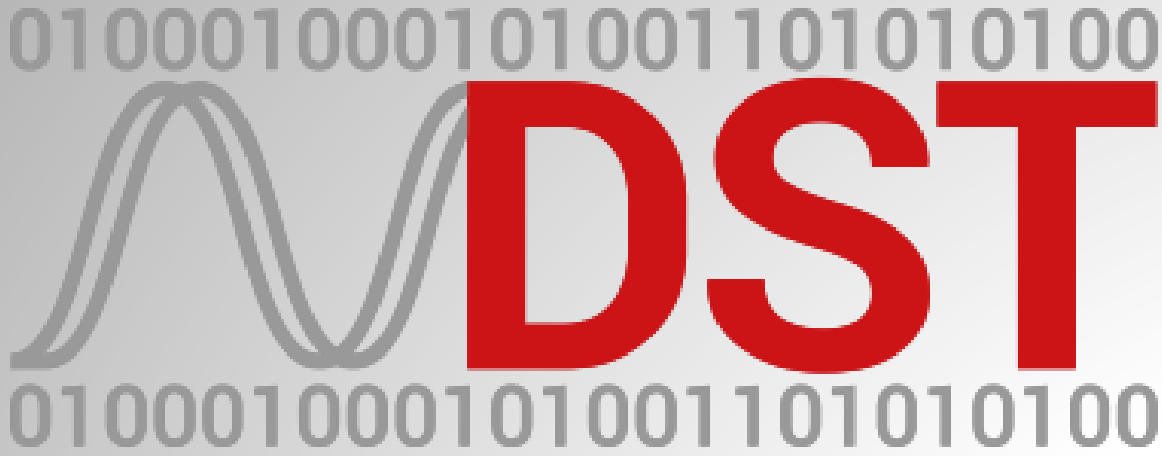
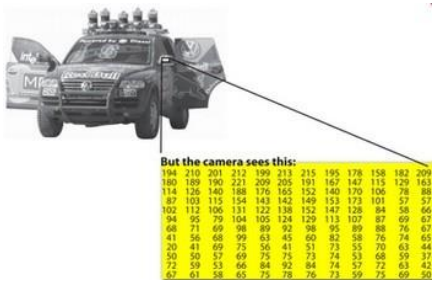


Chapter 2: Thresholding

Excursion: cv::Mat



cv::Mat class methods:

- bool Mat::**isContinuous**()
  - The method returns true, if the matrix elements are stored continuously without gaps at the end of each row. Otherwise, it returns false.
  - If you extract a part of the matrix (e.g. subpart of an image), the matrix is not continuous.

- template<typename _Tp> _Tp* Mat::**ptr**(int **i0**=0)
  - Access image data
  - **i0**: A 0-based row index.

- A full reference of cv::Mat is available here:
  https://docs.opencv.org/4.0.1/d3/d63/classcv_1_1Mat.html

- Real world images are transformed into digital images e.g via digital cameras, scanners
- Every real world information (intensity) is stored in a list (called array)
- This “container” is called Mat in OpenCV

Line by line representation:

0	134	135	255	255	136	128	142	60	42
60	60	67	128	254	136	128	70	30	0
10	34	16	15	0	13	35	78	56	22
5	68	203	181	164	142	100	42	30	8

- Every single row is saved as individual object in memory

Continuous representation:

0	134	135	255	255	136	128	142	60	42	60	60	67	128	254	136	128	70	30	0	. . .

- The whole picture is saved as a single object in memory

- Why is it important to check for continuity?
  - Extracted image is not continuous in cv::Mat

- Processing the extracted image needs pointer algorithms to get the first cell of the next row
- To get the next pixel after the cell with the value 136 the pointer has to be increased by 8 in this example

Some knowledge of cv::Mat and OpenCV is assumed, please brief yourself on the following website:

https://docs.opencv.org/4.0.1/d6/d6d/tutorial_mat_the_basic_image_container.html

Programming techniques: Access to the image data from a cv::Mat:

- There are 3 ways to access a pixel:
  1. template<typename T> T& Mat::**at**(int i, int j) const method
  2. Pointer with index
  3. Pointer without index
- The result is always the same, but the speed of the data access is different!

1. template<typename T> T& Mat::at(int i, int j) const

```
cv::Mat img = cv::imread("lena.tiff")

for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
        std::cout << img.at<uchar>(r, c) << std::endl;
    }
}
```

→ easy, but slow

2. Pointer with index:

```
cv::Mat img = cv::imread("lena.tiff")

// check for continuous data in memory
if (img.isContinuous()) {
    cols = rows*cols
    rows = 1;
}

for (int r = 0; r < rows; ++r) {
    // pointer to the data
    const uchar *pInput = img.ptr<uchar>(r);

    for (int c = 0; c < cols; ++c) {
        // access image element
        std::cout << pInput[c] << std::endl;
    }
}
```

→ faster, but not the fastest solution

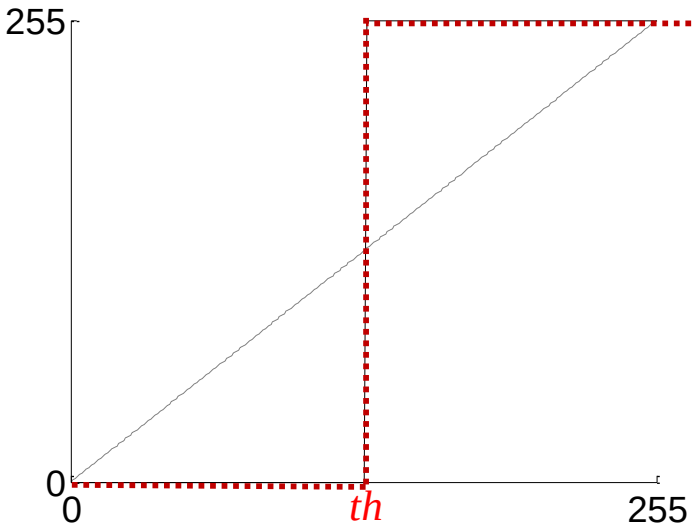
3. Pointer without index:

```
cv::Mat img = cv::imread("lena.tiff")

// check for continuous data in memory
if (img.isContinuous()) {
    cols = rows*cols
    rows = 1;
}

for (int r = 0; r < rows; ++r) {
    // pointer to the data
    const uchar *pInput = img.ptr<uchar>(r);

    for (int c = 0; c < cols; ++c) {
        // access image element
        std::cout << *pInput  << std::endl;

       // increment data address
       ++pInput;
    }
}
```

→ the fastest solution

Repetition Thresholding

$$J(m,n) = \begin{cases} 0, & \text{if } I(m,n) < th \\ 255, & \text{if } I(m,n) \geq th \end{cases}$$



- The result of thresholding an image is a binary image
- It can be used for simple segmentation tasks

$th = 64$

$th = 128$

$th = 192$

Second exercise

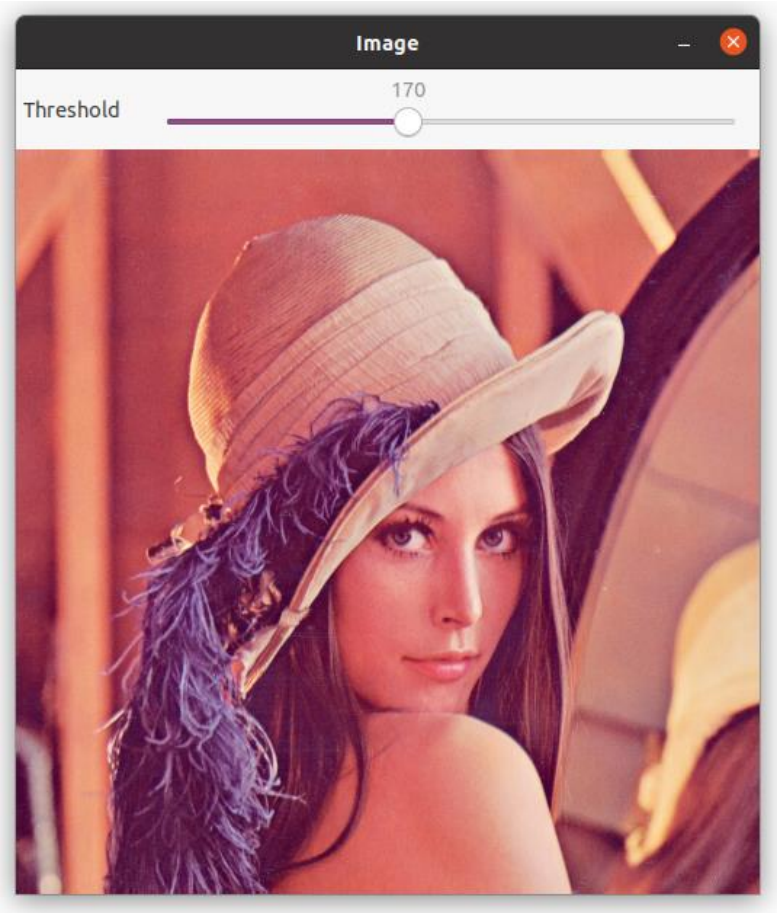
- Implement thresholding with all 3 access methods

The exercise program has 6 windows:

- Main window (colored image) with threshold slider
- Grayscale image
- 4 threshold images
  - 1 with OpenCV (this is already done)
  - 3 windows for the 3 access methods

→ initially black, it is **your task** to implement the algorithms

→ all 4 threshold images should be **identical**



Exercise 2: Main window with slider

Expected Output