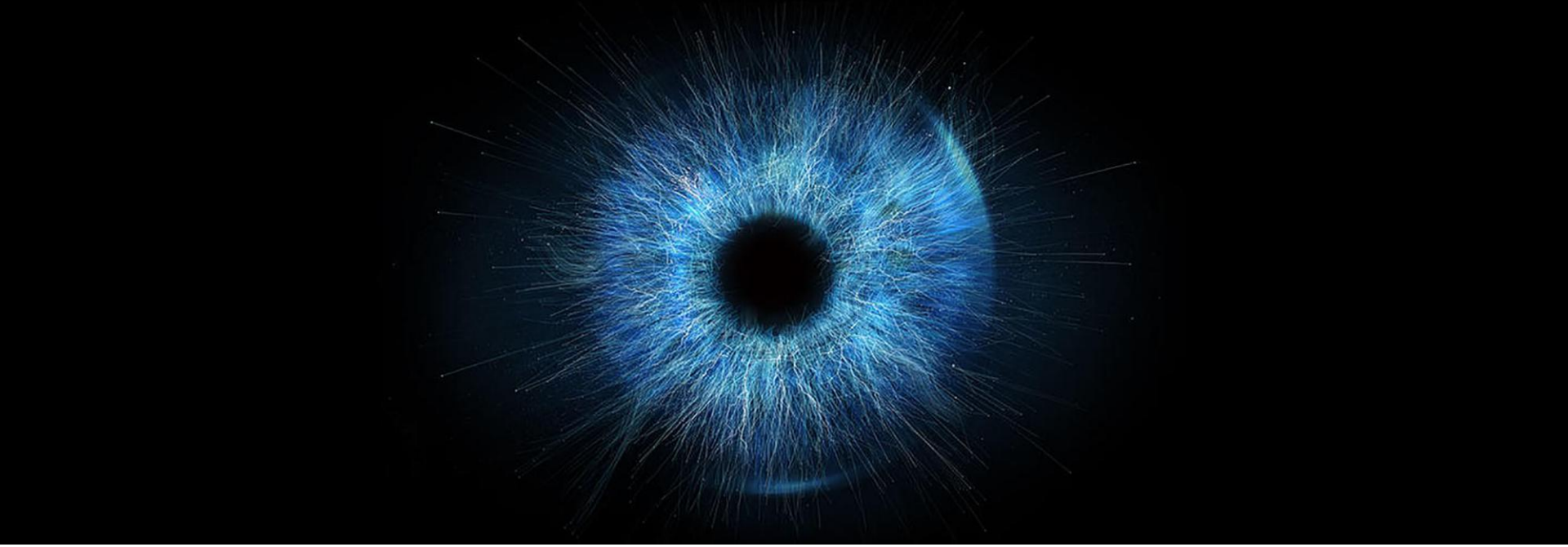


Exercise – OpenCV basics

Machine vision algorithms

Doc.dr.sc. Filip Šuligoj
fsuligoj@fsb.hr

University of Zagreb
FSB Faculty of mechanical engineering and naval arhitecture

OpenCV modules

OpenCV Overview

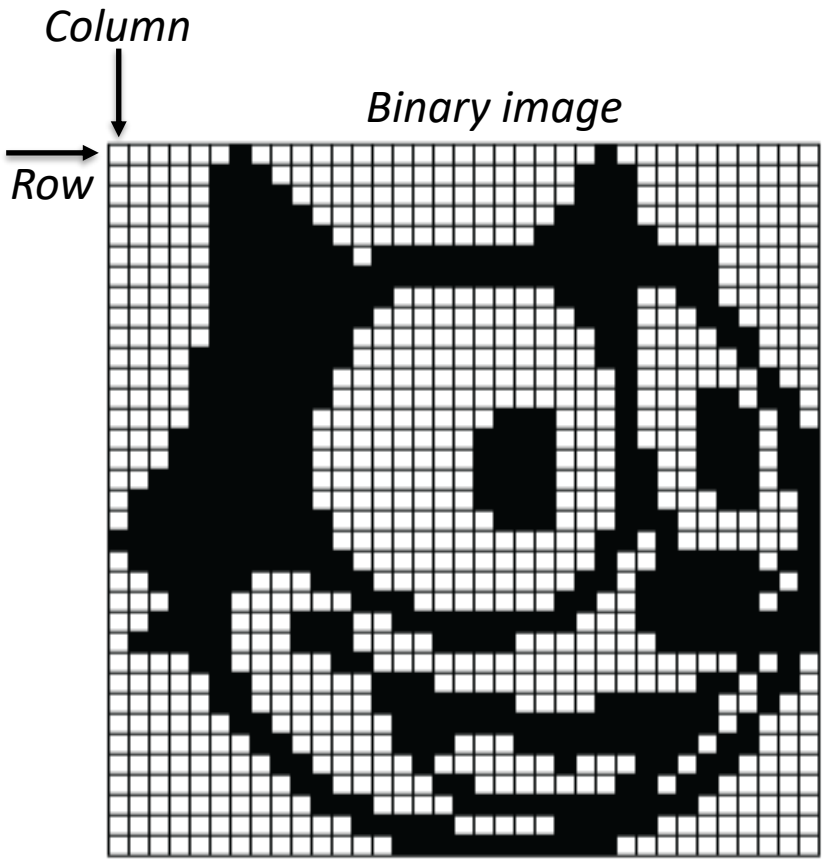
OpenCV (Open Source Computer Vision Library) is a widely used open-source library offering hundreds of computer vision algorithms. Primarily based on a C++ API, it supports real-time image and video processing across various platforms.

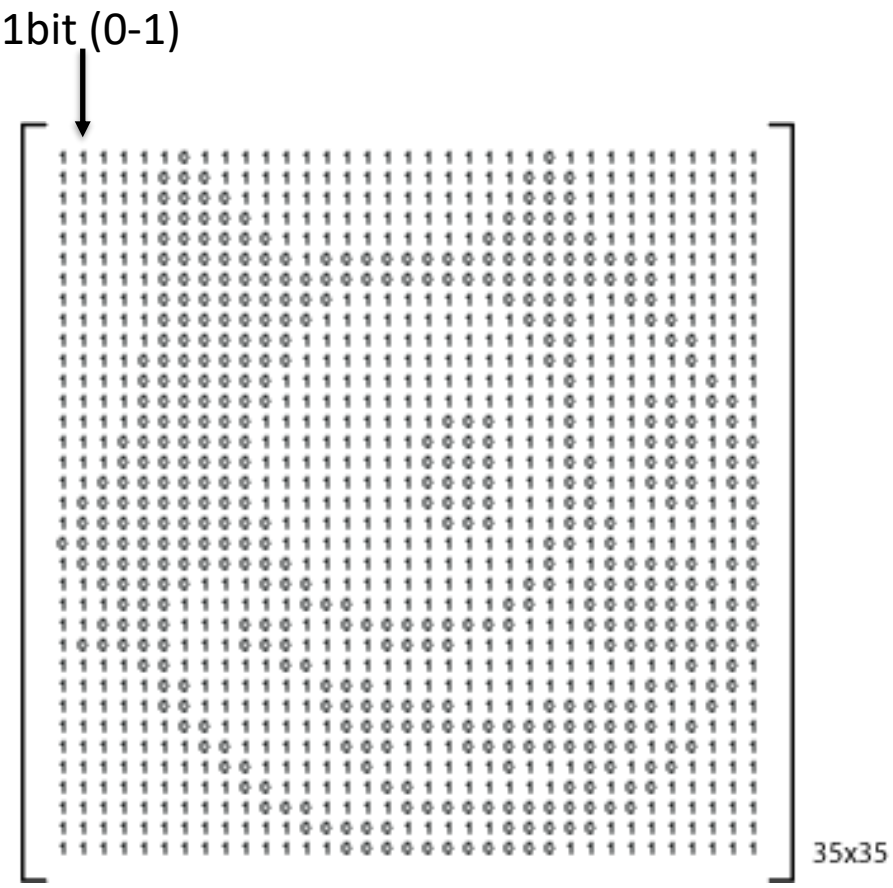
Key Modules:

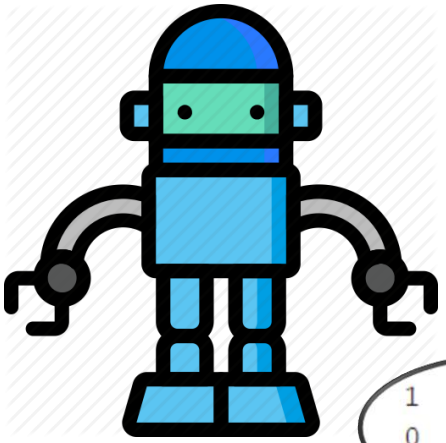
- ***core*** *– Basic data structures (e.g., Mat) and functions.*
- ***imgproc*** *– Image processing (filters, transformations, color conversion).*
- ***video*** *– Motion analysis, background subtraction, tracking.*
- ***calib3d*** *– Camera calibration, pose estimation, 3D reconstruction.*
- ***features2d*** *– Feature detection, description, and matching.*
- ***objdetect*** *– Object detection (e.g., faces, people, cars).*
- ***highgui*** *– Simple GUI for image/video display.*
- ***videoio*** *– Video capture and codec interface.*
- ***Others*** *– Python bindings, FLANN, testing tools, etc.*

