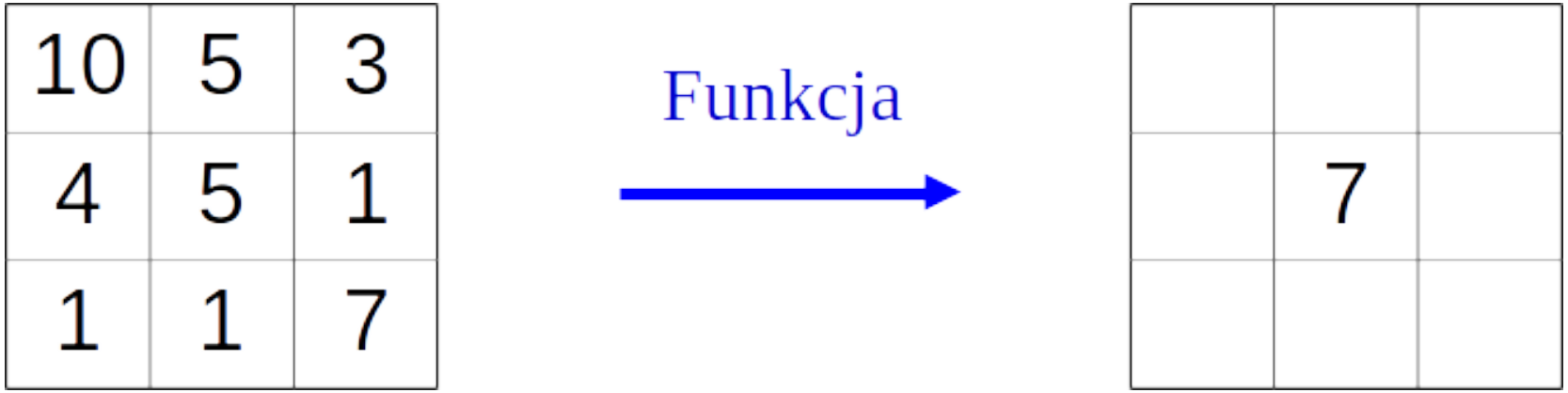
Machine Vision

Image Processing - Filtering lecture 3

Adam Szmigielski
aszmigie@pjwstk.edu.pl
materials: $ftp(public) : //aszmigie/WMA$

Image filter

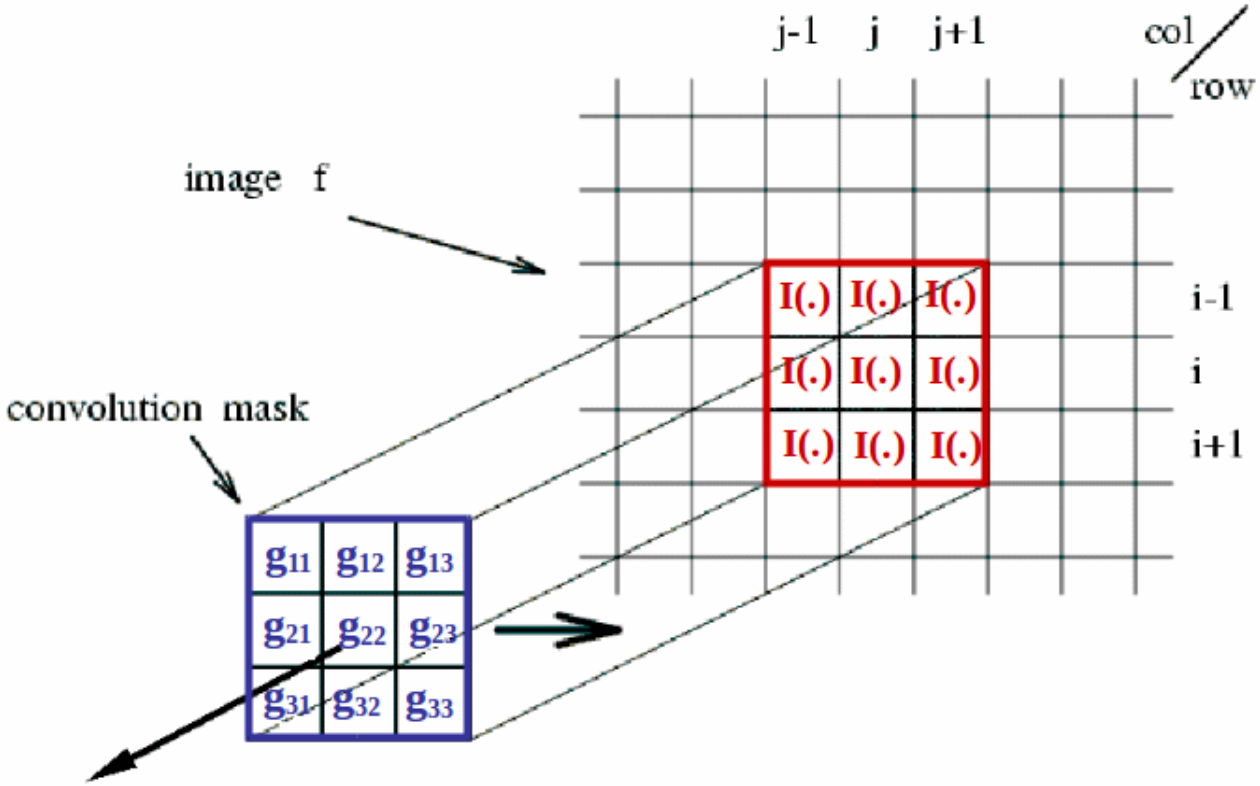
Modifies image pixels based on neighborhood pixels



