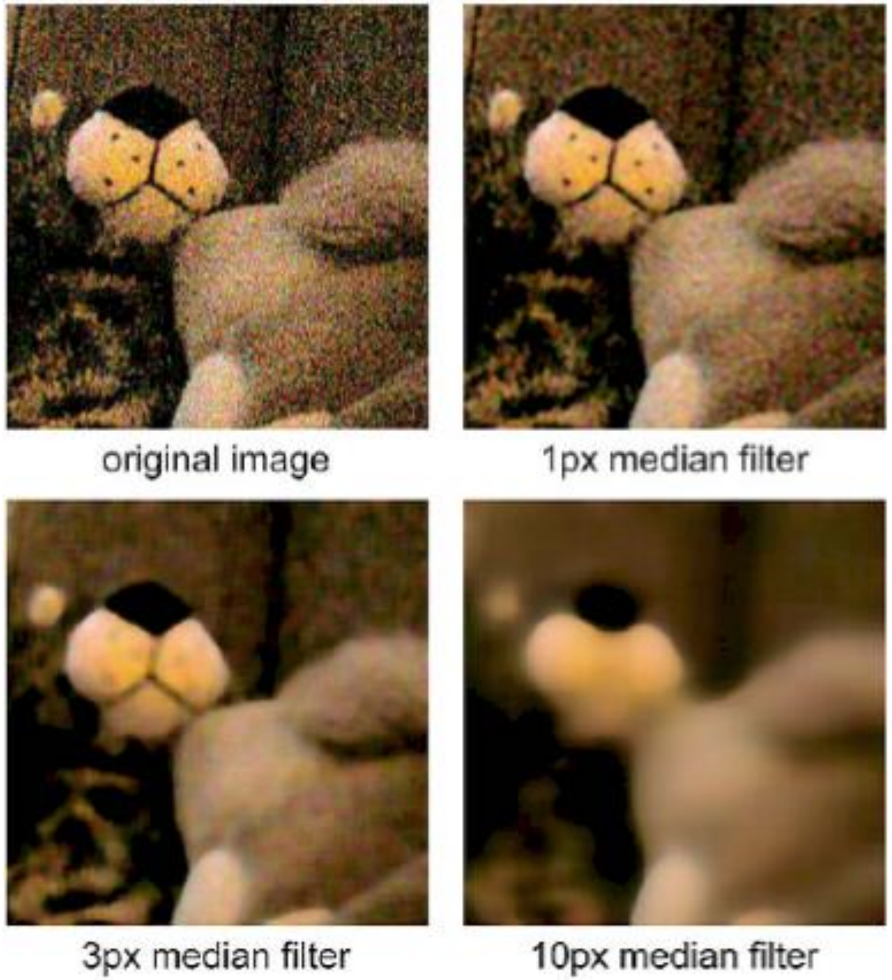
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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
   }
}
```



- **Basic algorithm for NxN support region:**
  - **Sort N2 elements in support region, pick median O(N2log(N2)) work per pixel**