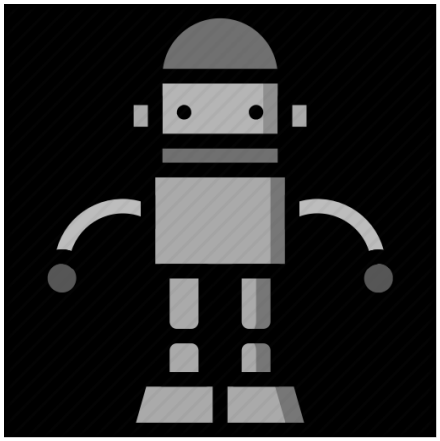
More info: opencv.org

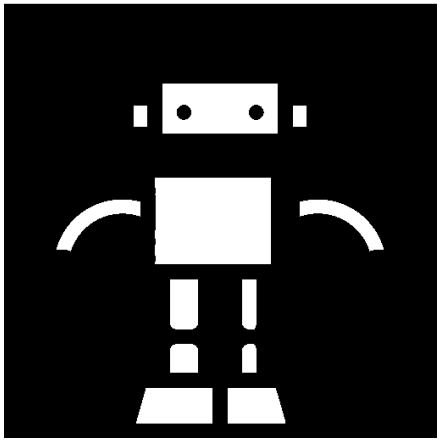
2D image matrix











Color 3 channels

Grayscale 1 channel

Monochrome 1 channel

OpenCV minimum project example

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(OpenCV)

set(CMAKE_CXX_STANDARD          17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS        OFF)

find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )

add_executable(OpenCV src/test.cpp)

target_link_libraries(OpenCV ${OpenCV_LIBS} )
```

test.cpp

```
#include <opencv2/opencv.hpp>
#include <iostream>
int main() {
   // Print OpenCV version
   std::cout << "OpenCV version: " << CV_VERSION <<
std::endl;
   // Create a simple image
   cv::Mat image = cv::Mat::zeros(300, 300, CV_8UC3);
   // Draw a red circle
   cv::circle(image, cv::Point(150, 150), 50, cv::Scalar(0, 0,
255), -1);
   // Show the image
   cv::imshow("OpenCV Test", image);
   cv::waitKey(0); // Wait for key press

   return 0;
}
```

Basic variables and commands in OpenCV

Mat → Image matrix variable

- Imread(„image_name”,1) → read from file
- Declare matrix of size and type → Mat::zeros(Size(500, 500), CV_8U)

Mat A, C; // creates just the header parts
A = imread(”image_name”, IMREAD_COLOR); // here we'll know the method used (allocate matrix)
Mat B(A); // Use the copy constructor
C = A; // Assignment operator

All the objects, in the end, point to the same single data matrix and making a modification using any of them will affect all the other ones as well.

Mat F = A.clone();
Mat G;
A.copyTo(G);

Now modifying F or G will not affect the matrix pointed by the A's header

Imshow(„window_name”,mat_name) → **show image matrix in a window**

waitKey() → **The function waitKey waits for a key event infinitely (when delay≤0) or for delay milliseconds, when it is positive**

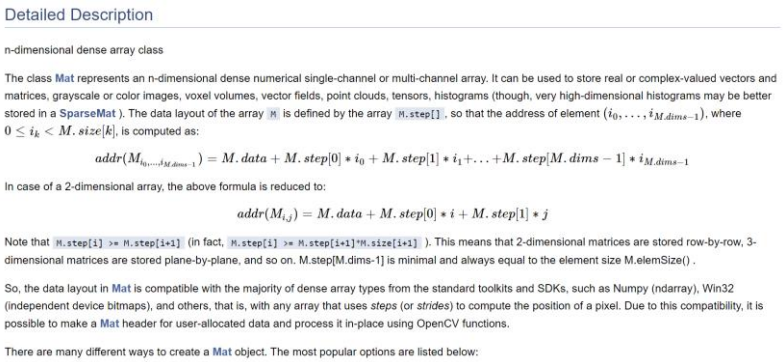
Color conversion → cvtColor(image,image,COLOR_RGB2GRAY);

resize() → Resizes an image.
resize (InputArray src, OutputArray dst, Size dsize, double fx = 0, double fy = 0, int interpolation = INTER_LINEAR)

Use OpenCV documentation → https://docs.opencv.org/4.5.5/d1/dfb/intro.html

OpenCV documentation - examples

Mat → https://docs.opencv.org/4.5.5/d3/d63/classcv_1_1Mat.html#details



Detailed Description

n-dimensional dense array class

The class **Mat** represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or color images, voxel volumes, vector fields, point clouds, tensors, histograms (though, very high-dimensional histograms may be better stored in a **SparseMat**). The data layout of the array `M` is defined by the array `M.step[]` , so that the address of element $(i_0,\ldots,i_{M.dims-1})$, where $0 \le i_k < M.size[k]$, is computed as:

$$addr(M_{i_0,\ldots,i_{M.dims-1}}) = M.data + M.step[0]*i_0 + M.step[1]*i_1 + \ldots + M.step[M.dims-1]*i_{M.dims-1}$$

In case of a 2-dimensional array, the above formula is reduced to:

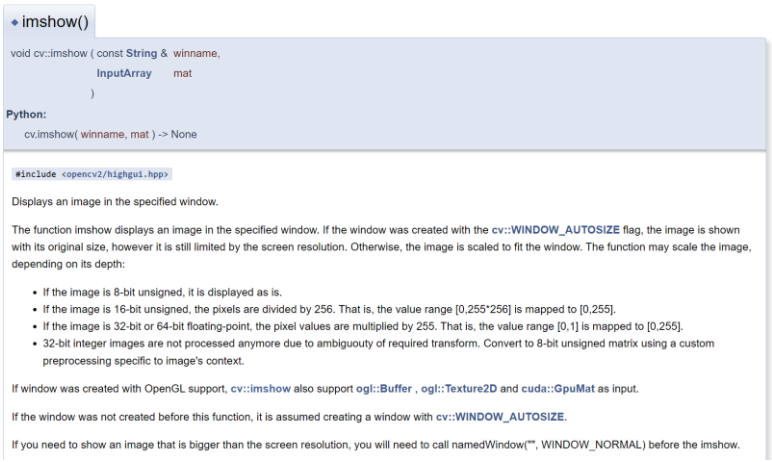
$$addr(M_{i,j}) = M.data + M.step[0]*i + M.step[1]*j$$

Note that `M.step[i] >= M.step[i+1]` (in fact, `M.step[i] >= M.step[i+1]*M.size[i+1]`). This means that 2-dimensional matrices are stored row-by-row, 3-dimensional matrices are stored plane-by-plane, and so on. M.step[M.dims-1] is minimal and always equal to the element size M.elemSize() .

So, the data layout in **Mat** is compatible with the majority of dense array types from the standard toolkits and SDKs, such as Numpy (ndarray), Win32 (independent device bitmaps), and others, that is, with any array that uses *steps* (or *strides*) to compute the position of a pixel. Due to this compatibility, it is possible to make a **Mat** header for user-allocated data and process it in-place using OpenCV functions.

There are many different ways to create a **Mat** object. The most popular options are listed below:

Imshow(„window_name”,mat_name) →
https://docs.opencv.org/4.5.5/d7/dfc/group__highgui.html#ga453d42fe4cb60e5723281a89973ee563



imshow()

void cv::imshow (const **String** & winname,
InputArray mat
)

Python:
cv.imshow(winname, mat) -> None

`#include <opencv2/highgui.hpp>`

Displays an image in the specified window.

The function imshow displays an image in the specified window. If the window was created with the **cv::WINDOW_AUTOSIZE** flag, the image is shown with its original size, however it is still limited by the screen resolution. Otherwise, the image is scaled to fit the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit or 64-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].
- 32-bit integer images are not processed anymore due to ambiguouty of required transform. Convert to 8-bit unsigned matrix using a custom preprocessing specific to image's context.

If window was created with OpenGL support, **cv::imshow** also support **ogl::Buffer** , **ogl::Texture2D** and **cuda::GpuMat** as input.

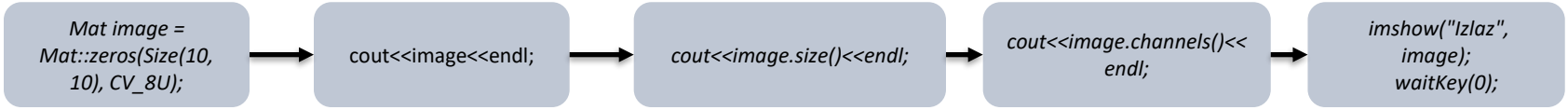
If the window was not created before this function, it is assumed creating a window with **cv::WINDOW_AUTOSIZE**.

If you need to show an image that is bigger than the screen resolution, you will need to call namedWindow("", WINDOW_NORMAL) before the imshow.

Use OpenCV documentation → https://docs.opencv.org/4.5.5/d1/dfb/intro.html

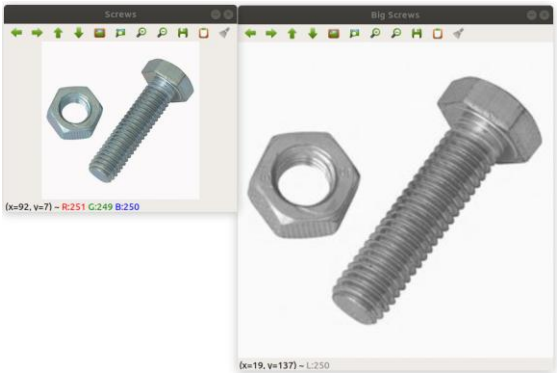
Practical task 1

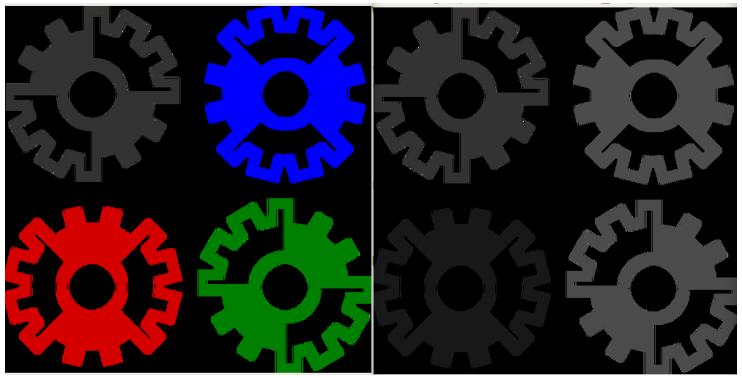
1. *Create black image of size (10,10). Write out the content of Mat, size and number of channels. Show the image in the output window.*



2. *Read images from files (gear.png and screws.jpg) and show them in the window. Write out empty(), size and number of channels for the image read from the file.*

3. *Copy (clone) mat of gear.png, convert color to grayscale, resize (twice the size) and show image.*





Practical task 2

Basic processing on grayscale images:

- *Brighten, darken and contrast grayscale image.*

Version 1

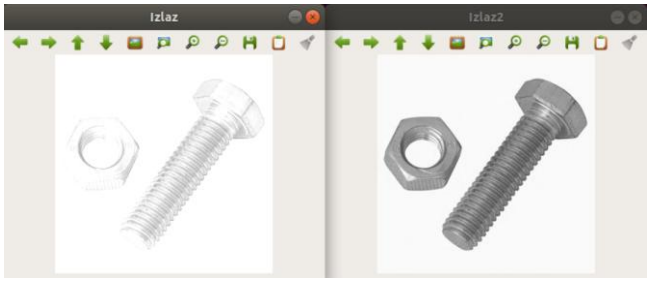
Load image screws.jpg from file → *Check number of channels (mat file). If color (3) convert to grayscale.* → *Add, subtract and multiply scalar value from image Mat.* → *Imshow() original and resulting images* → *Use Imwrite() to save the resulting image*

Comment – error if values out of scope of matrix!!!

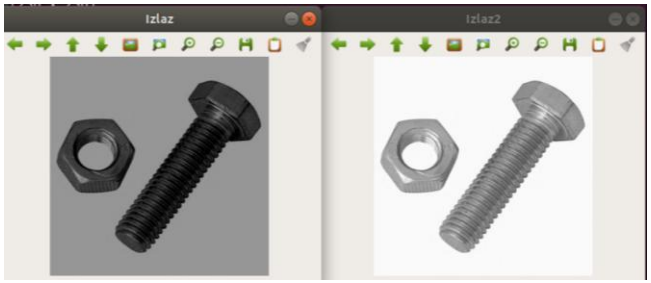
Version 2

Load image screws.jpg from file → *Check number of channels (mat file). If color (3) convert to grayscale.* → *Use add(), subtract() and image.convertTo()* → *Imshow() original and resulting images* → *Use Imwrite() to save the resulting image*

add(image,Mat::ones(image.size(), CV_8U)*100,image);



subtract(image,Mat::ones(image.size(), CV_8U)*100,image);



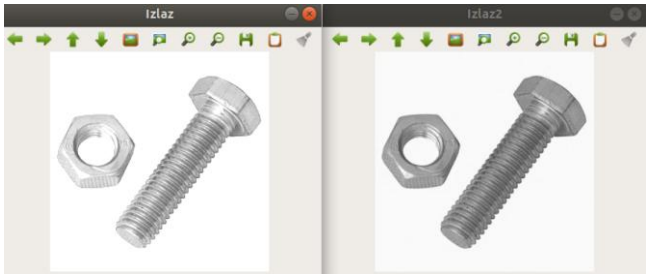


image.convertTo(image,-1,0.5,0);

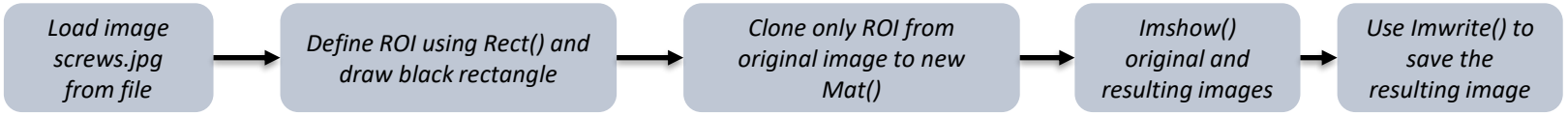
$$g(i,j) = \alpha \cdot f(i,j) + \beta$$

File name — Variable of type Mat()

imwrite("processed_image.jpg",image);

Practical task 3

Extract rectangular ROI from the image:




```
// Setup a rectangle to define your region of interest
Rect myROI(50, 50, 100, 100);
```

Upper left row

Upper left column

Width

Height

```
Mat image_ROI=image(myROI).clone();
```

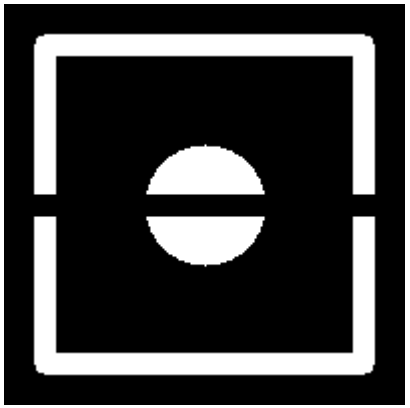
Crop and clone the original image with previously defined ROI

Practical task 4

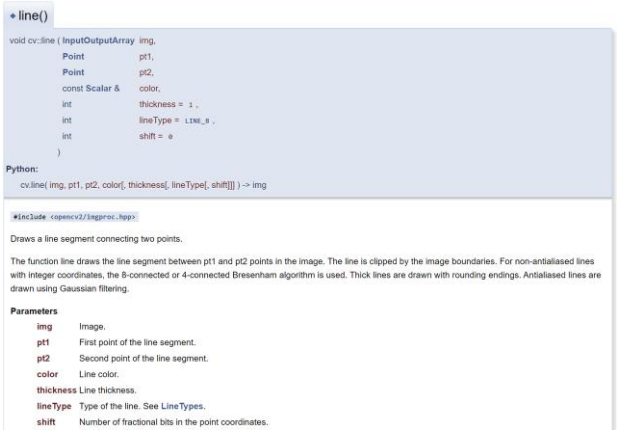
Draw circle, rectangle and line:

Version 1

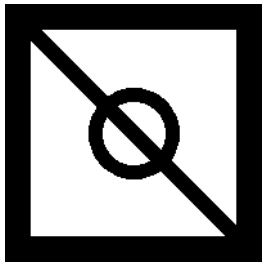
Construct grayscale black image, Size(200,200) → Draw white circle(full), white rectangle rectangle (not full) and black line → Show image



Mat image = Mat::zeros(Size(200, 200), CV_8U);
circle(image, Point(100,100),30,Scalar(255),-1,LINE_8,0);
rectangle(image, Rect(20,20,160,160),Scalar(255),10,LINE_8,0);
line(image,Point(0,100),Point(200,100),Scalar(0),10,LINE_8,0);

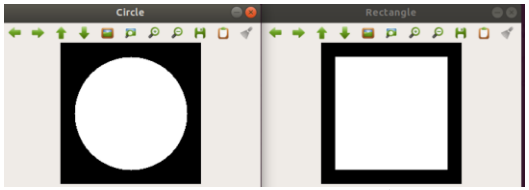


Student exercise. Do it yourself →



Practical task 5

Test bitwise operands on binary circle and rectangle:




```
void circle(InputOutputArray img, Point center, int radius, const Scalar &color, int thickness = 1, int lineType = LINE_8, int shift = 0)
circle(circle1,cv::Point(100,100),80,Scalar(255),-1,8,0)
1 of 2 void rectangle(InputOutputArray img, Rect rec, const Scalar &color, int thickness = 1, int lineType = LINE_8, int shift = 0)
rectangle(rectangle1,Rect(20,20,160,160),Scalar(255),-1,8,0);
```

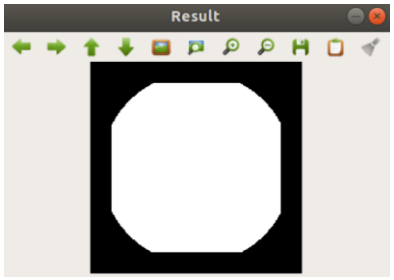
Construct 2 grayscale black images, Size(200,200) → *Draw white circle and rectangle in separate Mats* → *Test bitwise_and(), bitwise_or(), bitwise_xor(),...* → *Show images* → *Concatenate and save resulting images*

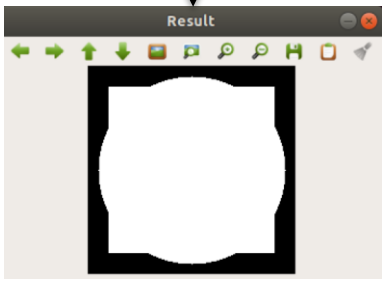
```
void bitwise_or(InputArray src1, InputArray src2, OutputArray dst, InputArray mask = noArray())
bitwise_or(circle1,rectangle1,result);
```

Bitwise OR

Bitwise AND

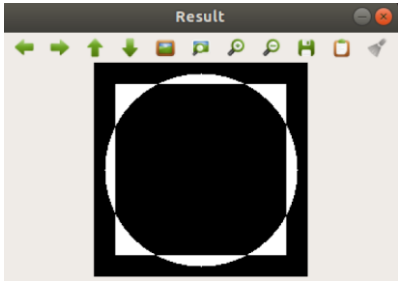
```
void bitwise_and(InputArray src1, InputArray src2, OutputArray dst, InputArray mask = noArray())
bitwise_and(circle1,rectangle1,result);
```





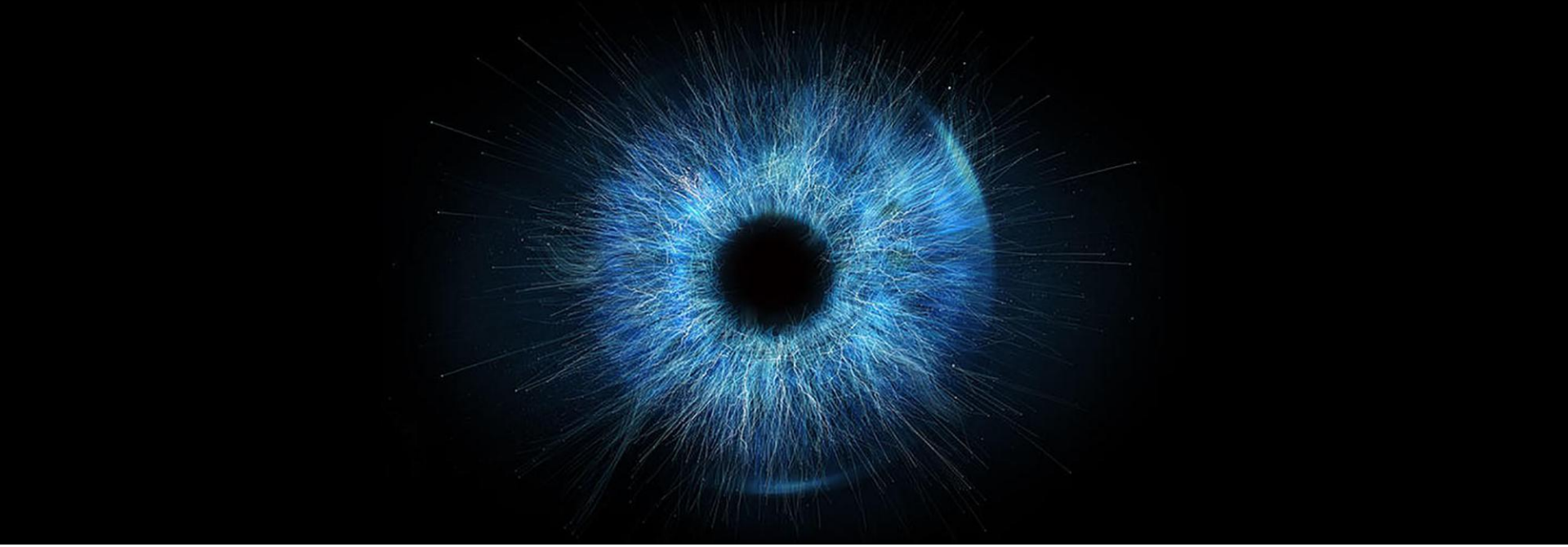
Bitwise XOR

```
void bitwise_xor(InputArray src1, InputArray src2, OutputArray dst, InputArray mask = noArray())
bitwise_xor(circle1,rectangle1,result);
```




```
1 of 3 void hconcat(InputArrayOfArrays src, OutputArray dst)
hconcat(image,image,connected);
```

→ *Horizontal concatenating of image matrices*



Exercise – Filters

Machine vision algorithms

Doc.dr.sc. Filip Šuligoj
fsuligoj@fsb.hr

University of Zagreb
FSB Faculty of mechanical engineering and naval arhitecture

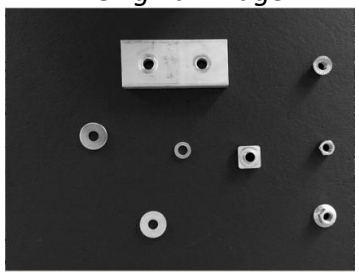
Practical task 1

Apply image filters to the original images

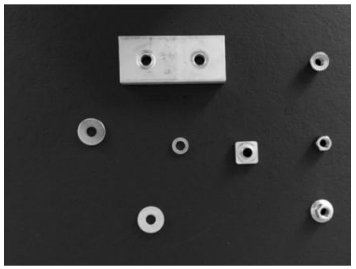
Load image parts.png from file → *Check number of channels (mat file). If color (3) convert to grayscale.* → *Apply filters with different kernel sizes (3,7,15) to original image (mean, gauss, median).* → *Imshow() original and resulting images.*

Results:

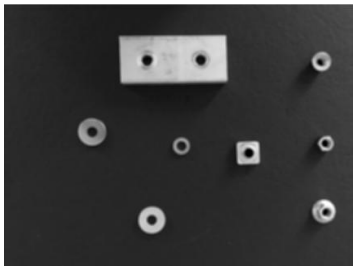
Original image



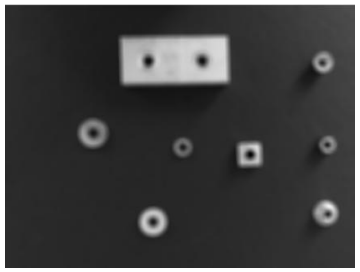
Mean 3x3



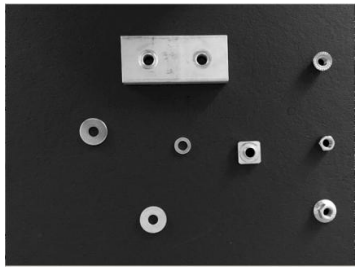
Mean 7x7



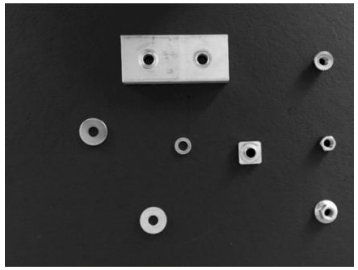
Mean 15x15



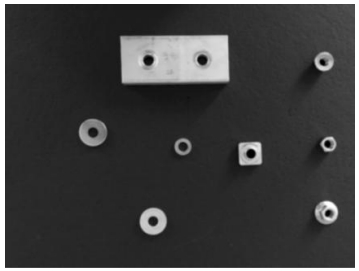
Original image



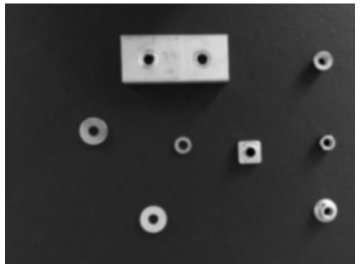
Gauss 3x3



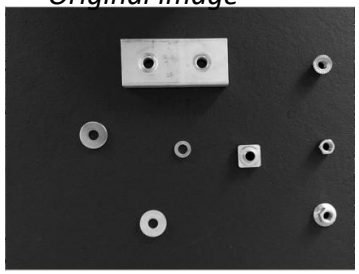
Gauss 7x7



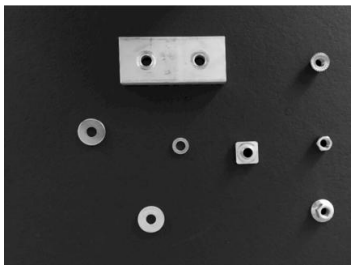
Gauss 15x15



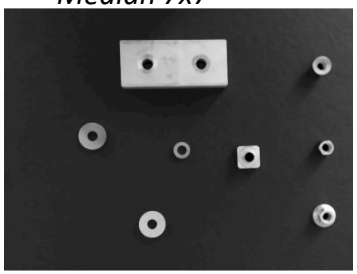
Original image



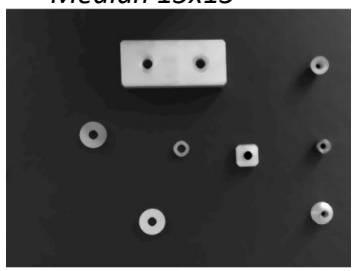
Median 3x3



Median 7x7



Median 15x15



Practical task 1

https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga8c45db9afe636703801b0b2e440fce37

◆ blur()

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

Python:

```
dst = cv.blur( src, ksize[, dst[, anchor[, borderType]]] )
```

```
#include <opencv2/imgproc.hpp>
```

Blurs an image using the normalized box filter.

The function smooths an image using the kernel:

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

The call `blur(src, dst, ksize, anchor, borderType)` is equivalent to `boxFilter(src, dst, src.type(), ksize, anchor, true, borderType)`.

Parameters

src	input image; it can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
ksize	blurring kernel size.
anchor	anchor point; default value Point(-1,-1) means that the anchor is at the kernel center.
borderType	border mode used to extrapolate pixels outside of the image, see **BorderTypes**. **BORDER_WRAP** is not supported.

See also

boxFilter, **bilateralFilter**, **GaussianBlur**, **medianBlur**

Examples:

samples/cpp/edge.cpp, **samples/cpp/laplace.cpp**, and **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**.

Practical task 1

https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#gaabe8c836e97159a9193fb0b11ac52cf1

◆ GaussianBlur()

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

Python:

```
dst = cv.GaussianBlur( src, ksize, sigmaX[, dst[, sigmaY[, borderType]]] )
```

```
#include <opencv2/imgproc.hpp>
```

Blurs an image using a Gaussian filter.

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

Parameters

src	input image; the image can have any number of channels, which are processed independently, but the depth should be CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	output image of the same size and type as src.
ksize	Gaussian kernel size. ksize.width and ksize.height can differ but they both must be positive and odd. Or, they can be zero's and then they are computed from sigma.
sigmaX	Gaussian kernel standard deviation in X direction.
sigmaY	Gaussian kernel standard deviation in Y direction; if sigmaY is zero, it is set to be equal to sigmaX, if both sigmas are zeros, they are computed from ksize.width and ksize.height, respectively (see **getGaussianKernel** for details); to fully control the result regardless of possible future modifications of all this semantics, it is recommended to specify all of ksize, sigmaX, and sigmaY.
borderType	pixel extrapolation method, see **BorderTypes**. **BORDER_WRAP** is not supported.

See also

sepFilter2D, **filter2D**, **blur**, **boxFilter**, **bilateralFilter**, **medianBlur**

Examples:

samples/cpp/laplace.cpp, **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**, and **samples/cpp/tutorial_code/ImgTrans/Sobel_Demo.cpp**.

Practical task 1

https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga564869aa33e58769b4469101aac458f9

medianBlur()

```
void cv::medianBlur ( InputArray  src,
                      OutputArray dst,
                      int         ksize
                    )
```

Python:

```
dst = cv.medianBlur( src, ksize[, dst] )
```

```
#include <opencv2/imgproc.hpp>
```

Blurs an image using the median filter.

The function smoothes an image using the median filter with the **ksize** × **ksize** aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

Note

The median filter uses **BORDER_REPLICATE** internally to cope with border pixels, see **BorderTypes**

Parameters

src	input 1-, 3-, or 4-channel image; when ksize is 3 or 5, the image depth should be CV_8U, CV_16U, or CV_32F, for larger aperture sizes, it can only be CV_8U.
dst	destination array of the same size and type as src.
ksize	aperture linear size; it must be odd and greater than 1, for example: 3, 5, 7 ...

See also

bilateralFilter, **blur**, **boxFilter**, **GaussianBlur**

Examples:

samples/cpp/laplace.cpp, **samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp**, and **samples/cpp/tutorial_code/ImgTrans/houghcircles.cpp**.

Practical task 2

Add noise (gaussian and salt&pepper) to image

Load image parts.png from file → Check number of channels (mat file). If color (3) convert to grayscale. → Add gaussian and salt&pepper noise to the image. Show noisy images. → Apply filters with different kernel sizes (3,7,15) to noisy image (mean, gauss, median). → Imshow() original and resulting images.

Adding gaussian noise to the image:

```
Mat gaussian_noise = Mat (image.size(),image.type());
randn(gaussian_noise,50,10);
add(image,gaussian_noise,image);
```

void cv::randn *(InputOutputArray dst, InputArray mean, InputArray stddev)*
Fills the array with normally distributed random numbers. The function cv::randn fills the matrix dst with normally distributed random numbers with the specified mean vector and the standard deviation matrix. The generated random numbers are clipped to fit the value range of the output array data type.
void cv::randu *(InputOutputArray dst, InputArray low, InputArray high)*
Generates a single uniformly-distributed random number or an array of random numbers. Non-template variant of the function fills the matrix dst with uniformly-distributed random numbers from the specified range.

Adding salt&pepper noise to the image:

```
Mat saltpepper_noise = Mat::zeros(image.rows, image.cols,CV_8U);
randu(saltpepper_noise,0,255);

Mat black = saltpepper_noise < 30;
Mat white = saltpepper_noise > 225;

image.setTo(255,white);
image.setTo(0,black);
```

`yy.setTo(0)` will set all the pixels to 0.

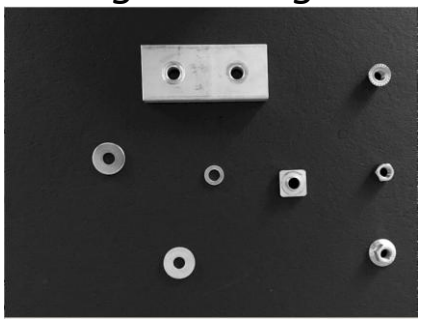
`yy.setTo(0, xx)` will set all the pixels who have a corresponding pixel with a non-zero value in the xx `Mat` to 0.

Example:

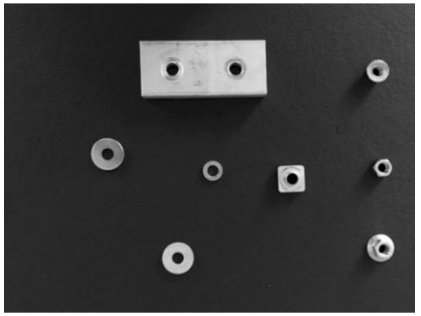
```
yy =
2 2 2
2 2 2
2 2 2