Linear filtration

- linear is the simplest and most useful,
- Replaces each pixel with a linear combination of neighbors,
- The function (way) for a linear combination is called a *convolution kernel*.

Line filter - convolution



$$f(i,j) = g_{11}\, I(i-1,j-1) + g_{12}\, I(i-1,j) + g_{13}\, I(i-1,j+1) + g_{21}\, I(i,j-1) + g_{22}\, I(i,j) + g_{23}\, I(i,j+1) + g_{31}\, I(i+1,j-1) + g_{32}\, I(i+1,j) + g_{33}\, I(i+1,j+1)$$

Neighborhood operator as linear filter

45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

$f(x,y)$

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

$h(x,y)$

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

$g(x,y)$

- The image on the left is convolved with the filter in the middle to yield the image on the right.
- The light blue pixels indicate the source neighborhood for the light green destination pixel.
- Output pixel's value is determined as a weighted sum of input pixel

values within a small neighborhood N:

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l).$$

- The entries in the weight *kernel* or *mask* $h(k,l)$ are often called the *filter coefficients*. The above **correlation operator** can be notated as

$$g = f \otimes h.$$

A common variant on this formula is:

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l),$$

where the sign of the offsets in f has been reversed, This is called the **convolution operator**,

$$g = f * h$$

and h is then called the *impulse response function* - The continuous version of convolution can be written as $g(x) = \int f(x-u)h(u)du$.

Correlation and convolution as matrix-vector multiplying

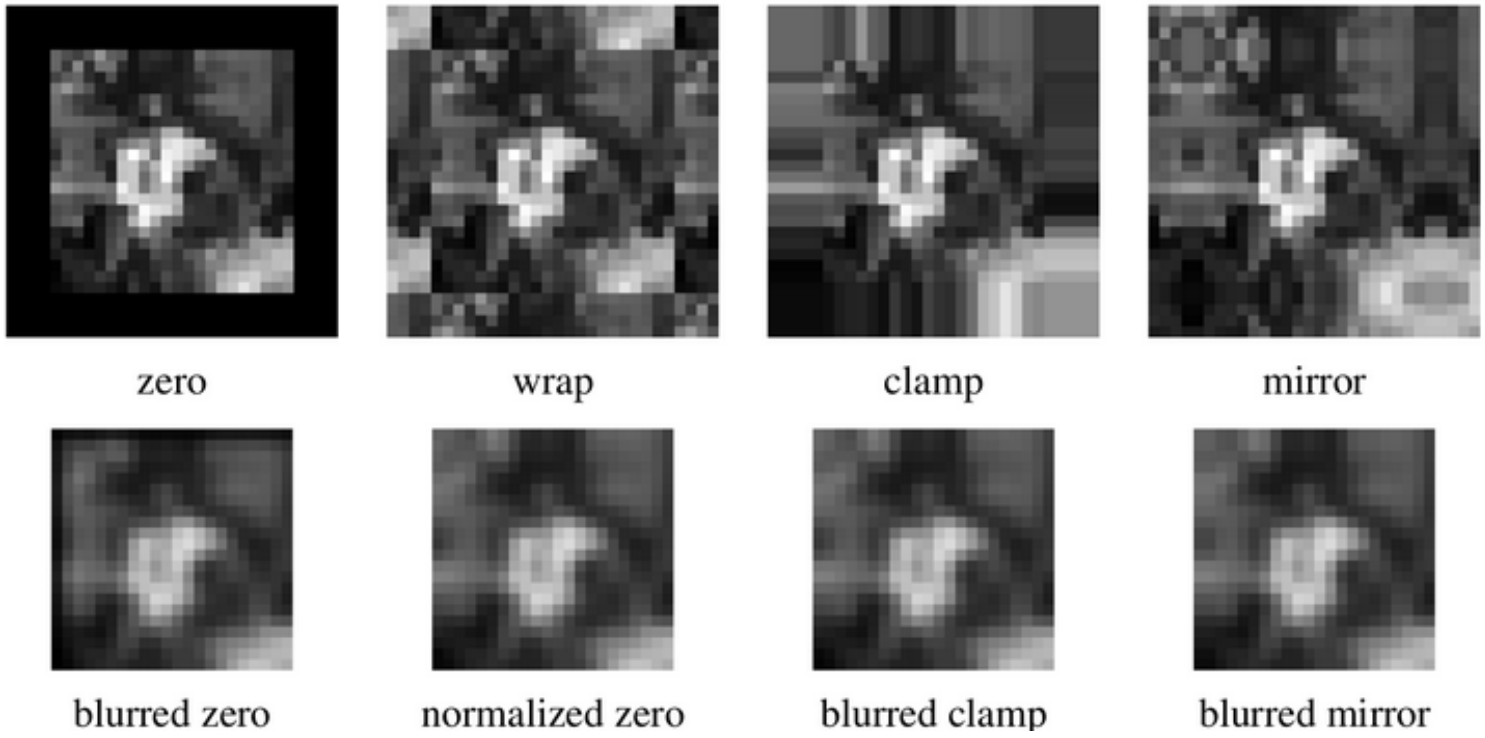
$$\boxed{72}\boxed{88}\boxed{62}\boxed{52}\boxed{37} * \boxed{1/4}\boxed{1/2}\boxed{1/4} \quad \Leftrightarrow \quad \frac{1}{4}\begin{bmatrix} 2 & 1 & . & . & . \\ 1 & 2 & 1 & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 1 & 2 & 1 \\ . & . & . & 1 & 2 \end{bmatrix} \begin{bmatrix} 72 \\ 88 \\ 62 \\ 52 \\ 37 \end{bmatrix}$$

- One-dimensional convolution can be represented in matrix-vector form.
- *Correlation* and *convolution* can both be written as a matrix-vector multiply, if we first convert the two-dimensional images $f(i, j)$ and $g(i, j)$ into raster-ordered vectors f and g:

$$g = Hf,$$

where the (sparse) H matrix contains the convolution kernels.

Padding (border effects)



- Correlation result is smaller than the original image, which may not be desirable in many applications.
- This is because the neighborhoods of typical correlation and convolution operations extend beyond the image boundaries

- A number of different padding or extension modes have been developed for neighborhood operations:
  - *zero*: set all pixels outside the source image to 0 (a good choice for alpha-matted cutout images);
  - *constant (border color)*: set all pixels outside the source image to a specified border value;
  - *clamp (replicate or clamp to edge)*: repeat edge pixels indefinitely;
  - *(cyclic) wrap (repeat or tile)*: loop "around" the image in a "toroidal" configuration;
  - *mirror*: reflect pixels across the image edge;
  - *extend*: extend the signal by subtracting the mirrored version of the signal from the edge pixel value.

Separable filtering

$\frac{1}{K^2}$
1	1	⋯	1
1	1	⋯	1
⋮	⋮	1	⋮
1	1	⋯	1

$\frac{1}{16}$
1	2	1
2	4	2
1	2	1

$\frac{1}{256}$
1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

$\frac{1}{8}$
−1	0	1
−2	0	2
−1	0	1

$\frac{1}{4}$
1	−2	1
−2	4	−2
1	−2	1

$\frac{1}{K}$ (1 1 ⋯ 1) $\frac{1}{4}$ (1 2 1) $\frac{1}{16}$ (1 4 6 4 1) $\frac{1}{2}$ (−1 0 1) $\frac{1}{2}$ (1 −2 1)

(a) box, $K = 5$ (b) bilinear (c) "Gaussian" (d) Sobel (e) corner

- The process of performing a convolution requires K^2 (multiply-add) operations per pixel, where K is the size
- In many cases, this operation can be significantly sped up by first performing a one-dimensional horizontal convolution followed by a one-dimensional vertical convolution, which requires a total of $2K$ operations per pixel.

- A convolution kernel for which this is possible is said to be separable.
- Two-dimensional kernel K corresponding to successive convolution with a horizontal kernel h and a vertical kernel v is the outer product of the two kernels,

$$K = vh^T$$

- Design of convolution kernels for computer vision applications is often influenced by their separability.
- This can often be done by inspection or by looking at the analytic form of the kernel
- more direct method is to treat the 2D kernel as a 2D matrix K and to take its singular value decomposition (SVD),

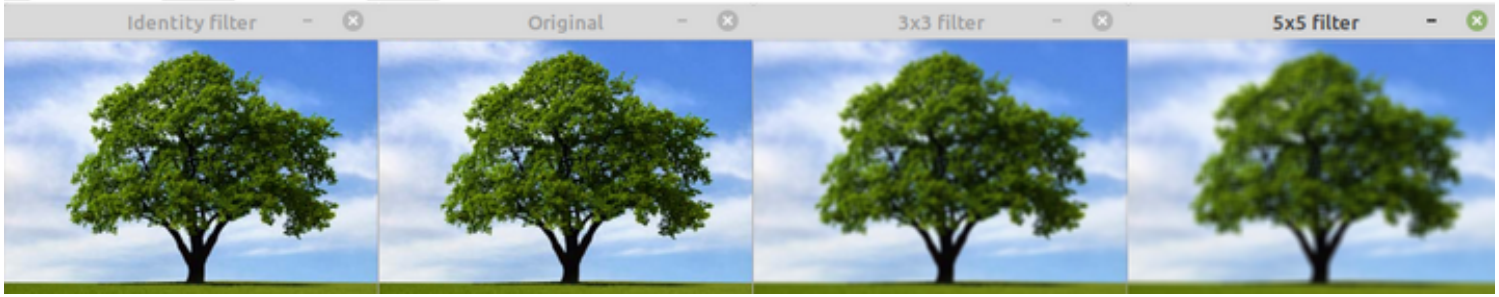
Median filtering

- Median filter selects the median value from each pixel's neighborhood,
- Median values can be computed in expected linear time,
- Since the noise value usually lies well outside the true values in the neighborhood, the median filter is able to filter away such bad pixels.
- Another possibility is to compute a weighted median - this is equivalent to minimizing the weighted objective function:

$$\sum_{k,l} w(k,l)|f(i+k,j+l) - g(i,j)|^p,$$

where $g(i,j)$ is the desired output value and $p = 1$ for the weighted median. The value $p = 2$ is the usual weighted mean, which is equivalent to correlation.

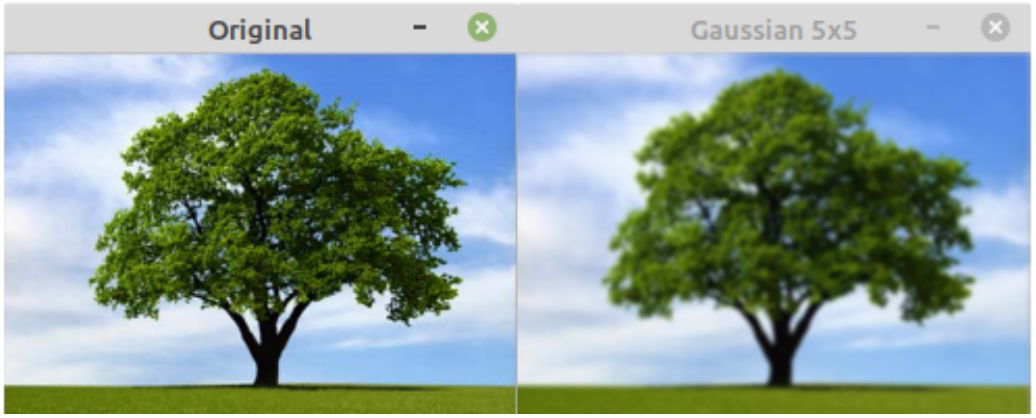
2D Convolution (Image Filtering) - Python




```
import cv2
import numpy as np

img = cv2.imread('images/tree_input.png', cv2.IMREAD_COLOR)

kernel_identity = np.array([[0,0,0], [0,1,0], [0,0,0]])
kernel_3x3 = np.ones((3,3), np.float32) / 9.0
kernel_5x5 = np.ones((5,5), np.float32) / 25.0

cv2.imshow('Original', img)
output = cv2.filter2D(img, -1, kernel_identity) # value -1 is to maintain source image
cv2.imshow('Identity filter', output)
output = cv2.filter2D(img, -1, kernel_3x3)
cv2.imshow('3x3 filter', output)
output = cv2.filter2D(img, -1, kernel_5x5)
cv2.imshow('5x5 filter', output)
cv2.waitKey(0)
```

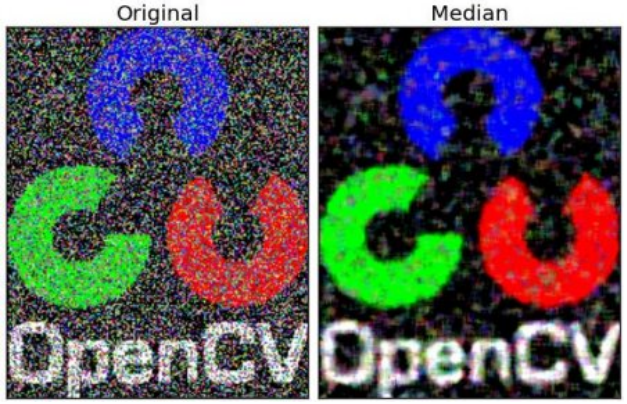
Image Blurring (Image Smoothing)

- **Averaging -** This is done by convolving an image with a normalized box filter e.g $K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
- **Gaussian Blurring -** Instead of a box filter, a Gaussian kernel is used.




```
blur = cv2.GaussianBlur(img,(5,5),0)
```

- **Median Blurring -** takes the median of all the pixels under the kernel area and the central element is replaced with this median value. This is highly effective against salt-and-pepper noise in an image.




```
median = cv2.medianBlur(img,5)
```

Non-linear filtering

1	2	1	2	4
2	1	3	5	8
1	3	7	6	9
3	4	8	6	7
4	5	7	8	9

(a) median = 4

1	2	1	2	4
2	1	3	5	8
1	3	7	6	9
3	4	8	6	7
4	5	7	8	9

(b) α-mean= 4.6

	2	1	0	1	2
2	0.1	0.3	0.4	0.3	0.1
1	0.3	0.6	0.8	0.6	0.3
0	0.4	0.8	1.0	0.8	0.4
1	0.3	0.6	0.8	0.6	0.3
2	0.1	0.3	0.4	0.3	0.1

(c) domain filter

0.0	0.0	0.0	0.0	0.2
0.0	0.0	0.0	0.4	0.8
0.0	0.0	1.0	0.8	0.4
0.0	0.2	0.8	0.8	1.0
0.2	0.4	1.0	0.8	0.4

(d) range filter

- In linear filter output pixel is a weighted summation of some number of input pixels.
- In many cases, however, better performance can be obtained by using a non-linear combination of neighboring pixels.

Bilateral filtering

- In the bilateral filter, the output pixel value depends on a weighted combination of neighboring pixel values

$$g(i,j) = \frac{\sum_{k,l} f(k,l) w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

The weighting coefficient $w(i,j,k,l)$ depends on the product of a domain kernel ,

$$d(i,j,k,l) = exp(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2})$$

and a data-dependent range kernel:

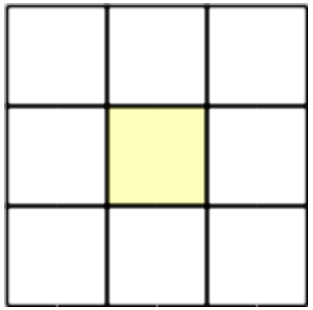
$$d(i,j,k,l) = exp(-\frac{||f(i,j) - f(k,l)||^2}{2\sigma_r^2})$$

When multiplied together, these yield the data-dependent bilateral weight function

$$d(i,j,k,l) = exp(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2} - \frac{||f(i,j)-f(k,l)||^2}{2\sigma_r^2})$$

Bilateral Filtering $cv.bilateralFilter()$ is highly effective in noise removal while keeping edges sharp. But the operation is slower compared to other filters.

Morphological operations



- The basic concept of morphological transformation is the so-called structural element of the image,
- This is a certain section of the image (a subset of elements) with one point - the central point,
- Structural element (circle with unit radius) on a square grid.
- The structuring element can be any shape, from a simple 3×3 box filter, to more complicated disc structures.

Structuring Element - Python

We may create a structuring elements with help of Numpy. In some cases, you may need elliptical or circular shaped kernels. For this purpose, OpenCV has a function, *cv.getStructuringElement*().

```
# Rectangular Kernel
>>> cv.getStructuringElement(cv.MORPH_RECT,(5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)
# Elliptical Kernel
>>> cv.getStructuringElement(cv.MORPH_ELLIPSE,(5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)
# Cross-shaped Kernel
>>> cv.getStructuringElement(cv.MORPH_CROSS,(5,5))
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

General algorithm of morphological transformation

- The structural element is moved around the whole image and for each point of the image a comparison of image points and the structural element is made,
- At each point of the image, it is checked if the actual configuration of the pixels of the image in the vicinity of this point is consistent with the model structural element,
- If a match between the image pixel pattern and the structural element template is detected, some operation is performed on the tested point.

Morphological operations

- **Erosion** - an eroded figure is a set of all centers of circles with a radius r that are entirely contained within the area of X
- **Dylation** - the dylation figure is a set of centers of all B circles for which at least one point coincides with any point of the initial figure
- **Opening** - involves rolling wheel B on the inside of the figure's edge and rejecting all those points that cannot be reached by the circle.
- **Closing** - involves rolling wheel B on the outside of the figure's edge and adding to it all those points that cannot be reached by the circle.

Morphology - examples

The most common binary image operations are called morphological operations, since they change the shape of the underlying binary objects.

(a) (b) (c) (d) (e) (f)

- Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing. The structuring element for all examples is a 5×5 square. The effects of majority are a subtle rounding of sharp corners. Opening fails to eliminate the dot, since it is not wide enough.

Erosion

The basic idea of erosion is just like soil erosion only, it erodes away the boundaries of foreground object .

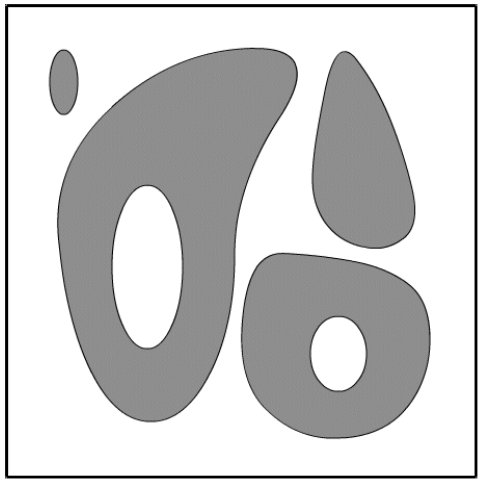


figura przed erozją

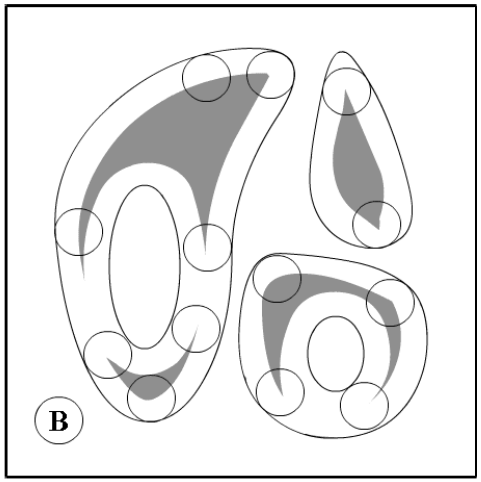


figura po erozji

```
import cv2 as cv
import numpy as np
img = cv.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv.erode(img,kernel,iterations = 1))
```

Dilation

It is just opposite of erosion. It increases the region in the image or size of foreground object increases

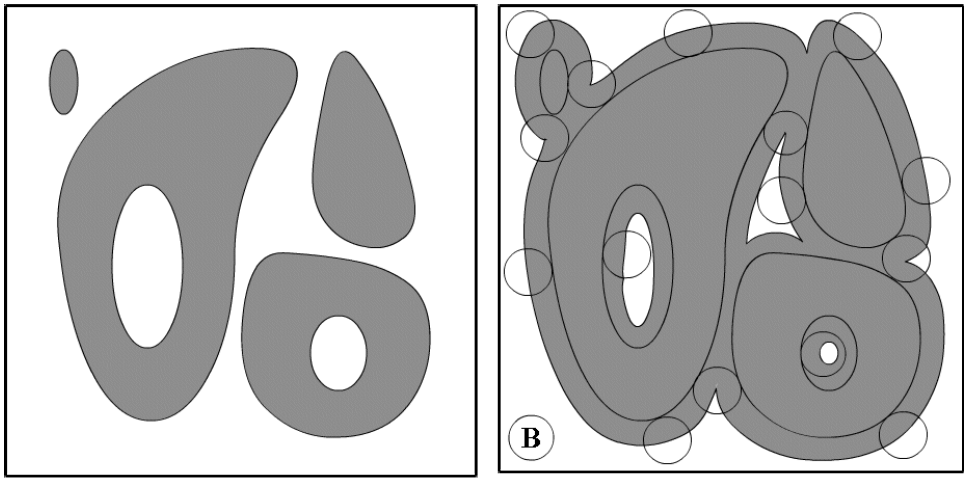


Figura przed dylatacją

figura po dylatacji

```
dilation = cv.dilate(img,kernel,iterations = 1)
```

Opening

Opening is just another name of erosion followed by dilation. It is useful in removing noise, as we explained above.

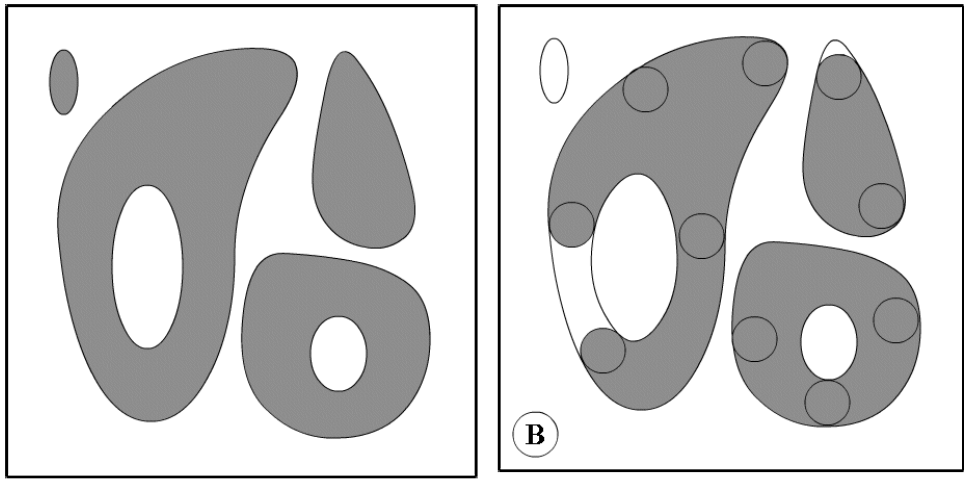


figura przed otwarciem

figura po otwarcia

- Opening = erosion + dylation: $O(x) = D(E(x))$

```
opening = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)
```

Closing

Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small points on the object.

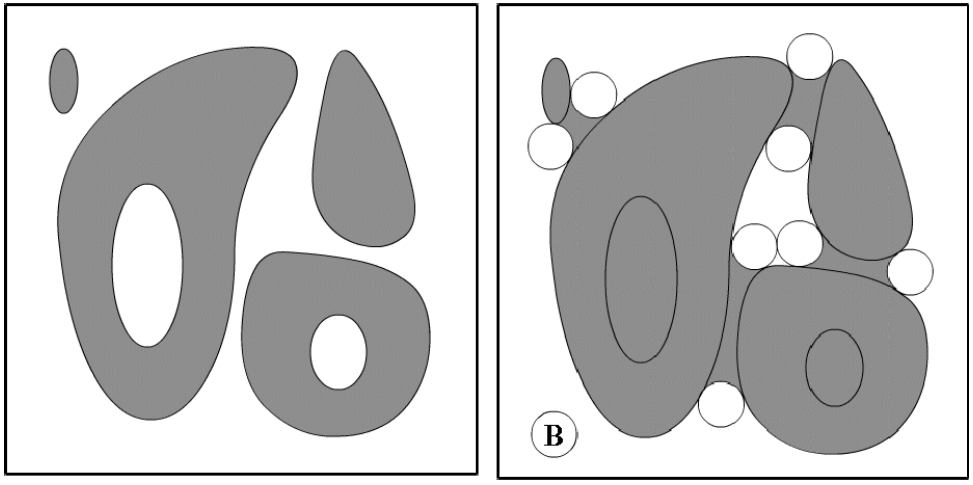


figura przed zamknięciem

figura po zamknięciu

- Closing = dylation + erosion: $C(x) = E(D(x))$

```
closing = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)
```

Morphological Gradient

It is the difference between dilation and erosion of an image.

The result will look like the outline of the object

- Gradient = dylation - erosion: $G(x) = D(x) - E(x)$

```
gradient = cv.morphologyEx(img, cv.MORPH_GRADIENT, kernel)
```

Top Hat

It is the difference between input image and Opening of the image. Below example is done for a 9×9 kernel.

- TopHat = Image - opening: $TH(x) = x - O(x)$

```
tophat = cv.morphologyEx(img, cv.MORPH_TOPHAT, kernel)
```

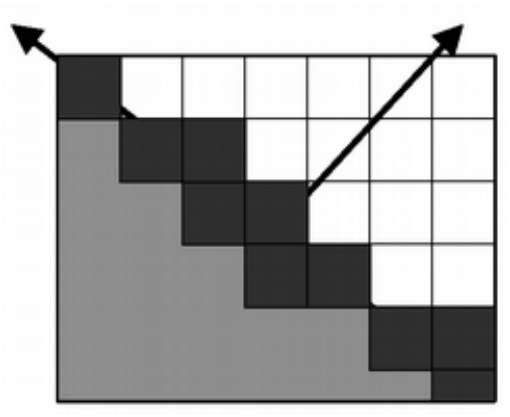
Black Hat

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

- BlackHat = Image - opening: $BH(x) = C(x) - x$

```
blackhat = cv.morphologyEx(img, cv.MORPH_BLACKHAT, kernel)
```

Normal to the edge



- Edge Size: $S = \sqrt{dx^2 + dy^2}$
- Edge direction: $\alpha = arctan(\frac{dy}{dx})$

Sobel's operator

$S_1 =$

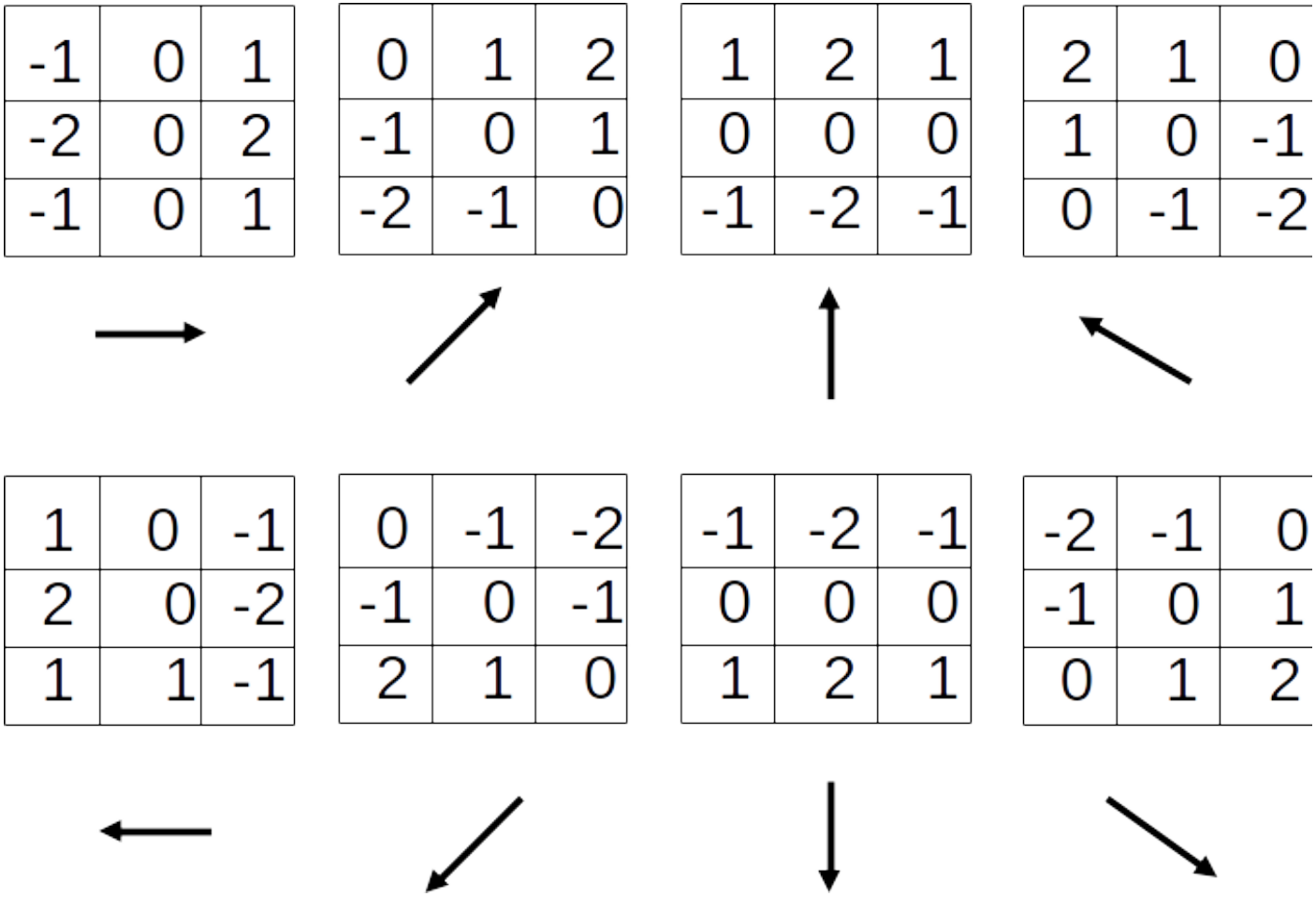
-1	-2	-1
0	0	0
1	2	1

$S_2 =$

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

- Edge Size: $\sqrt{S_1^2 + S_2^2}$
- Edge direction: $arctan(\frac{S_1}{S_2})$

Different directions of edge filtration



Other linear filters

1	1	1
1	-2	1
-1	-1	-1

Prewitt 1

5	5	5
-3	0	-3
-3	-3	-3

Kirsch

-1	$-\sqrt{2}$	-1
0	0	0
1	$\sqrt{2}$	1

Frei & Chen

1	1	1
0	0	0
-1	-1	-1

Prewitt 2

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

Sobel

- You can design your own linear filter.

Canny edge detection

Filtration stages:

1. stage 1 - smoothing,
2. stage 2 - calculating the gradient,
3. stage 3 - remove the maximum pixels.

Corner detection

Corner in the concept of " sharply curving edge ", intersection of 2 edges, lines

- Type of points of interest, points of interest (other: line endings, light or dark points),
- The first algorithms that detect corners:
  - edge detection,
  - tracing the extracted edges and detecting a sudden change in their direction,
- Newer generation of algorithms - high gradient curvature,
- It is recommended to smooth the image earlier.

Distance transforms

The distance transform is useful in quickly precomputing the distance to a curve or set of points.

Two commonly used metrics:

- city block

$$d_1(k,l) = |k| + |l|$$

- Manhattan distance

$$d_2(k,l) = \sqrt{(k^2 + l^2)}$$

Connected components

Another useful semi-global image operation is finding connected components, which are defined as regions of adjacent pixels that have the same input value or label.

Such statistics include for each individual region R:

- the area (number of pixels),
- the perimeter (number of boundary pixels),
- the centroid (average x and y values),
- the second moments,

$$M = \sum_{(x,y) \in R} \begin{bmatrix} x - \overline{x} \\ y - \overline{y} \end{bmatrix} \cdot \begin{bmatrix} x - \overline{x} & y - \overline{y} \end{bmatrix}$$

  from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis.