**2065-34**

**Advanced Training Course on FPGA Design and VHDL for Hardware Simulation and Synthesis**

***26 October - 20 November, 2009***

# Two-dimensional digital signal processing

Fabio Mammano

*University of Padua*
*Faculty of Medicine and Surgery*
*Department of Physics*
*via Marzolo 8*
*35131 Padova*
*Italy*

## *Two-dimensional digital signal processing*

## Digital images as matrices

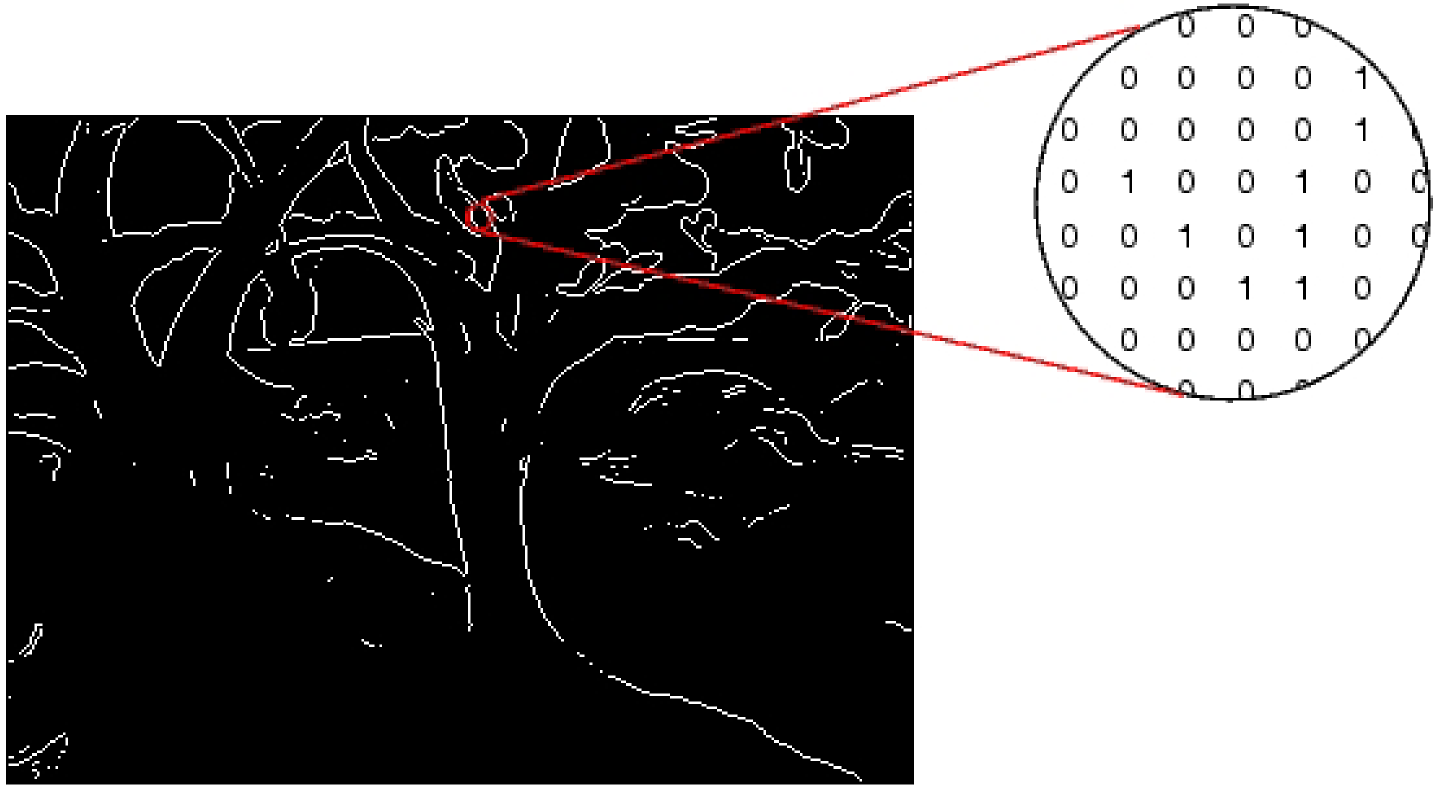
**Images are stored as two-dimensional arrays (i.e., *matrices*), in which each element of the matrix corresponds to a single *pixel* in the displayed image.**

**Pixel is derived from *picture element* and usually denotes a single dot on a computer display. For example, an image composed of 200 rows and 300 columns of different colored dots would be stored as a 200-by-300 matrix.**

**This convention makes working with images similar to working with any other type of matrix data, and makes the full power of matrix manipulation software available for image processing applications. For example, you can select a single pixel from an image matrix using normal matrix subscripting like *M*(2,15). This command returns the value of the pixel at *row* 2, *column* 15 of the image *M*.**

## Representing images as matrices: binary images

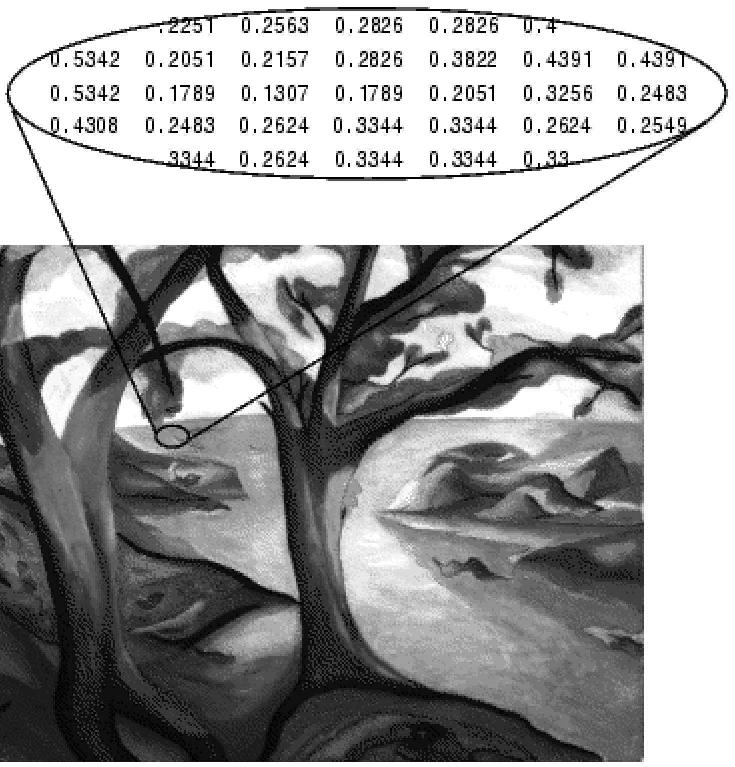
In a binary image, each pixel assumes one of only two discrete values. Essentially, these two values correspond to on and off. A binary image is stored as a logical array of 0's (off pixels) and 1's (on pixels). The figure below depicts a binary image.



## Intensity images

**An intensity image is a data matrix, *M*, whose values represent intensities within some range, with each element of the matrix corresponding to one image pixel. The matrix can be of class *double*, *uint8*, or *uint16*.**

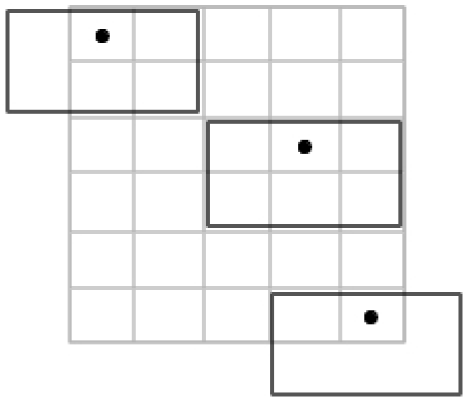
**The elements in the intensity matrix represent various *intensities*, or *gray levels*, where the intensity 0 usually represents black and the intensity 1, 255, or 65535 usually represents full intensity, or white. The figure below depicts an intensity image of class double.**



```
        .2251   0.2563  0.2826  0.2826  0.4
0.5342  0.2051  0.2157  0.2826  0.3822  0.4391  0.4391
0.5342  0.1789  0.1307  0.1789  0.2051  0.3256  0.2483
0.4308  0.2483  0.2624  0.3344  0.3344  0.2624  0.2549
        3344    0.2624  0.3344  0.3344  0.33
```

## Sliding neighborhood operations

**A *sliding neighborhood operation* is an operation that is performed *a pixel at a time*, with the value of any given pixel in the output image being determined by applying some algorithm to the values of the corresponding input pixel's *neighborhood*. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel, which is called the *center pixel*. The neighborhood is a *rectangular block*, and as you move from one element to the next in an image matrix, the neighborhood block *slides* in the same direction.**



**The figure shows the neighborhood blocks for some of the elements in a 6-by-5 matrix with 2-by-3 sliding blocks. The center pixel for each neighborhood is marked with a dot.**

**The center pixel is the actual pixel in the input image being processed by the operation. If the neighborhood has an odd number of rows and columns, the center pixel is actually in the center of the neighborhood. If one of the dimensions has even length, the center pixel is just to the left of center or just above center. For example, in a 2-by-2 neighborhood, the center pixel is the upper left.**

**Notice that, in the previous figure, the upper left and bottom right neighborhoods include "pixels" that are not part of the image. To process these neighborhoods, sliding neighborhood operations pad the borders of the image, usually with 0's.**

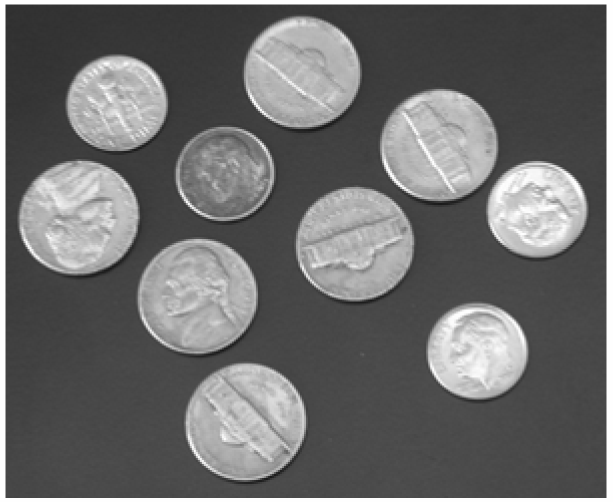
**In other words, these functions process the border pixels by assuming that the image is surrounded by additional rows and columns of 0's. These rows and columns do not become part of the output image and are used only as parts of the neighborhoods of the actual pixels in the image.**

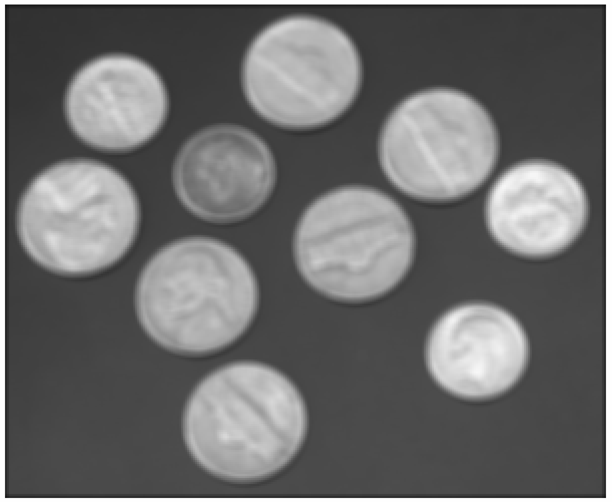
**To perform a sliding neighborhood operation:**

1. **Select a single pixel.**
2. **Determine the pixel's neighborhood.**
3. **Apply a function to the values of the pixels in the neighborhood. This function must return a scalar.**
4. **Find the pixel in the output image whose position corresponds to that of the center pixel in the input image.**
5. **Set this output pixel to the value returned by the function.**
6. **Repeat steps 1 through 4 for each pixel in the input image.**

## Averaging filter

For example, consider an *averaging* operation. The function might sum the values of the *N* neighborhood pixels and then divide by *N*. The result is the value of the output pixel.

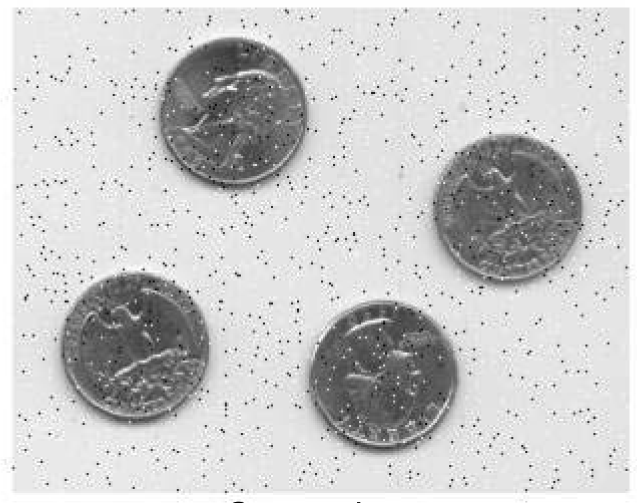

Source image

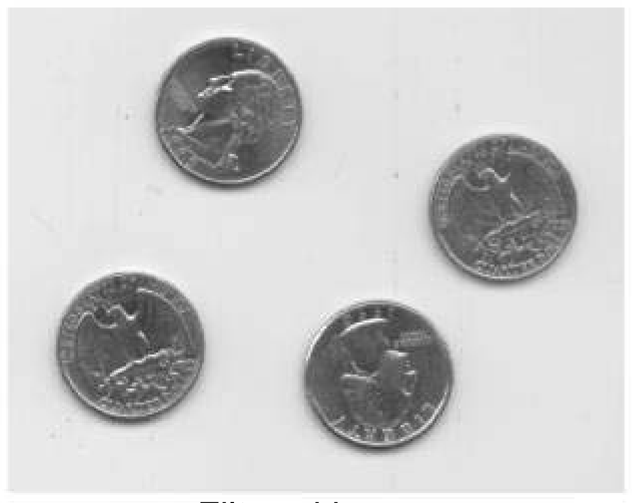

Filtered image

This example filters an image with a 5-by-5 pixel neighborhood containing equal weights. Such a filter is often called an averaging filter.

## Median filter

Median filtering (MF) is similar to using an averaging filter, but with MF the value of an output pixel is determined by the median of the neighborhood pixels, rather than the mean.


Source image


Filtered image

The median is much less sensitive than the mean to extreme values (called outliers). MF is therefore better able to remove these outliers without reducing the sharpness of the image. MF is an example of non-linear filtering applied through a sliding neighborhood operation.

## Convolution in two dimensions

**2D linear filtering is applied to digital images through a sliding neighborhood operations of two dimensional *convolution*.**

**The value of an output pixel is computed as a weighted sum of neighboring pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter* or *point spread function*.**

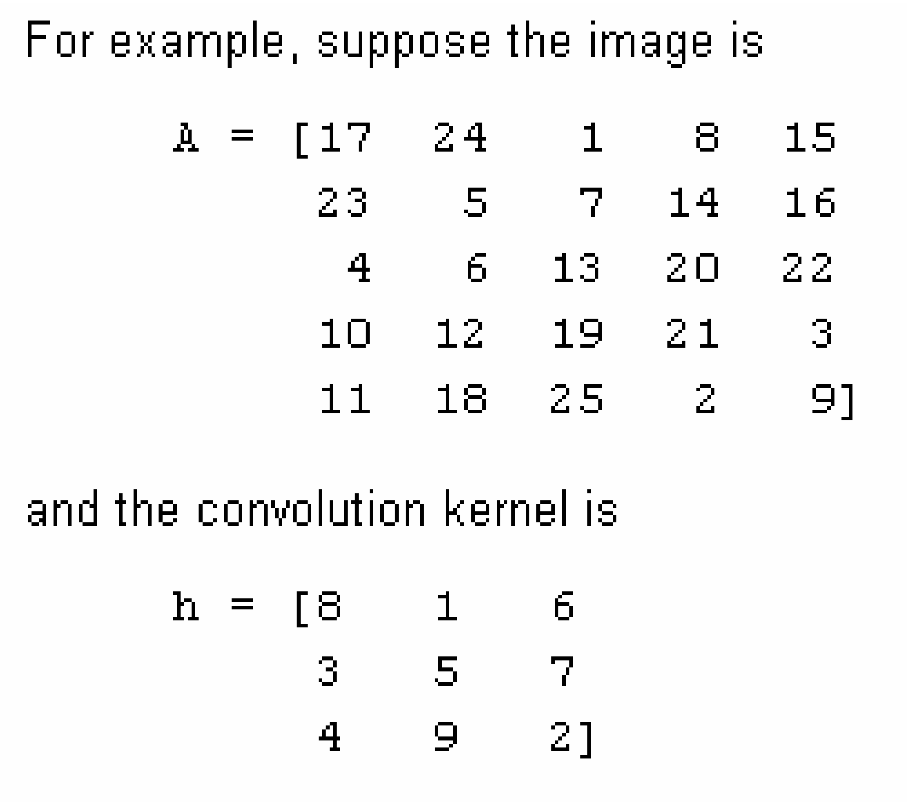
$$g = f \otimes h$$

$$\Downarrow$$

$$g(x,y) = \sum_{j,k} f(j,k)\,h(x-j,y-k)$$

**Notice that the filter kernel can itself be considered as an image, even if its actual meaning is more a collection of topologically ordered coefficients (i.e. they are not intensity values, but rather the coefficients that linearly combine image intensity values).**

**Linear filtering essentially involves only products and sums, and is therefore easily and efficiently implemented digitally. Notice, though, that unlike point transformations that could be performed "on place" using the same image both for the original and the results, image filtering needs two separate locations for original image and resulting image.**



For example, suppose the image is

```
A = [17  24   1   8  15
     23   5   7  14  16
      4   6  13  20  22
     10  12  19  21   3
     11  18  25   2   9]
```

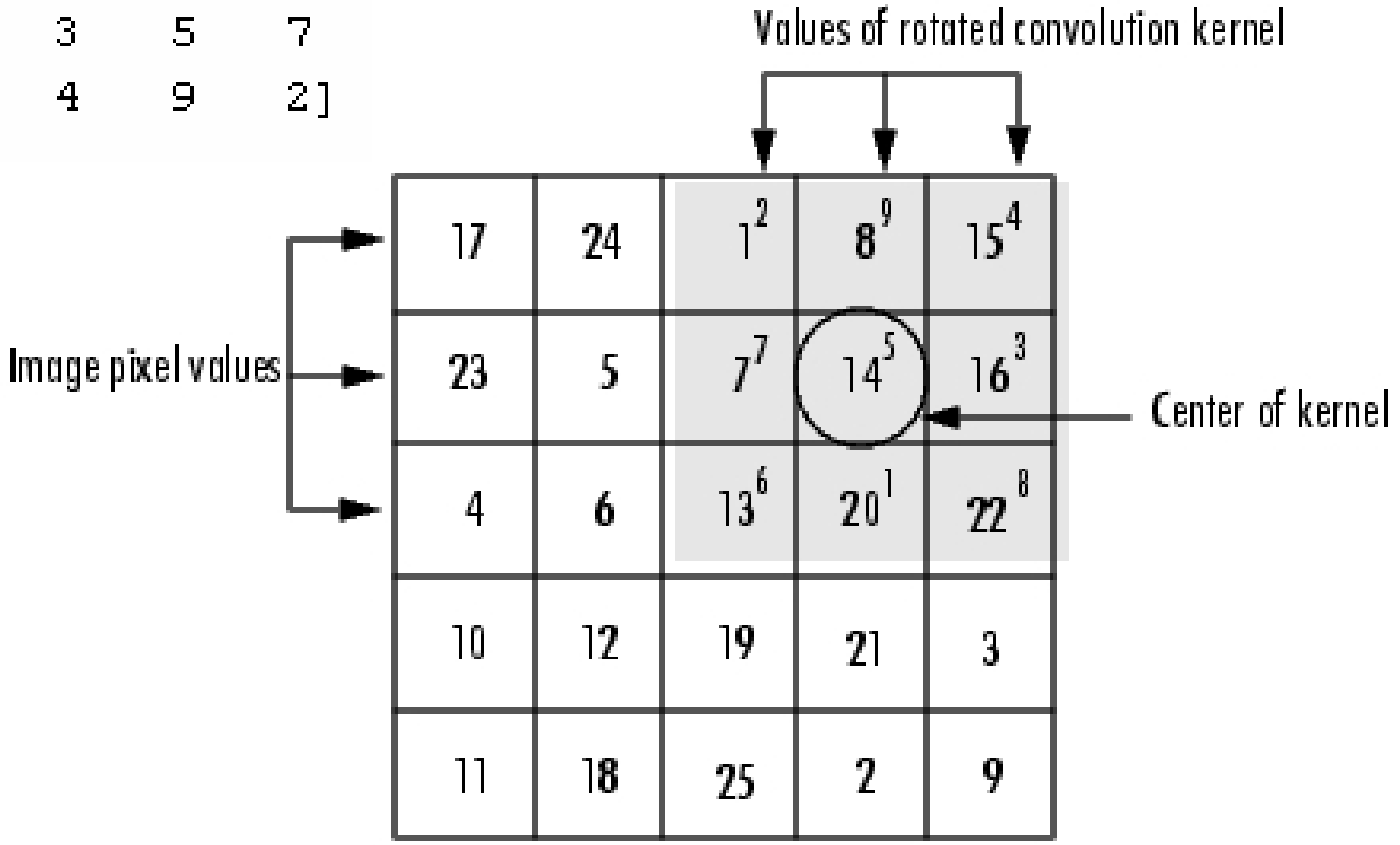
and the convolution kernel is

```
h = [8   1   6
     3   5   7
     4   9   2]
```

**To compute the (*m*,*n*) output pixel we need to:**

- **Rotate the convolution kernel 180 degrees about its center element.**
- **Slide the center element of the convolution kernel so that it lies on top of the (*m*,*n*) element of A.**
- **Multiply each weight in the rotated convolution kernel by the pixel of A underneath and sum up the individual products.**

```
h = [8   1   6
     3   5   7
     4   9   2]
```

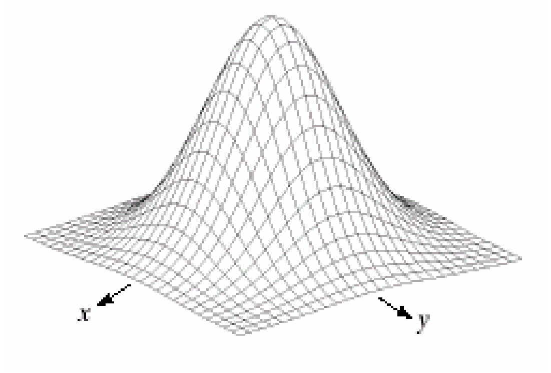


**For example, the output pixel (2, 4) is:**

$$1 \cdot 2 + 8 \cdot 9 + 15 \cdot 4 + 7 \cdot 7 + 14 \cdot 5 + 16 \cdot 3 + 13 \cdot 6 + 20 \cdot 1 + 22 \cdot 8 = 575$$

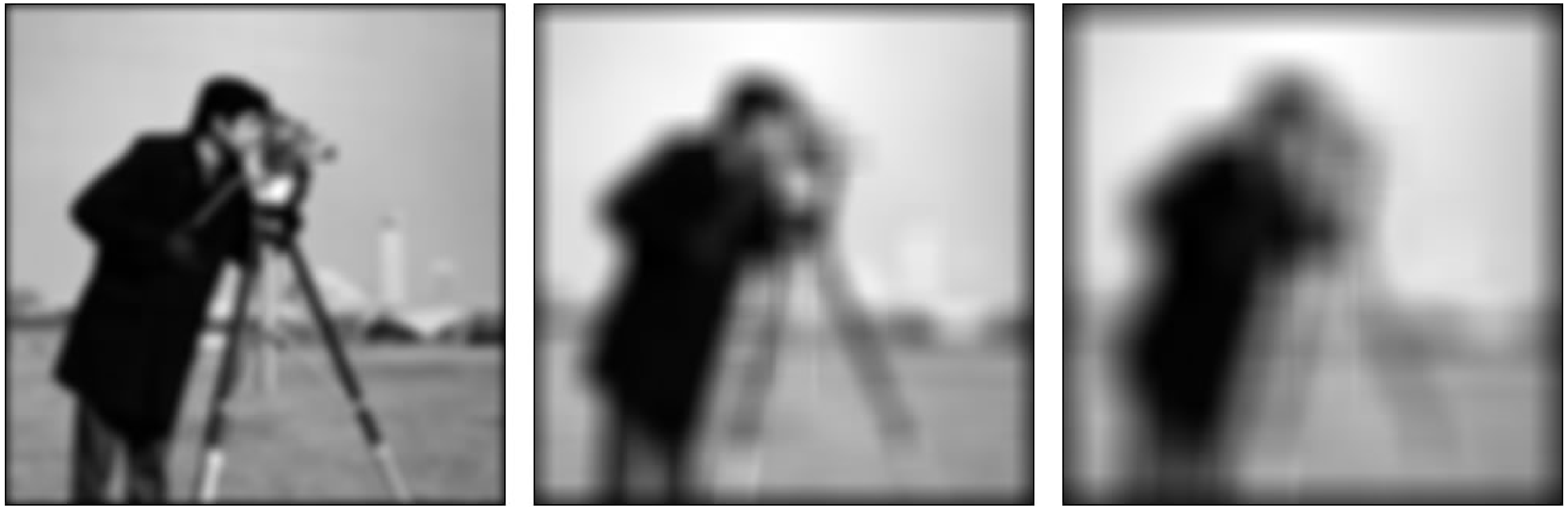
## Gaussian kernel

Source image



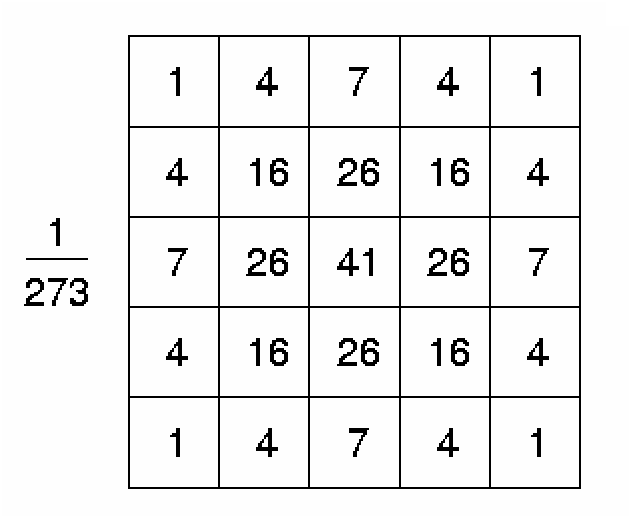
$$h(m,n)=\frac{1}{K}e^{-\frac{1}{2}\left(\frac{m^2+n^2}{\sigma^2}\right)}$$

Filtered images



**Photographer picture convolved with kernels having σ = (from left to right) 3, 11, 20.**

**In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point.**



$\frac{1}{273}$

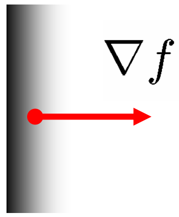
| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 4 | 1 |
| 4 | 16 | 26 | 16 | 4 |
| 7 | 26 | 41 | 26 | 7 |
| 4 | 16 | 26 | 16 | 4 |
| 1 | 4 | 7 | 4 | 1 |

**Shown is a suitable integer-valued convolution kernel that approximates a Gaussian with a $\sigma$ of 1.0.**

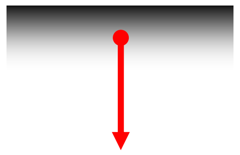
## Image gradient

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

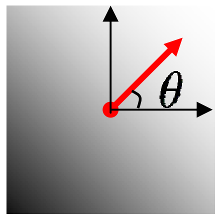
**The gradient points in the direction of most rapid change in image intensity**



$$\nabla f = \left[\frac{\partial f}{\partial x}, 0\right]$$



$$\nabla f = \left[0, \frac{\partial f}{\partial y}\right]$$



$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} \Big/ \frac{\partial f}{\partial x}\right)$$

**Possible discrete approximations are provided by finite differences such as**

$$\frac{\partial f(x,y)}{\partial x} \approx f(x+1,y) - f(x,y)$$

$$\frac{\partial f(x,y)}{\partial y} \approx f(x,y+1) - f(x,y)$$

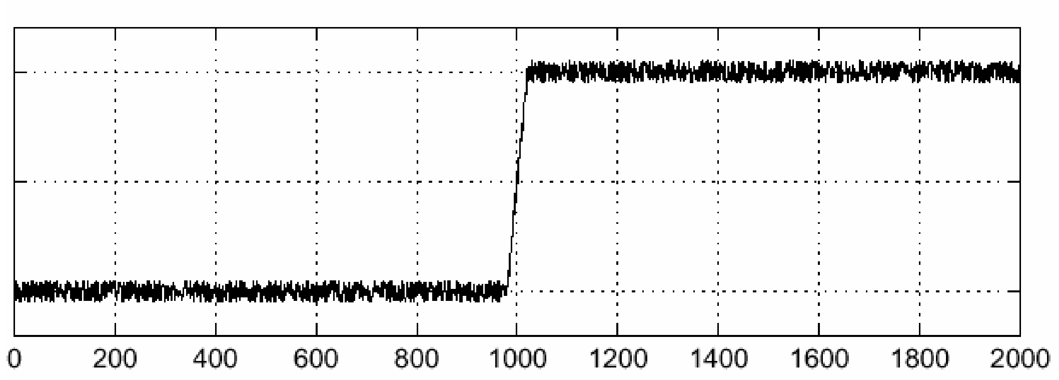
**The corresponding 2×2 convolution kernels are known as the <u>Roberts operators</u>**

$$h_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad h_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

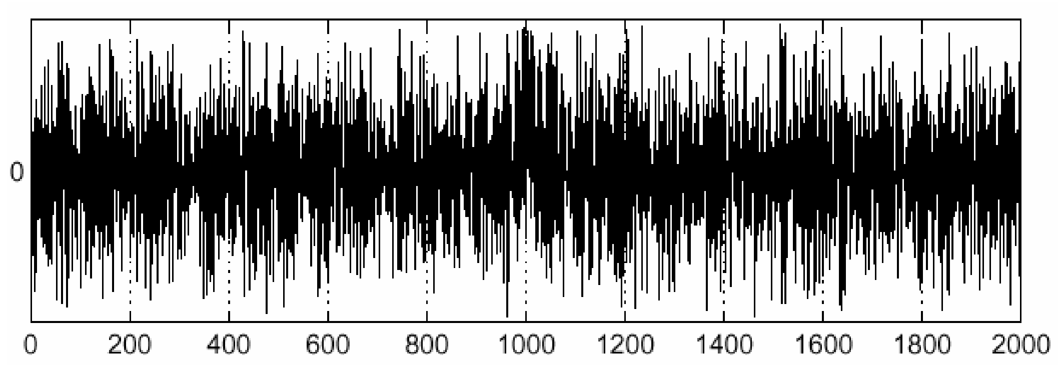
## Edge detection: Effect of noise

- Consider a single row or column of the image
  - Plotting intensity as a function of position gives a signal

$f(x)$



$\frac{\partial f(x)}{\partial x}$



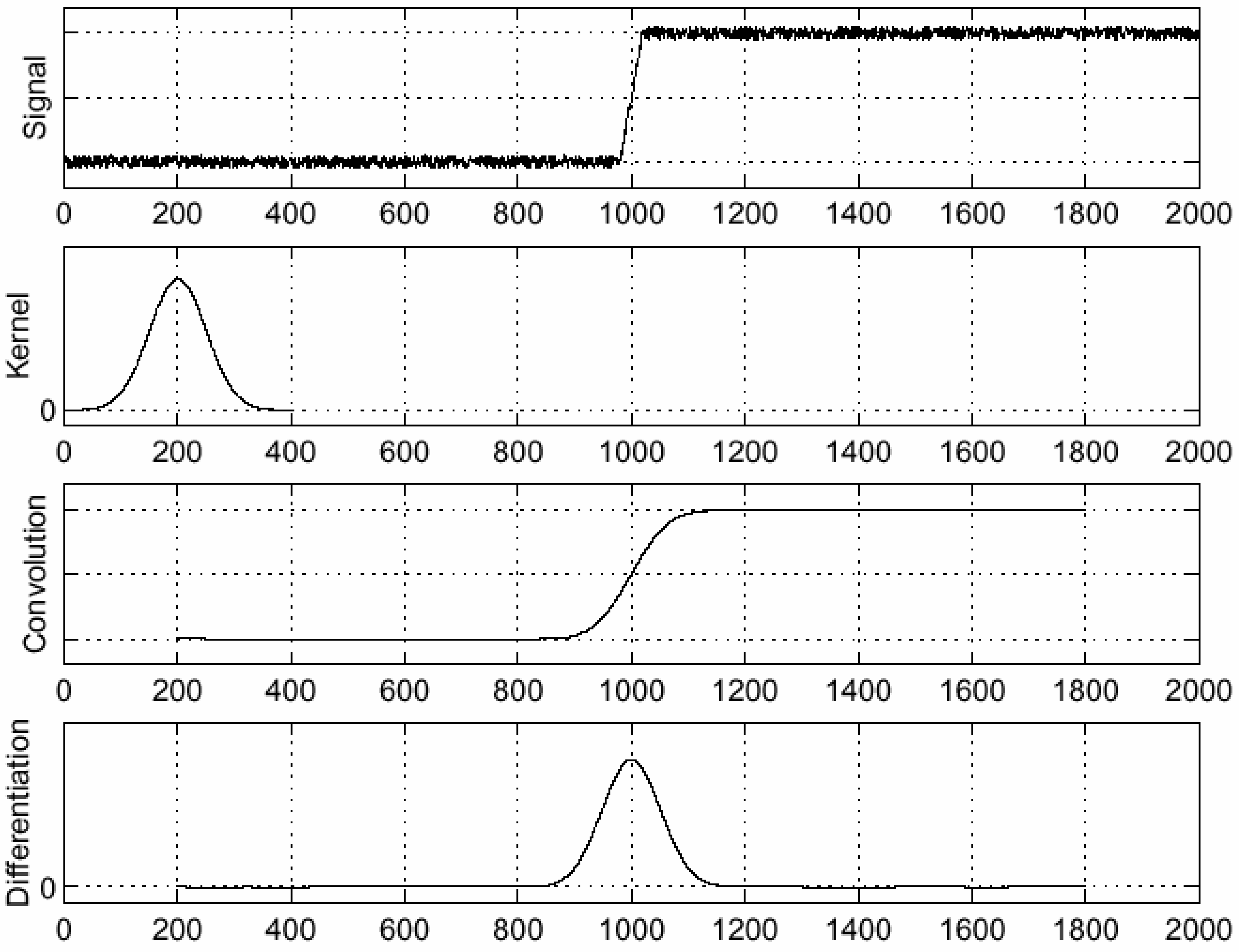
## Solution: smooth first

$f$

$h$

$g = f \otimes h$

$\dfrac{\partial g}{\partial x}$



## The Sobel operator

**If we define A as the source image, and $G_x$ and $G_y$ are two images which at each point contain the horizontal and vertical derivative approximations, the computations are as follow:**

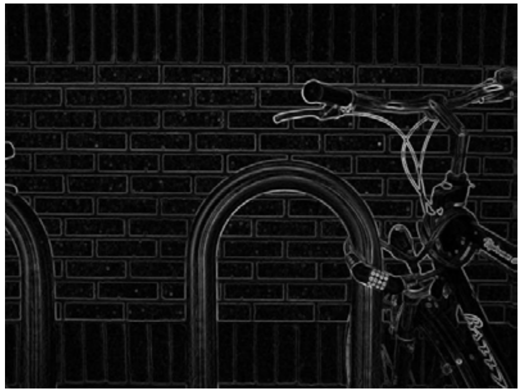
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \otimes \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \otimes \mathbf{A}$$

**Convolution with the 3×3 kernels above are slower to compute than with the smaller Roberts operators, but the larger size smooths the image to a greater extent, making it less sensitive to noise. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, or edge strength, using:**

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

**The result of the Sobel operator G is a 2–dimensional map of the gradient at each point that can be processed and viewed as though it is itself an image, with the areas of high gradient (the likely edges) visible as white lines.**

Source image


Filtered image

## Filtering in the frequency domain

An alternative to spatial domain filtering is to implement the filter in the frequency domain. Recall that, if

$$c(x,y) = g(x,y) \otimes h(x,y)$$

after Fourier transformation (*F*) the convolution relationships takes the form

$$\boxed{C(f_X, f_Y) = G(f_X, f_Y)\,H(f_X, f_Y)}$$

where

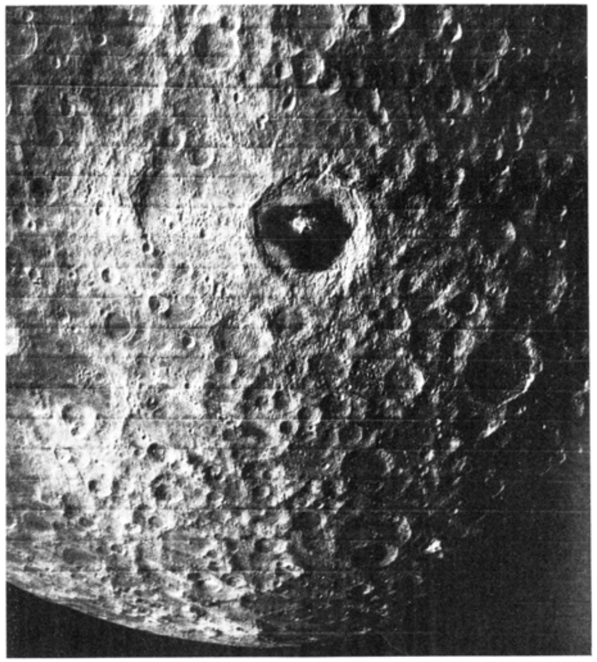
$$C(f_X, f_Y) = F\{c(x,y)\},\; G(f_X, f_Y) = F\{g(x,y)\}$$

Thus, it is multiplication by the <u>Fourier transform of the kernel function</u>

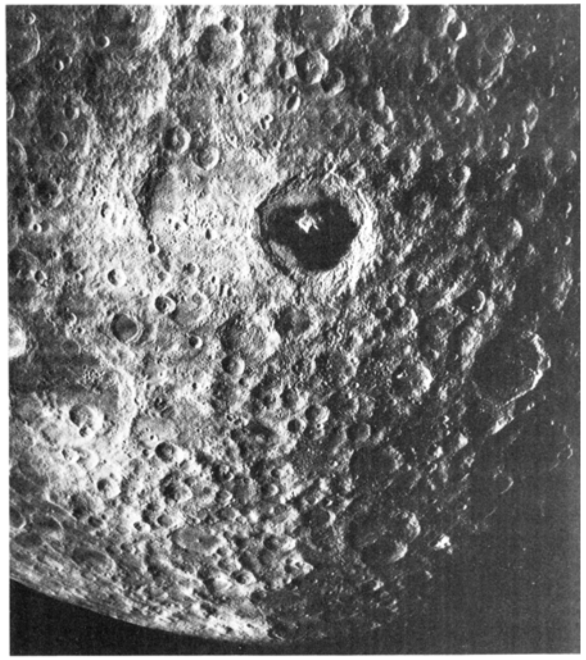
$$H(f_X, f_Y) = F\{h(x,y)\}$$

that produces the alteration in the *frequency spectrum* of the input converting it into that of the output spectrum (*filtering*).

Source image


(a)

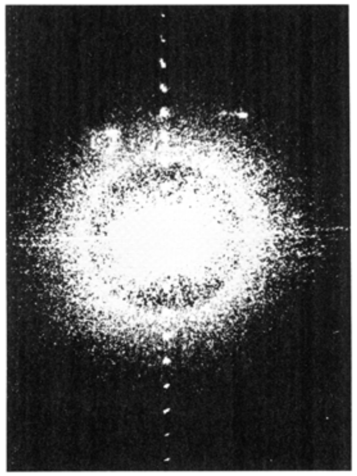
Filtered image

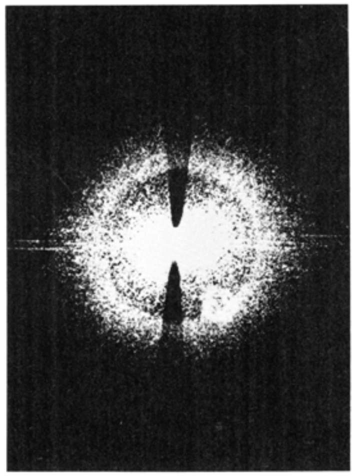

(b)

**The slide in (a) is damaged by horizontal scratches (maybe you were a careless user).**

**The amplitude of the two dimensional Fourier transform is shown in (*c*).**

**The grating-like horizontal line pattern generate the broad-bandwidth, vertical-frequency distribution evident in (c).**


(c)


(d)

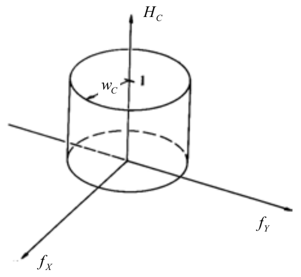
**These frequency components can be selectively deleted (*d*); the resulting image is shown in (*b*).**

**This process of altering the freqency spectrum of the image is known as *spatial filtering*.**

## Low pass and high pass filter transfer functions

**An *ideal low pass filter* is designed by assigning a (radial frequency) cutoff value $w_c$ such that**

$$H_C\left(f_X, f_Y\right) = \mathrm{cyl}\left(\frac{\sqrt{f_X^2 + f_Y^2}}{w_C}\right)$$

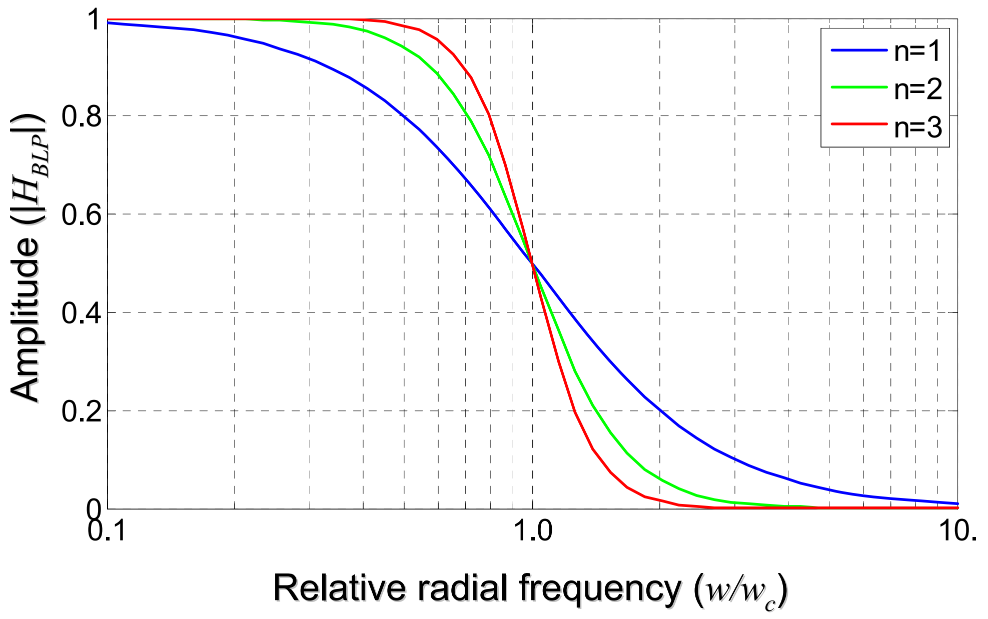


**However, this causes *ringing* artifacts in the spatial domain, therefore filters with smoother roll off are used instead, for example the *Butterworth low pass* filter of order $n$, defined by**

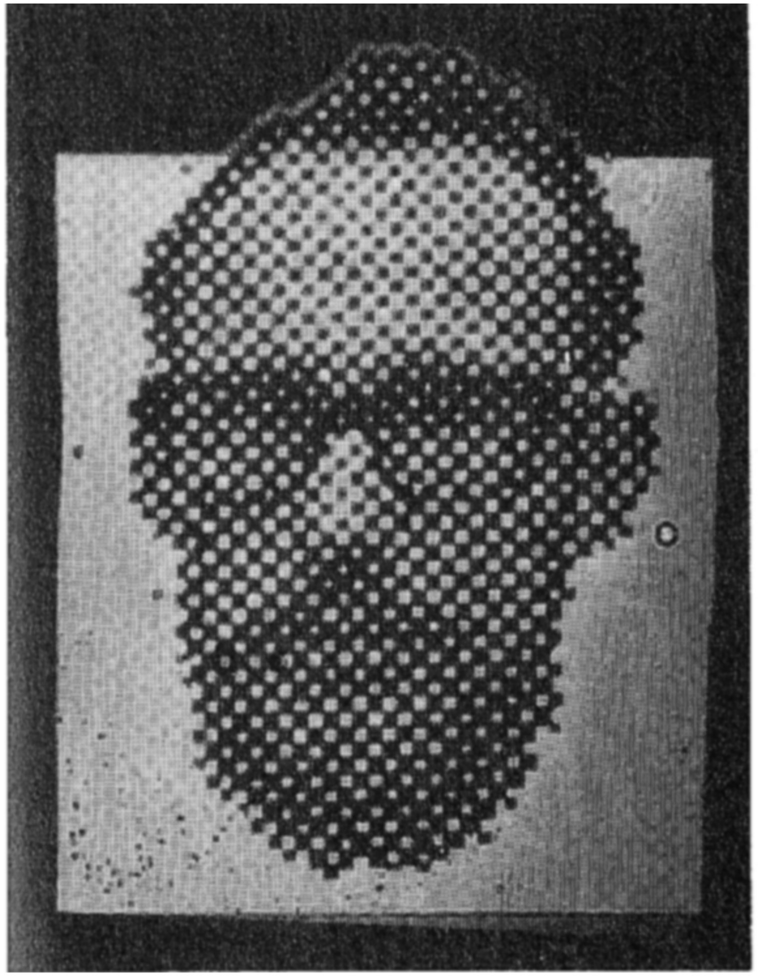
$$H_{BLP}(w) = \frac{1}{1 + \left[w/w_C\right]^{2n}}$$

**where**

$$w = \sqrt{f_X^2 + f_Y^2}$$

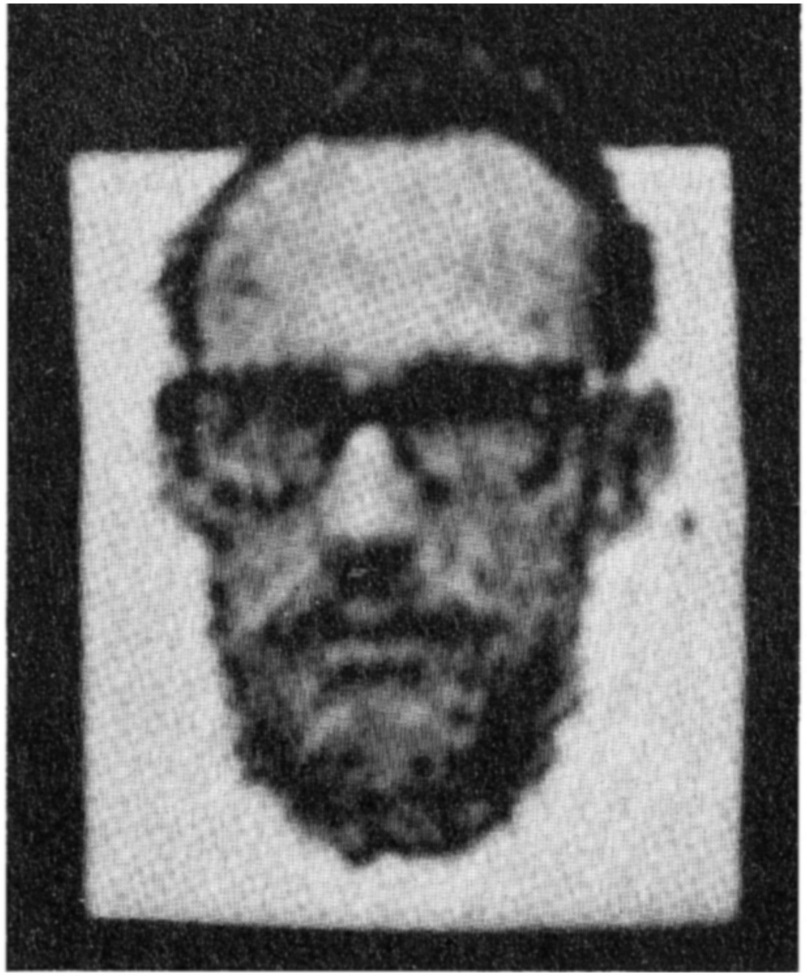


Source image



The object consists of only black and white regions on a halftone.

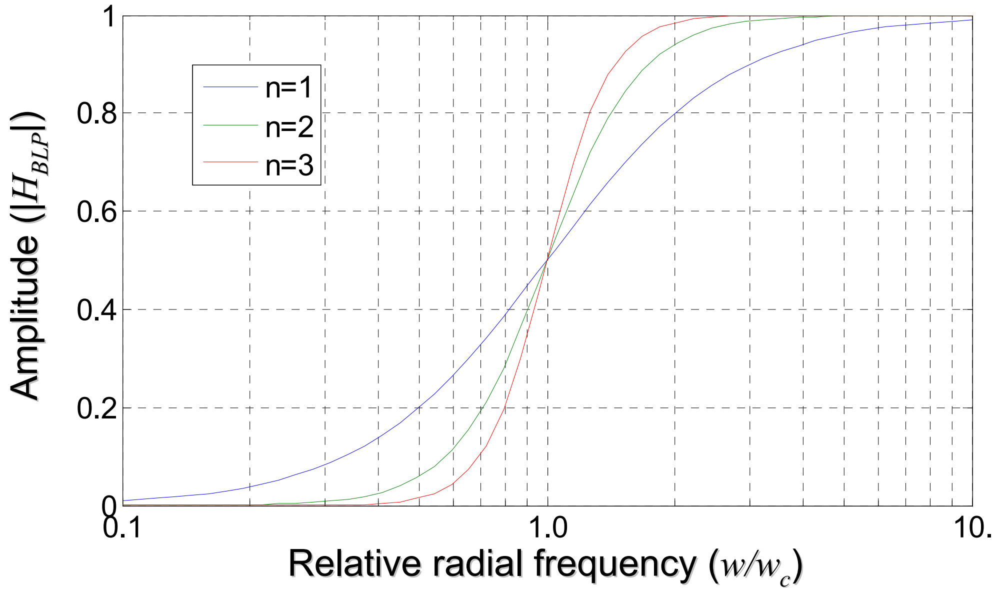
Filtered image



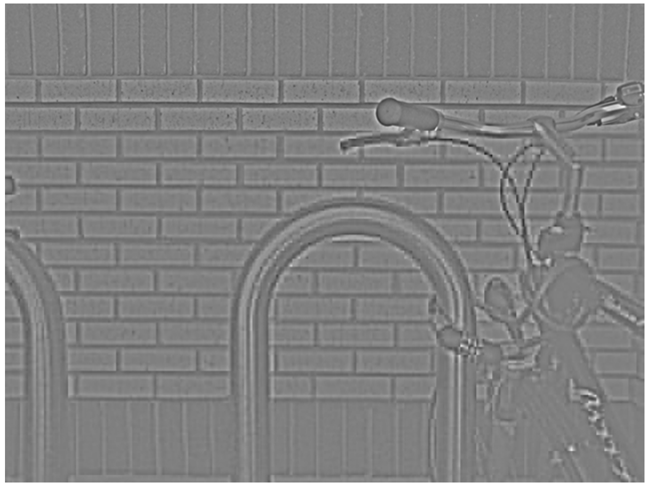
When the high frequencies are filtered out, shades of grey appear and the sharp boundaries vanish.

The corresponding ***Butterworth high pass*** **filter is**

$$H_{BHP}(w) = \frac{1}{1 + [w_C / w]^{2n}}$$



Source image

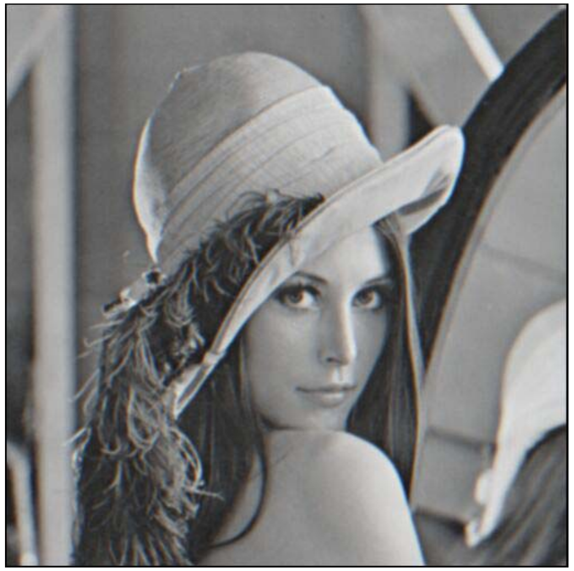

Filtered image

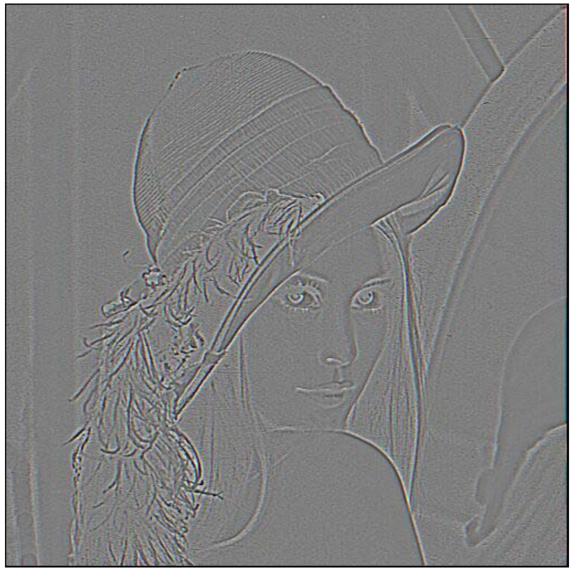
**Note that**

$$H_{BHP}(w) = 1 - H_{BLP}(w)$$

**thus one can always make a high pass filter by subtracting a blur (low pass filter).**

Source image


Blur (5×5 gaussian)


Source – Blur = High pass

**Finally, both type of filters can be applied through convolution**

Examples of discrete kernel masks for spatial filtering

| Low-Pass Filter | High-Pass Filter |
|---|---|
| $w_{i,j} = \frac{1}{10}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ | $w_{i,j} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ |