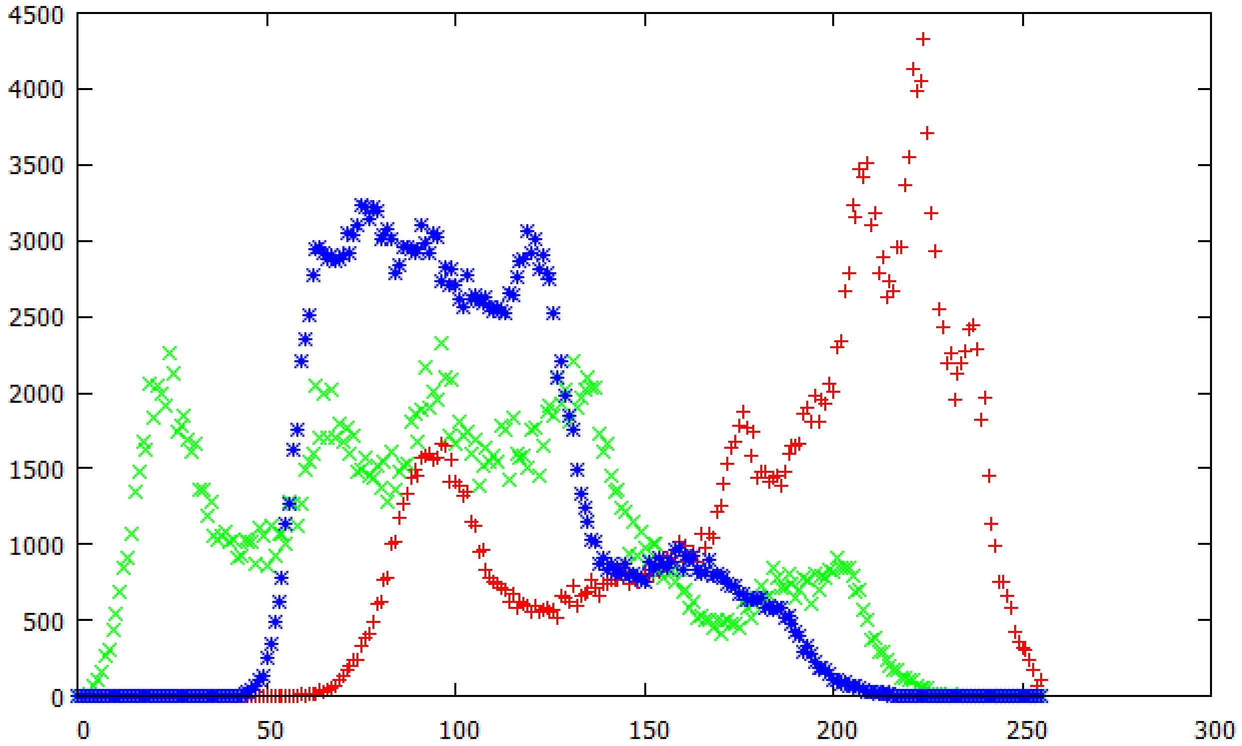
# Histogram and LUT operations

Local operation

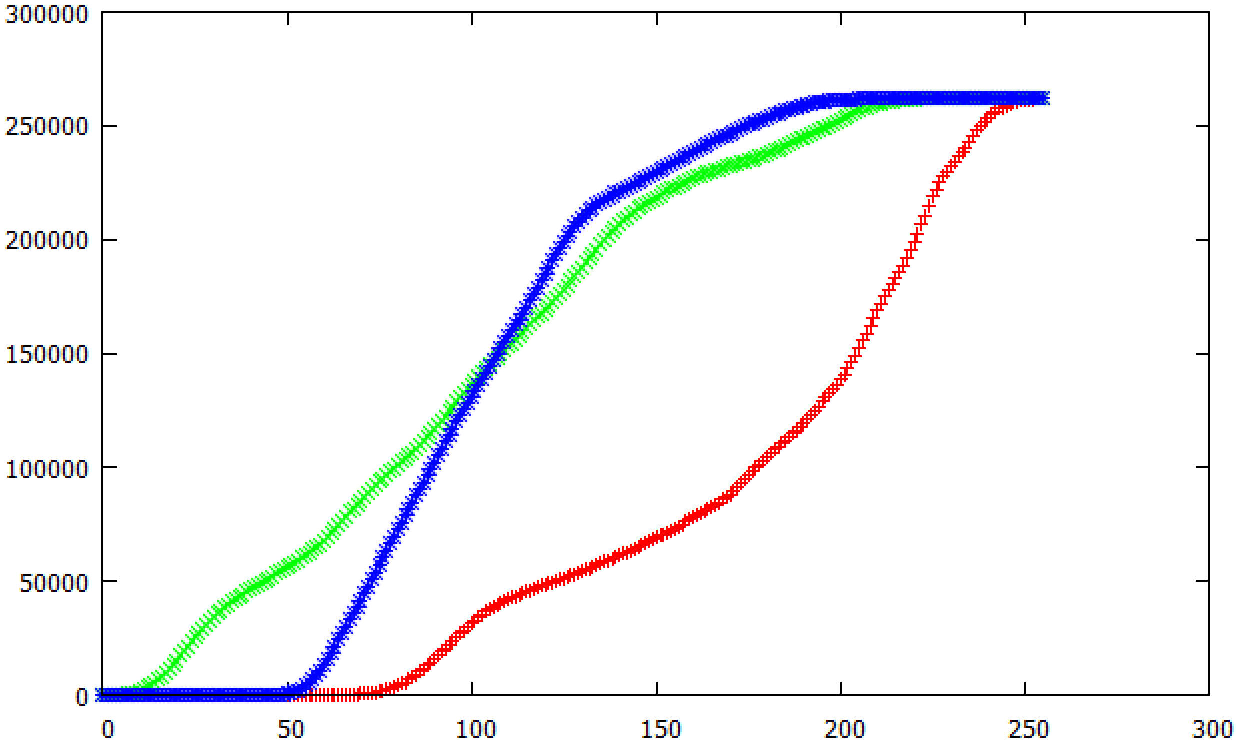
# Histogram




H[v] is the number of pixel of the image with value v (v is usually in the range 0-255) H is an array

We usually evaluate a different histogram for each channel
(or we merge the channels in a single «gray» channel)

# Cumulative function




C[v] is the number of pixel of the image with value less or equal to v

From a programming point of view
C[0] = H[0]
C[v] = C[v-1] + H[v] → C[v] ≥ C[v-1] (for v>0)
With the common range (0-255)
C[255] is the number of pixels of the image (width x height)

# Local operation

The value of the pixels of the new image depends only on the value of the corresponding pixel of the original image

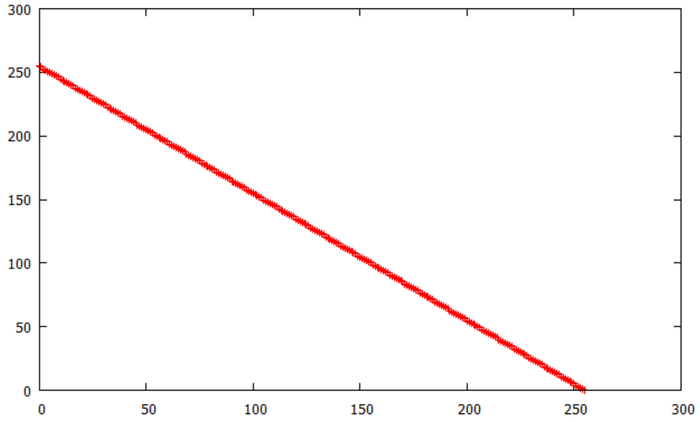
newImage(i,j) = f(originalImage(i,j))

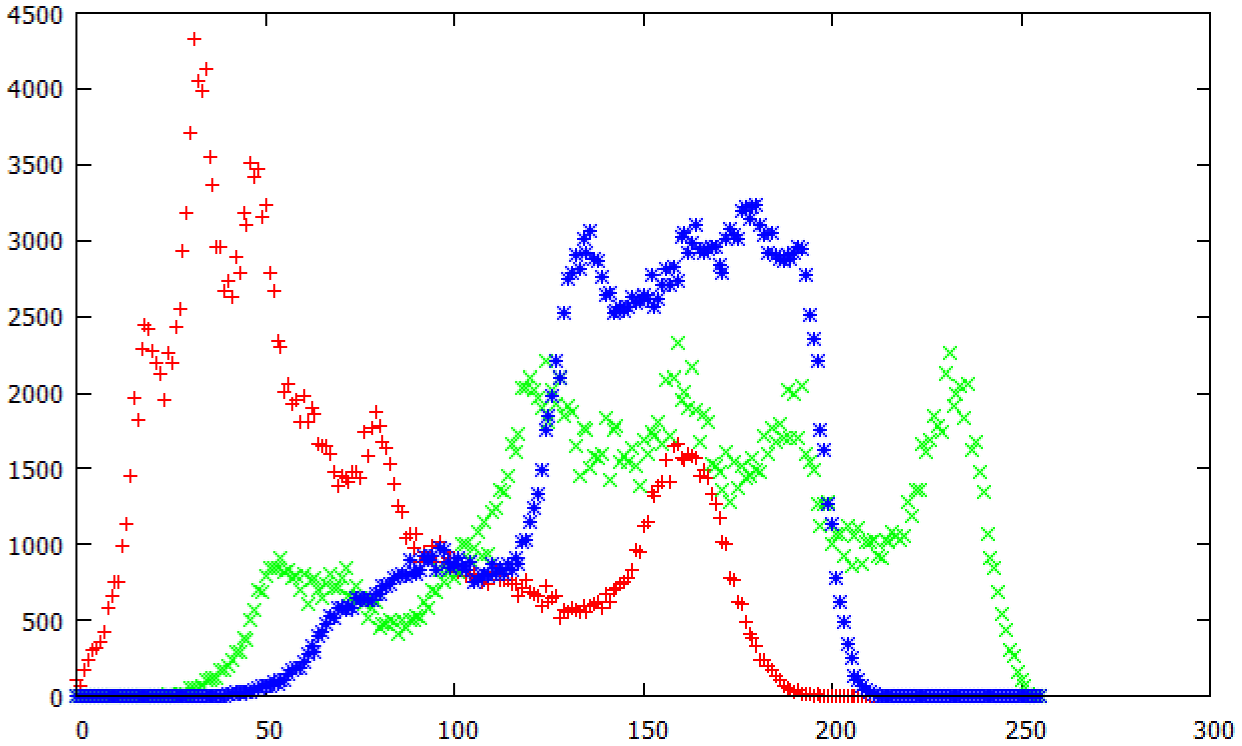
It is easily implemented by a Look Up Table (an array of integer)

newImage(i,j) = LUT[originalImage(i,j)] (Common programming language notation)

# A simple example: the negative image

$$LUT(n) = 255 - n$$

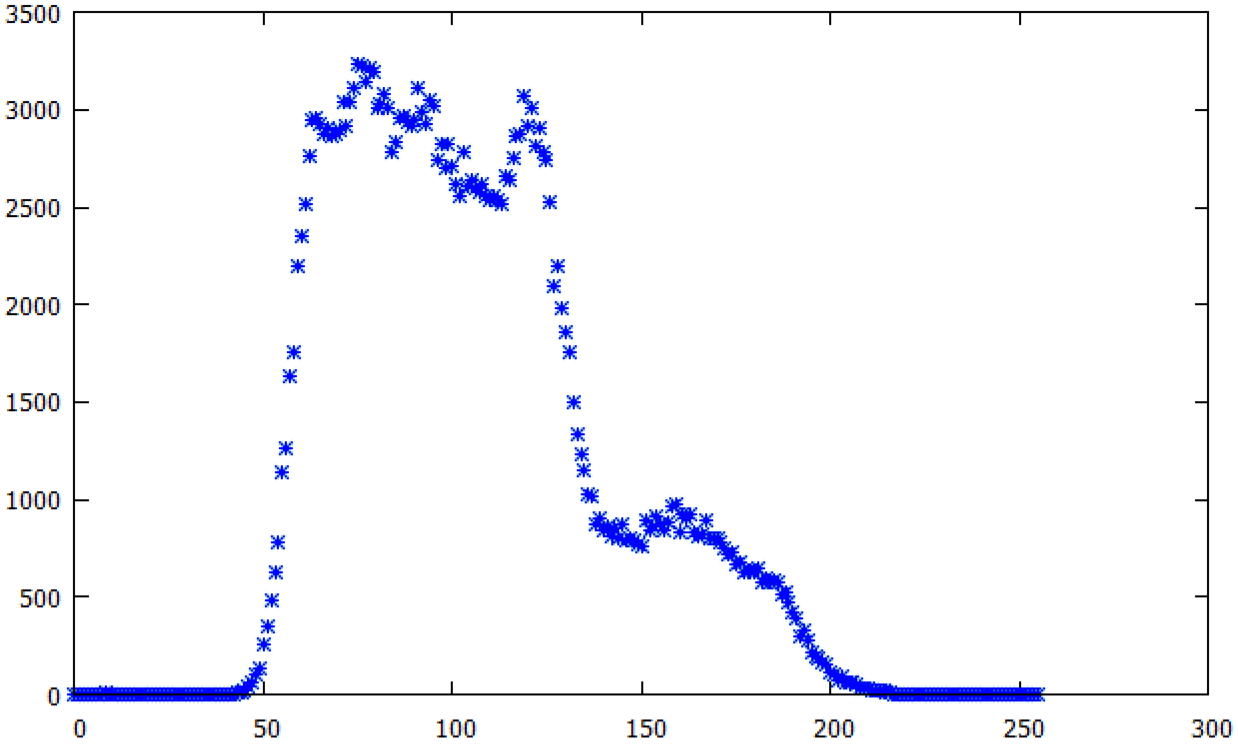



The histogram is reflected respect to the original

The original image can be obtained by applying a second time the LUT

# An image with a limited range



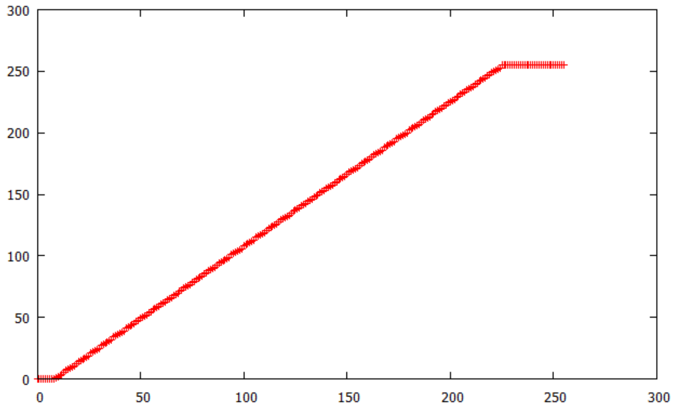

Only a small range is used in the image, very clear and very dark pixels are absent

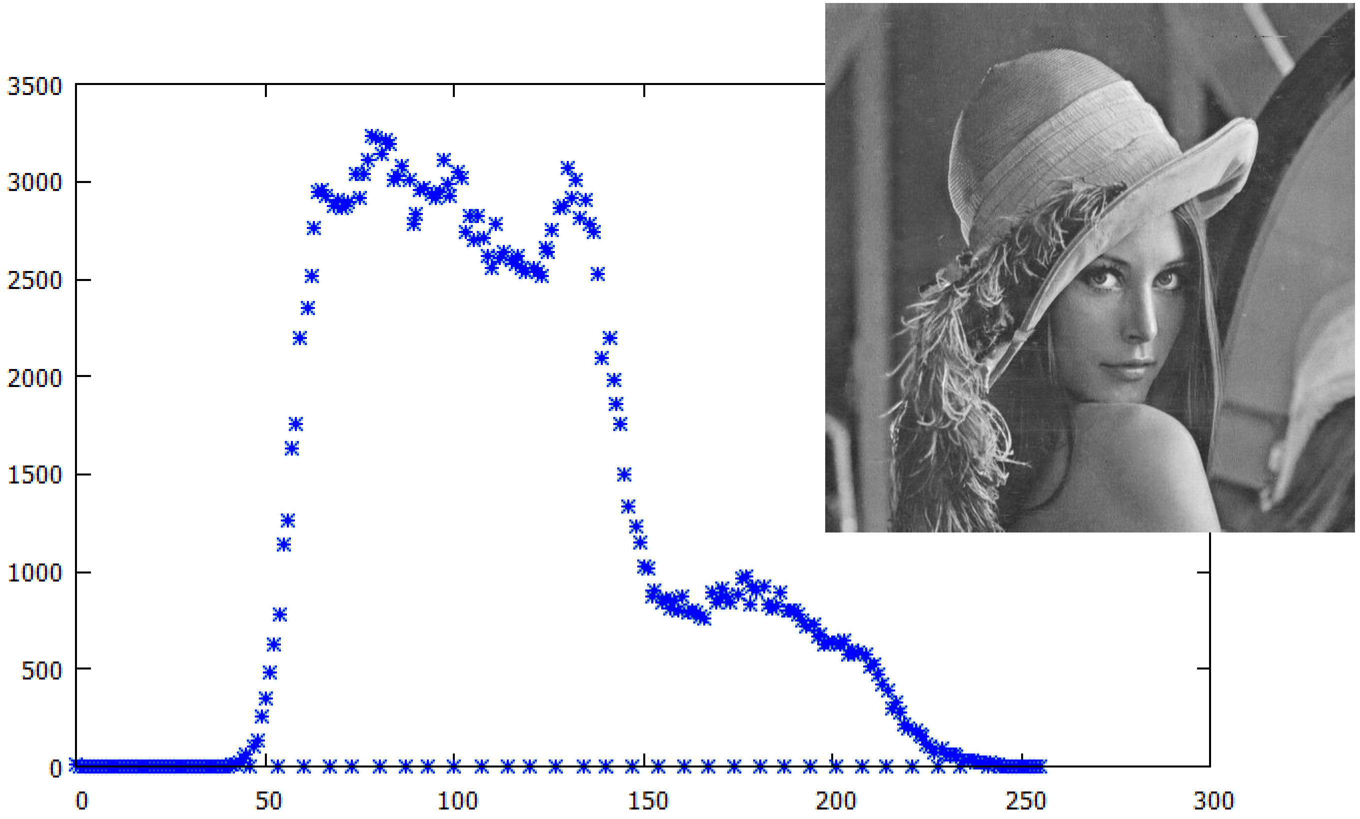
In this example the blue channel has been selected

# Extended dynamics

The pixel distribution may be extended by stretching with the following LUT (min e max, are the minimum and maximum values):

$$LUT(n) = 255 \times \frac{n - \min}{\max - \min}$$

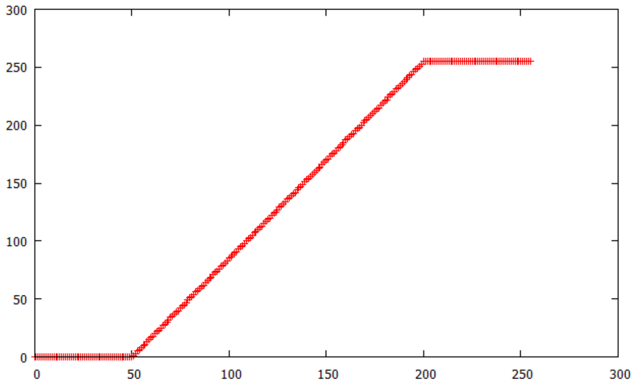


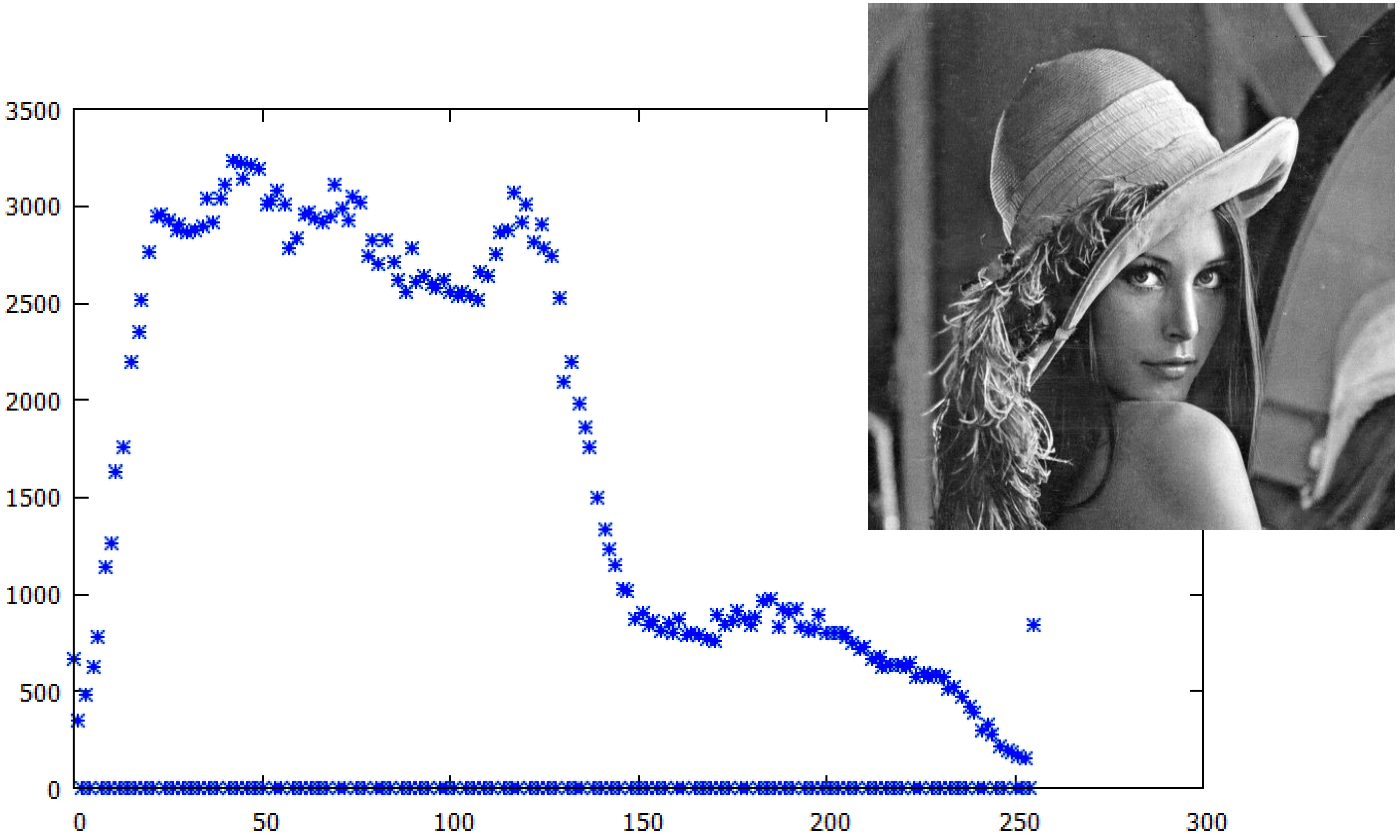



# Highlight contrast in a range

A small range may be enhanced (in this case a=50, b=200)

$$LUT(n) = \begin{cases} 0 & se \quad n < a \\ 255 \times \dfrac{n-a}{b-a} & se \quad a \le n \le b \\ 255 & se \quad n > b \end{cases}$$

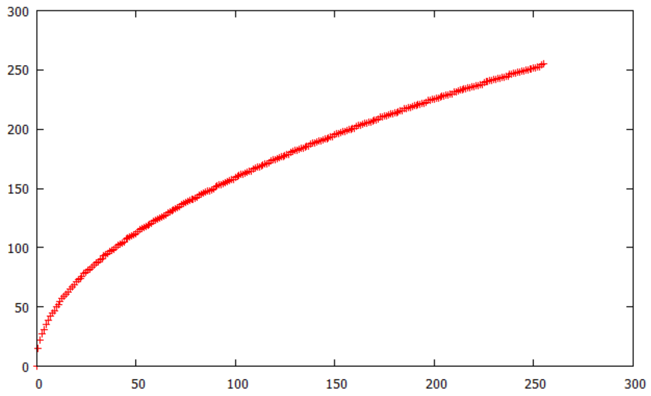


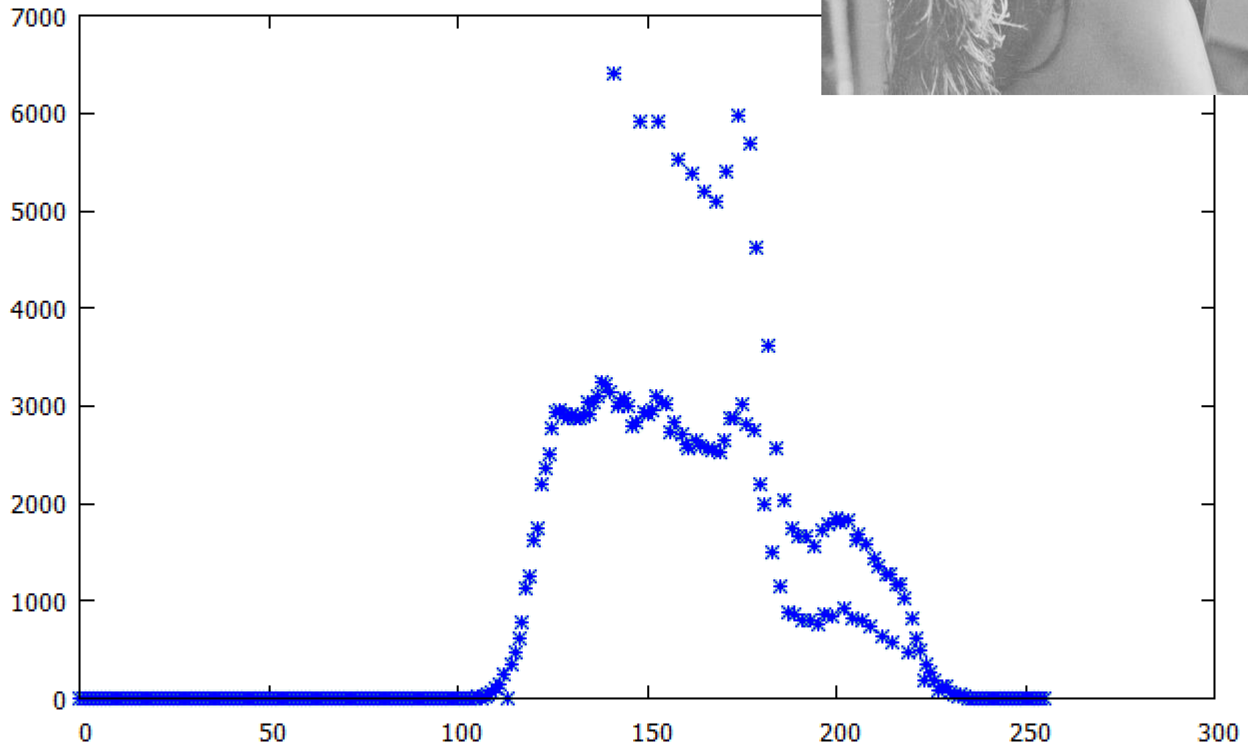



# Bright image

The histogram is compressed towards high values

$$LUT(n) = \sqrt{255 \times n} = 255\sqrt{n/255}$$

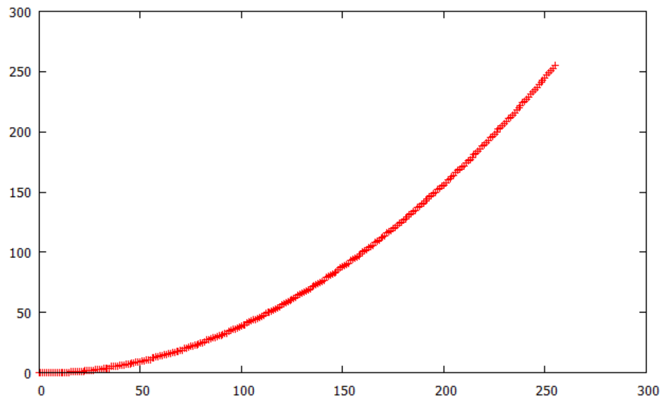


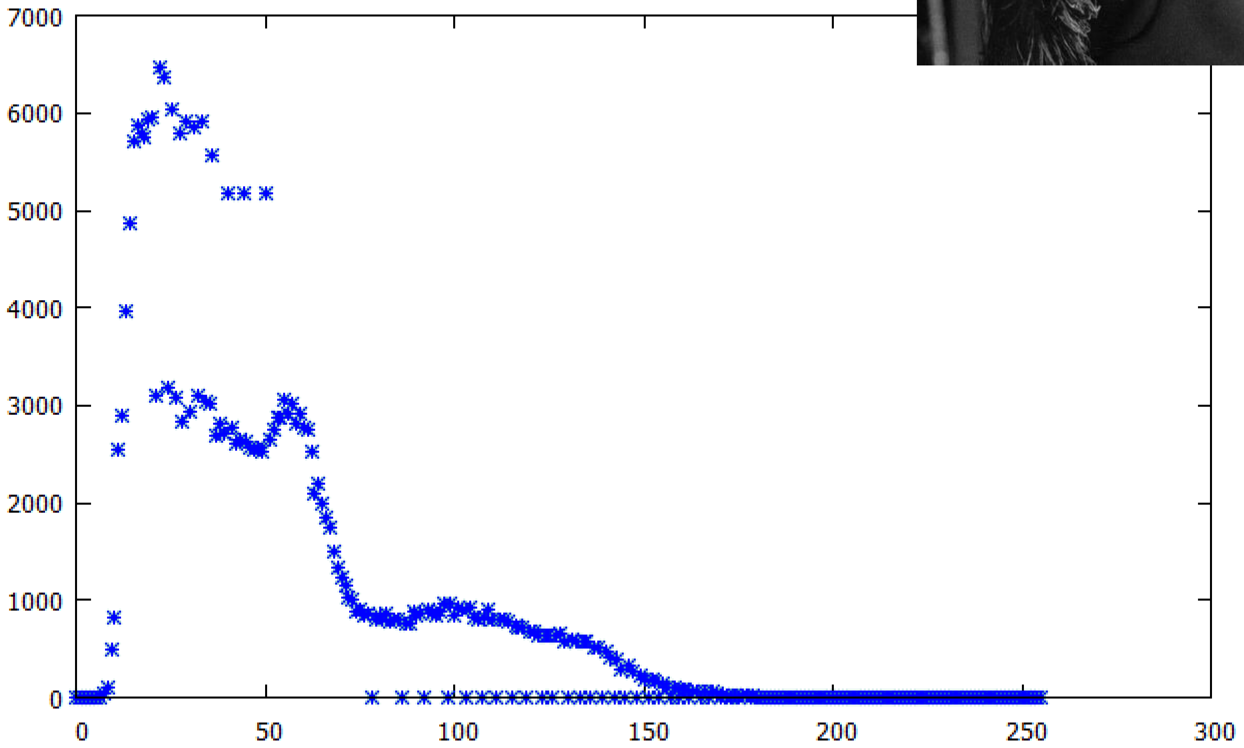




# Dark image

The histogram is compressed towards low values

$$LUT(n) = \frac{n^2}{255} = 255(n/255)^2$$



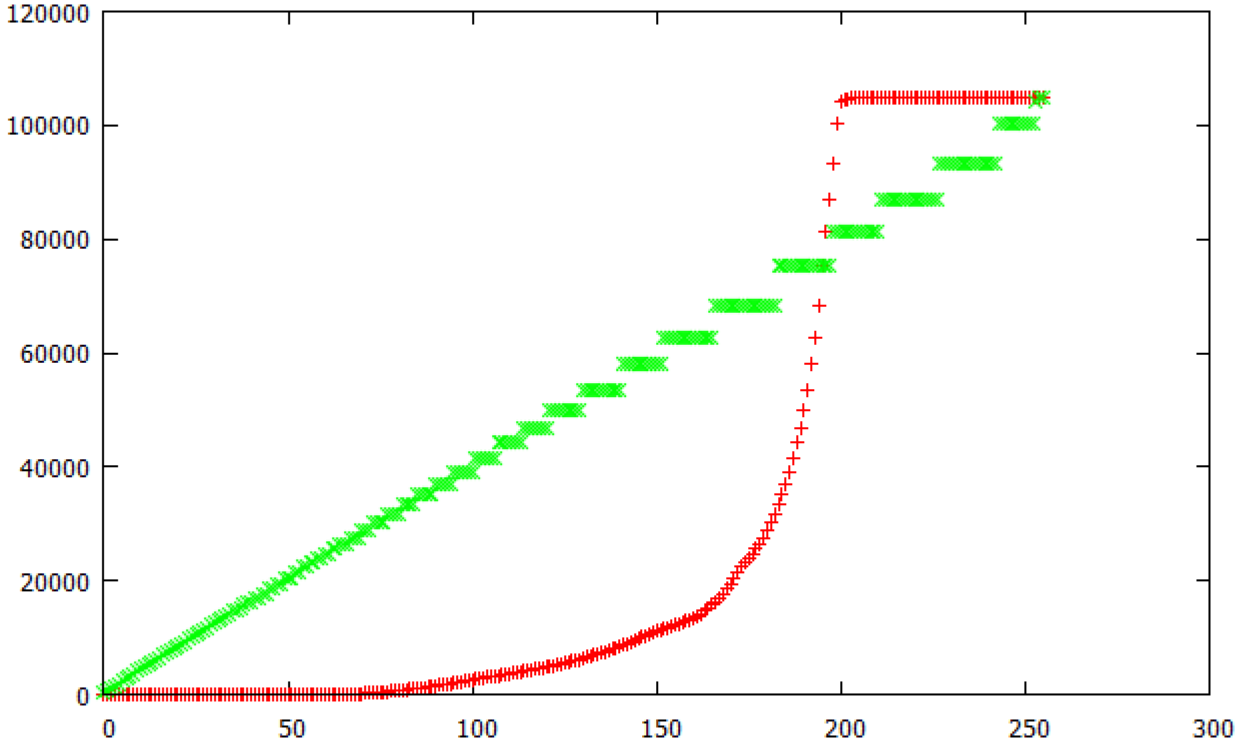




# Image equalization

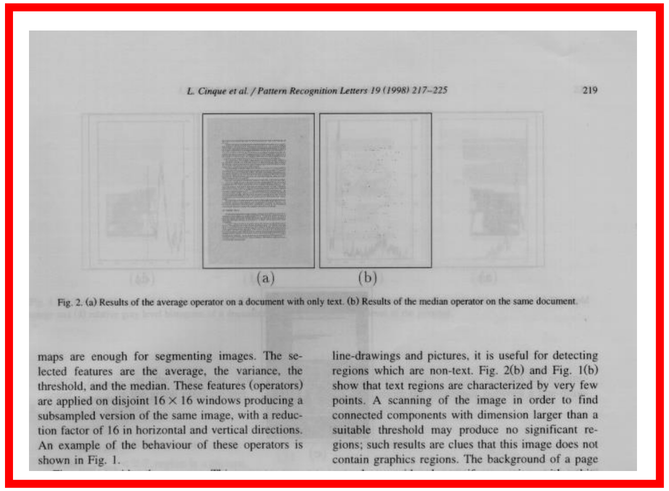
To obtain a uniform distribution of image contrast a technique known as equalization is employed. This technique consists in making the grey level distribution as close as possible to a uniform distribution, in an adaptive way. The more uniform is a grey level distribution, the better contrasted is the associated image

$$LUT(n) = 255 \times \frac{\sum_{i=0}^{n} H(i)}{\sum_{i=0}^{255} H(i)}$$

# Image equalization




Cumulative function: original equalized



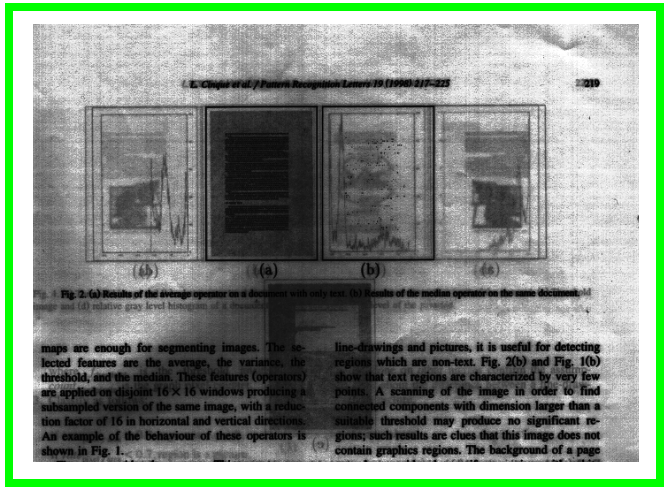
L. Cinque et al. / Pattern Recognition Letters 19 (1998) 217–225 219

(a) (b)

Fig. 2. (a) Results of the average operator on a document with only text. (b) Results of the median operator on the same document.

maps are enough for segmenting images. The selected features are the average, the variance, the threshold, and the median. These features (operators) are applied on disjoint 16 × 16 windows producing a subsampled version of the same image, with a reduction factor of 16 in horizontal and vertical directions. An example of the behaviour of these operators is shown in Fig. 1.

line-drawings and pictures, it is useful for detecting regions which are non-text. Fig. 2(b) and Fig. 1(b) show that text regions are characterized by very few points. A scanning of the image in order to find connected components with dimension larger than a suitable threshold may produce no significant regions; such results are clues that this image does not contain graphics regions. The background of a page



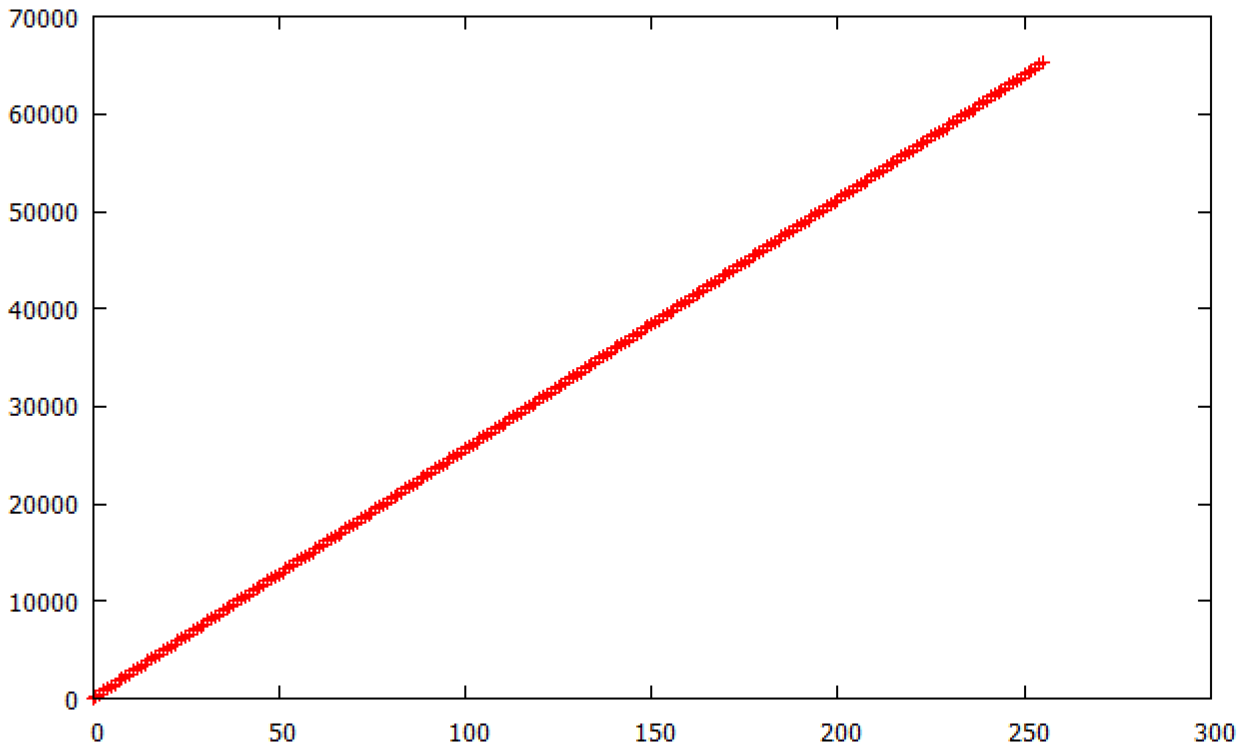

L. Cinque et al. / Pattern Recognition Letters 19 (1998) 217–225 219

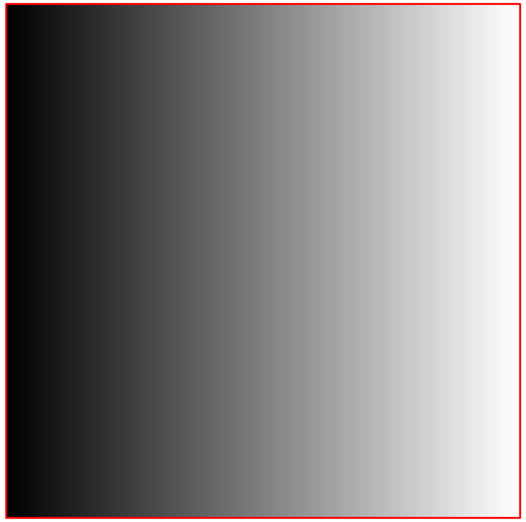
Fig. 2. (a) Results of the average operator on a document with only text. (b) Results of the median operator on the same document.

maps are enough for segmenting images. The selected features are the average, the variance, the threshold, and the median. These features (operators) are applied on disjoint 16 × 16 windows producing a subsampled version of the same image, with a reduction factor of 16 in horizontal and vertical directions. An example of the behaviour of these operators is shown in Fig. 1.

line-drawings and pictures, it is useful for detecting regions which are non-text. Fig. 2(b) and Fig. 1(b) show that text regions are characterized by very few points. A scanning of the image in order to find connected components with dimension larger than a suitable threshold may produce no significant regions; such results are clues that this image does not contain graphics regions. The background of a page

# Perfect uniform distribution



256x256 image
Each column has a unique gray value
Img(i,j) = i

```
magick -size 256x256 canvas: -fx "i/255" ideal.gif
```

The pixel has a value in the range 0.0:1.0

# Back to color images

- Color images are made by pixels of the three different channels (red, green, blue)
  - RGB sequence: RGBRGBRGBRGBRGBRGBRGBRGB …
    - Remember PPM images
  - But it is also used an alternative choice:
  - BGR sequence: BGRBGRBGRBGRBGRBGRBGRBGR …
- Usually each value is a single byte (so 256x256x256 different colors are possible ~16 millions)

# A practical example: Java

- Java memorizes a pixel value by an «`int`» (4 bytes)
  - The less significant byte for the blue channel, then the green channel and the red channel
  - The most significant byte is used for the transparency of the pixel (alpha channel: 0 → invisible pixel, 255 → completly visible)

```
int pixel = 0xFFRRGGBB;
```

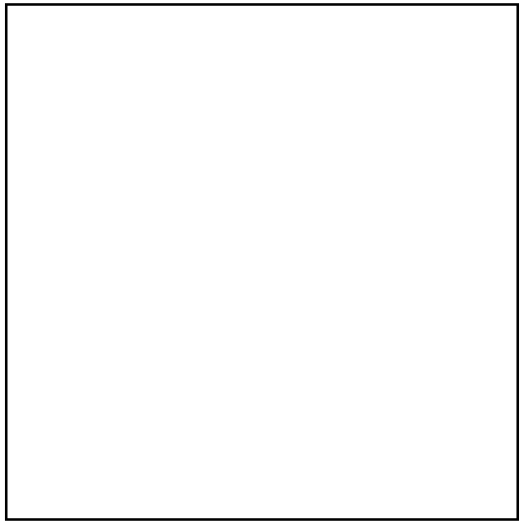
# Some channels are zeroed

java cv.imageframe.Bool 0xFFFF0000 lena-c256.png

java cv.imageframe.Bool 0xFF00FF00 lena-c256.png

java cv.imageframe.Bool 0xFF0000FF lena-c256.png

java cv.imageframe.Bool 0x00FFFFFF lena-c256.png



# Get the value of each channel

- V is the int value of each pixel
  - R = (V >> 16) & 255; // R = (V/0x10000) & 255
  - G = (V >> 8) & 255;
  - B = (V) & 255;
  - Of course $0 \leq R,G,B \leq 255$
- In a similar way we get the pixel value from the channels
  - V = (R<<16) | (G<<8) | (B) | 0xFF000000;

# Gray scale images

- A gray scale image is an image where all channels are equal
  - A simple way to get the gray value (or brightness) from color images is

$$G = (R+G+B)/3$$

  - But the human eyes have different sensibilities for different colors, a common choise is

$$G = 0.299*R + 0.587*G + 0.114*B$$

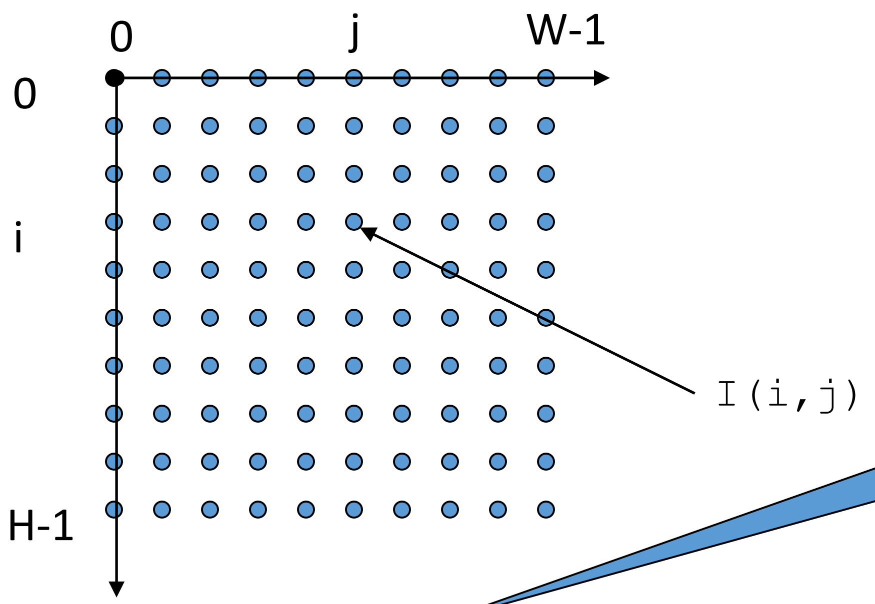
  - The pixel values for a gray image is

$$V = \text{0xFF000000} \mid (G \ll 16) \mid (G \ll 8) \mid G$$

# Simple visualization

- See cv.imageframe.ImageFrame

# Standard reference system (get pixels)



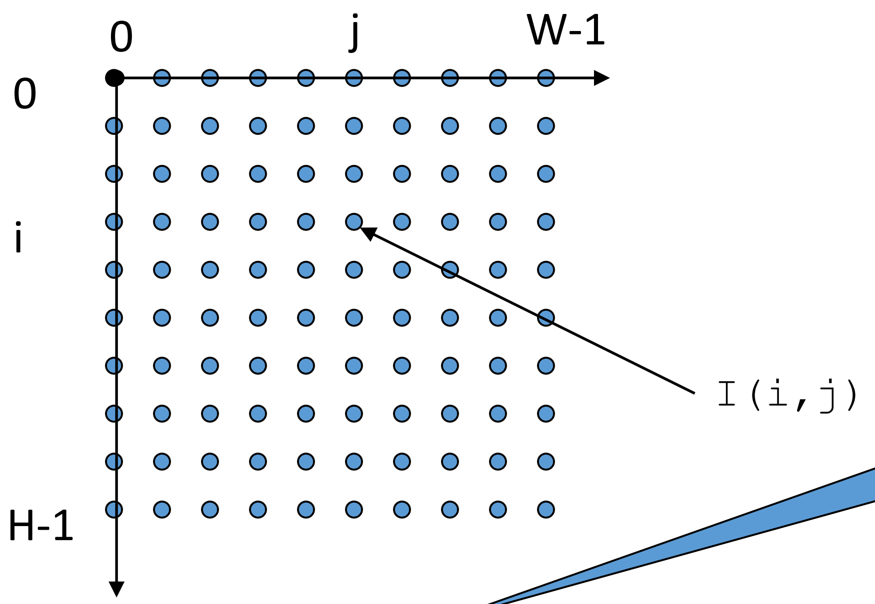

Get a window of size w and h

```
int[] rgbArray = img.getRGB(startX, startY, w, h, null, offset, scansize );
int pixel = rgbArray[offset + (y-startY)*scansize + (x-startX)];
rgbArray.length:(offset + (h-startY)*scansize)
int[] rgbArray = img.getRGB(0, 0, W, H, null, 0, W );
int pixel = pix[y*W + x]; (W*H)
```

Get the complete image

See Java API java/awt/image/BufferedImage.html

# Standard reference system (set pixels)




```
img.setRGB(startX, startY, w, h, rgbArray, offset, scansize );
rgbArray[offset + (y-startY)*scansize + (x-startX)] = pixel;
img.setRGB(0, 0, W, H, rgbArray, 0, W );
```

Set a window of size w and h

Set the complete image, rgbArray is the array with the new values

See Java API awt/image/BufferedImage.html

# Get an image as a matrix

- Get row by row:

```
int[][] mat = new int[h][w];
for(int i=0; i<h; i++) mat[i] = img.getRGB(0,i,w,h,null,0,w);

int pixel = mat[i][j]; // to handle a single pixel
mat[i][j] = LUT[pixel];
```

See cv.imageframe.FlipVertical

# Other approach

- Get/set a single pixel

```
int value = img.getRGB(x, y);
img.setRGB(x, y, value);  // x, y the coordinates of the pixel
```

See cv.imageframe.FlipHorizontal, cv.imageframe.Transpose

# Automatic thresholding

- Simple approach:
- Start from a first threshold (as an example 128, (max+min)/2, …)
  - Evaluate the means of the two regions
  - Evaluate a new threshold $(m_1+m_2)/2$
  - Repeat if the two thresholds differ more then a fixed constant

- Otsu's algorithm:
  - Optimize respect a threshold $p_1(m_1-m)^2+p_2 (m_2-m)^2$
  - m mean of the image, $p_i$ is the probability that a pixel belongs to region i
  - It may be generalized respect to 2 thresholds (and 3 regions)
    - $p_1(m_1-m)^2+p_2 (m_2-m)^2+p_3 (m_3-m)^2$
- Both algorithms may be optimized using the histogram of the image (not the image)