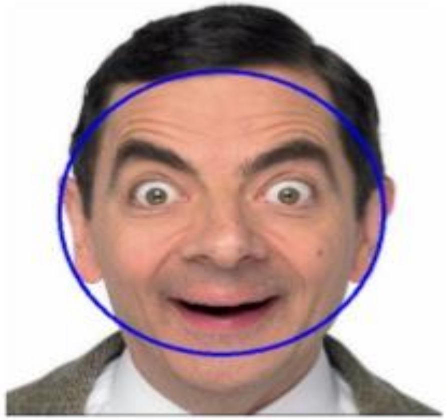
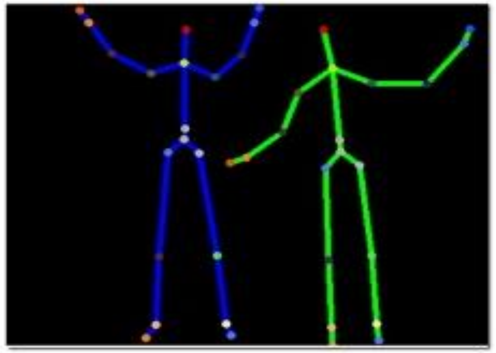
# OpenCV Workshop



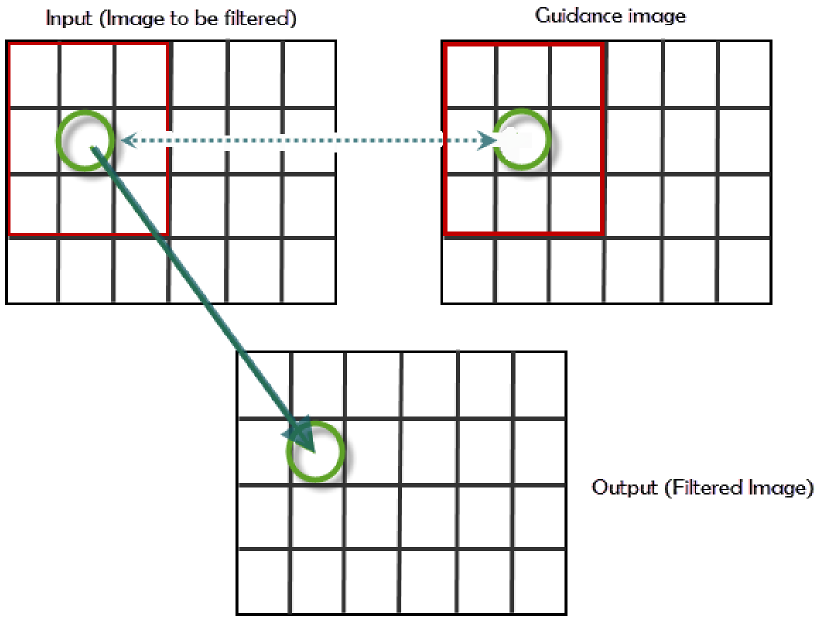


**Session 3: filtering images and image histograms**

**Elham Shabaninia**

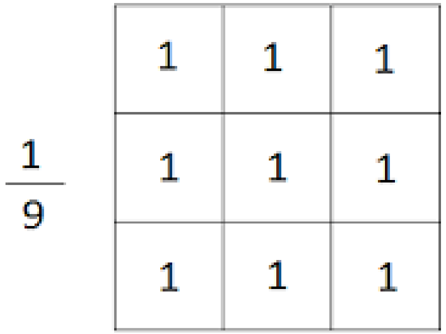
# Filter Images

- The image filtering is a neighborhood operation in which the value of any given pixel in the output image is determined by applying a certain algorithm to the pixel values in the vicinity of the corresponding input pixel.
- This technique is commonly used to smooth, sharpen and detect edges of images and videos.



# Filter Images

- **Kernel**
  Kernel is usually a matrix with an odd height and an odd width. It is also known as convolution matrix, mask or filter.
- Kernel is used to define a neighborhood of a pixel in a image filtering operation. Kernels can be defined with different sizes. But large kernels result in a large processing time.



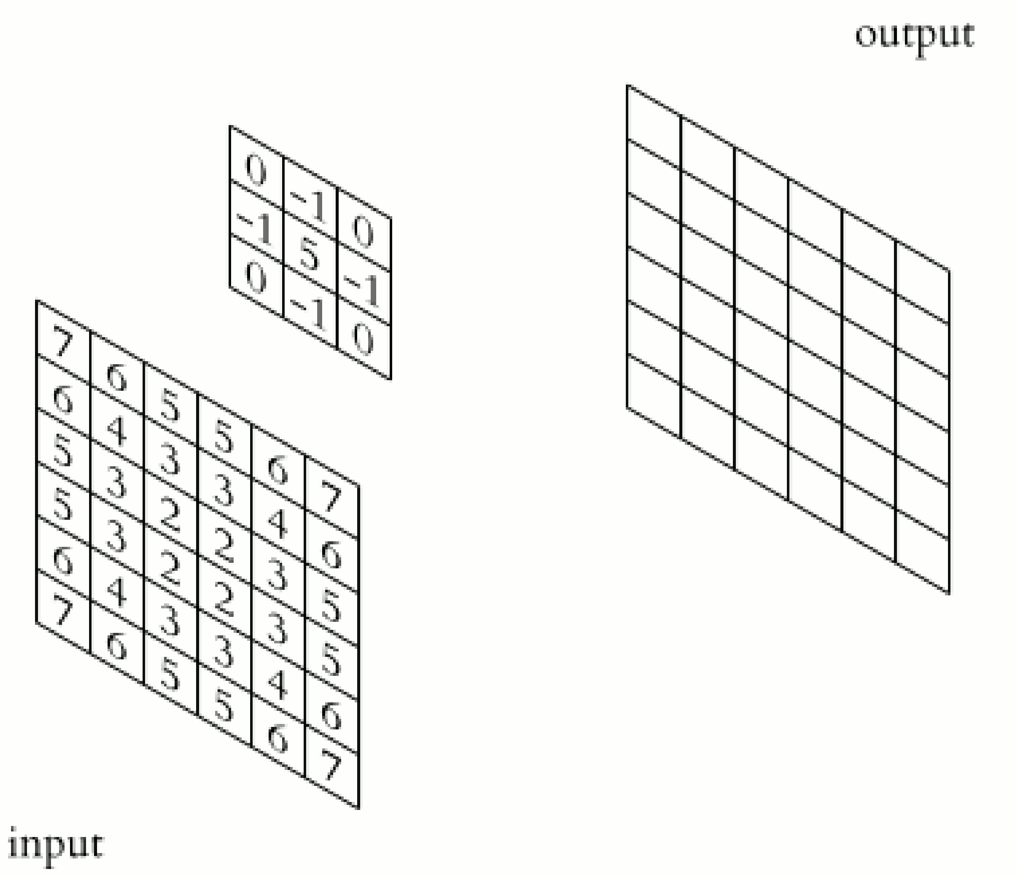
$$\frac{1}{9}\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

a 3 x 3 Normalized Box filter. This kernel is used in homogeneous smoothing (blurring).

# Filter Images

- **Convolution**

Convolution is a mathematical operation performed on images by sliding a kernel across the whole image and calculating new values for each pixels based on the value of the kernel and the value of overlapping pixels of original image.



|   |   |   |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 5 | -1 |
| 0 | -1 | 0 |

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 7 | 6 | 5 | 5 | 6 | 7 |
| 6 | 4 | 3 | 3 | 4 | 6 |
| 5 | 3 | 2 | 2 | 3 | 5 |
| 5 | 3 | 2 | 2 | 3 | 5 |
| 6 | 4 | 3 | 3 | 4 | 6 |
| 7 | 6 | 5 | 5 | 6 | 7 |

# blur

Blurs an image using the normalized box filter.

**Python:**
**cv2.blur(src, ksize, dst, anchor, borderType) → dst**

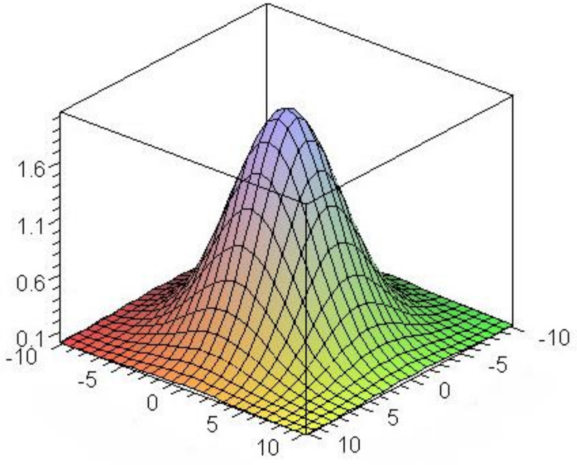
**C++:**
**void blur(InputArray src, OutputArray dst, Size ksize, Point anchor=Point(-1,-1), int borderType=BORDER_DEFAULT )**

- **src** – input image;
- **dst** – output image of the same size and type as src.
- **ksize** – blurring kernel size.
- **anchor** – anchor point; default value is at the kernel center.
- **borderType** – border mode used to extrapolate pixels outside of the image.

$$K = \frac{1}{\mathtt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ & & & \cdots\cdots\cdots & & \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

# GaussianBlur

Blurs an image using a Gaussian filter.

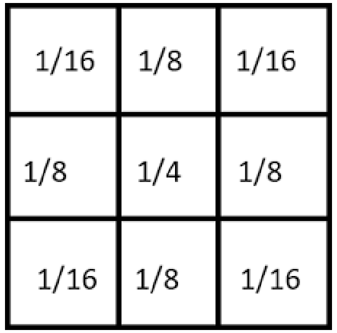


**Python: cv2.GaussianBlur(src, ksize, sigmaX, dst, sigmaY, borderType) → dst**

**C++:**
**void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY=0, int borderType=BORDER_DEFAULT )**

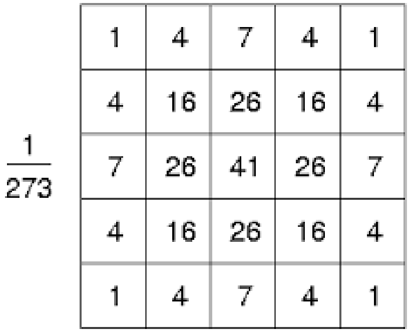
sigmaX – Gaussian kernel standard deviation in X direction.
sigmaY – Gaussian kernel standard deviation in Y direction;

3*3Gaussian kernel



| | | |
|---|---|---|
| 1/16 | 1/8 | 1/16 |
| 1/8 | 1/4 | 1/8 |
| 1/16 | 1/8 | 1/16 |

5*5Gaussian kernel



$\frac{1}{273}$

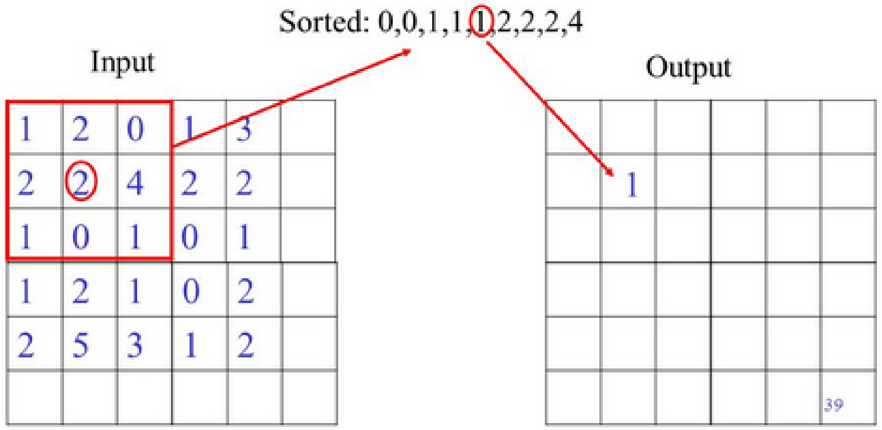
| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 4 | 1 |
| 4 | 16 | 26 | 16 | 4 |
| 7 | 26 | 41 | 26 | 7 |
| 4 | 16 | 26 | 16 | 4 |
| 1 | 4 | 7 | 4 | 1 |

# medianBlur

Blurs an image using the median filter.

**Python: cv2.medianBlur(src, ksize[, dst]) → dst**

**C++: void medianBlur(InputArray src, OutputArray dst, int ksize)**



Sorted: 0,0,1,1,1,2,2,2,4

Input

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 0 | 1 | 3 | |
| 2 | 2 | 4 | 2 | 2 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 2 | 1 | 0 | 2 | |
| 2 | 5 | 3 | 1 | 2 | |
| | | | | | |

Output

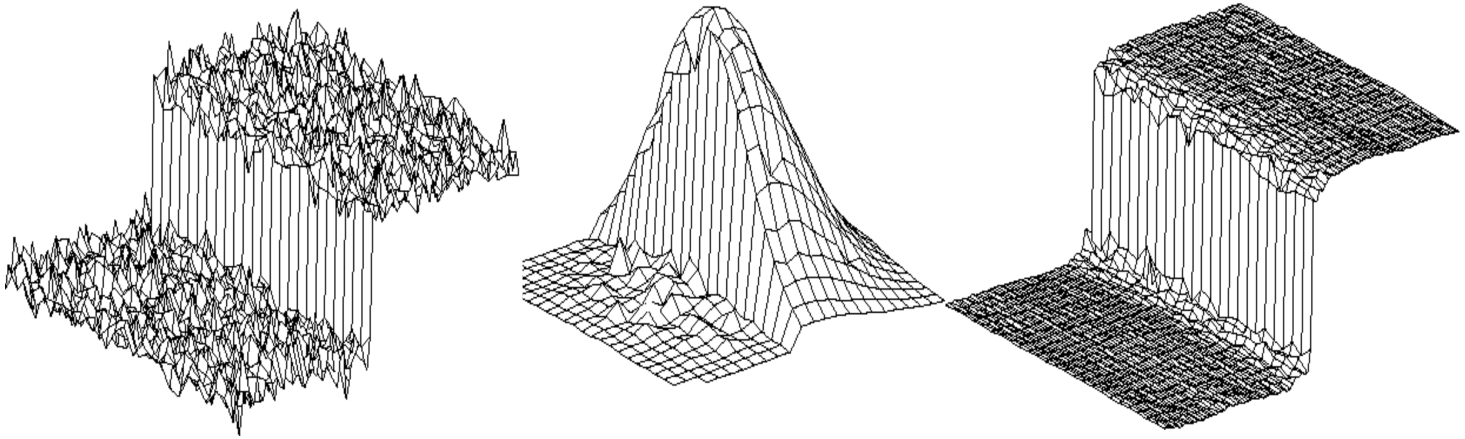
| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | 1 | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | 39 | |

# bilateral filter

Applies the bilateral filter to an image.
bilateralFilter can reduce unwanted noise very well while keeping edges fairly sharp.
However, it is very slow compared to most filters.

**Python: cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace, dst, borderType) → dst**

**C++: void bilateralFilter(InputArray src, OutputArray dst, int d, double sigmaColor, double sigmaSpace, int borderType=BORDER_DEFAULT )**



# Let’s write some code…

Removing noise using Gaussian, median, and bilateral filters

# Image Gradients

we will learn to:

- Find Image gradients, edges etc
- We will see following functions : **cv2.Sobel()**, **cv2.Scharr()**, **cv2.Laplacian()** etc

- The gradian *grad(x,y)* of an image with pixel intensity values *I(x,y)* is given by:

$$grad\,(I) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]^{T}$$

- The Laplacian *L(x,y)* of an image with pixel intensity values *I(x,y)* is given by:

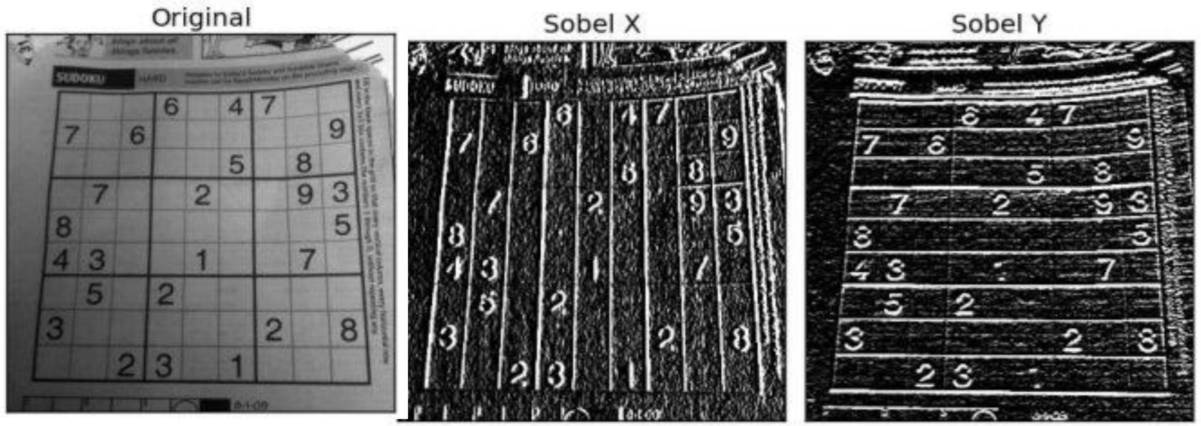
$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

# sobel

Calculates the image derivatives using a sobel operator.

**Python: cv2.Sobel(src, ddepth, dx, dy, dst, ksize, scale, delta, borderType) → dst**

**C++: void Sobel(InputArray src, OutputArray dst, int ddepth, int dx, int dy, int ksize=3, double scale=1, double delta=0, int borderType=BORDER_DEFAULT )**



| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

# Let’s write some code…

Let’s see sobel in a program

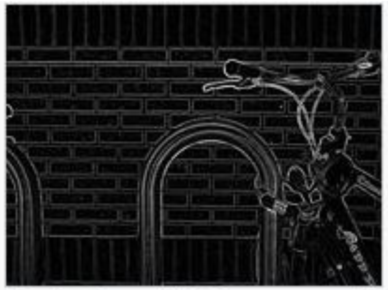
# Scharr

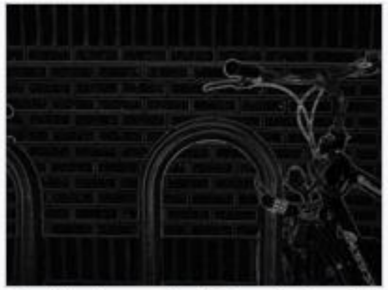
Calculates the first x- or y- image derivative using Scharr operator.

**Python: cv2.Scharr(src, ddepth, dx, dy, dst, scale, delta, borderType) → dst**

**C++: void Scharr(InputArray src, OutputArray dst, int ddepth, int dx, int dy, double scale=1, double delta=0, int borderType=BORDER_DEFAULT )**

Grayscale test image of brick wall and bike rack


Gradient magnitude from Sobel–Feldman operator


Gradient magnitude from Scharr operator

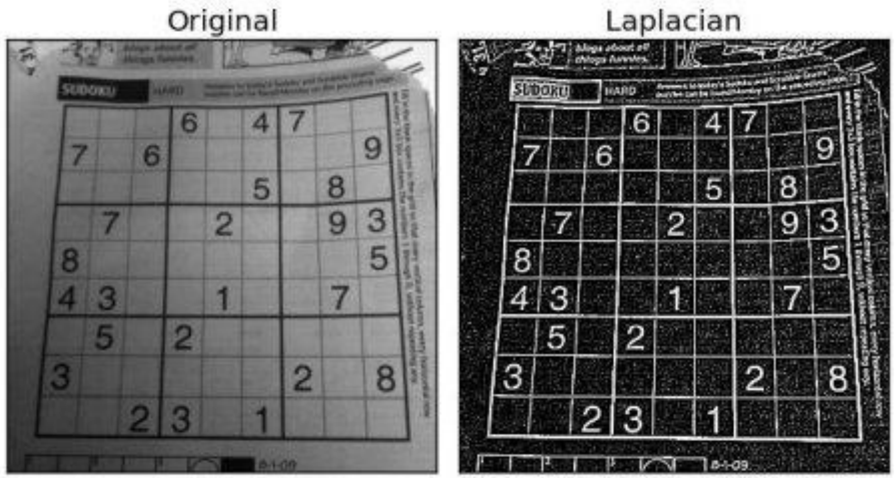
$$\begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \qquad \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

# Laplacian

It calculates the Laplacian of the image given by the relation,

$$\Delta src = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2}$$

where each derivative is found using Sobel derivatives. for 3*3 kernel, then following kernel is used for filtering:



$$kernel = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

# Canny : a popular edge detection algorithm

It is a multi-stage algorithm and we will go through each stages.

- **Noise Reduction**: first step is to remove the noise in the image with a Gaussian filter.
- **Finding Intensity Gradient of the Image**: Smoothened image is then filtered with a Sobel kernel.
- **Non-maximum Suppression**: a full scan of image is done to remove any unwanted pixels which may not constitute the edge.
- **Hysteresis Thresholding:** two threshold values, **threshold1 and threshold2** are considered. Any edges who lie between these two thresholds are classified edges or non-edges based on their connectivity.

**Python: cv2.Canny(image, threshold1, threshold2, edges, apertureSize, L2gradient) → edges**

**C++: void Canny(InputArray image, OutputArray edges, double threshold1, double threshold2, int apertureSize=3, bool L2gradient=false )**

# Let’s write some code…

Let’s see canny in a program…

# You may have your own kernel

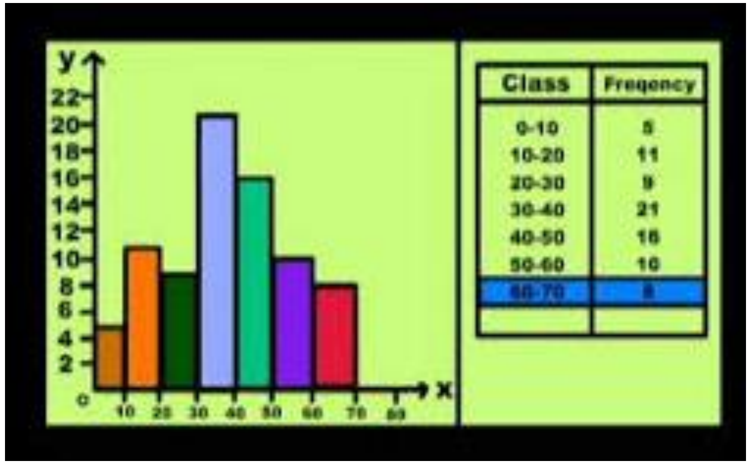
**filter2D** Convolves an image with the kernel.

**Python: cv2.filter2D(src, ddepth, kernel, dst, anchor, delta, borderType) → dst**

**C++: void filter2D(InputArray src, OutputArray dst, int ddepth, InputArray kernel, Point anchor=Point(-1,-1), double delta=0, int borderType=BORDER_DEFAULT )**

# Histogram: a graphical representation of the underlying data

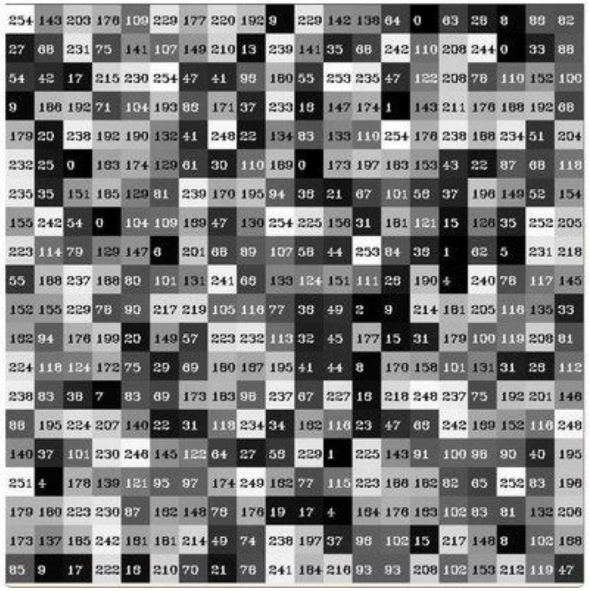
- Histograms are collected *counts* of data organized into a set of predefined *bins*
- When we say *data* we are not restricting it to be intensity values. The data collected can be whatever feature you find useful to describe your image.



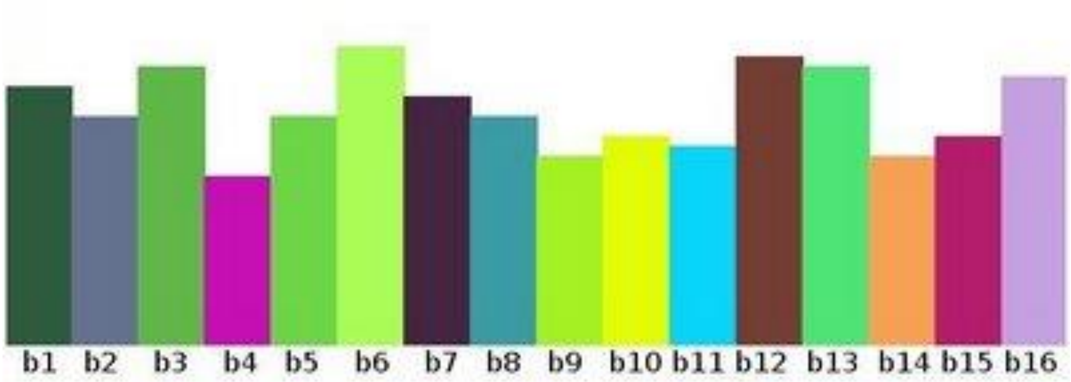
| Class | Freqency |
|---|---|
| 0-10 | 5 |
| 10-20 | 11 |
| 20-30 | 9 |
| 30-40 | 21 |
| 40-50 | 16 |
| 50-60 | 10 |
| 60-70 | 8 |
| | |

# An example

Imagine that a Matrix contains information of an image (i.e. intensity in the range 0-255)



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 254 | 143 | 203 | 176 | 109 | 229 | 177 | 220 | 192 | 9 | 229 | 142 | 138 | 64 | 0 | 63 | 28 | 8 | 88 | 82 |
| 27 | 68 | 231 | 75 | 141 | 107 | 149 | 210 | 13 | 239 | 141 | 35 | 68 | 242 | 110 | 208 | 244 | 0 | 33 | 88 |
| 54 | 42 | 17 | 215 | 230 | 254 | 47 | 41 | 98 | 180 | 55 | 253 | 235 | 47 | 122 | 208 | 78 | 110 | 152 | 100 |
| 9 | 186 | 192 | 71 | 104 | 193 | 88 | 171 | 37 | 233 | 18 | 147 | 174 | 1 | 143 | 211 | 176 | 188 | 192 | 68 |
| 179 | 20 | 238 | 192 | 190 | 132 | 41 | 248 | 22 | 134 | 83 | 133 | 110 | 254 | 176 | 238 | 188 | 234 | 51 | 204 |
| 232 | 25 | 0 | 183 | 174 | 129 | 61 | 30 | 110 | 189 | 0 | 173 | 197 | 183 | 153 | 43 | 22 | 87 | 68 | 118 |
| 235 | 35 | 151 | 185 | 129 | 81 | 239 | 170 | 195 | 94 | 38 | 21 | 67 | 101 | 58 | 37 | 196 | 149 | 52 | 154 |
| 155 | 242 | 54 | 0 | 104 | 109 | 169 | 47 | 130 | 254 | 225 | 156 | 31 | 181 | 121 | 15 | 126 | 35 | 252 | 205 |
| 223 | 114 | 79 | 129 | 147 | 6 | 201 | 68 | 89 | 107 | 58 | 44 | 253 | 84 | 38 | 1 | 62 | 5 | 231 | 218 |
| 55 | 188 | 237 | 188 | 80 | 101 | 131 | 241 | 68 | 133 | 124 | 151 | 111 | 28 | 190 | 4 | 240 | 78 | 117 | 145 |
| 152 | 155 | 229 | 78 | 90 | 217 | 219 | 105 | 116 | 77 | 38 | 49 | 2 | 9 | 214 | 181 | 205 | 116 | 135 | 33 |
| 182 | 94 | 176 | 199 | 20 | 149 | 57 | 223 | 232 | 113 | 32 | 45 | 177 | 15 | 31 | 179 | 100 | 119 | 208 | 81 |
| 224 | 118 | 124 | 172 | 75 | 29 | 69 | 180 | 187 | 195 | 41 | 44 | 8 | 170 | 158 | 101 | 131 | 31 | 28 | 112 |
| 238 | 83 | 38 | 7 | 83 | 69 | 173 | 183 | 98 | 237 | 67 | 227 | 16 | 218 | 248 | 237 | 75 | 192 | 201 | 146 |
| 88 | 195 | 224 | 207 | 140 | 22 | 31 | 118 | 234 | 34 | 182 | 116 | 23 | 47 | 68 | 242 | 189 | 152 | 116 | 248 |
| 140 | 37 | 101 | 230 | 246 | 145 | 122 | 64 | 27 | 58 | 229 | 1 | 225 | 143 | 91 | 100 | 98 | 90 | 40 | 195 |
| 251 | 4 | 178 | 139 | 121 | 95 | 97 | 174 | 249 | 182 | 77 | 115 | 223 | 188 | 182 | 82 | 65 | 252 | 83 | 198 |
| 179 | 180 | 223 | 230 | 87 | 182 | 148 | 78 | 176 | 19 | 17 | 4 | 184 | 176 | 183 | 102 | 83 | 81 | 132 | 206 |
| 173 | 137 | 185 | 242 | 181 | 181 | 214 | 49 | 74 | 238 | 197 | 37 | 98 | 102 | 15 | 217 | 148 | 8 | 102 | 188 |
| 85 | 9 | 17 | 222 | 18 | 210 | 70 | 21 | 78 | 241 | 184 | 216 | 93 | 93 | 208 | 102 | 153 | 212 | 119 | 47 |

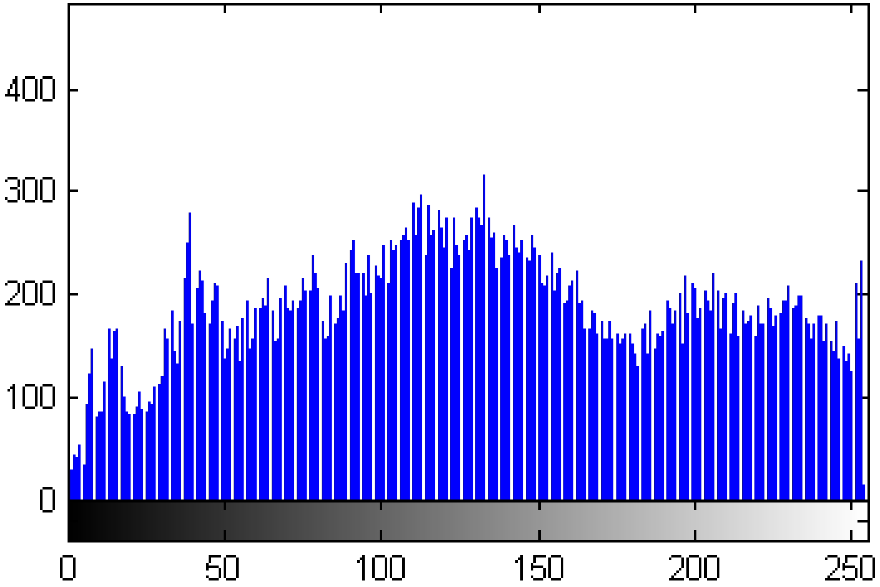


$$[0, 255] = [0, 15] \cup [16, 31] \cup .... \cup [240, 255]$$
$$\mathrm{range} = \mathrm{bin}_1 \cup \mathrm{bin}_2 \cup .... \cup \mathrm{bin}_{n=15}$$

bins

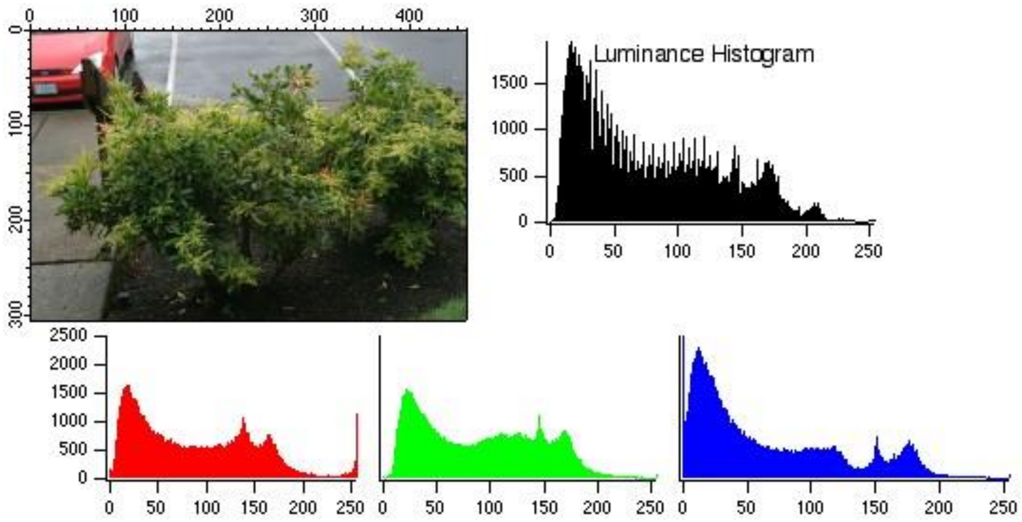
# Another example

For example in an grayscale image histogram has one dimension (dims = 1) because we are only counting the intensity values of each pixel.



# What if you want to count more features?

- for two features your resulting histogram would be a 3D plot.
- The same would apply for more features .
- For example for a color image the histogram would be a 4D plot.
- Some times the histogram is calculated separately for each channel.



# calcHist

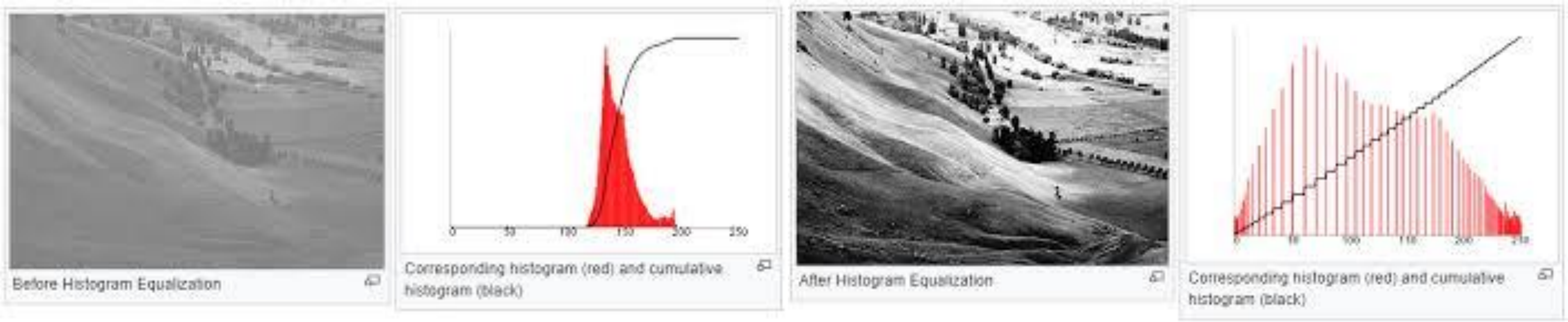
Calculates a histogram of a set of arrays.

**Python: cv2.calcHist(images, channels, mask, histSize, ranges, hist, accumulate) → hist**

**C++: void calcHist(const Mat* images, int nimages, const int* channels, InputArray mask, OutputArray hist, int dims, const int* histSize, const float** ranges, bool uniform=true, bool accumulate=false )**

# Equalizing image histograms

- low-contrast images have histograms where bins are clustered near a value: most of the pixels have their values within a narrow range. Low-contrast images are harder to work with because small details are poorly expressed. There is a technique that is able to address this issue. It's called histogram equalization.



Before Histogram Equalization

Corresponding histogram (red) and cumulative histogram (black)

After Histogram Equalization

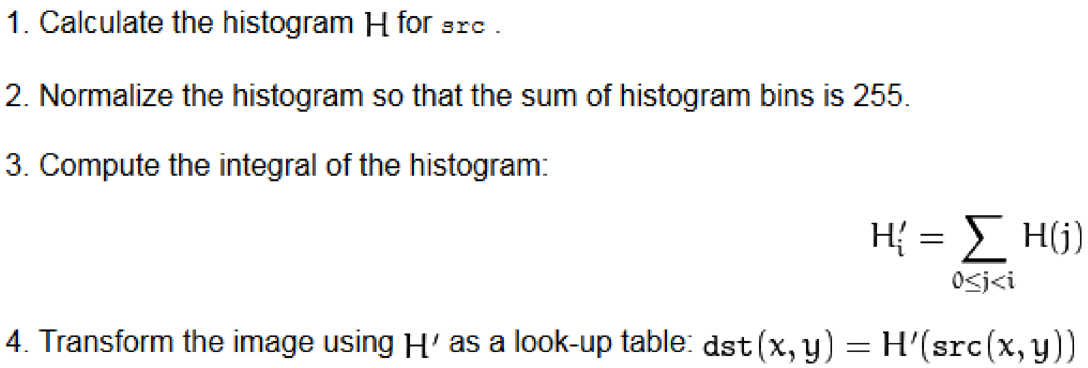
Corresponding histogram (red) and cumulative histogram (black)

# equalizeHist

Equalizes the histogram of a grayscale image.
The algorithm normalizes the brightness and increases the contrast of the image.

**Python: cv2.equalizeHist(src, dst) → dst**

**C++: void equalizeHist(InputArray src, OutputArray dst)**



1. Calculate the histogram $H$ for `src` .
2. Normalize the histogram so that the sum of histogram bins is 255.
3. Compute the integral of the histogram:

$$H'_i = \sum_{0 \le j < i} H(j)$$

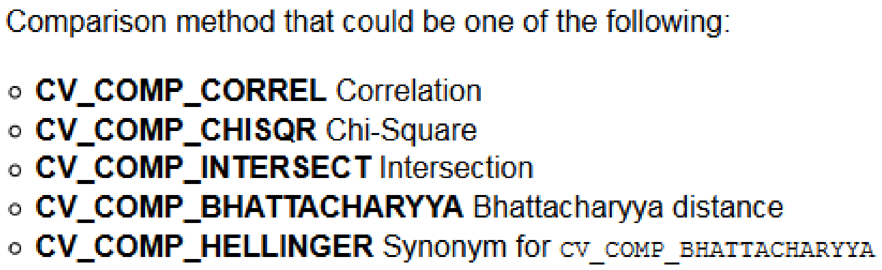
4. Transform the image using $H'$ as a look-up table: $\texttt{dst}(x, y) = H'(\texttt{src}(x, y))$

# compareHist

Compares two histograms.

**Python: cv2.compareHist(H1, H2, method) → retval**

**C++: double compareHist(InputArray H1, InputArray H2, int method)**



Comparison method that could be one of the following:

- **CV_COMP_CORREL** Correlation
- **CV_COMP_CHISQR** Chi-Square
- **CV_COMP_INTERSECT** Intersection
- **CV_COMP_BHATTACHARYYA** Bhattacharyya distance
- **CV_COMP_HELLINGER** Synonym for `CV_COMP_BHATTACHARYYA`

# THE END