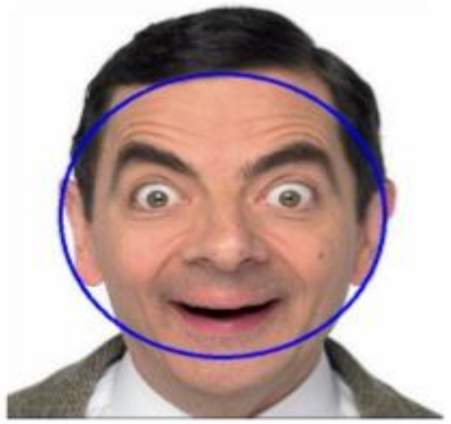
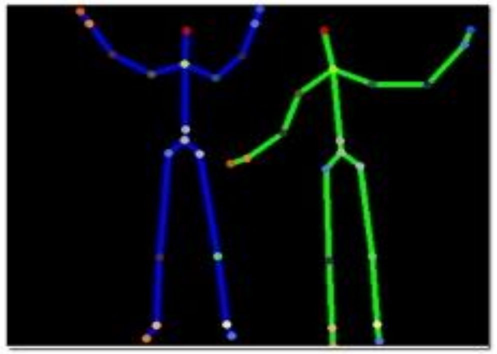
OpenCV Workshop



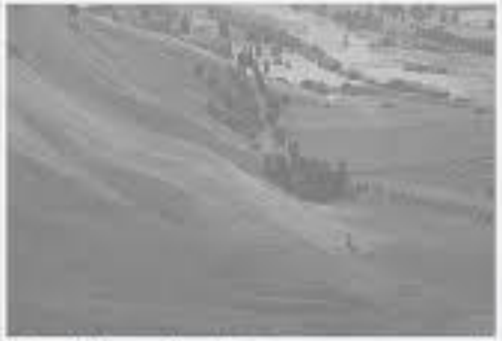


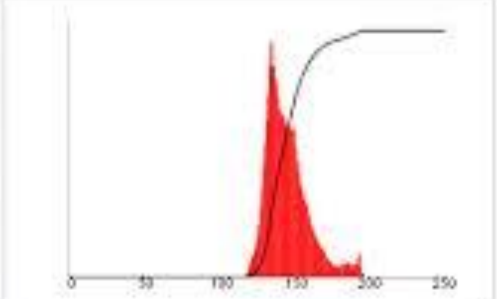
Session 4: histograms (continued) & Morphological Operations

Elham Shabaninia

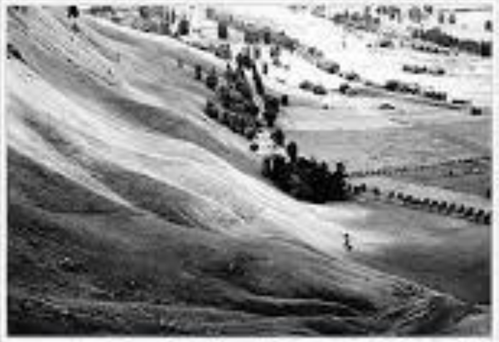
Equalizing image histograms

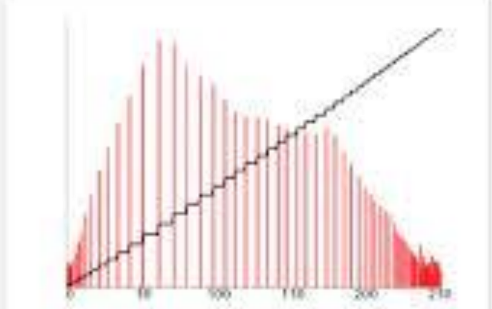
- low-contrast images have histograms where bins are clustered near a value: most of the pixels have their values within a narrow range. Low-contrast images are harder to work with because small details are poorly expressed. There is a technique that is able to address this issue. It's called histogram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


Before Histogram Equalization



Corresponding histogram (red) and cumulative histogram (black)


After Histogram Equalization



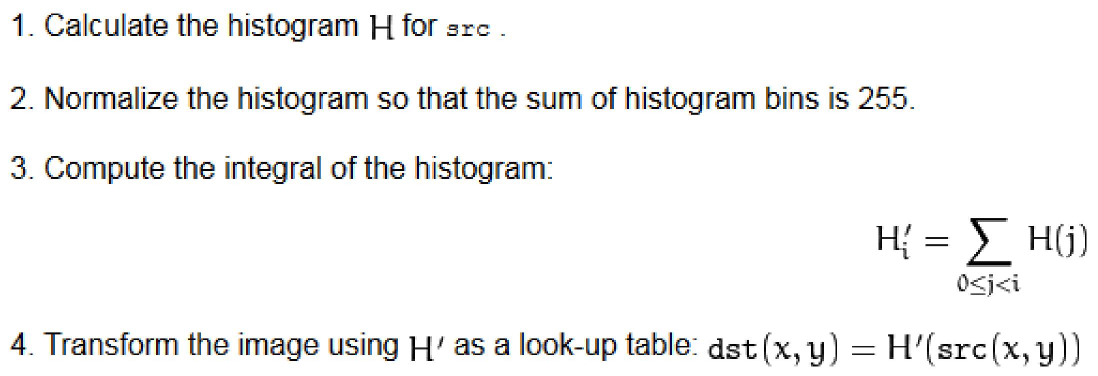
Corresponding histogram (red) and cumulative histogram (black)

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)



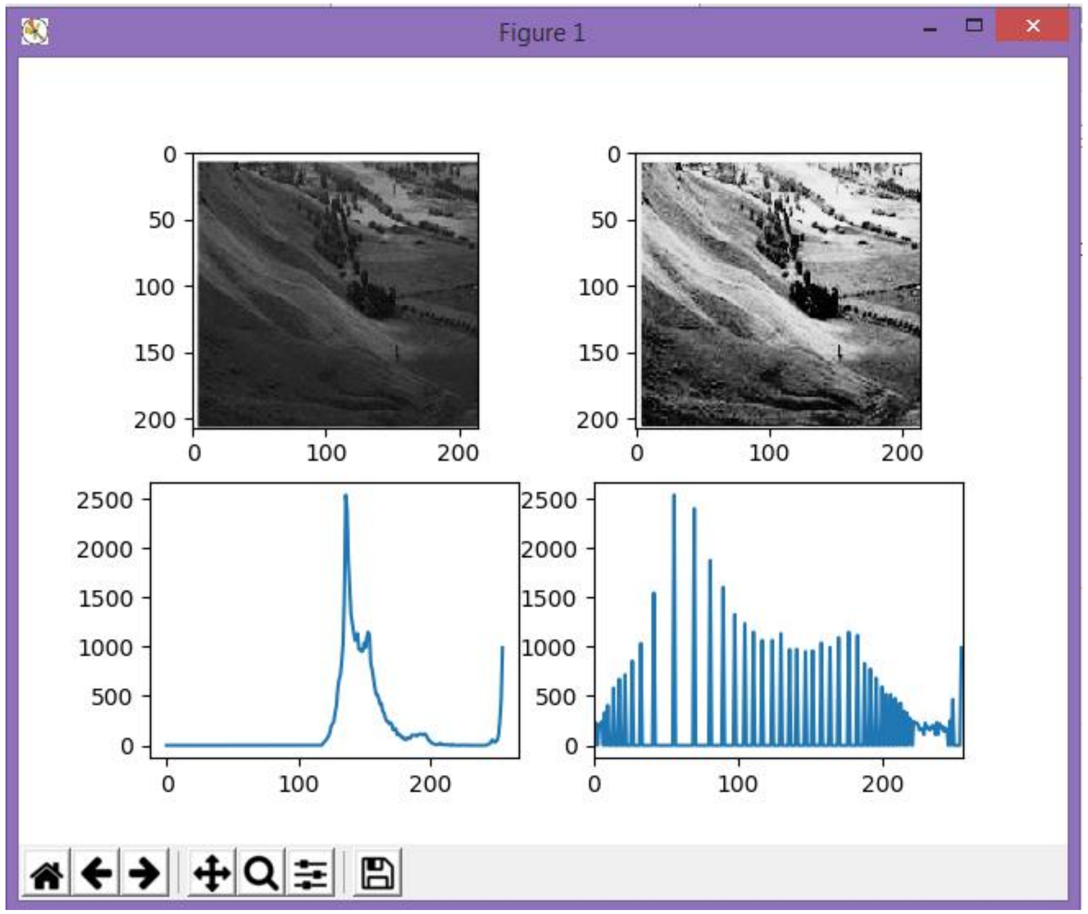
1. Calculate the histogram H for `src` .
2. Normalize the histogram so that the sum of histogram bins is 255.
3. Compute the integral of the histogram:

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

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

Let’s write some code…

Let’s see histogram equalization in an image

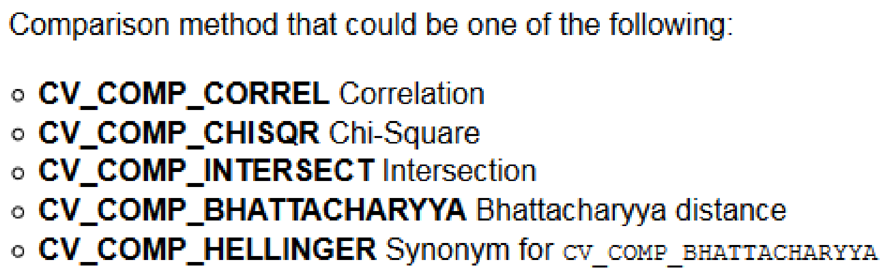


compareHist

Compares two histograms.

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

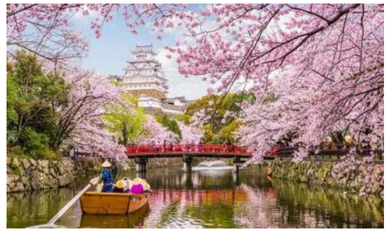
C++: double compareHist(InputArray H1, InputArray H2, int method)



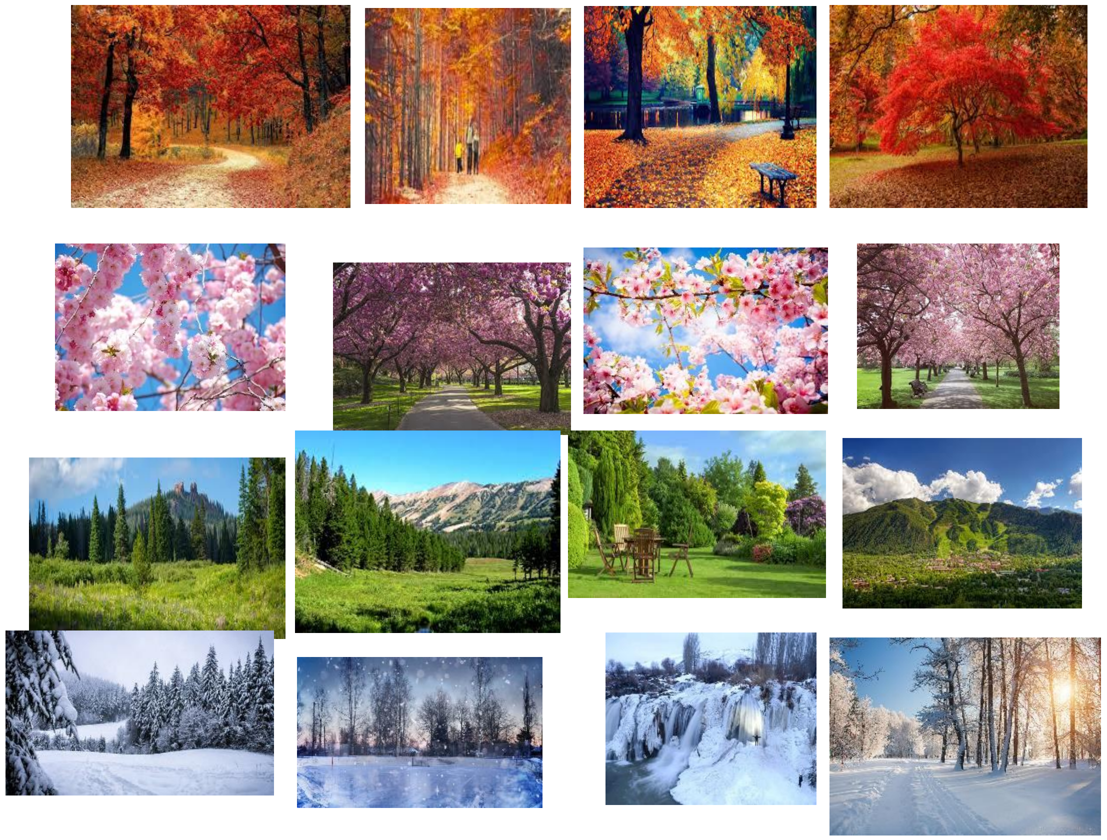
Comparison method that could be one of the following:

- **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`

Exercise



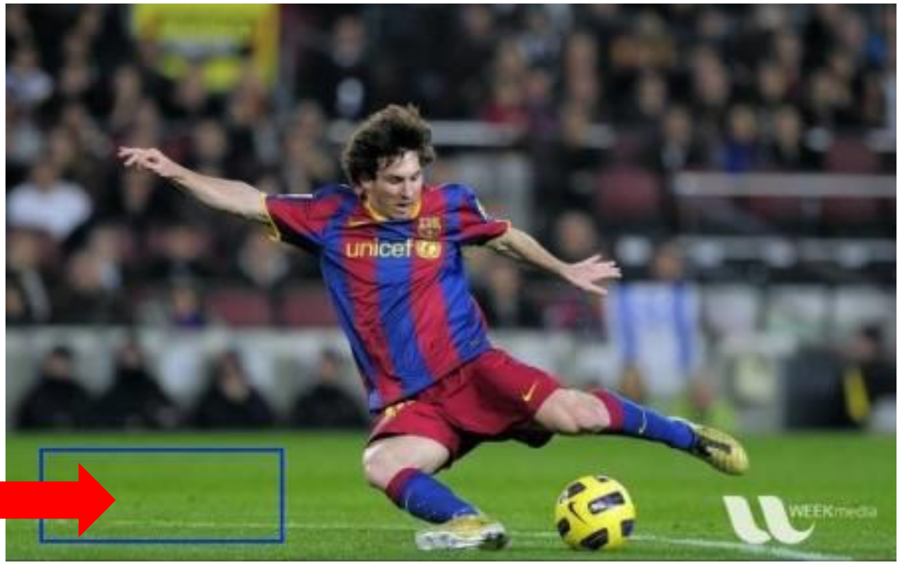
You are given a test image and a collection of images. Find the similarity of image with all images in the collection. Finally sort images from the most similar image to the least similar one.

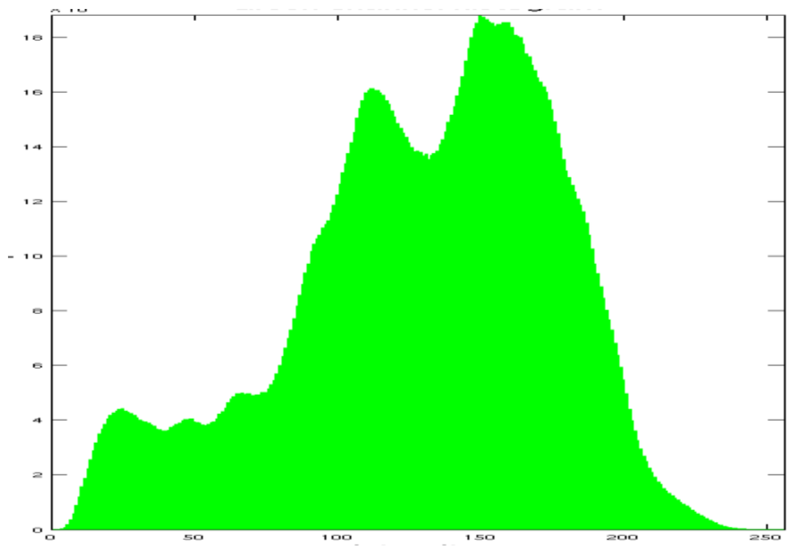


histogram back projection

- Suppose you have an image and you wish to detect specific content inside it.
- if we extract the histogram of this ROI, by normalizing this histogram, we obtain a function that gives us the probability that a pixel of a given intensity value belongs to the defined area.

Region of interest





histogram back projection

- It is used for finding objects of interest in an image.
- In simple words, it creates an image of the same size (but single channel) as that of our input image, where each pixel corresponds to the probability of that pixel belonging to our object.

calcBackProject

Calculates the back projection of a histogram.

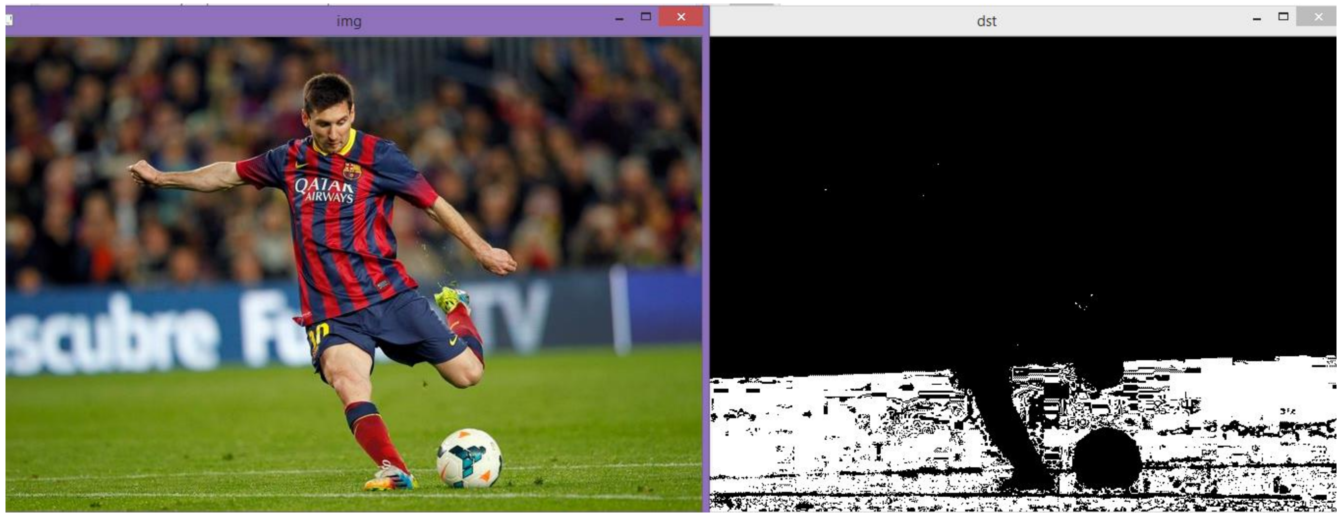
Python: cv2.calcBackProject(images, channels, hist, ranges, scale, dst) → dst

C++: void calcBackProject(const Mat* images, int nimages, const int* channels, InputArray hist, OutputArray backProject, const float** ranges, double scale=1, bool uniform=true)

calcBackProject calculate the back project of the histogram.
That is, similarly to calcHist , at each location (x, y) the function collects the values from the selected channels in the input images and finds the corresponding histogram bin.
But instead of incrementing it, the function reads the bin value, scales it by scale , and stores in backProject(x,y) .
In terms of statistics, the function computes probability of each element value in respect with the empirical probability distribution represented by the histogram.

Let’s write some code…

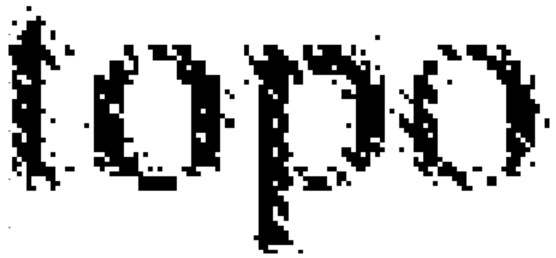
Let’s see histogram back projection in an image



morphology

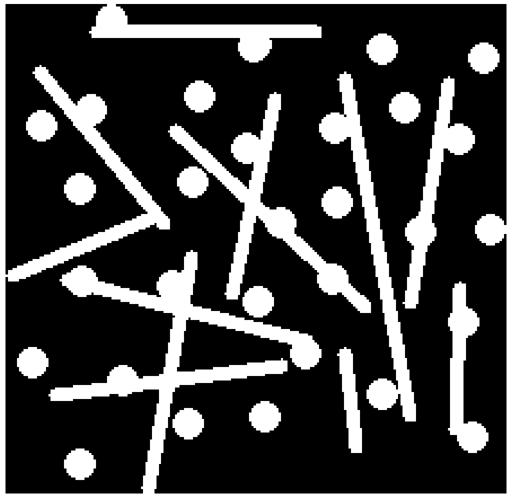
morphology

- Mathematical morphology is a theory that was developed for the analysis and processing of discrete images.
- It defines a series of operators that transform an image by probing it with a predefined shape element (a structuring element).
- The way this shape element intersects the neighborhood of a pixel determines the result of the operation.
- A structuring element can be simply defined as a configuration of pixels (the square shape in the following figure) on which an origin is defined (also called an anchor point).
- structuring elements can be useful to emphasize or eliminate regions of particular shapes.



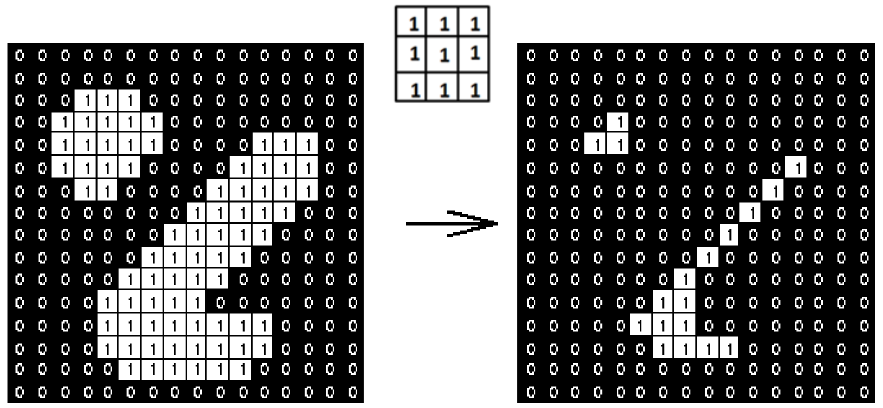
How to remove unwanted noise?

How to separate circles and lines?



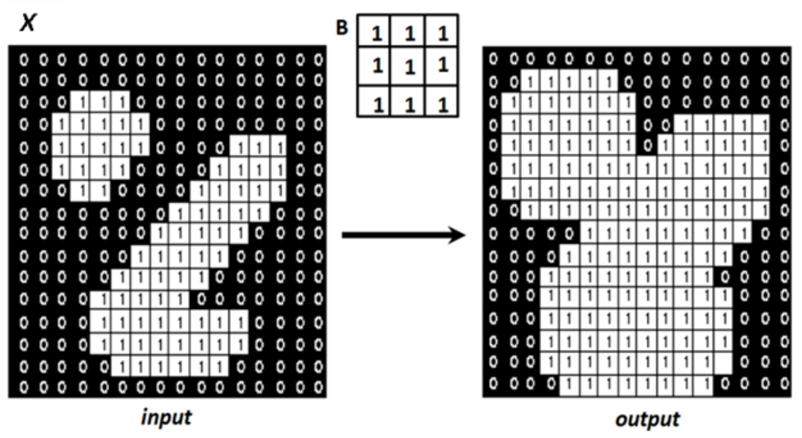
Eroding and dilating images

Erosion replaces the current pixel with the minimum pixel value found in the defined pixel set.



$$\mathrm{dst}(x, y) = \min_{(x',y'):\,\mathrm{element}(x',y')\neq 0} \mathrm{src}(x + x', y + y')$$

Dilation replaces the current pixel with the maximum pixel value found in the defined pixel set.



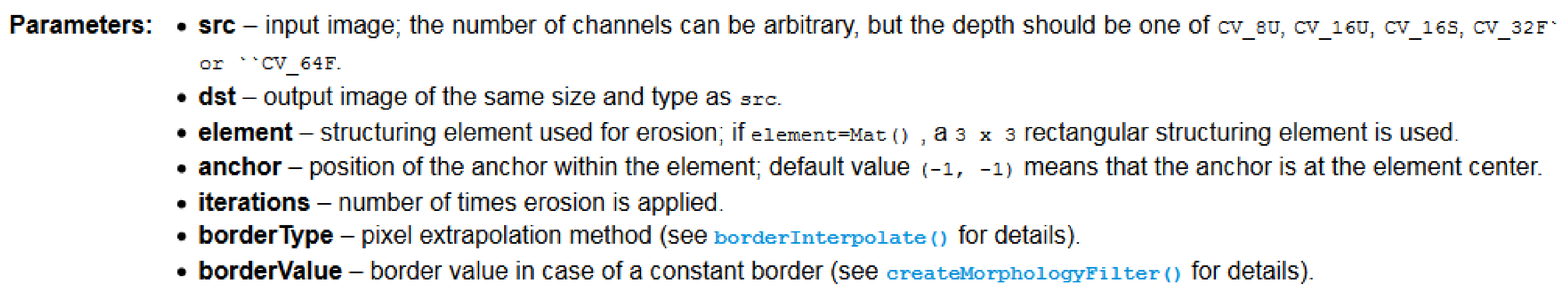
$$\mathrm{dst}(x, y) = \max_{(x',y'):\,\mathrm{element}(x',y')\neq 0} \mathrm{src}(x + x', y + y')$$

Erosion

erode()

Python: cv2.erode(src, kernel, dst, anchor, iterations, borderType, borderValue) → dst

C++: void erode(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())



Parameters:

- **src** – input image; the number of channels can be arbitrary, but the depth should be one of `CV_8U`, `CV_16U`, `CV_16S`, `CV_32F`` or ``CV_64F`.
- **dst** – output image of the same size and type as `src`.
- **element** – structuring element used for erosion; if `element=Mat()` , a `3 x 3` rectangular structuring element is used.
- **anchor** – position of the anchor within the element; default value `(-1, -1)` means that the anchor is at the element center.
- **iterations** – number of times erosion is applied.
- **borderType** – pixel extrapolation method (see `borderInterpolate()` for details).
- **borderValue** – border value in case of a constant border (see `createMorphologyFilter()` for details).

Dilation

dilate()

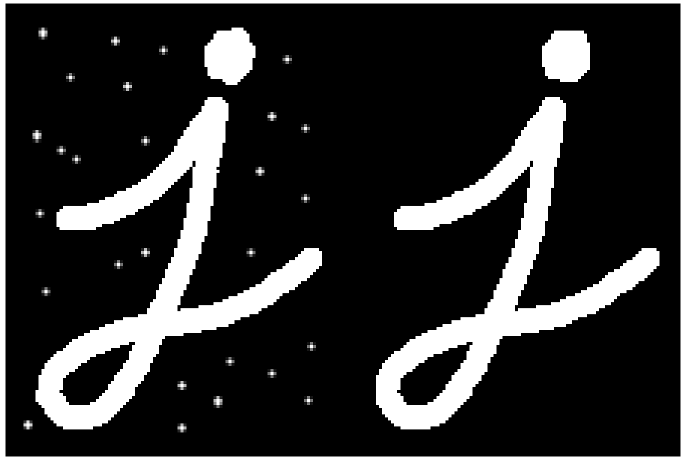
Dilates an image by using a specific structuring element.

Python: cv2.dilate(src, kernel, dst, anchor, iterations, borderType, borderValue) → dst

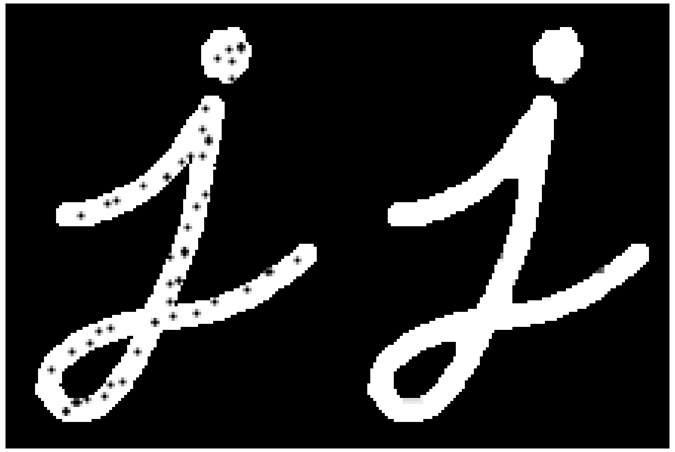
C++: void dilate(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())

Advanced morphological transformations

Opening is defined as an erosion followed by a dilation *using the same structuring element for both operations*.

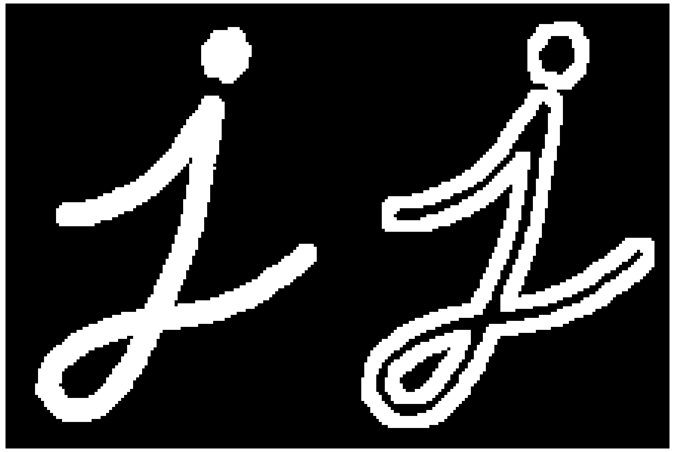


Closing is defined as a dilation followed by an erosion *using the same structuring element for both operations*



Advanced morphological transformations

Morphological Gradient
It is the difference between dilation and erosion of an image



Black Hat
It is the difference between the closing of the input image and input image.

Top Hat
It is the difference between input image and Opening of the image.

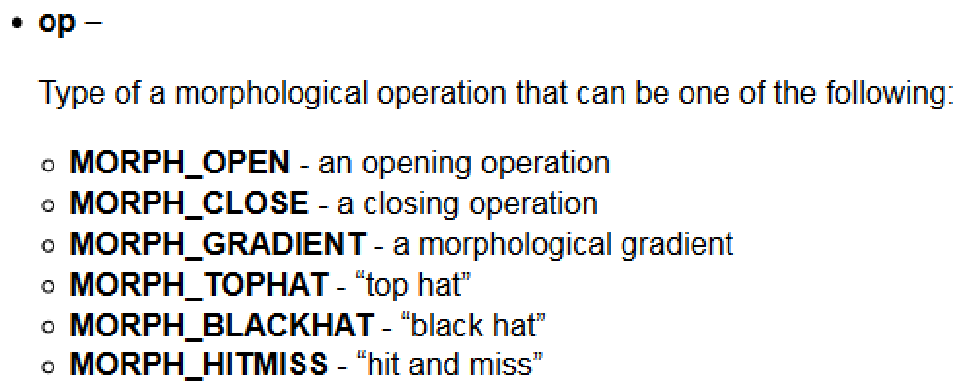
Advanced morphological transformations

morphologyEx()

- Performs advanced morphological transformations.

Python: cv2.morphologyEx(src, op, kernel, dst, anchor, iterations, borderType, borderValue) → dst

C++: void morphologyEx(InputArray src, OutputArray dst, int op, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())



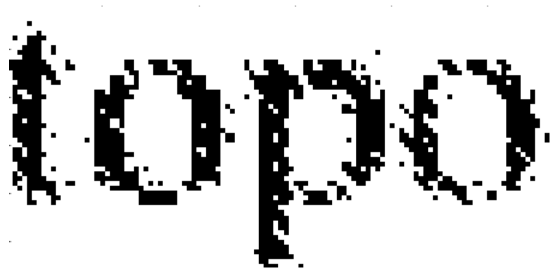
- **op** –

  Type of a morphological operation that can be one of the following:

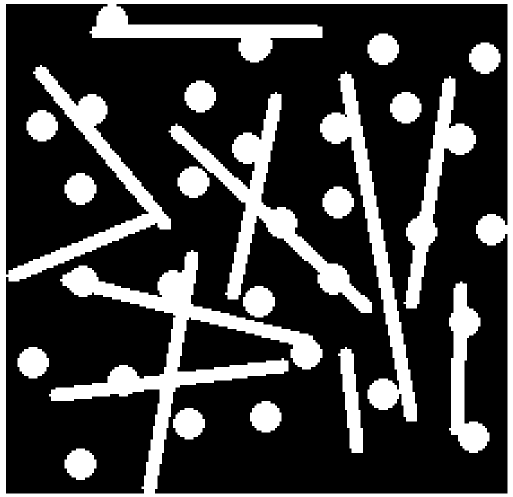
  - **MORPH_OPEN** - an opening operation
  - **MORPH_CLOSE** - a closing operation
  - **MORPH_GRADIENT** - a morphological gradient
  - **MORPH_TOPHAT** - “top hat”
  - **MORPH_BLACKHAT** - “black hat”
  - **MORPH_HITMISS** - “hit and miss”

Let’s write some code…

Let’s remove unwanted noise from this image



Let’s separate circles.



Let’s write some code…

Let’s write a color detector …

THE END