
Basics of Computer Vision

Chapter Goals

After completing this chapter, you should be able to understand :

- Image transformations: (crop, resize, flip)
- Contrast improvement: histogram equalization, contrast stretching
- Image filtering: blurring, edge detection
- Thresholding: simple and adaptive
- Apply all the previous, using different libraries of python

Image representation

Gray image representation

Pixel: the smallest element
in the digital image



What we see

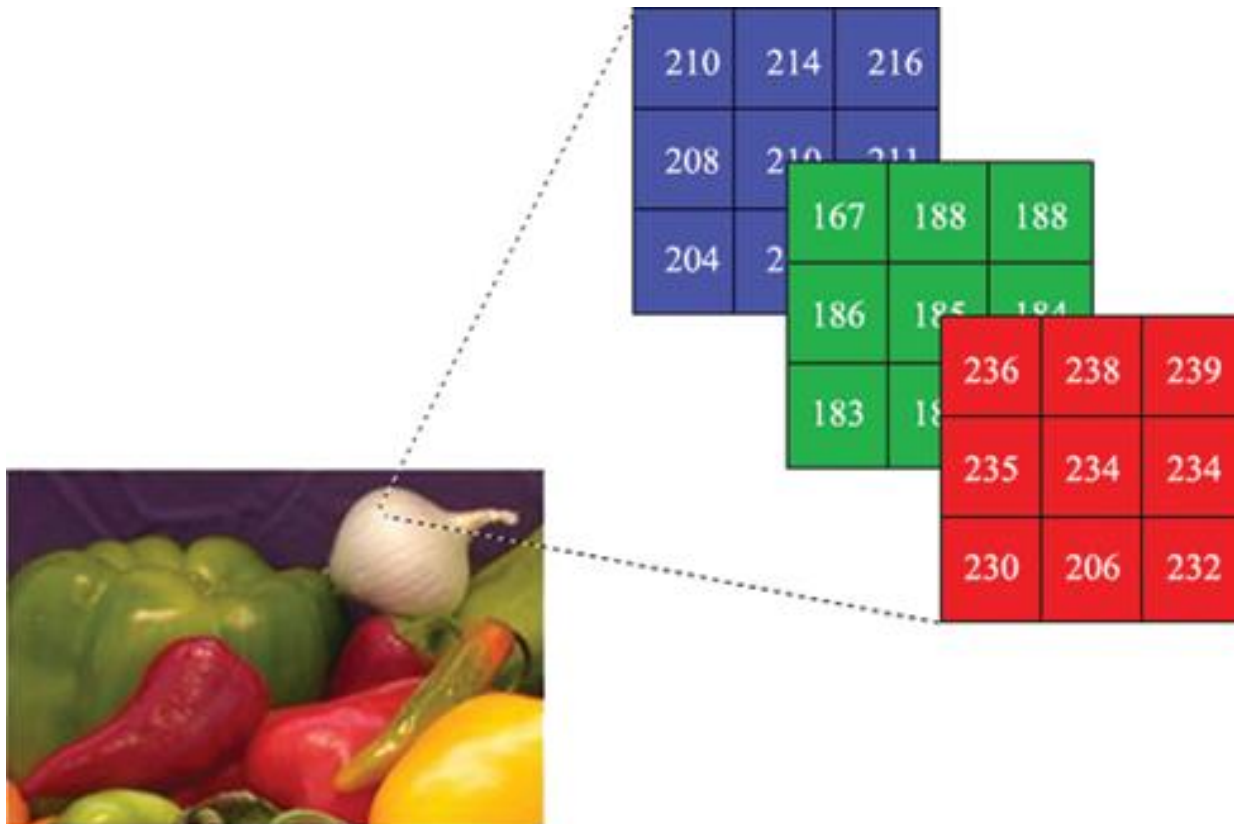
157	153	174	168	150	152	129	151	172	151	155	156
155	182	163	74	75	62	33	17	113	210	180	154
180	186	50	14	34	6	10	30	48	106	153	181
206	106	6	124	131	111	120	234	166	16	66	180
194	66	137	251	227	239	229	228	227	87	71	231
172	106	207	233	239	214	220	239	228	98	74	236
188	86	179	205	185	215	211	156	129	75	20	159
189	97	166	84	10	148	134	11	50	62	22	148
199	146	191	193	158	227	178	143	182	106	36	190
206	174	166	263	236	231	149	178	228	43	66	234
190	216	115	149	235	187	85	150	79	38	218	241
190	224	147	108	227	210	127	192	56	191	268	224
190	214	173	56	103	143	55	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	268	211
183	202	237	145	0	0	12	108	200	136	243	236
195	206	123	297	177	121	123	290	175	13	96	218

157	153	174	168	150	152	129	151	172	151	155	156
155	182	163	74	75	62	33	17	113	210	180	154
180	186	50	14	34	6	10	30	48	106	153	181
206	106	6	124	131	111	120	234	166	16	66	180
194	66	137	251	227	239	229	228	227	87	71	231
172	106	207	233	239	214	220	239	228	98	74	236
188	86	179	205	185	215	211	156	129	75	20	159
189	97	166	84	10	148	134	11	50	62	22	148
199	146	191	193	158	227	178	143	182	106	36	190
206	174	166	263	236	231	149	178	228	43	66	234
190	216	115	149	235	187	85	150	79	38	218	241
190	224	147	108	227	210	127	192	56	191	268	224
190	214	173	56	103	143	55	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	268	211
183	202	237	145	0	0	12	108	200	136	243	236
195	206	123	297	177	121	123	290	175	13	96	218

What computers see

- Each pixel value represents the brightness of the pixel. With 8-bit image, the pixel value of each pixel is 0 ~ 255
- Matrix representation: An image of MxN pixels is represented by an MxN array, each element being an unsigned integer of 8 bits

Color image representation



Pixels of a color image have Red, Green, and Blue gray values

Color image representation

Original



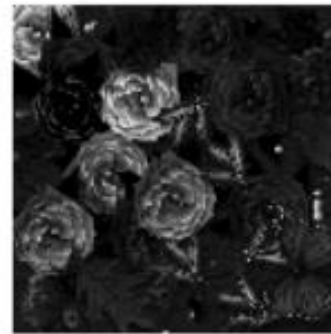
Red



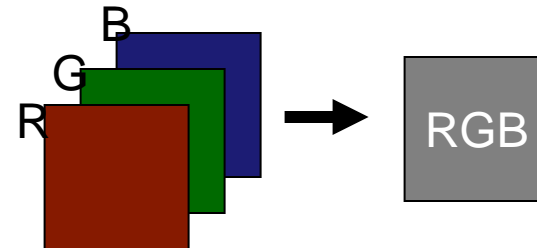
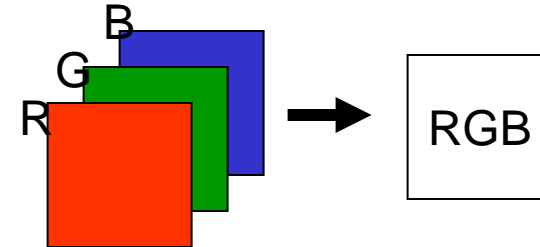
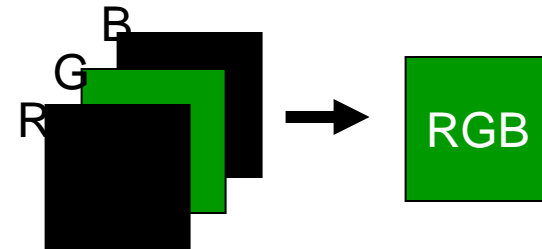
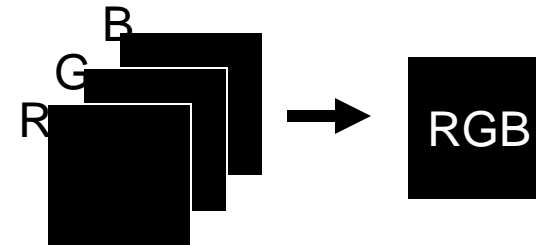
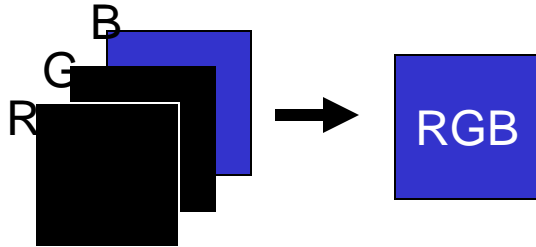
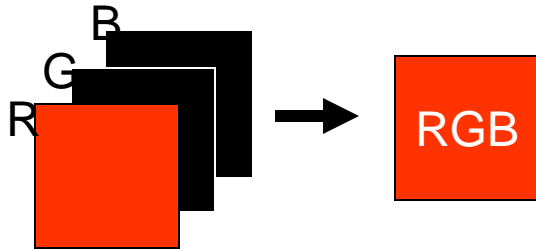
Green



Blue



Color image representation



Key Libraries for Image Processing in Python

1. Matplotlib

- **Functionality:** Primarily used for data visualization, Matplotlib can also handle image files. It allows users to read, write, and display images using simple commands.
- **Installation:** Typically included with scientific Python distributions, but can be installed via [pip install matplotlib](#).

2. Pillow (PIL Fork)

- **Functionality:** Pillow is a maintained fork of the original Python Imaging Library (PIL) and provides a comprehensive suite of tools for opening, manipulating, and saving many different image file formats. It supports operations like resizing, cropping, and filtering.
- **Installation:** Installable via [pip install Pillow](#)

3. OpenCV

- **Functionality:** OpenCV is a powerful library for computer vision tasks that includes capabilities for real-time image processing, object detection, and feature extraction. It is highly optimized and widely used in both academic and industrial applications.
- **Installation:** Can be installed using [pip install opencv-python](#)

4. Scikit-Image

- **Functionality:** Built on top of NumPy and SciPy, Scikit-Image provides a collection of algorithms for image processing tasks such as filtering, segmentation, and geometric transformations. It is user-friendly and well-documented.
- **Installation:** Available through [pip install scikit-image](#)

5. NumPy

- **Functionality:** While not exclusively an image processing library, NumPy is essential for handling image data as it allows images to be represented as multi-dimensional arrays. This makes it easy to perform mathematical operations on pixel values.
- **Installation:** Installable via [pip install numpy](#), often included with other scientific packages

6. SciPy

- **Functionality:** Similar to NumPy but with additional functionality for scientific computing. The `scipy.ndimage` module provides tools for image processing such as filtering and morphological operations.
- **Installation:** Can be installed via [pip install scipy](#)

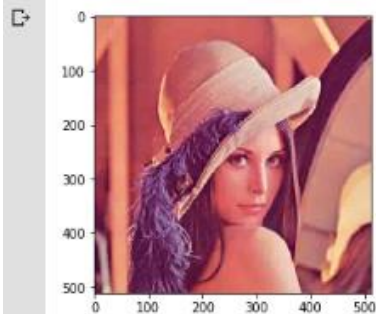
Image properties

Opencv library

```
import cv2

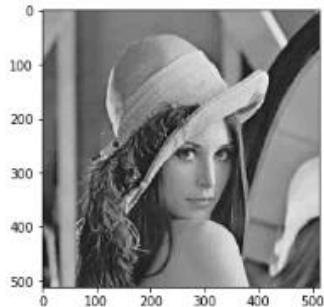
image = cv2.imread('lena10.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

cv2.waitKey(0)
cv2.destroyAllWindows()
```



```
[26] plt.imshow(gray, cmap=cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7fa932cdce80>
```



```
[27] gray.shape
```

```
(512, 512)
```

```
image.shape
```

```
(512, 512, 3)
```

```
gray.shape[0] # number of rows
```

```
512
```

```
gray.shape[1] # number of columns
```

```
512
```

Resizing Images: Zoom

•What is Zooming?

- Zooming, also called upsampling, increases the number of pixels in an image, making it appear larger.

•Zooming Process:

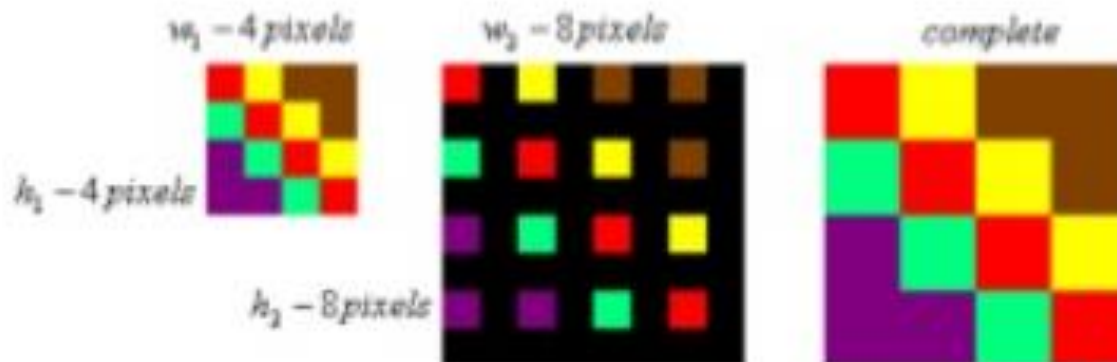
- **Creating New Pixel Locations:** Adds additional pixel positions between existing ones.
- **Interpolating Pixel Values:** Assigns values to the new pixel locations based on surrounding pixel data.

•Nearest Neighbor Interpolation:

- The black pixels in the middle image represent spaces where interpolation is required.
- The final image (on the right) is completed by assigning values to these spaces based on the nearest known pixel.

•Definition of Interpolation:

- It is the process of estimating values for unknown pixels using the values of nearby known pixels.



Resizing Images: Zoom- Interpolation Techniques

•Nearest Neighbor Interpolation:

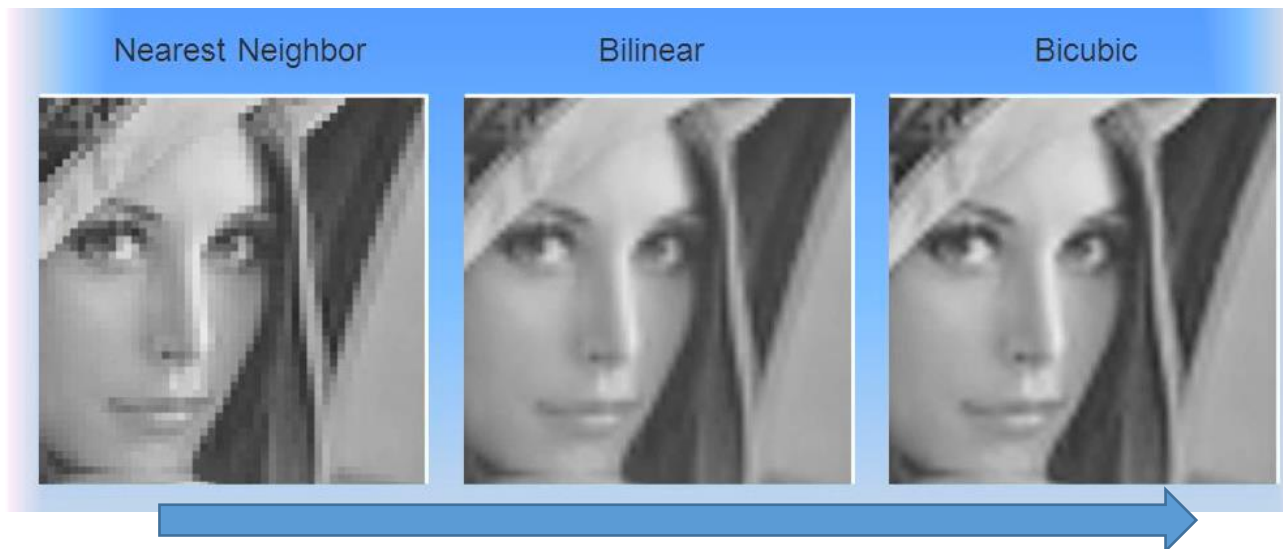
- Simplest method, where a pixel's value is directly assigned from the nearest known pixel.
- Results in blocky or pixelated images, suitable for low-complexity applications like thumbnails.

•Bilinear Interpolation:

- Uses a weighted average of the 2x2 neighboring pixels to calculate a new pixel's value.
- Produces smoother results than nearest neighbor, balancing quality and computational effort.

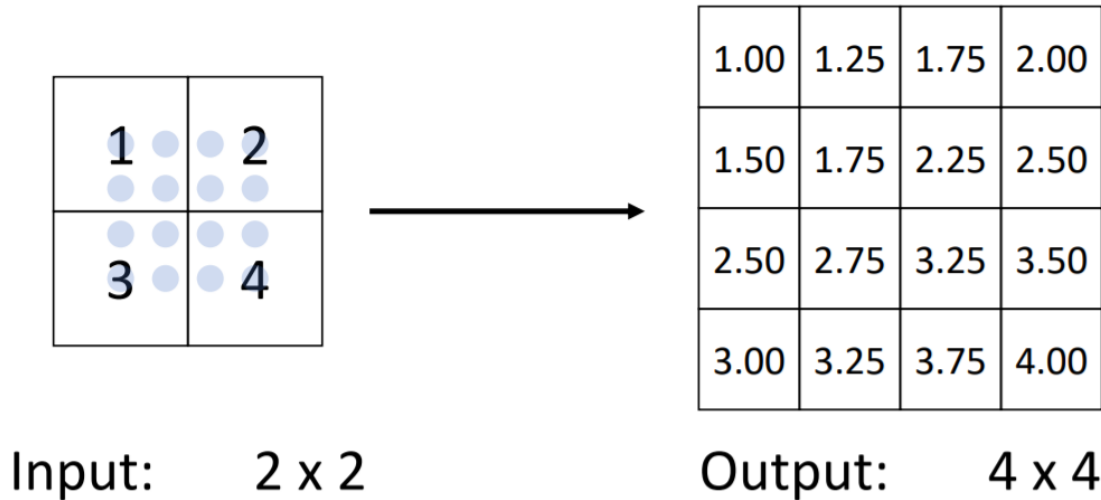
•Bicubic Interpolation:

- Considers the nearest 4x4 pixel neighborhood (16 pixels) for interpolation.
- Delivers high-quality, smooth images, but requires significantly more computational resources.
- Often used in applications requiring precision, like photo editing or computer vision tasks.



Complexity of the algorithm

Bilinear Interpolation



Use two closest neighbors in x and y to construct linear approximations
Smoother results than nearest neighbor upsampling

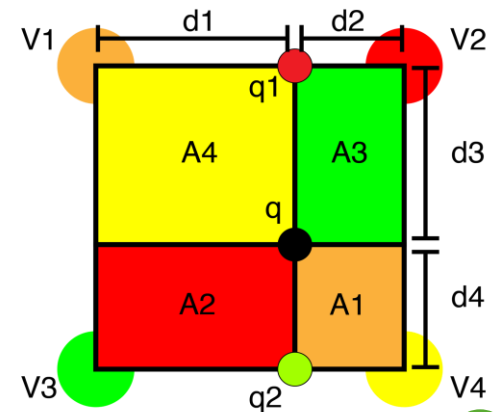
$$A1 = d2 * d4$$

$$A2 = d1 * d4$$

$$A3 = d2 * d3$$

$$A4 = d1 * d3$$

$$q = V1 * A1 + V2 * A2 + V3 * A3 + V4 * A4$$



Resizing Images: Shrink

Resizing Images: Shrink (Downsampling)

Downsampling reduces the resolution of an image by decreasing the number of pixels. This involves selecting or calculating fewer pixels to represent the original image, typically to reduce size for storage, transmission, or processing efficiency.

Key Steps in Downsampling:

- 1. Selection of Target Pixels:** Fewer pixels are retained based on the target resolution.
- 2. Interpolation:** The values of the resulting pixels are derived from nearby pixels in the original image using one of the following methods:

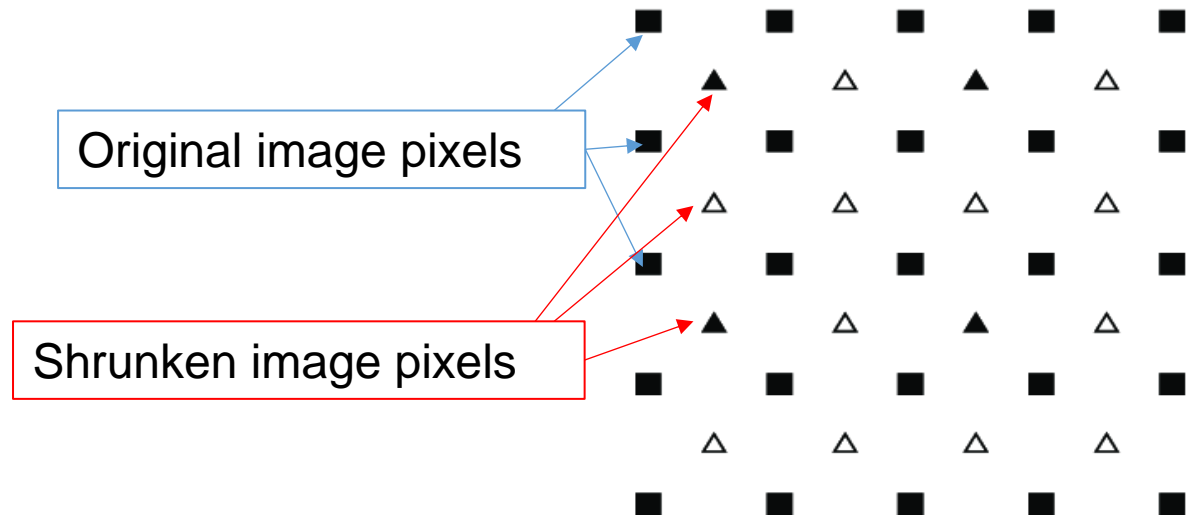
- 1. Nearest Neighbor**

Interpolation: Simplest approach; selects the value of the closest original pixel.

- 2. Bilinear Interpolation:**

Computes a weighted average of the 2x2 neighboring pixels to estimate the value.

- 3. Bicubic Interpolation:** Uses the closest 4x4 pixel neighborhood for a smoother and higher-quality result.



Resizing Images: Zoom

```
[18] scale_percent = 10 # percent of original size
      width = int(image.shape[1] * scale_percent / 100)
      height = int(image.shape[0] * scale_percent / 100)
      dim = (width, height)
      # resize image
      resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
```

```
plt.imshow(cv2.cvtColor(resized, cv2.COLOR_BGR2RGB))
plt.title('Resized')
```



Some Opencv interpolation methods

- cv2.INTER_NEAREST: Interpolation by using the nearest neighbor technique
- cv2.INTER_LINEAR: Bilinear interpolation technique. This is extensively used for zooming. This is the default interpolation technique.
- cv2.INTER_AREA: Performs resampling using the pixel-area relationship. This is Extensively used to shrink the image
- cv2.INTER_CUBIC: Bicubic interpolation technique. This is extensively used for zooming

Histogram operations

Image contrast

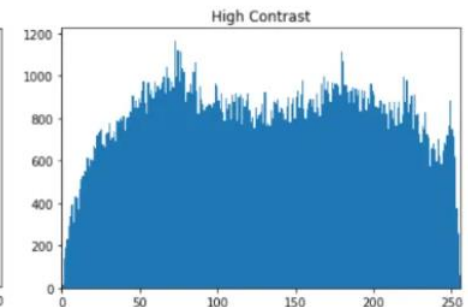
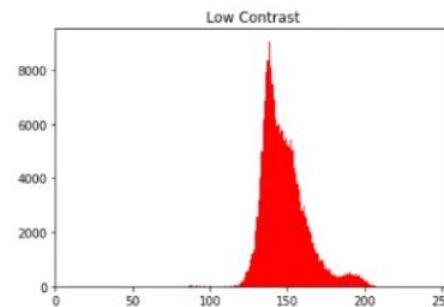
- **Contrast** refers to the difference in brightness or color that allows objects in an image to stand out from one another. Higher contrast makes it easier to see and differentiate details, while lower contrast makes an image look flat or washed out.
- For example:
- **Low Contrast:** In the left image, it's hard to identify details because the brightness levels are too similar.
- **High Contrast:** In the right image, the differences in brightness and colors make the details much easier to see.
- A real-world analogy would be comparing a sunny day to a foggy day:
- On a **sunny day**, objects appear clear and vivid, showing high contrast.
- On a **foggy day**, everything appears dull and similar in brightness, resulting in low contrast.
- To objectively determine contrast, we can analyze the **image histogram**, which shows the distribution of brightness (intensity) values in the image:
- In the histogram of the left image, the intensity values are grouped within a narrow range, indicating low contrast. This is because it's difficult to differentiate similar intensity levels (e.g., 150 vs. 148).
- In contrast, an image with higher contrast would have intensity values spread over a wider range (e.g., 50 vs. 200), making the differences more noticeable.
- By examining the histogram, we can clearly see why the left image has low contrast.



Low Contrast Image



High Contrast Image



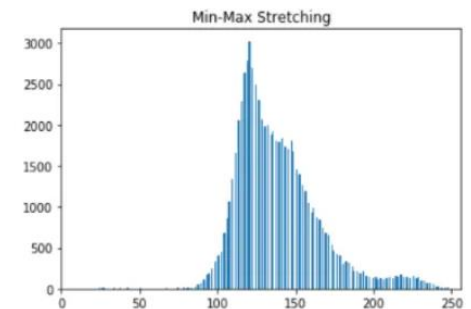
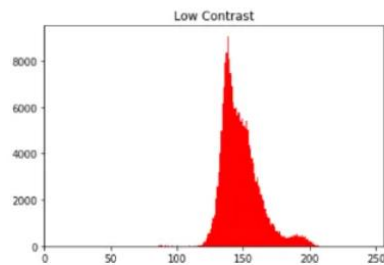
Contrast Enhancement

- **Linear: Contrast stretching**
- In **Min-Max Stretching**, the lower and upper values of the input image are made to span the full dynamic range. In other words, Lower value of the input image is mapped to 0 and the upper value is mapped to 255.

```

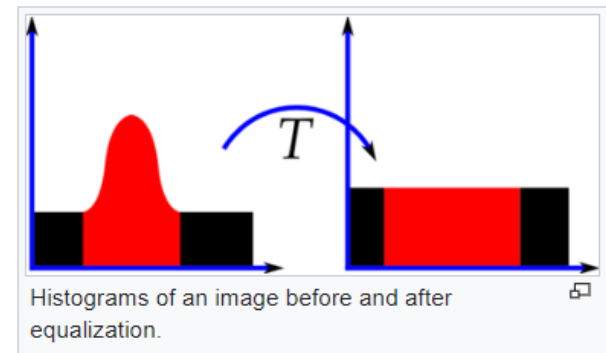
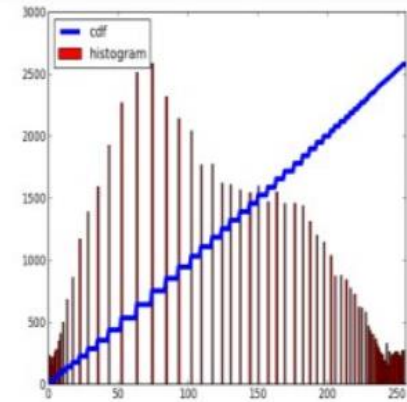
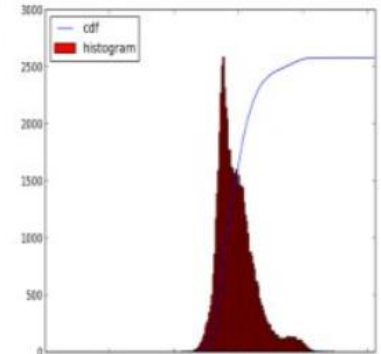
1 import cv2
2 import numpy as np
3
4 # Read the image
5 img1 = cv2.imread('D:/downloads/contrast.PNG',0)
6
7 # Create zeros array to store the stretched image
8 minmax_img = np.zeros((img1.shape[0],img1.shape[1]),dtype = 'uint8')
9
10 # Loop over the image and apply Min-Max formulae
11 for i in range(img1.shape[0]):
12     for j in range(img1.shape[1]):
13         minmax_img[i,j] = 255*(img1[i,j]-np.min(img1))/(np.max(img1)-np.min(img1))
14
15 # Displat the stretched image
16 cv2.imshow('Minmax',minmax_img)
17 cv2.waitKey(0)

```



Contrast Enhancement: Histogram equalization

- **Histogram Equalization** is a technique used to improve the contrast of an image by redistributing the pixel intensity values. The goal is to spread out the most frequent intensity values so that the image appears clearer and more detailed.
- **How It Works:**
 - 1. Input Histogram:** An image's histogram shows the distribution of pixel intensities (e.g., how many pixels have a brightness level of 0, 50, 100, etc.). A poorly contrasted image might have pixel intensities concentrated in a narrow range.
 - 2. Cumulative Distribution Function (CDF):** Histogram equalization calculates the cumulative distribution of intensity values. This step ensures the mapping of intensity values to better utilize the available range (e.g., 0 to 255 for an 8-bit image).
 - 3. Mapping New Intensity Values:** Each pixel in the image is reassigned an intensity value based on the CDF, effectively stretching and redistributing the intensity range. This makes darker areas appear brighter and increases the distinction between features.
 - 4. Output Histogram:** The histogram of the enhanced image is more uniformly distributed, meaning the intensity values are spread across the entire range, improving contrast.



Histogram equalization-Python

```
img = cv2.imread('wiki.jpg',0)
equ = cv2.equalizeHist(img)
res = np.hstack((img,equ)) #stacking images side-by-side
cv2.imwrite('res.png',res)
```



Image Filtering

Image Filtering

- One simple version of filtering: linear filtering (convolution)
 - Replace each pixel by a linear combination (a weighted sum) of its neighbors
- The prescription for the linear combination is called the “kernel” (or “mask”, “filter”)

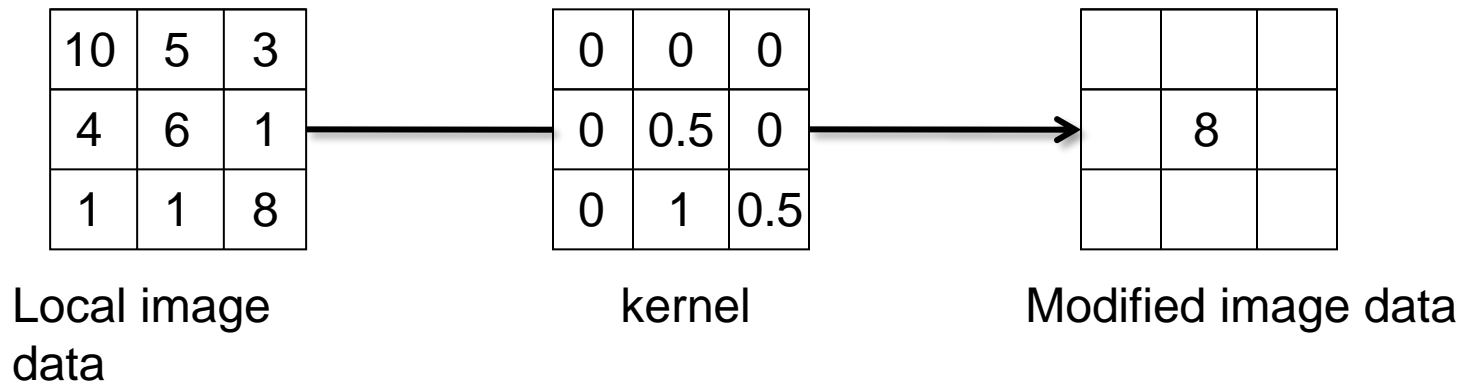
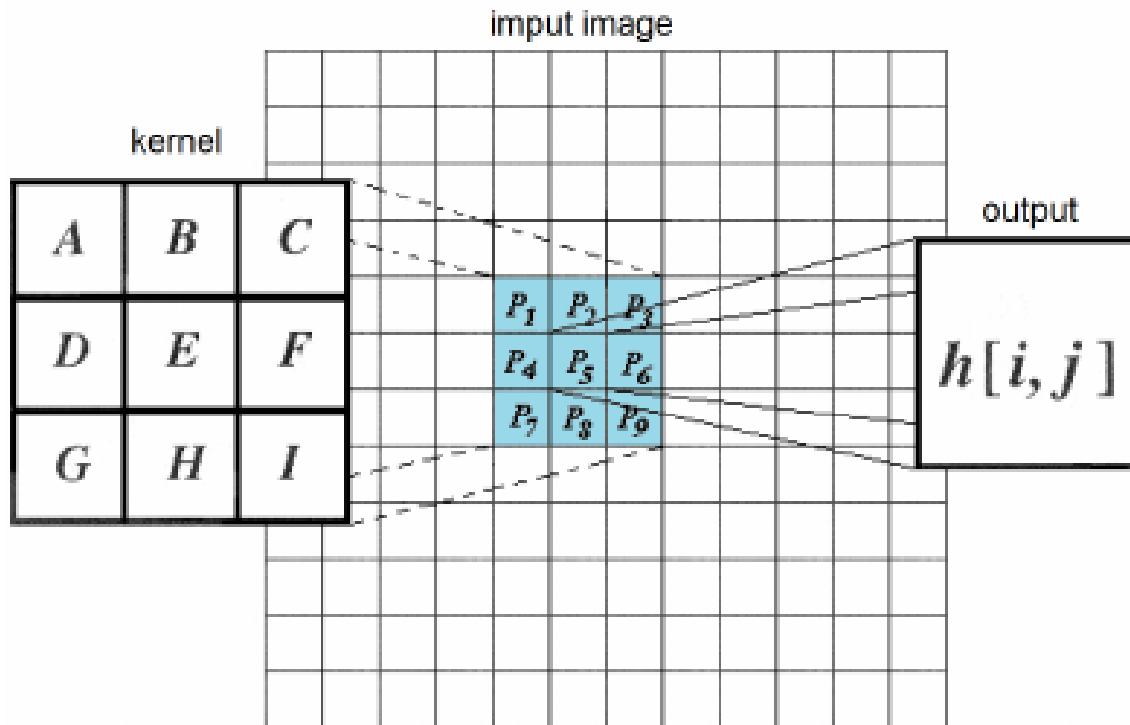
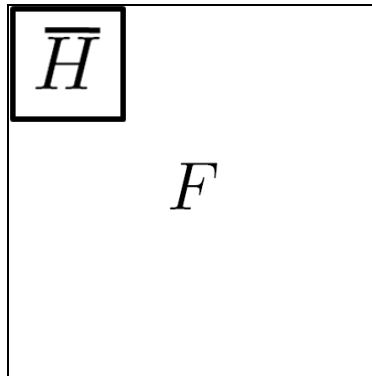


Image Filtering

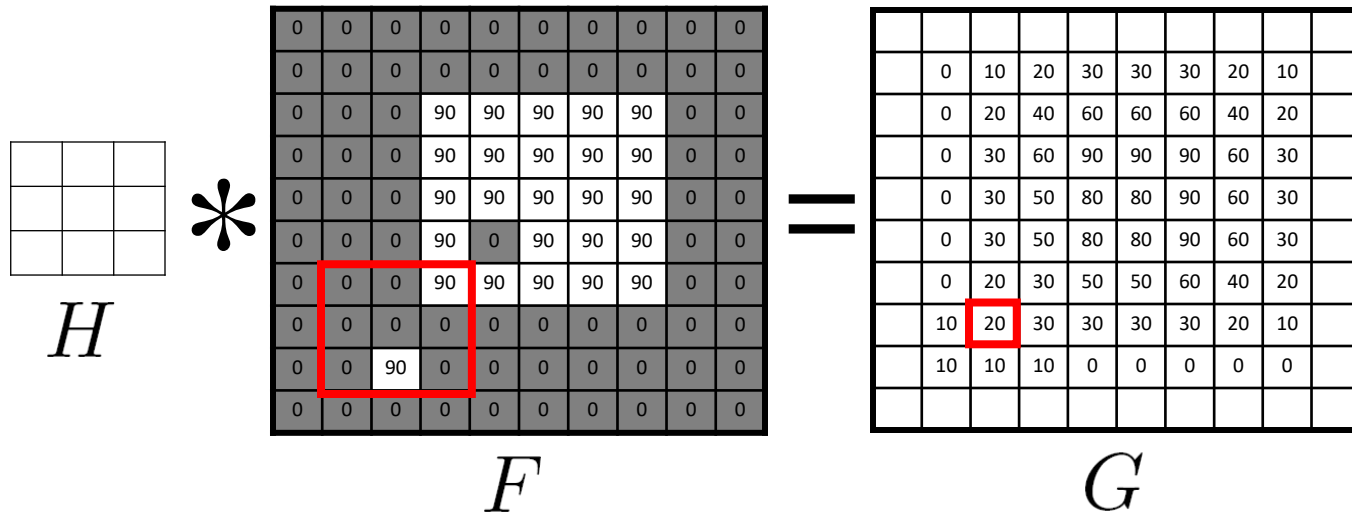


$$h(i,j)=A*P_1+B*P_2+C*P_3+D*P_4+E*P_5+F*P_6+G*P_7+H*P_8+I*P_9$$

Convolution



Mean filtering



Mean filtering/Moving average

$$F[x, y]$$

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		90	90	90	90	90	0	0	
0	0	0		90	90	90	90	90	0	0	
0	0	0		90	90	90	90	90	0	0	
0	0	0		90	0	90	90	90	0	0	
0	0	0		90	90	90	90	90	0	0	
0	0	0		0	0	0	0	0	0	0	0
0	0	90		0	0	0	0	0	0	0	0
0	0	0		0	0	0	0	0	0	0	0

$$G[x, y]$$

A 10x10 grid with a red square in the top-left corner. The red square is located in the first row and first column, with a side length of 1 unit. The grid is composed of 10 columns and 10 rows of squares. The red square is the top-leftmost square in the grid.

Mean filtering/Moving average

$$F[x, y]$$
[illegible]
$$G[x, y]$$
[illegible]

Mean filtering/Moving average

$$F[x, y]$$
[illegible]
$$G[x, y]$$
[illegible]

Mean filtering/Moving average

$$F[x, y]$$
[illegible]
$$G[x, y]$$
[illegible]

Mean filtering/Moving average

$$F[x, y]$$
[illegible]
$$G[x, y]$$
[illegible]

Mean filtering/Moving average

$$F[x, y]$$

[illegible]

$$G[x, y]$$

[illegible]

Linear filters: examples

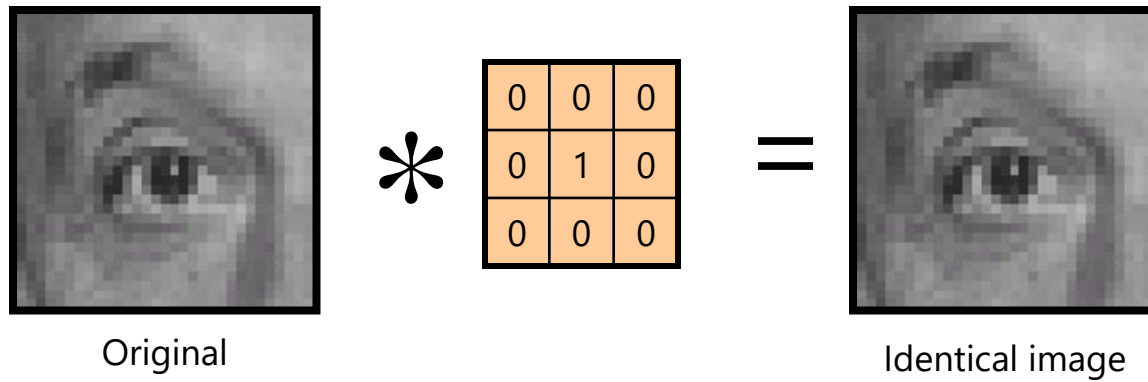


Original



0	0	0
0	1	0
0	0	0

Linear filters: examples



Source: D. Lowe

Linear filters: examples



Original



0	0	0
1	0	0
0	0	0

Linear filters: examples



Original



0	0	0
1	0	0
0	0	0



Shifted left by 1 pixel

Linear filters: examples

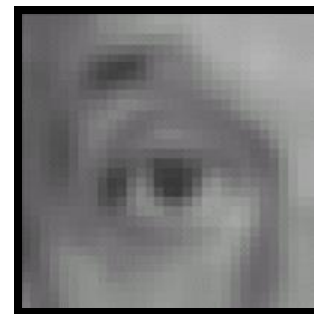


Original



$\frac{1}{9}$

1	1	1
1	1	1
1	1	1



Blur (with a mean filter)

Linear filters: examples



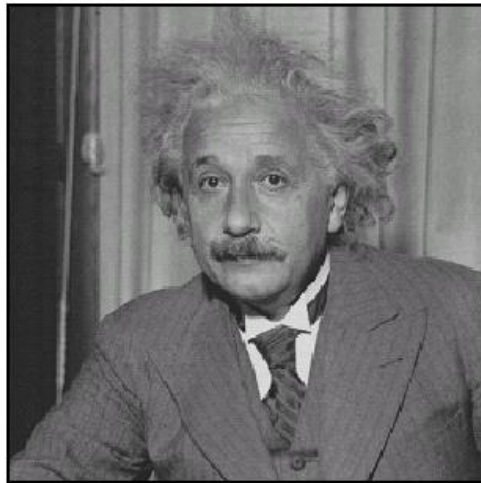
Original

$$* \left(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \right) =$$

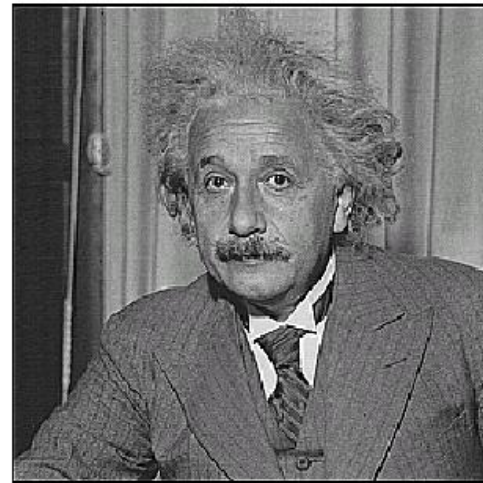


Sharpening filter
(accentuates edges)

Sharpening



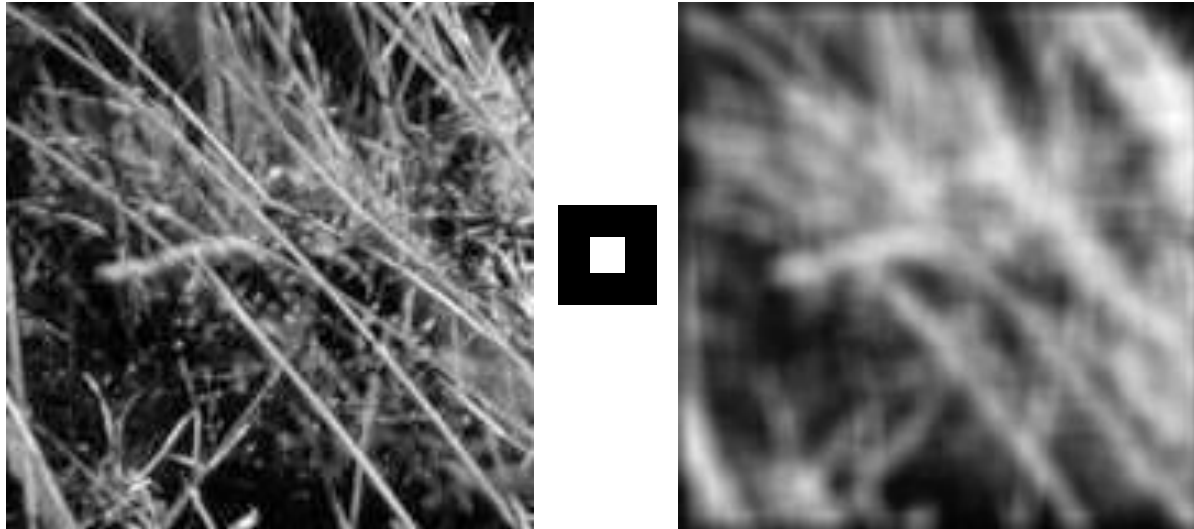
before



after

Source: D. Lowe

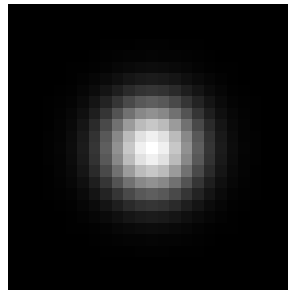
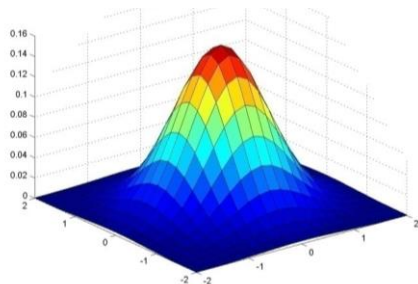
Smoothing with box filter revisited



Source: D. Forsyth

Gaussian Kernel

$$G_{\sigma} = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$



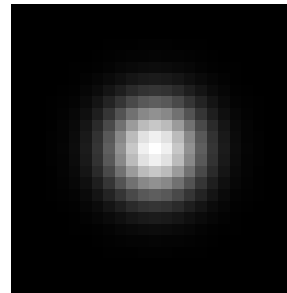
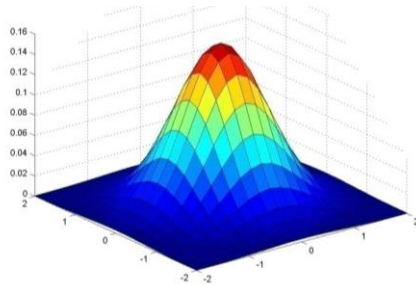
0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

5 x 5, $\sigma = 1$

- Constant factor at front makes volume sum to 1 (can be ignored, as we should re-normalize weights to sum to 1 in any case)

Source: C. Rasmussen

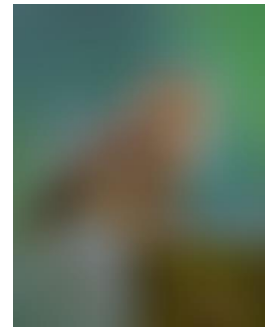
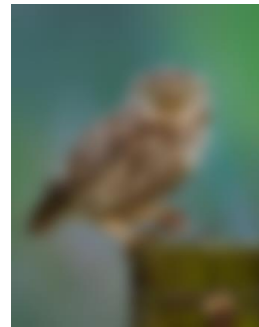
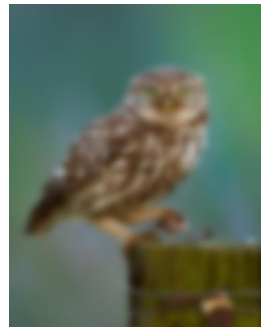
Gaussian kernel



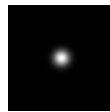
$$G_{\sigma} = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Source: C. Rasmussen

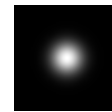
Gaussian filters



$\sigma = 1$ pixel



$\sigma = 5$ pixels



$\sigma = 10$ pixels



$\sigma = 30$ pixels

Mean vs. Gaussian filtering

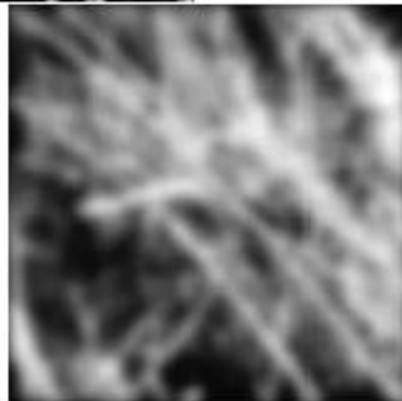
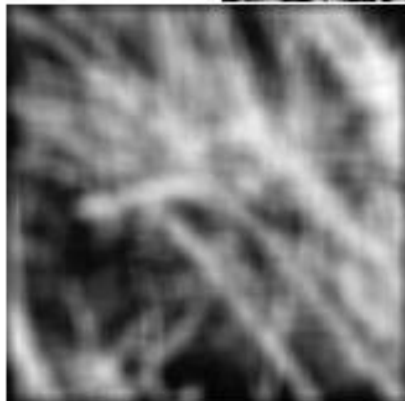


Image filtering-Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

kernel = np.ones((5,5),np.float32)/25
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([], plt.yticks([]))
plt.show()
```

Result:

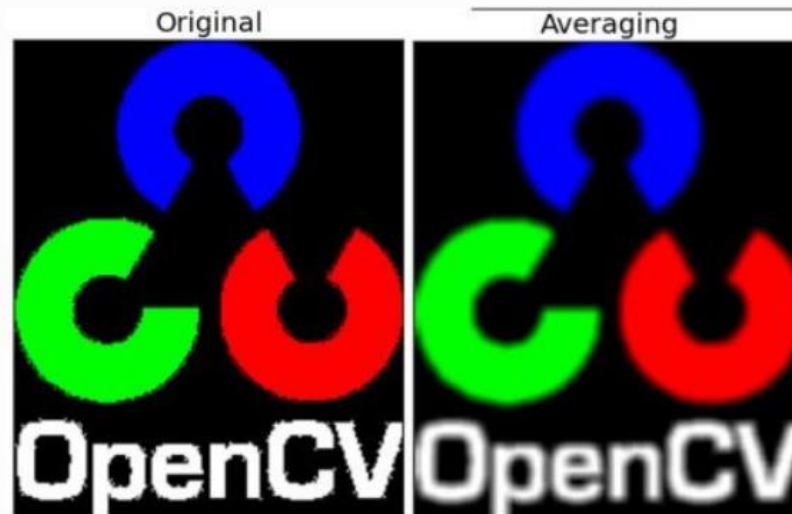
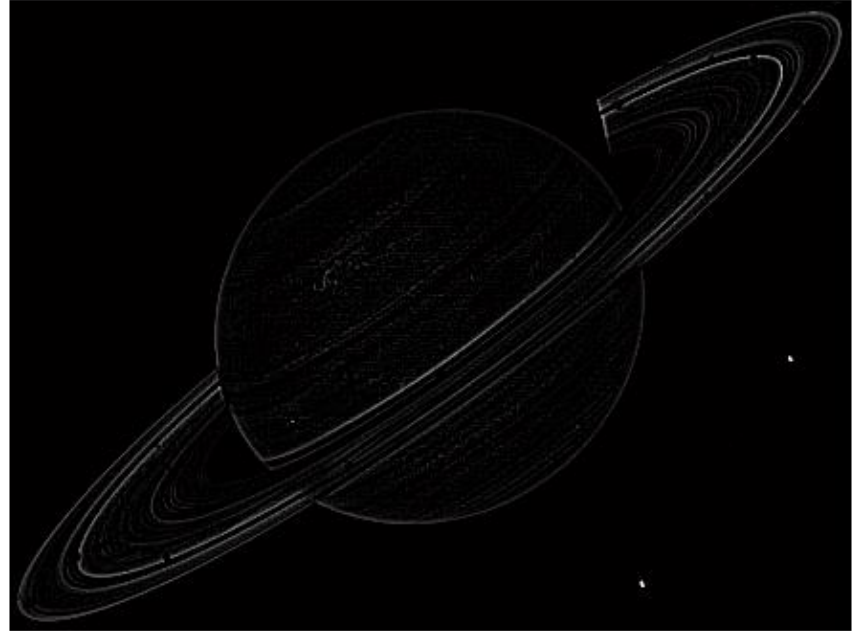
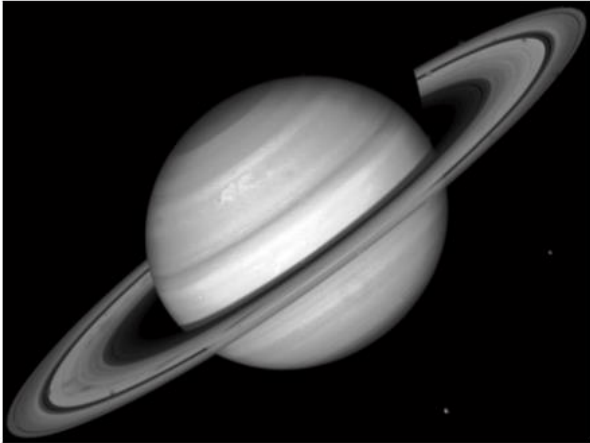


Image Filtering



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

Sobel Edge Detection

Sobel approximates the derivatives of the image along x and y axis

- The *Sobel* operators below are very commonly used

$$G_x = \frac{1}{8} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \quad G_y = \frac{1}{8} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$$

s_x s_y

- At each point of the image, we calculate an approximation of the *gradient* in that point by combining image convolution with kernels above:

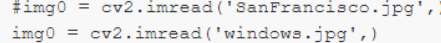
$$G = \sqrt{G_x^2 + G_y^2}$$

- Using this information, we can also calculate the gradient's direction:

$$\Theta = \text{atan}\left(\frac{G_y}{G_x}\right)$$

Sobel Edge Detection-Python

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# loading image

img0 = cv2.imread('SanFrancisco.jpg',)
img0 = cv2.imread('windows.jpg',)

# converting to gray scale
gray = cv2.cvtColor(img0, cv2.COLOR_BGR2GRAY)

# remove noise
img = cv2.GaussianBlur(gray, (3,3),0)

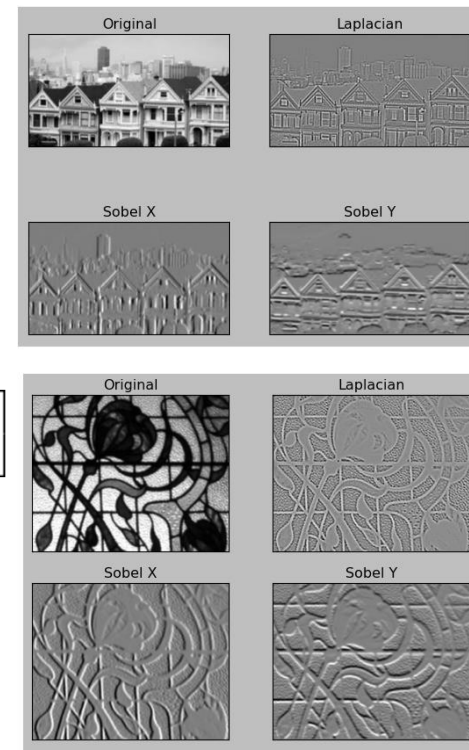
# convolute with proper kernels
laplacian = cv2.Laplacian(img,cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5) # x
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5) # y

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([], plt.yticks([]))
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([], plt.yticks([]))

plt.show()
```

Unlike the Sobel edge detector, the Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass.

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



Thresholding

Simple thresholding

- In an image, for every pixel, the same threshold value is applied. If the pixel value is smaller than the threshold, it is set to **0**, otherwise it is set to a maximum value (**255**).

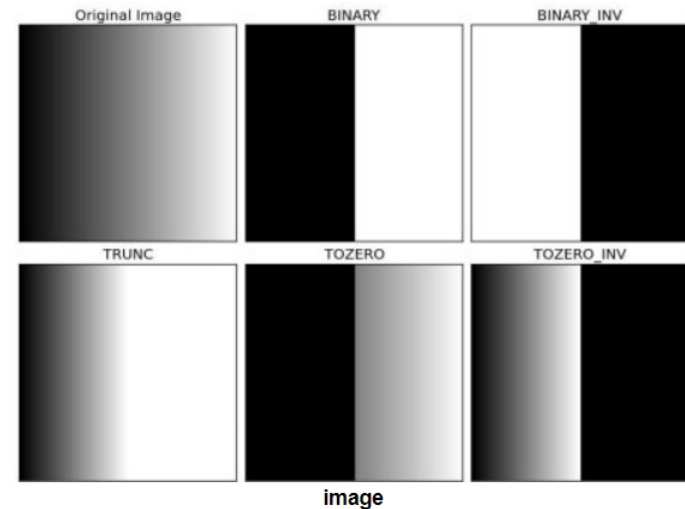
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('gradient.png',0)
ret,thresh1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
ret,thresh2 = cv.threshold(img,127,255,cv.THRESH_BINARY_INV)
ret,thresh3 = cv.threshold(img,127,255,cv.THRESH_TRUNC)
ret,thresh4 = cv.threshold(img,127,255,cv.THRESH_TOZERO)
ret,thresh5 = cv.threshold(img,127,255,cv.THRESH_TOZERO_INV)

titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]

for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([],plt.yticks([]))

plt.show()
```



Adaptive thresholding

- **Adaptive Thresholding** is a technique used in image processing to segment an image by converting it to a binary image (black and white). Unlike global thresholding, which applies a single threshold value to the entire image, **adaptive thresholding calculates different threshold values for different regions of the image**. This makes it particularly useful for images with varying lighting conditions or uneven illumination.
- **How Adaptive Thresholding Works:**
 1. **Divide the Image into Regions:** The image is divided into smaller regions (blocks or neighborhoods), and a local threshold is calculated for each region.
 2. **Calculate Local Thresholds:** The threshold value for each region is determined based on the pixel intensities in that region. Common methods include:
 1. **Mean Thresholding:** The threshold is the average intensity of the pixels in the region.
 2. **Gaussian Weighted Mean:** Similar to mean thresholding but applies weights to give more importance to pixels closer to the center of the region.
 3. **Median Thresholding:** Uses the median pixel intensity in the region.
 3. **Threshold Each Pixel:** Each pixel is compared to the local threshold of its region:
 1. If the pixel intensity is above the threshold, it is set to white (255).
 2. If it is below the threshold, it is set to black (0).
- **Benefits:**
 - **Handles Uneven Illumination:** It performs well on images where lighting changes across the scene.
 - **Improves Edge Detection:** It can highlight important features in images with poor or uneven lighting.

Adaptive Thresholding-Python

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('sudoku.png',0)
img = cv.medianBlur(img,5)

ret,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
th2 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C,\
    cv.THRESH_BINARY,11,2)
th3 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,\
    cv.THRESH_BINARY,11,2)

titles = ['Original Image', 'Global Thresholding (v = 127)',
    'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]

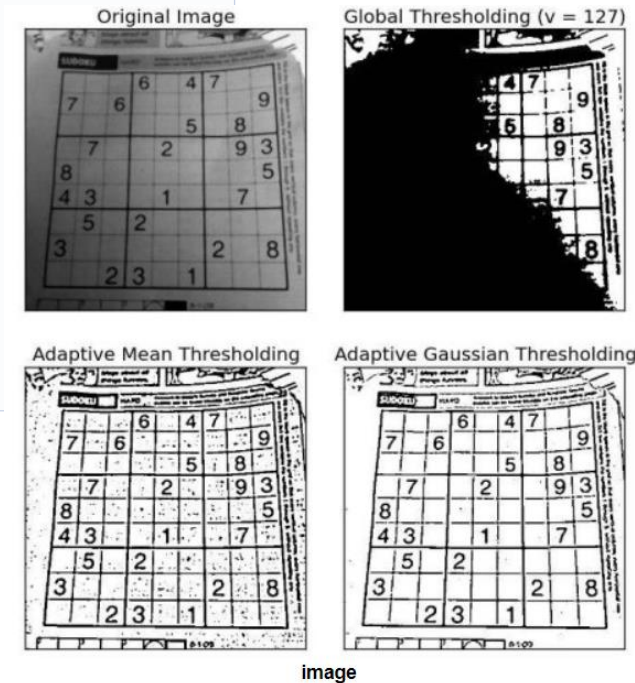
for i in xrange(4):
    plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([],plt.yticks([]))
plt.show()
```

The **adaptiveMethod** decides how the threshold value is calculated:

cv.ADAPTIVE_THRESH_MEAN_C: The threshold value is the mean of the neighborhood area minus the constant **C**.

cv.ADAPTIVE_THRESH_GAUSSIAN_C: The threshold value is a gaussian-weighted sum of the neighborhood values minus the constant **C**.

The **blockSize** determines the size of the neighborhood area and **C** is a constant that is subtracted from the mean or weighted sum of the neighborhood pixels.



Otsu's Thresholding

- In global thresholding, we used an arbitrary chosen value as a threshold. In contrast, Otsu's method avoids having to choose a value and determines it automatically.
- Consider an image with only two distinct image values (*bimodal image*), where the histogram would only consist of two peaks. A good threshold would be in the middle of those two values. Similarly, Otsu's method determines an optimal global threshold value from the image histogram

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('noisy2.png',0)

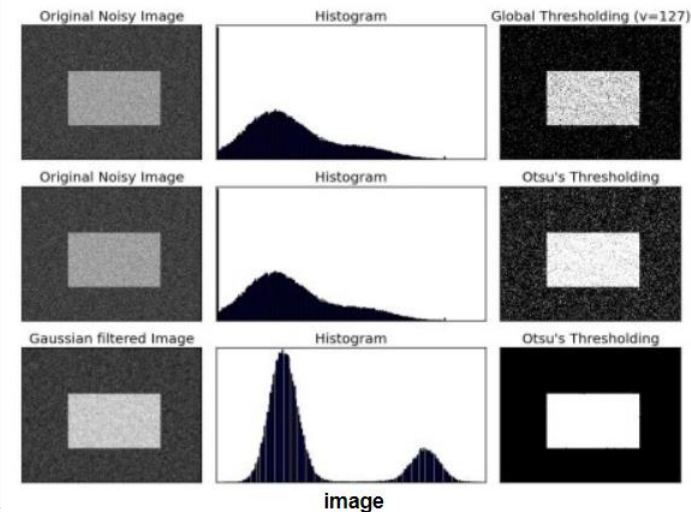
# global thresholding
ret1,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)

# Otsu's thresholding
ret2,th2 = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)

# Otsu's thresholding after Gaussian filtering
blur = cv.GaussianBlur(img,(5,5),0)
ret3,th3 = cv.threshold(blur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)

# plot all the images and their histograms
images = [img, 0, th1,
          img, 0, th2,
          blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
          'Original Noisy Image','Histogram','Otsu's Thresholding',
          'Gaussian filtered Image','Histogram','Otsu's Thresholding']

for i in xrange(3):
    plt.subplot(3,3,i*3+1),plt.imshow(images[i*3], 'gray')
    plt.title(titles[i*3]), plt.xticks([], plt.yticks([]))
    plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
    plt.title(titles[i*3+1]), plt.xticks([], plt.yticks([]))
    plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2], 'gray')
    plt.title(titles[i*3+2]), plt.xticks([], plt.yticks([]))
plt.show()
```



Chapter Summary

- We reviewed some Image transformations
- We visited some techniques for Contrast improvement
- We introduced Image filtering: blurring, edge detection
- We saw the different Thresholding techniques: simple and adaptive
- We saw how to apply these methods using python and common computer vision libraries