xx =
0 0 0
0 1 0
0 0 0

yy.setTo(0, xx) =>

yy =
2 2 2
2 0 2
2 2 2
```

Practical task 2 - results

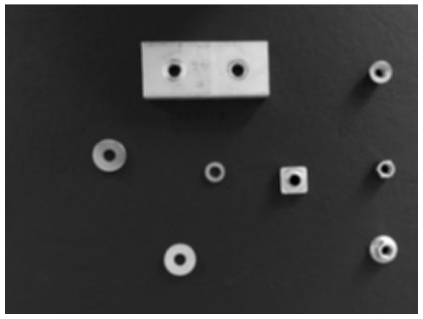
Original image



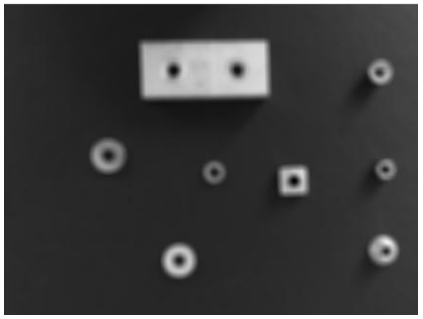
Mean 3x3



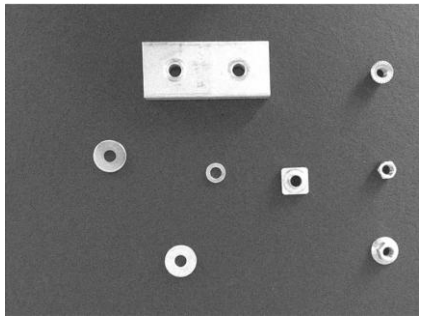
Mean 7x7



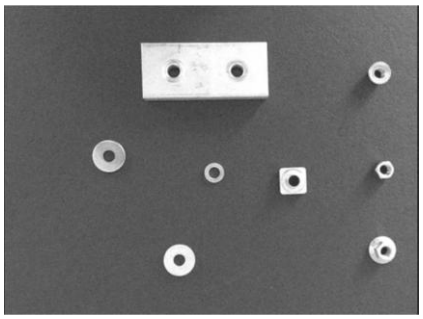
Mean 15x15



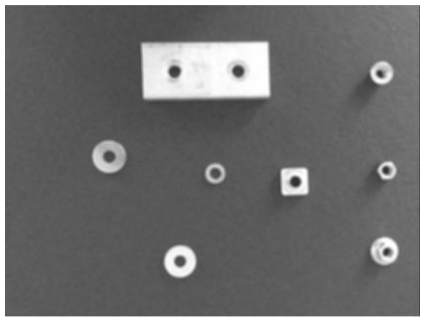
Gaussian noise



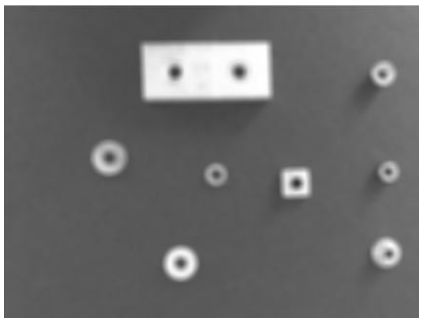
Mean 3x3



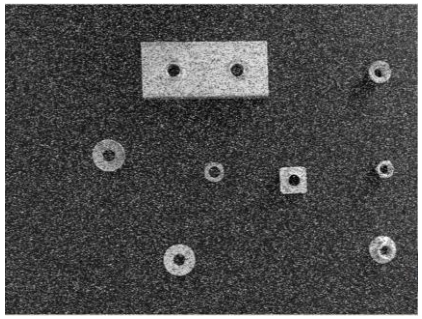
Mean 7x7



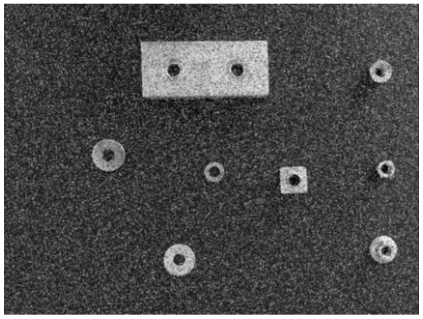
Mean 15x15



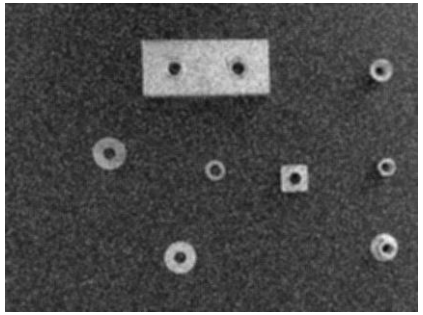
Salt and pepper noise



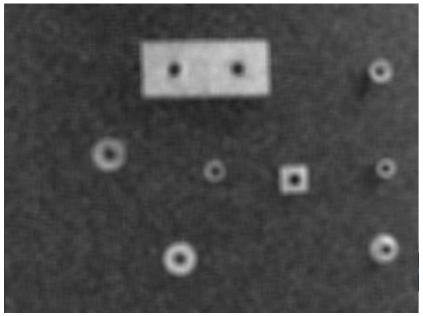
Mean 3x3



Mean 7x7

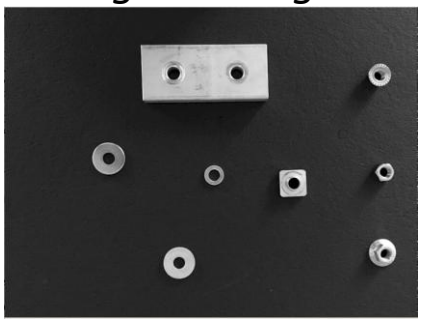


Mean 15x15

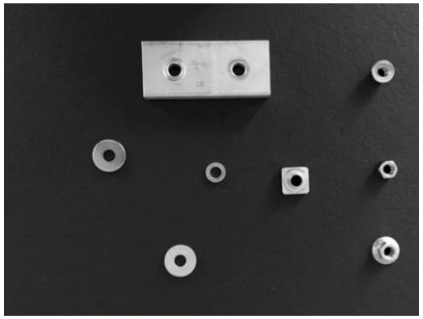


Practical task 2 - results

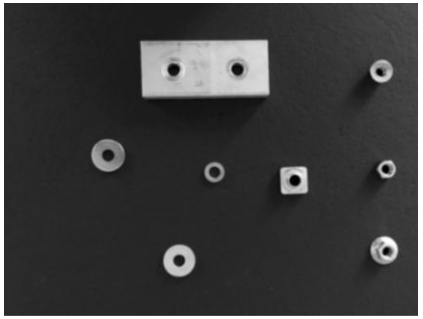
Original image



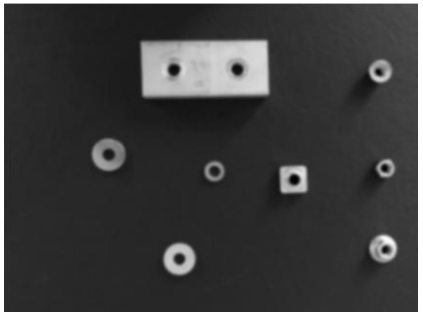
Gauss 3x3



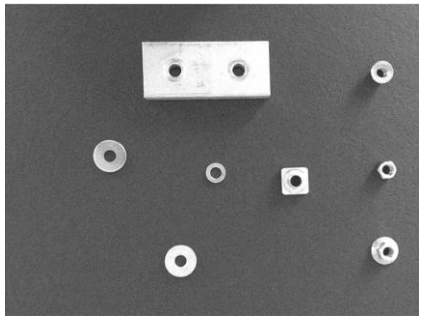
Gauss 7x7



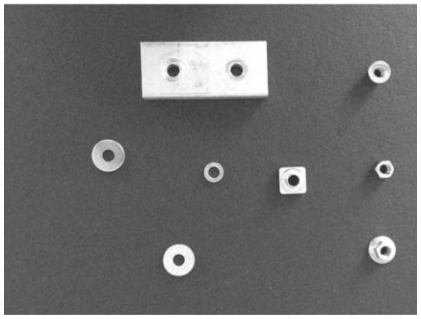
Gauss 15x15



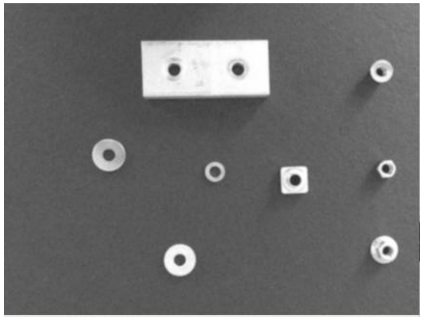
Gaussian noise



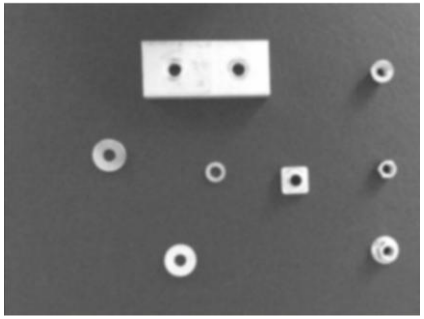
Gauss 3x3



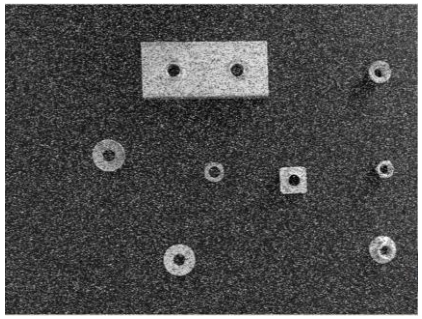
Gauss 7x7



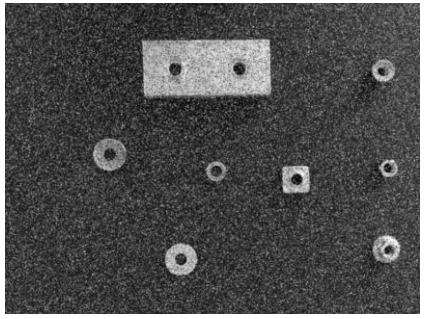
Gauss 15x15



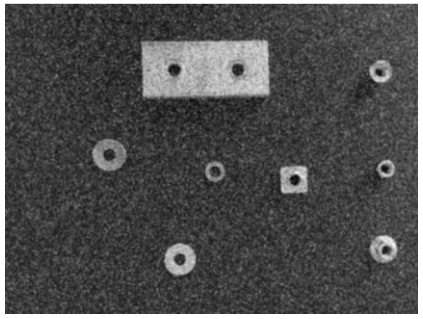
Salt and pepper noise



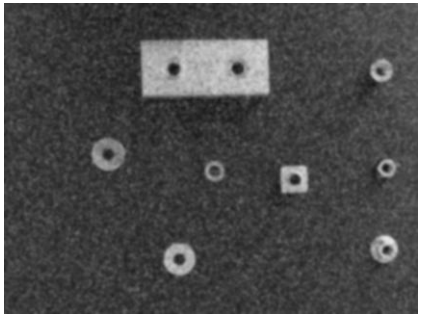
Gauss 3x3



Gauss 7x7

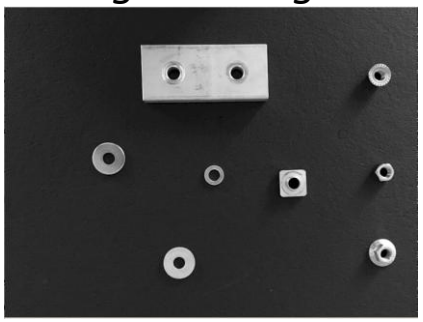


Gauss 15x15

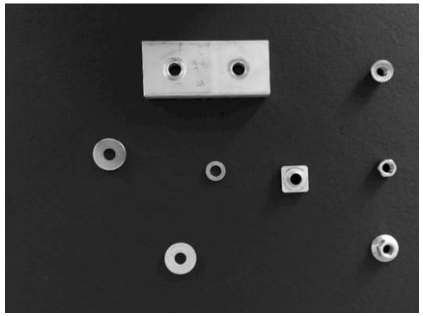


Practical task 2 - results

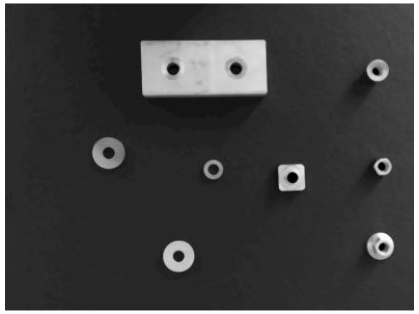
Original image



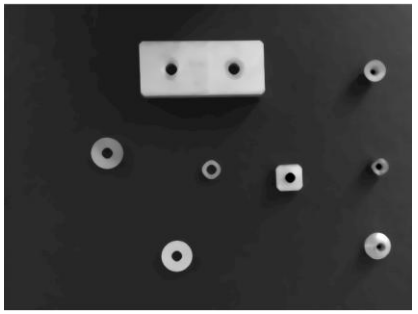
Median 3x3



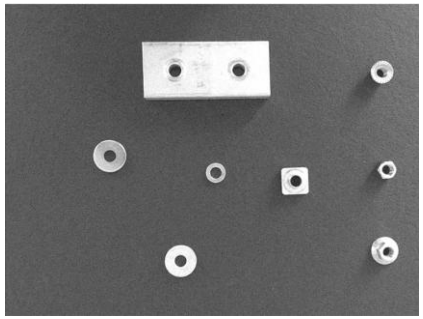
Median 7x7



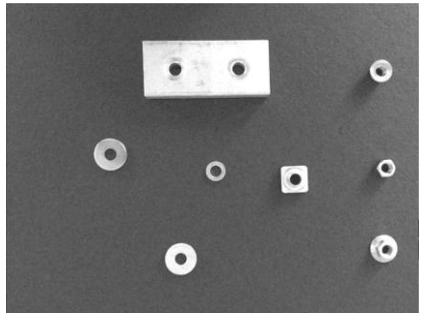
Median 15x15



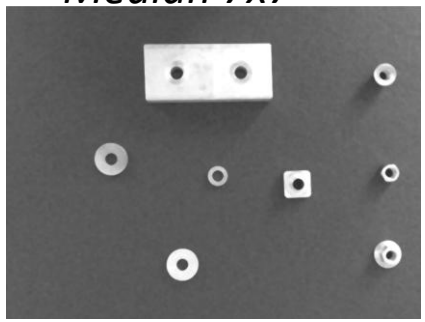
Gaussian noise



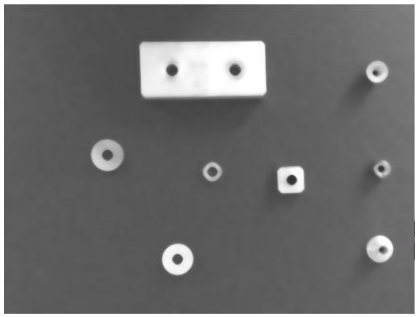
Median 3x3



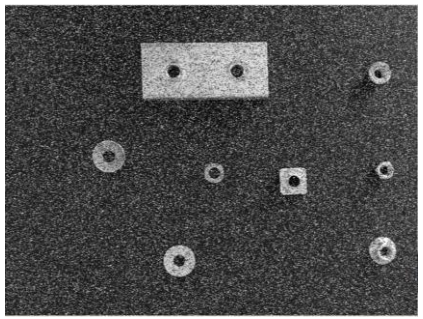
Median 7x7



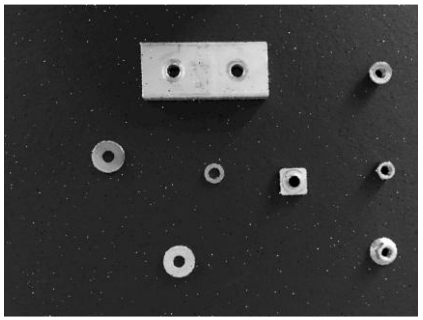
Median 15x15



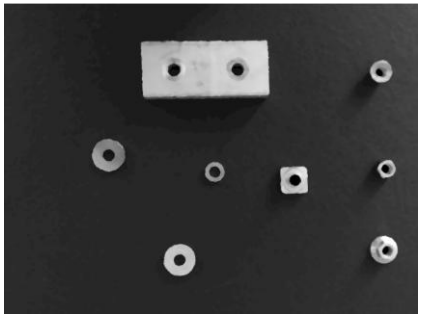
Salt and pepper noise



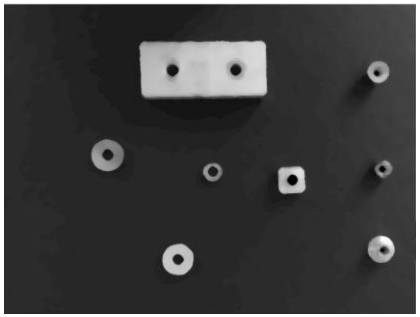
Median 3x3

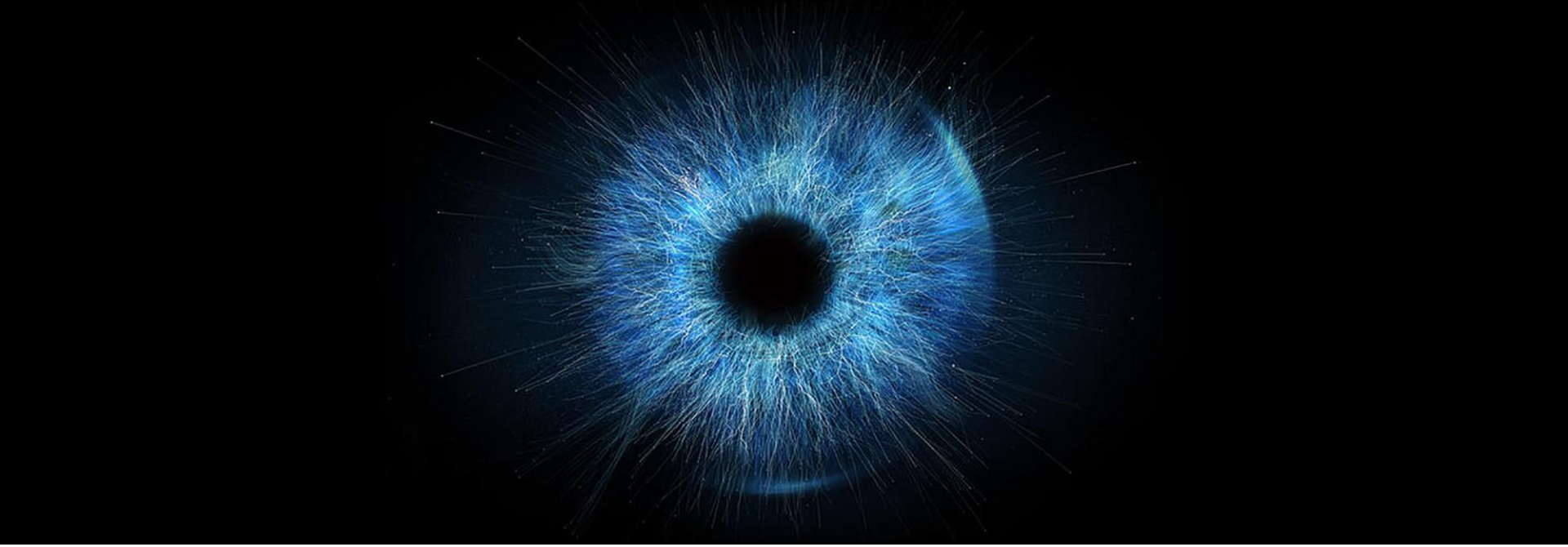


Median 7x7



Median 15x15





Exercise – Image morphology

Fundamentals of machine vision algorithms

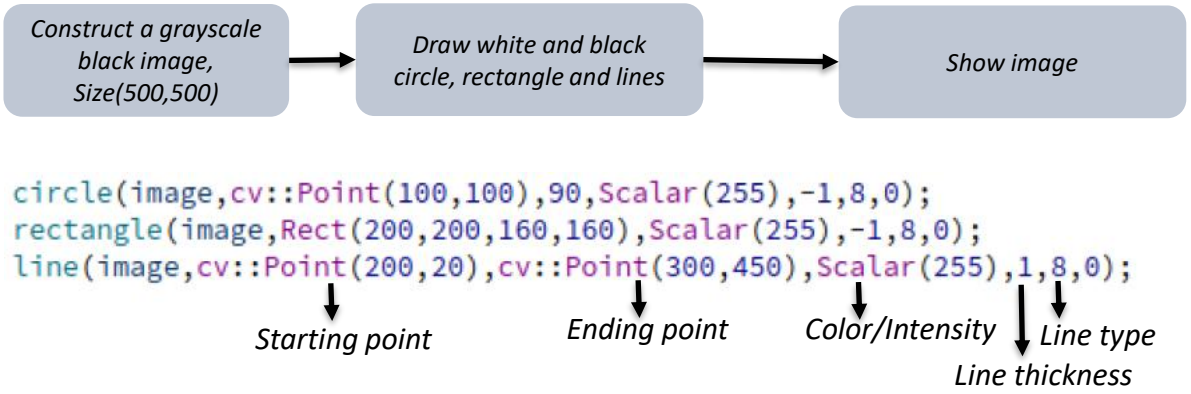
dr.sc. Filip Šuligoj
fsuligoj@fsb.hr

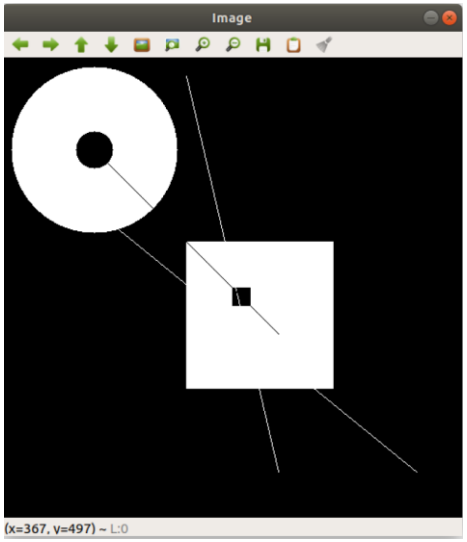
University of Zagreb
FSB Faculty of mechanical engineering and naval arhitecture

Practical task 1

Test morphological operators → erosion, dilatation, opening, closing, gradient, tophat, blackhat on the image with intersecting shapes.

Step 1 (create grayscale image with intersecting shapes)





Practical task 1

Test morphological operators → erosion, dilatation, opening, closing, gradient, tophat, blackhat on the image with intersecting shapes:

Use image with circles, rectangles and lines → *Define structuring element, e.g., rectangle –> MORPH_RECT* → *Apply morphological operations (0-6) and save results to different Mat()* → *Show results*

```
int element_type=MORPH_RECT;
int element_size=1;
Mat element = getStructuringElement( element_type,
                       Size( 2*element_size + 1, 2*element_size+1 ),
                       Point( element_size, element_size ) );

//erode( image, result, element);
//dilate(image,result2,element);
morphologyEx(image,result,0,element);
morphologyEx(image,result2,1,element);
morphologyEx(image,result3,2,element);
morphologyEx(image,result4,3,element);
morphologyEx(image,result5,4,element);
morphologyEx(image,result6,5,element);
morphologyEx(image,result7,6,element);
```

Structuring element type

Structuring element size

morphologyEx types →

0	MORPH_ERODE	see erode
1	MORPH_DILATE	see dilate
2	MORPH_OPEN	an opening operation
		dst=open(src,element)=dilate(erode(src,element))
3	MORPH_CLOSE	a closing operation
		dst=close(src,element)=erode(dilate(src,element))
4	MORPH_GRADIENT	a morphological gradient
		dst=morph_grad(src,element)=dilate(src,element)−erode(src,element)
5	MORPH_TOPHAT	"top hat"
		dst=tophat(src,element)=src−open(src,element)
6	MORPH_BLACKHAT	"black hat"
		dst=blackhat(src,element)=close(src,element)−src
7	MORPH_HITMISS	"hit or miss" .- Only supported for CV_8UC1 binary images. A tutorial can be found in the documentation

morphologyEx - documentation link

morphologyEx()

```
void cv::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()
                      )
```

Python:

```
cv.morphologyEx( src, op, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]] ) -> dst
```

The function **cv::morphologyEx** can perform advanced morphological transformations using an erosion and dilation as basic operations.

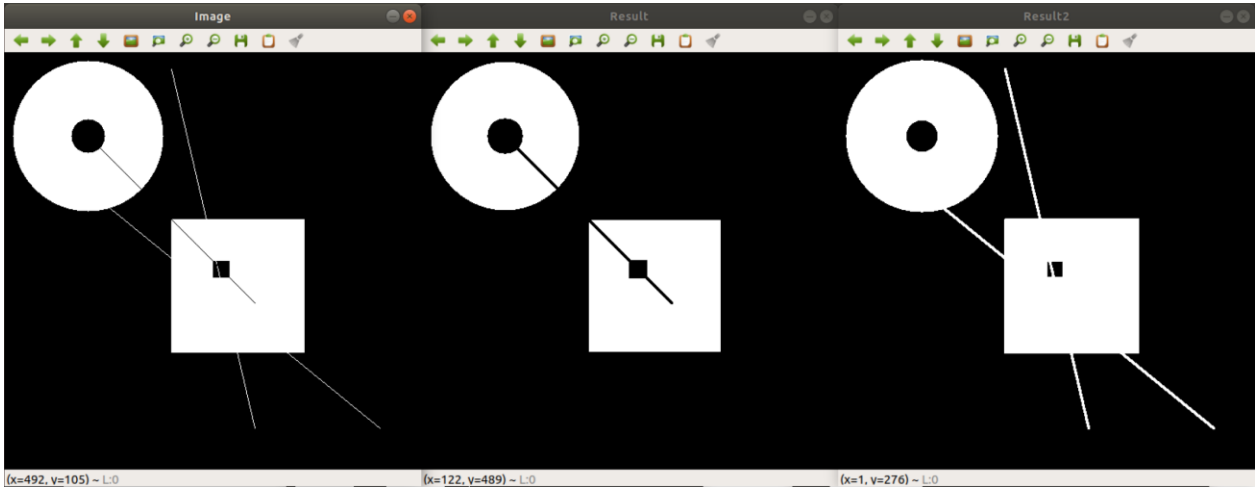
Any of the operations can be done in-place. In case of multi-channel images, each channel is processed independently.

Parameters

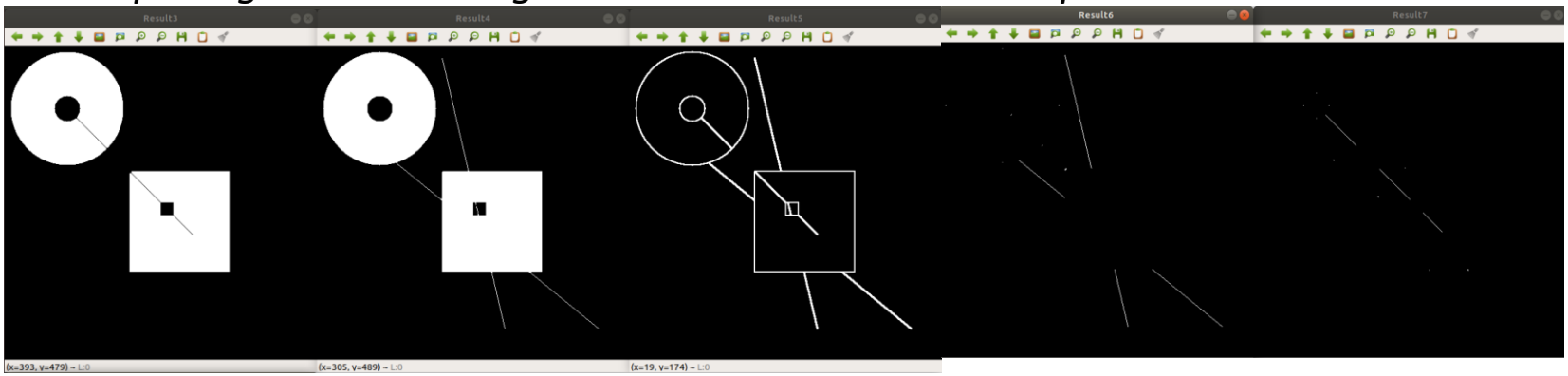
src	Source image. The number of channels can be arbitrary. The depth should be one of CV_8U, CV_16U, CV_16S, CV_32F or CV_64F.
dst	Destination image of the same size and type as source image.
op	Type of a morphological operation, see **MorphTypes**
kernel	Structuring element. It can be created using **getStructuringElement**.
anchor	Anchor position with the kernel. Negative values mean that the anchor is at the kernel center.
iterations	Number of times erosion and dilation are applied.
borderType	Pixel extrapolation method, see **BorderTypes**. **BORDER_WRAP** is not supported.
borderValue	Border value in case of a constant border. The default value has a special meaning.

Practical task 1 - results

Original *erosion* *dilatation*

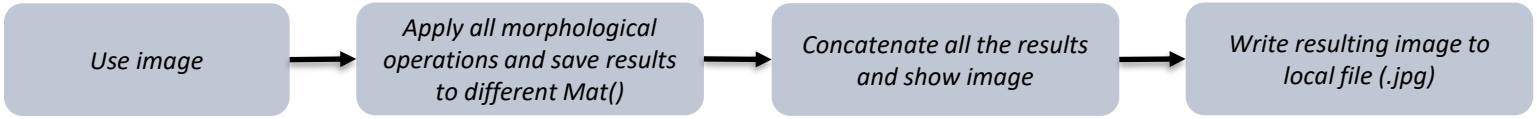


opening *closing* *gradient* *tophat* *blackhat*



Practical task 2

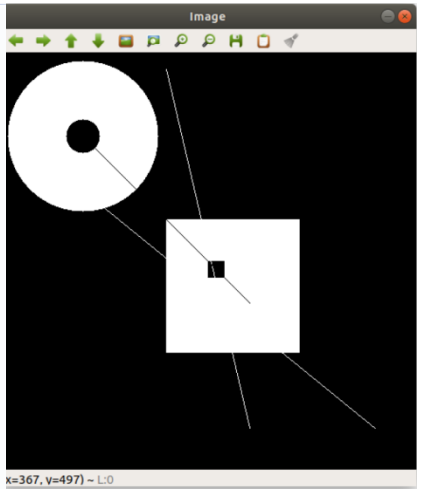
Test erosion, dilatation, opening, closing, gradient, tophat, blackhat with different structuring elements:



Test parameters:

- *Different structuring elements sizes (3x3,5x5,7x7)*
- *Different structuring element types (rect, ellipse, cross)*
- *Different line thicknesses (1,2,3,4 pixel wide) in the original image.*

```
# 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)
```



Practical task 3

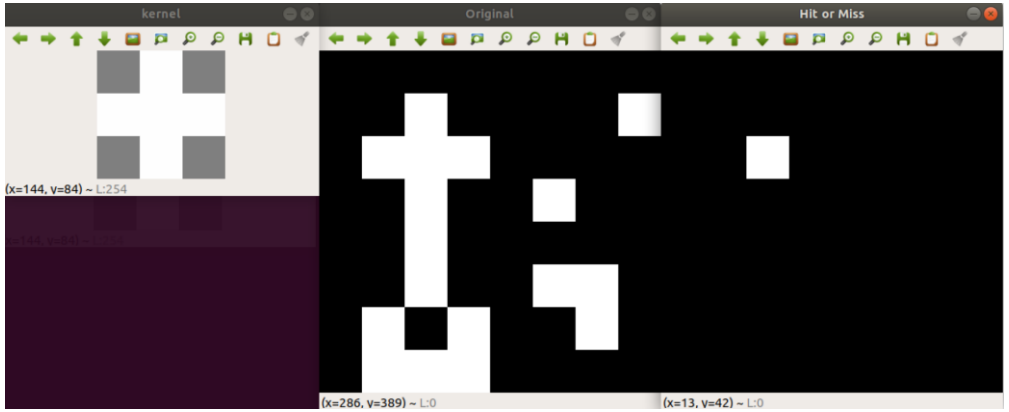
Use morphological hit-or-miss to find a shape of a cross:

Create image matrix and search kernel → *Apply morphological hit-or-miss* → *resize() the images 8x and show results* → *Write resulting image to local file (.jpg)*

```
morphologyEx(input_image, output_image, MORPH_HITMISS, kernel);
```

```
Mat input_image = (Mat_<uchar>(8, 8) <<
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 255, 0, 0, 0, 0, 255,
    0, 255, 255, 255, 0, 0, 0, 0,
    0, 0, 255, 0, 0, 255, 0, 0,
    0, 0, 255, 0, 0, 0, 0, 0,
    0, 0, 255, 0, 0, 255, 255, 0,
    0, 255, 0, 255, 0, 0, 255, 0,
    0, 255, 255, 255, 0, 0, 0, 0);
  Mat kernel = (Mat_<int>(3, 3) <<
    0, 1, 0,
    1, 1, 1,
    0, 1, 0);
```

Result



Student assignment (seminar)

Student assignment

Add trackbar and change the filter kernel size manually in code from Practical task 2. Add and test out the bilateral filter as well.

For trackbar use example :

https://docs.opencv.org/master/da/d6a/tutorial_trackbar.html

◆ bilateralFilter()

```
void cv::bilateralFilter ( InputArray  src,
                           OutputArray dst,
                           int         d,
                           double      sigmaColor,
                           double      sigmaSpace,
                           int         borderType = BORDER_DEFAULT
                         )
```

Python:

```
dst = cv.bilateralFilter( src, d, sigmaColor, sigmaSpace[, dst[, borderType]] )
```

```
#include <opencv2/imgproc.hpp>
```

Applies the bilateral filter to an image.

The function applies bilateral filtering to the input image, as described in http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html bilateralFilter can reduce unwanted noise very well while keeping edges fairly sharp. However, it is very slow compared to most filters.

Sigma values: For simplicity, you can set the 2 sigma values to be the same. If they are small (< 10), the filter will not have much effect, whereas if they are large (> 150), they will have a very strong effect, making the image look "cartoonish".

Filter size: Large filters (d > 5) are very slow, so it is recommended to use d=5 for real-time applications, and perhaps d=9 for offline applications that need heavy noise filtering.

This filter does not work inplace.

Parameters

src	Source 8-bit or floating-point, 1-channel or 3-channel image.
dst	Destination image of the same size and type as src .
d	Diameter of each pixel neighborhood that is used during filtering. If it is non-positive, it is computed from sigmaSpace.
sigmaColor	Filter sigma in the color space. A larger value of the parameter means that farther colors within the pixel neighborhood (see sigmaSpace) will be mixed together, resulting in larger areas of semi-equal color.
sigmaSpace	Filter sigma in the coordinate space. A larger value of the parameter means that farther pixels will influence each other as long as their colors are close enough (see sigmaColor). When d>0, it specifies the neighborhood size regardless of sigmaSpace. Otherwise, d is proportional to sigmaSpace.
borderType	border mode used to extrapolate pixels outside of the image, see BorderTypes

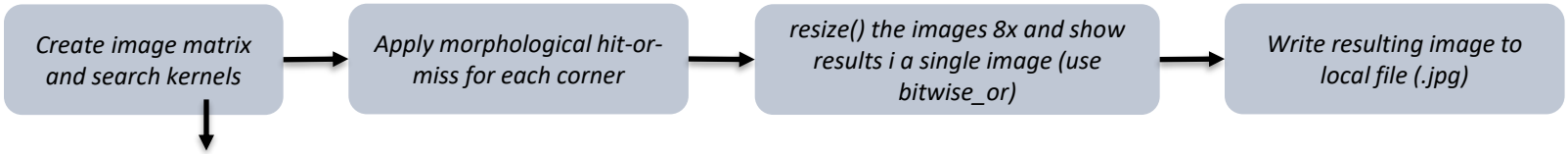
Examples:

samples/cpp/tutorial_code/ImgProc/Smoothing/Smoothing.cpp.

https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga9d7064d478c95d60003cf839430737ed

Student assignment

Use morphological hit-or-miss to find 4 corners of a square or a rectangle:



Hint: search kernels should also have negative fields!

Example: