TrabHCI *Technologies to Reduce the Access Barrier in Human Computer Interaction*
Erasmus Intensive Programme IP29588-1-1731-10

Image processing for gesture recognition: from theory to practice

2

Michela Goffredo
University Roma TRE
goffredo@uniroma3.it

Image processing

- At this point we have all of the basics at our disposal. We understand the structure of the library as well as the basic data structures it uses to represent images. We understand the HighGUI interface and can actually run a program and display our results on the screen. Now that we understand these primitive methods required to manipulate image structures, we are ready to learn some more sophisticated operations.
- We will now move on to higher-level methods that treat the images as images, and not just as arrays of colored (or grayscale) values.
- We said that **Computer vision** is the transformation of data from a still or video camera into either a decision or a new representation.
- **Image processing** is part of Computer Vision and aim at transforming the image so that information can be extracted.
- Processing can be divided into:
  - **global** operator: transforms the whole image
  - **local** operator: transform a region of the image

Noisy images

Images gathered for the real world are not clean and sharp as the synthetic ones.

But they present random variation of brightness or color information.

These discontinuities and local changes are called noise and depends on:

- Sensor
- Local change of light
- Sampling
- Quantization

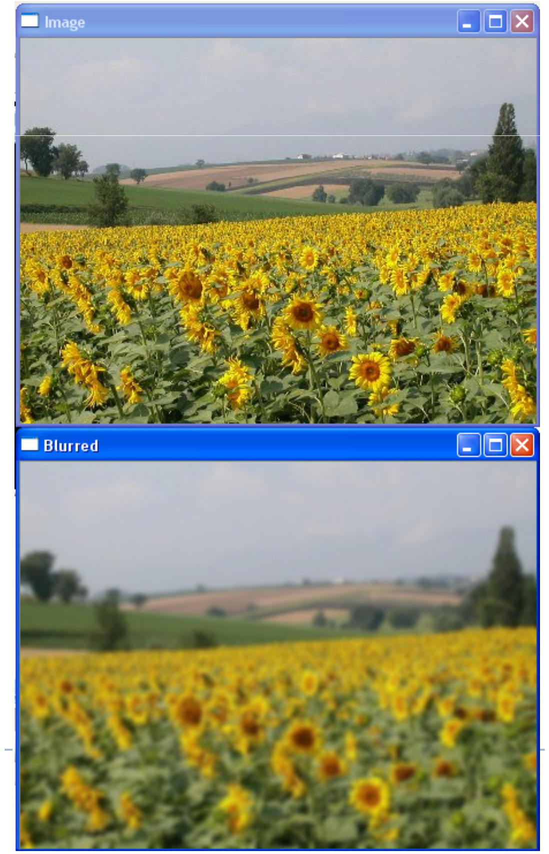
Smoothing

Smoothing is a basic image transformation used for blurring, for noise reduction and for camera artifacts decreasing.

Blurring is used in pre-processing steps, such as removal of small details from an image prior to (large) object extraction, and bridging of small gaps in lines or curves.

Different operators cause image smoothing:

- Mean filter
- Median filter
- Gaussian filter
- Bilateral filter



Smoothing

- <u>Mean filter</u>

Each pixel in the output is the mean of all of the pixels in a window around the corresponding pixel in the input.

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

3x3 mean filter:

Neighbour values:
124,126,127,120,150,125,115,119,123

Mean value (rounded):
125

The central pixel value of 150 is rather unrepresentative of the surrounding pixels and is replaced with the mean value: 125. A 3×3 square neighborhood is used here. Larger neighborhoods will produce more severe smoothing.

Smoothing

- <u>Median filter</u>

Each pixel in the output is the median of all of the pixels in a window around the corresponding pixel in the input.

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

3x3 mean filter:

Neighbour values:
124,126,127,120,150,125,115,119,123

Median value:
124

The central pixel value is replaced with the median value 124. The median filter does not "create" a different value. For this reason it is widely claimed to be 'edge-preserving' since it theoretically preserves step edges without blurring.

Mean vs Median

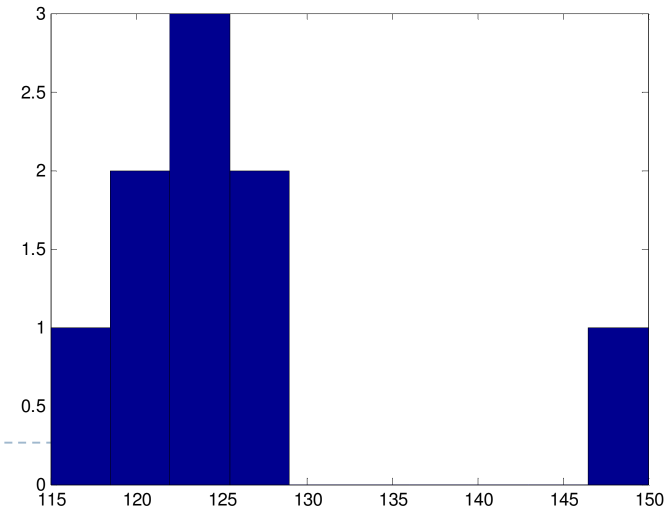
- Mean value: arithmetic average value of the values:

$$A := \frac{1}{n}\sum_{i=1}^{n} a_i$$

- Median value: numeric value separating the higher half of a sample from the lower half. It can be found by arranging all the observations from lowest value to highest value and picking the middle one.

124 126 127 120 150 125 115 119 123

115 119 120 123 124 125 126 127 150

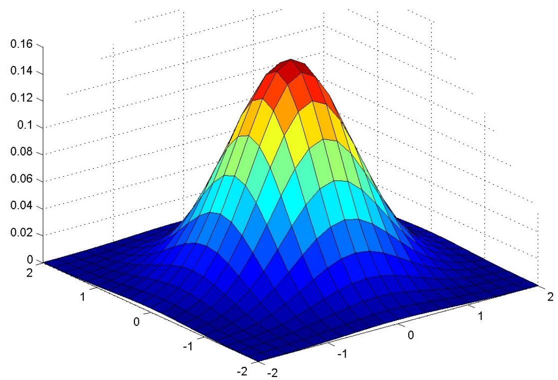


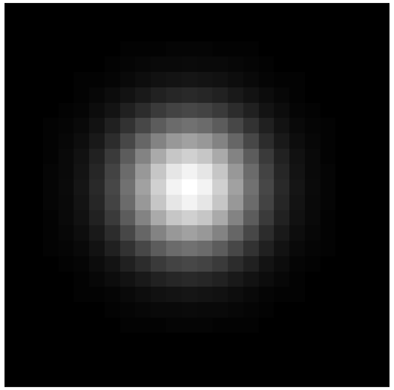
Smoothing

- <u>Gaussian filter</u>

It is achieved by convolving each pixel with a Gaussian window.

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$





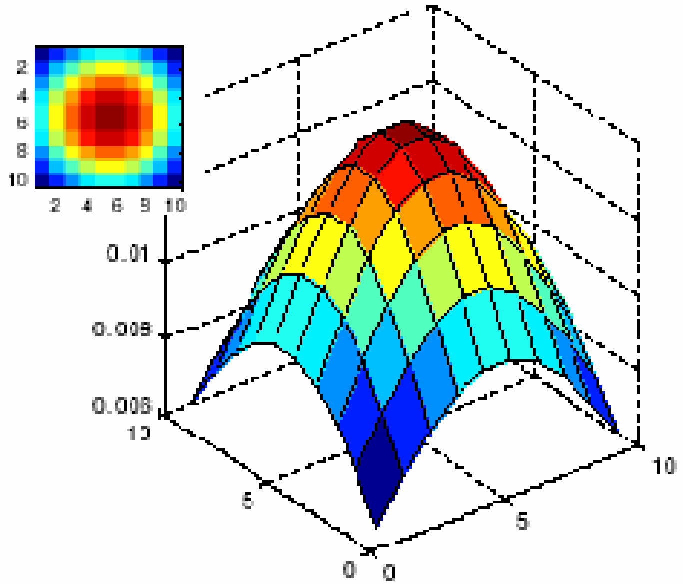
0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

5 x 5, $\sigma = 1$

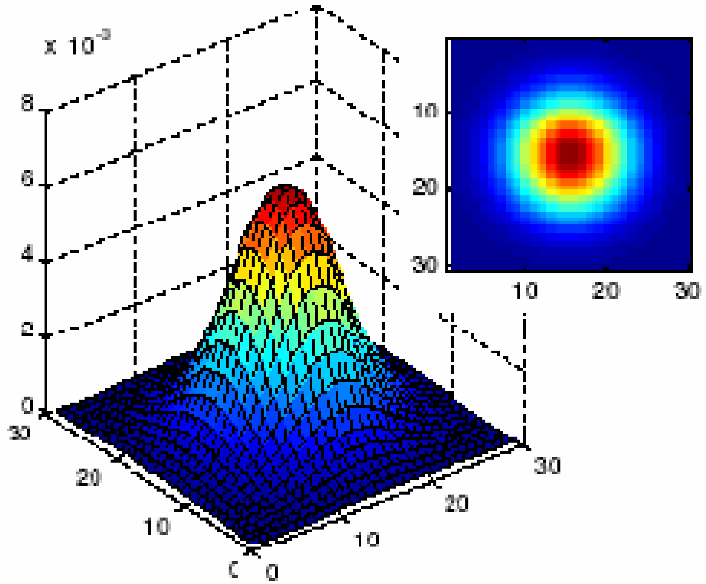
Smoothing

Defining the Gaussian kernel means fixing its size and σ.

Look what's going on by changing the kernel size:



$\sigma = 5$ with 10x10 kernel



$\sigma = 5$ with 30x30 kernel

Rule of thumb: set filter half-width to about 3σ

Smoothing

How does it work?

Assumptions:

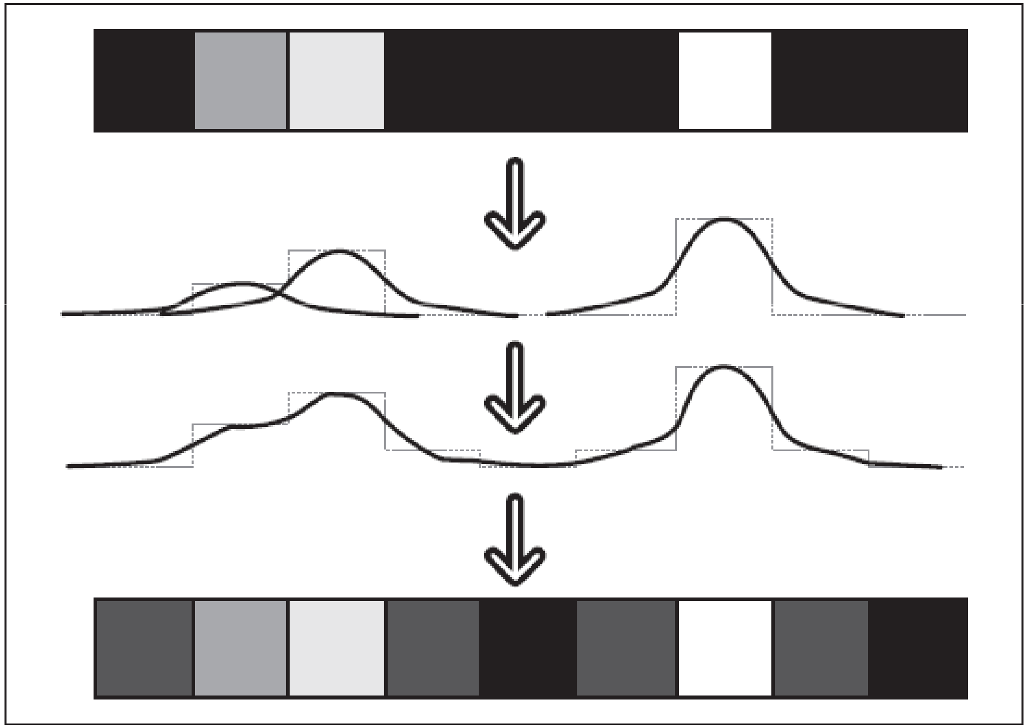
- pixels in a real image should vary slowly over space and thus be correlated to their neighbors;
- random vary greatly from one pixel to the next (i.e., noise is not spatially correlated).

Therefore:

- Gaussian smoothing reduces noise while preserving signal.

Smoothing

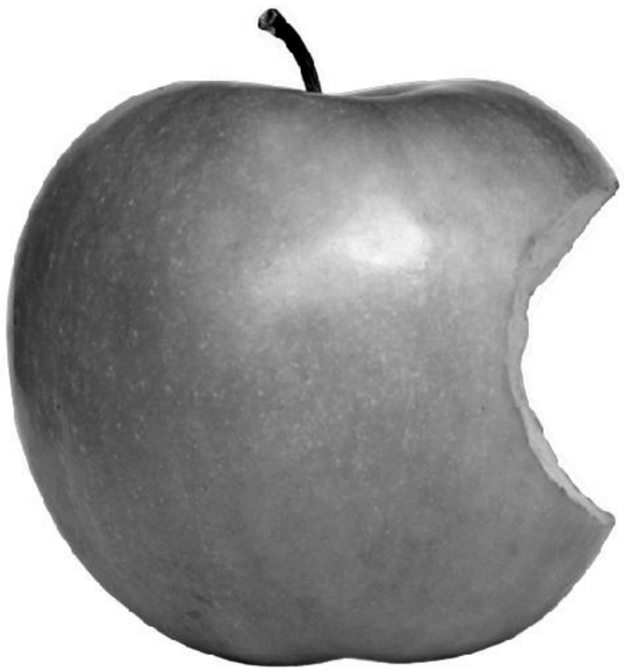
Let's have a look at the 1D case:



- Unfortunately, this method breaks down near edges, where you do expect pixels to be uncorrelated with their neighbours...

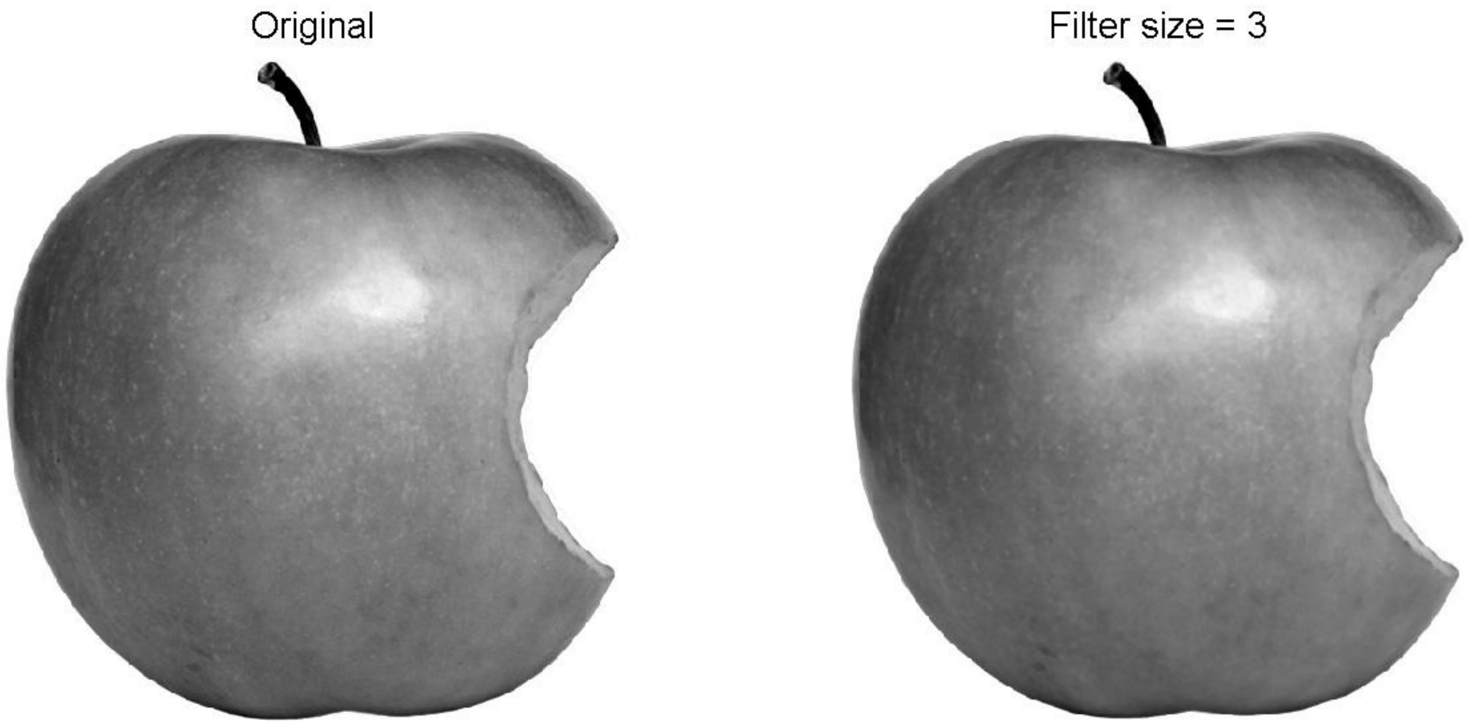
Smoothing

Example of Gaussian filter on a 564x528 image:



Smoothing

Differences with the median filter



Smoothing

With images having lots of details…

Gaussian

Median

3x3 5x5 7x7

Smoothing

- Bilateral filter

It is an edge-preserving and noise reducing smoothing filter.

The intensity value at each pixel in an image is replaced by a weighted average of intensity values from nearby pixels.

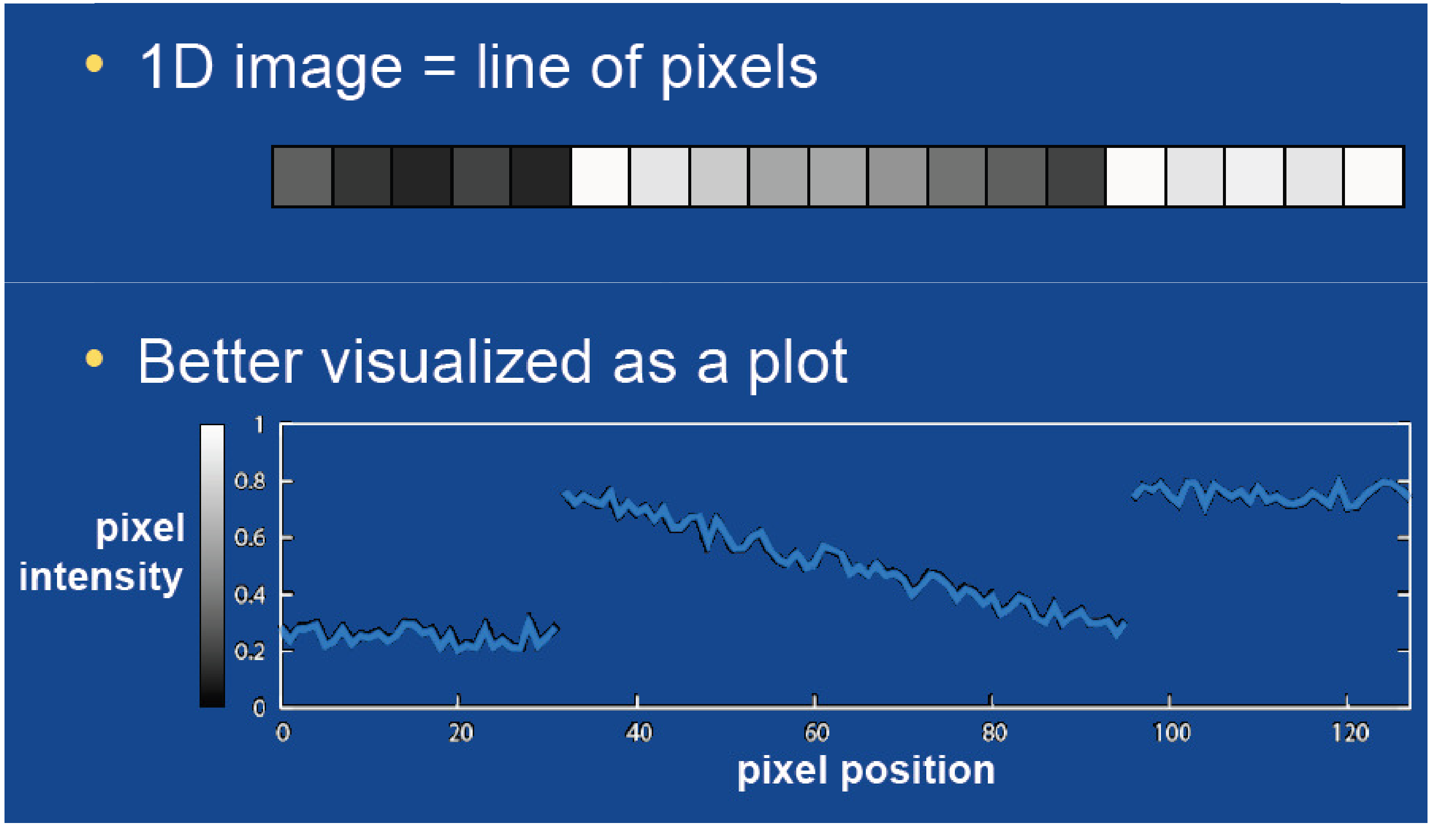
The weights depend not only on Euclidean distance but also on the radiometric differences (differences in the range of color/gray intensity).

We need to set 3 parameters:

- Filter size
- σ_1 (spatial-domain standard deviation, like the Gaussian filter)
- σ_2 (intensity-domain standard deviation)

Smoothing

- Bilateral filter



Smoothing

We need to set 3 parameters:

- Filter size w
- σ_1 (spatial-domain standard deviation, like the Gaussian filter)
- σ_2 (intensity-domain standard deviation)

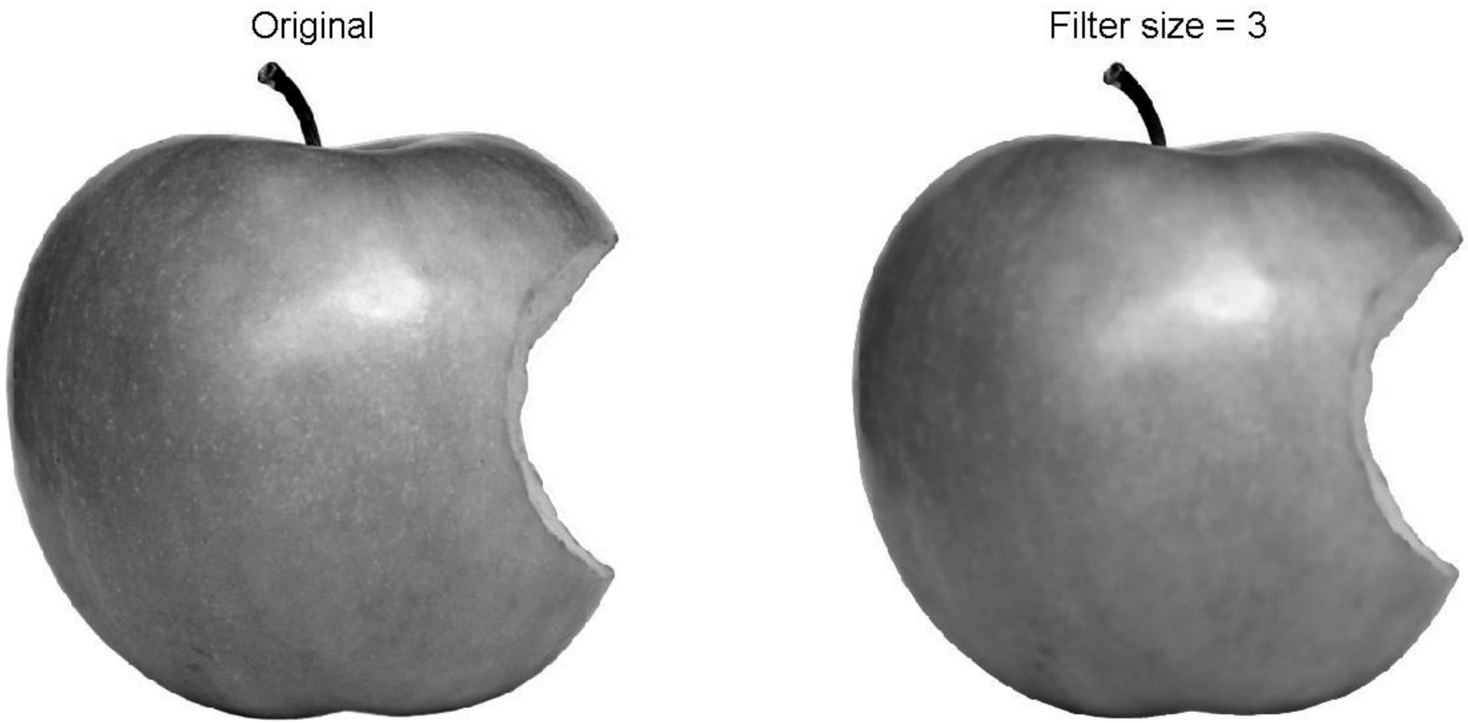
Rules of thumb:

$\sigma_1 = (w/2)*3$

σ_2 = the larger this parameter is, the broader is the range of intensities that will be included in the smoothing (and thus the more extreme a discontinuity must be in order to be preserved). 0.1 is suggested.

Smoothing

And the apple?



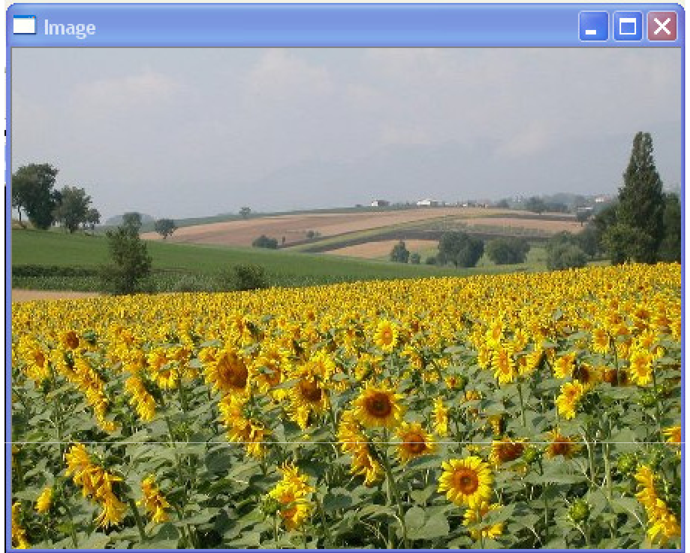
Smoothing

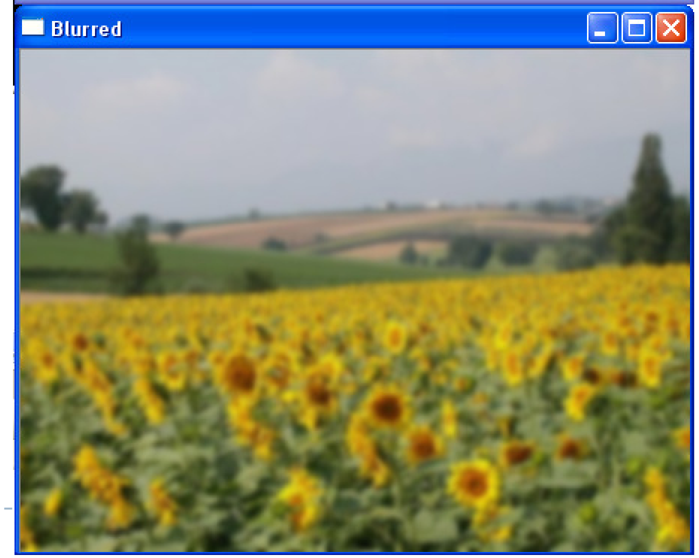
- **OpenCV**

```
void cvSmooth(
  const CvArr*   src,
  CvArr*         dst,
  int            smoothtype   = CV_GAUSSIAN,
  int            param1       = 3,
  int            param2       = 0,
  double         param3       = 0,
  double         param4       = 0
);
```

EXAMPLE

```
cvSmooth(img,img2,CV_GAUSSIAN,11,11);
```





Smoothing

Smooth type	Name	In place?	Nc	Depth of src	Depth of dst	Brief description
CV_BLUR	Simple blur	Yes	1,3	8u, 32f	8u, 32f	Sum over a param1×param2 neighborhood with sub-sequent scaling by 1/(param1×param2).
CV_BLUR_NO_SCALE	Simple blur with no scaling	No	1	8u	16s (for 8u source) or 32f (for 32f source)	Sum over a param1×param2 neighborhood.
CV_MEDIAN	Median blur	No	1,3	8u	8u	Find median over a param1×param1 square neighborhood.
CV_GAUSSIAN	Gaussian blur	Yes	1,3	8u, 32f	8u (for 8u source) or 32f (for 32f source)	Sum over a param1×param2 neighborhood.
CV_BILATERAL	Bilateral filter	No	1,3	8u	8u	Apply bilateral 3-by-3 filtering with color sigma=param1 and a space sigma=param2.

Practice 2/1

Write a program which:

- loads the image "noisy.jpg";
- shows the image on a window;
- apply 4 different smoothing algorithms;
- find a good set of parameters;
- shows the results on different windows;
- saves the best smoothed images

Threshold

- So far we've seen how to smooth and clean a noisy image.
- We said that the aim of image processing is **getting information** from the image itself.
- A simple and useful image processing method for getting information is segmenting pixels with respect to their values, i.e**. segmenting objects**.

- The **basic global threshold algorithm** aims at scanning the image pixel by pixel and labelling each pixel whether the gray level of that pixel is greater or less than a value T.
- If the gray level of the pixel is >= T, then it's set to a maximum value M (usually 255)
- If the gray level of the pixel is < T, then it's set to a minimum value m (usually 0)

Threshold

- See what's happening by varying T

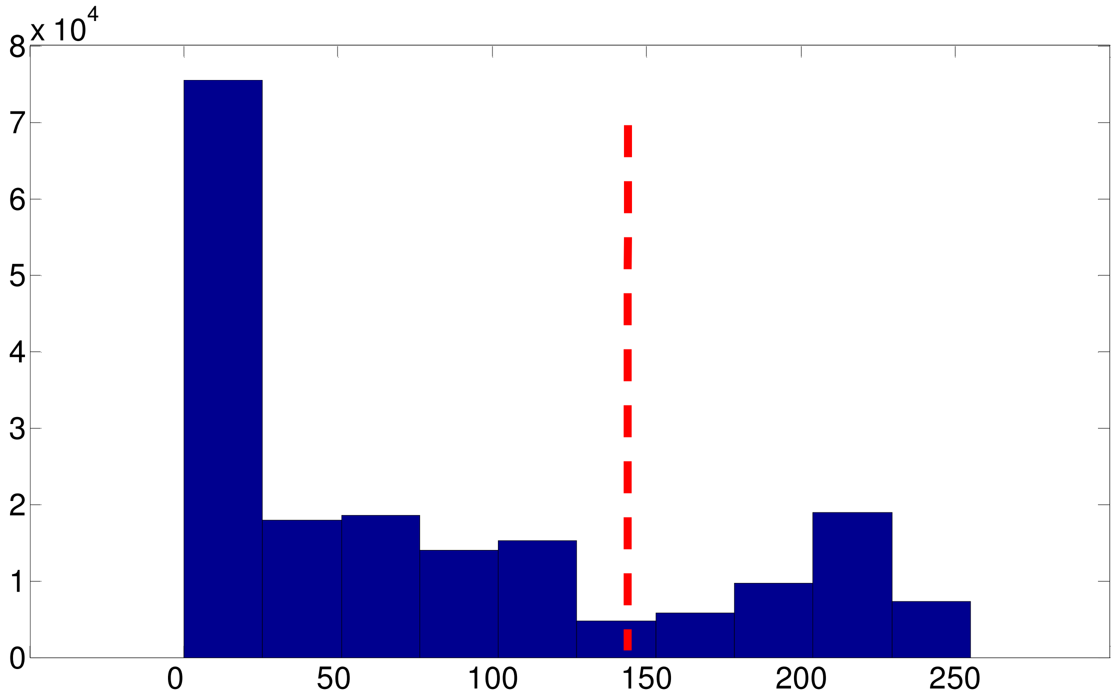
Original

T = 100

Threshold

- How can we set T?

Have a look to the histogram of the grey levels…

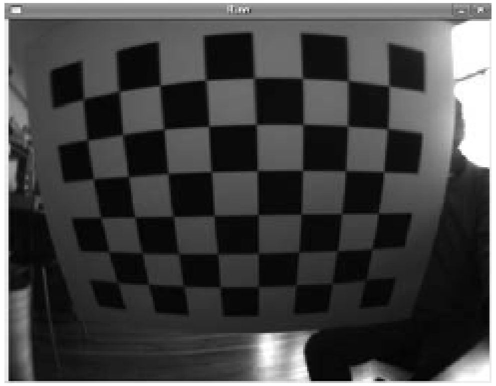


Threshold

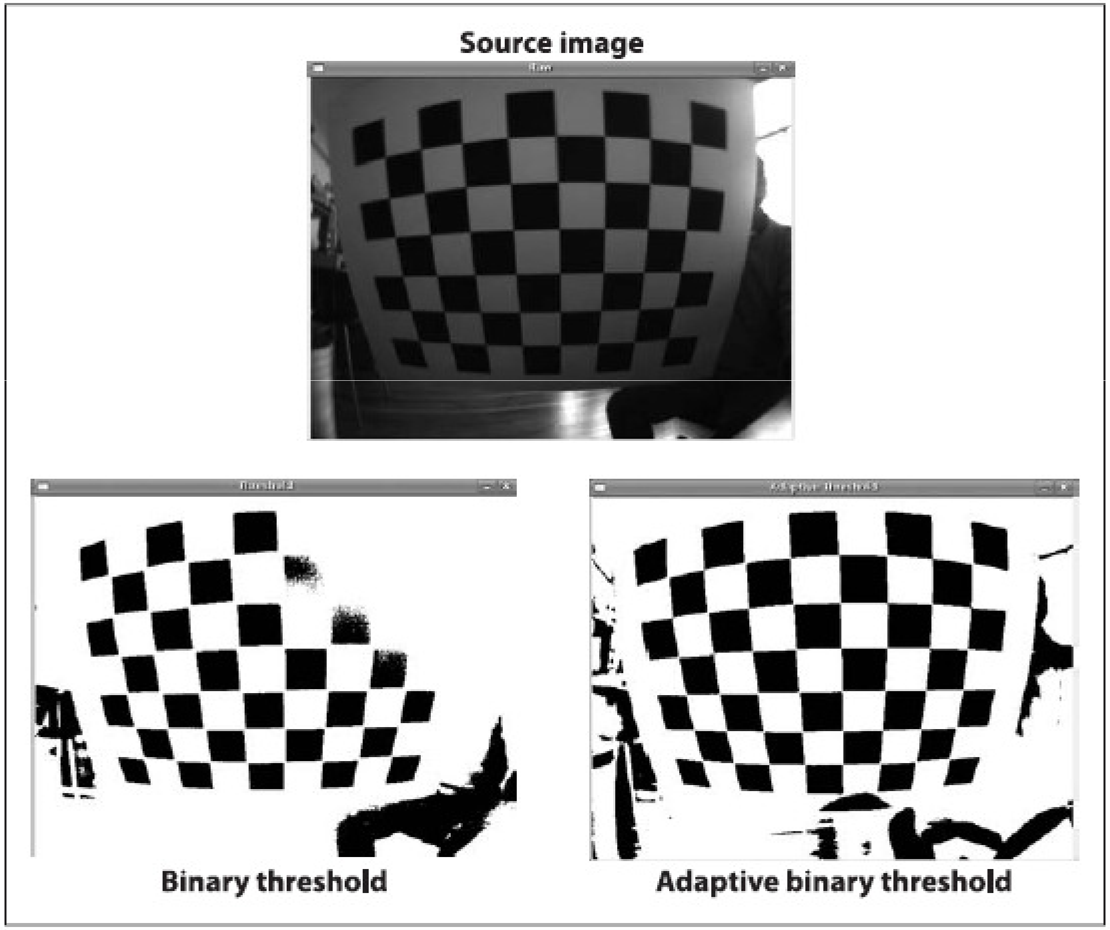
- How can we set T?

Alternatively, there's an approach which **automatically set a threshold T for each pixel** by computing a weighted average of a K-by-K region around each pixel location minus a constant C.

- The **adaptive threshold technique** is useful when there are strong illumination or reflectance gradients that you need to threshold relative to the general intensity gradient.



Threshold



Threshold

- There are several **adaptive threshold methods**, i.e.:

1. T=mean of K × K pixel neighborhood, subtracted by parameter C.

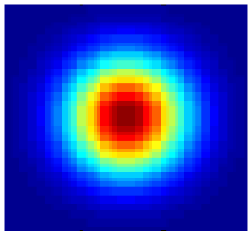
123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

Pixel value = 150
K=3 (3-by-3 window)
C = 5 (parameter)
Mean value: 125
T = 120
The pixel will be set to 255

Threshold

- There are several **adaptive threshold methods**, i.e.:

2. T=weighted sum (gaussian) of K × K pixel neighborhood, subtracted by parameter C.



123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

Pixel value = 150
K=3 (3-by-3 window)
C = 5 (parameter)
T = 134
The pixel will be set to 255

Double Threshold

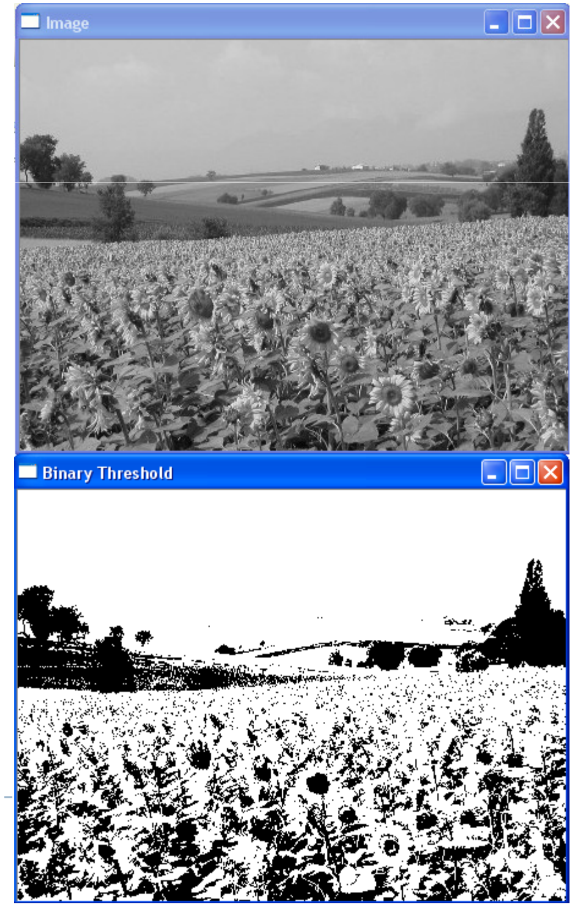
- Sometimes pixels belonging to the object of interest are in the range of 2 values.
- In this case, we need 2 thresholds
- Remember Beckham?

Thresholds: H(0-20); S (30-150); V(80-255)

Threshold

- **OpenCV**
  Comparison operation between the i^{th} source pixel and the threshold.
  The destination pixel may be set to 0, the source pixel, or the max_value.

```
double cvThreshold(
    CvArr*          src,
    CvArr*          dst,
    double          threshold,
    double          max_value,
    int             threshold_type
);
```

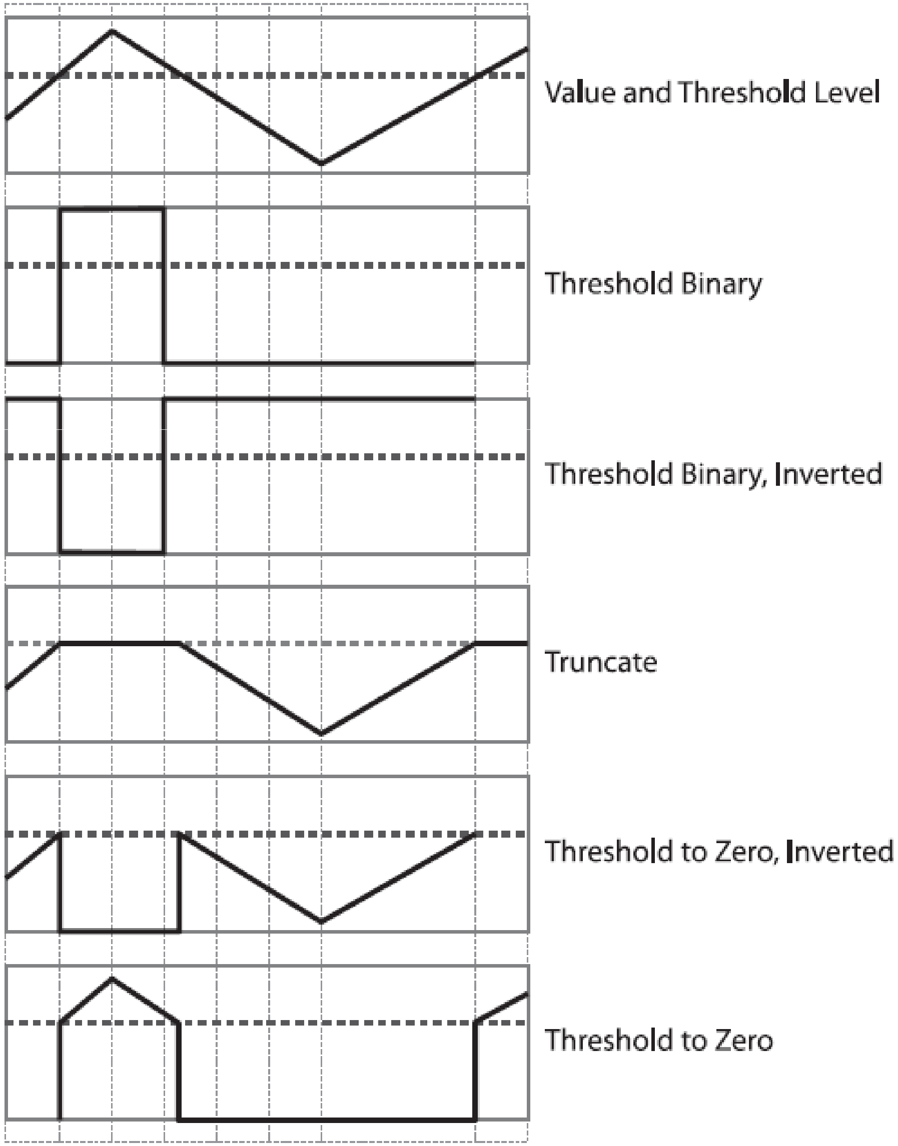


EXAMPLE

```
cvThreshold( img2, img3, 100, 255, CV_THRESH_BINARY );
```

Threshold

Threshold type	Operation
CV_THRESH_BINARY	$dst_i = (src_i > T) \; ? \; M : 0$
CV_THRESH_BINARY_INV	$dst_i = (src_i > T) \; ? \; 0 : M$
CV_THRESH_TRUNC	$dst_i = (src_i > T) \; ? \; M : src_i$
CV_THRESH_TOZERO_INV	$dst_i = (src_i > T) \; ? \; 0 : src_i$
CV_THRESH_TOZERO	$dst_i = (src_i > T) \; ? \; src_i : 0$



Threshold

- **Adaptive Threshold**

```
void cvAdaptiveThreshold(
    CvArr*          src,
    CvArr*          dst,
    double          max_val,
    int             adaptive_method = CV_ADAPTIVE_THRESH_MEAN_C
    int             threshold_type  = CV_THRESH_BINARY,
    int             block_size      = 3,
    double          param1          = 5
);
```

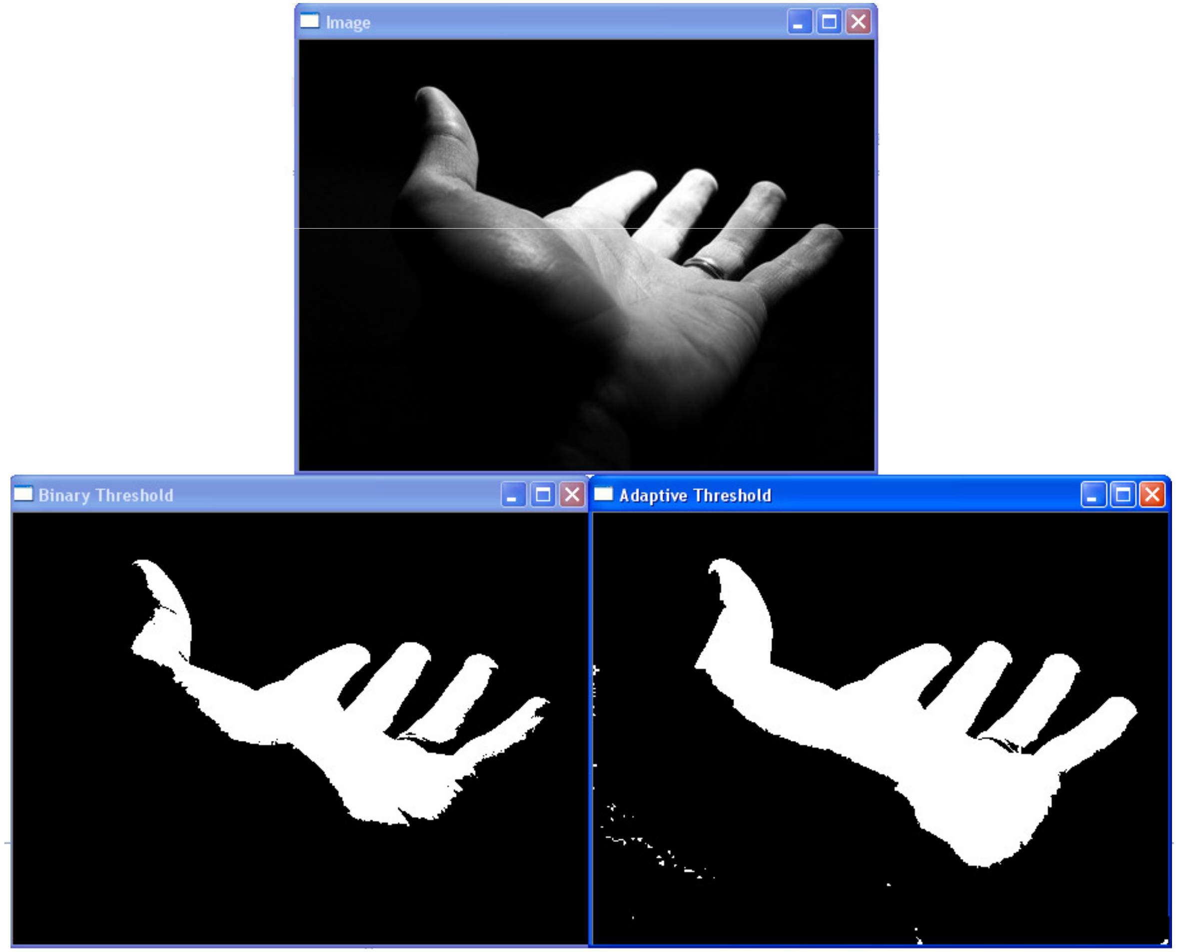
Methods:

- CV_ADAPTIVE_THRESH_MEAN_C: T=mean of block_size × block_size pixel neighborhood, subtracted by param1.
- CV_ADAPTIVE_THRESH_GAUSSIAN_C: T=weighted sum (gaussian) of block_size × block_size pixel neighborhood, subtracted by param1.

Threshold

EXAMPLE

```
cvAdaptiveThreshold(img2, img3, 255, CV_ADAPTIVE_THRESH_GAUSSIAN_C, CV_THRESH_BINARY, 3, 5);
```



InRange

- **2 thresholds**
  Comparison operation between the i[th] source pixel is in range between the values of the i[th] pixels in the lower and upper images or between TWO scalars:

```
void cvInRange(
    const CvArr* src,
    const CvArr* lower,
    const CvArr* upper,
    CvArr*       dst
);
void cvInRangeS(
    const CvArr* src,
    CvScalar     lower,
    CvScalar     upper,
    CvArr*       dst
);
```

- If the value in src is greater than or equal to the value in lower and also less than the value in upper, then the corresponding value in dst will be set to 1; otherwise, the value in dst will be set to 0.

Practice 2/2

Write a program which:

- loads the image "th.jpg";
- transforms the image in gray-levels colorspace;
- changes its size (1/2);
- shows the image on a window;
- finds the best threshold(s) for a good segmentation of the flowers
- saves the results as "th_binary.jpg"

Try these two thresholds:

Th_{min} =50

Th_{max} = 200

Is the binary image you got clean?

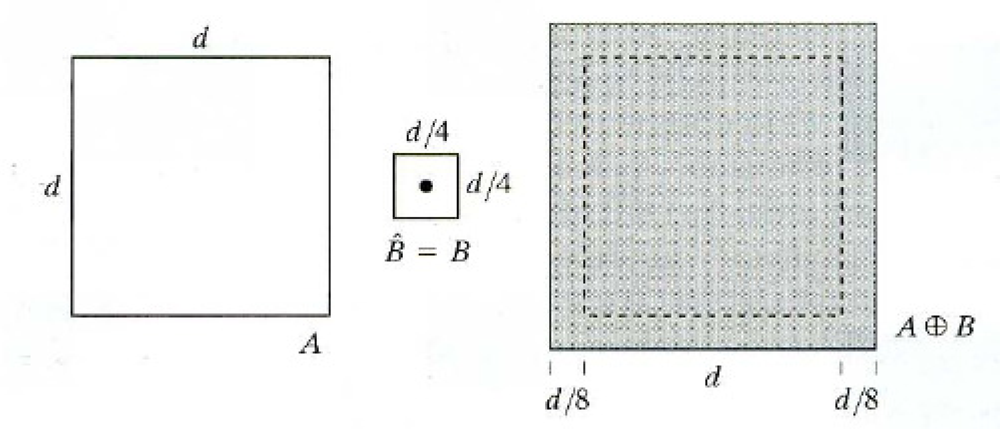
Morphology

- After image threshold, we usually have a **binary image** where white pixels (255) correspond to the object of interest (+ noise)

- Image processing presents very useful algorithms (called Morphological) which allow to **connect isolated pixels sufficiently close to other and/or deleting isolated pixels**.
- The basic morphological transformations are called dilation and erosion, and they arise in a wide variety of contexts such as removing noise, isolating individual elements, and joining disparate elements in an image.

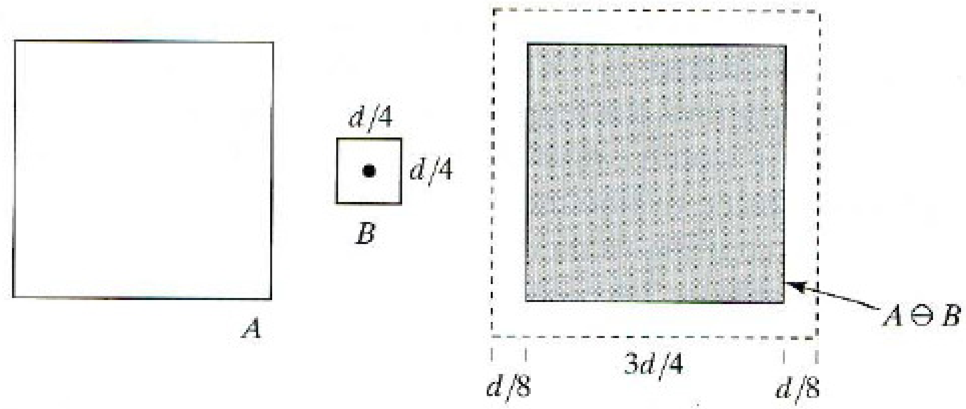
Morphology

- **Dilation** is a convolution of an image A with a kernel B.
- The kernel can be any shape or size.
- Most often, the kernel is a small solid square or disk
- As the kernel B is scanned over the image, we compute the maximal pixel value overlapped by B and replace the image pixel under the central point with that maximal value.
- This causes bright regions within an image to grow → this growth is the origin of the term "dilation operator".



Morphology

- **Erosion** is the converse operation.
- The action of the erosion operator is equivalent to computing a *local minimum over the area of the kernel.*
- As the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the central point with that minimal value.
- This causes dark regions within an image to grow -> this growth is the origin of the term "erosion operator".



Morphology

Usage:

- In general, whereas dilation expands region A, erosion reduces region A.
- The **erode** operation is often used to eliminate "**speckle**" noise in an image. The idea here is that the speckles are eroded to nothing while larger regions that contain visually significant content are not affected.
- The **dilate** operation is often used when attempting to find **connected components** (i.e., large discrete regions of similar pixel color or intensity). The utility of dilation arises because in many cases a large region might otherwise be broken apart into multiple components as a result of noise, shadows, or some other similar effect. A small dilation will cause such components to "**melt**" together into one.

Morphology

- **OpenCV**

```
void cvErode(
    IplImage*       src,
    IplImage*       dst,
    IplConvKernel*  B            = NULL,
    int             iterations = 1
);

void cvDilate(
    IplImage*       src,
    IplImage*       dst,
    IplConvKernel*  B            = NULL,
    int             iterations = 1
);
```

Morphology

- In the NULL case, the kernel used is a 3-by-3 kernel.
- But, you can make your own custom morphological kernels using:

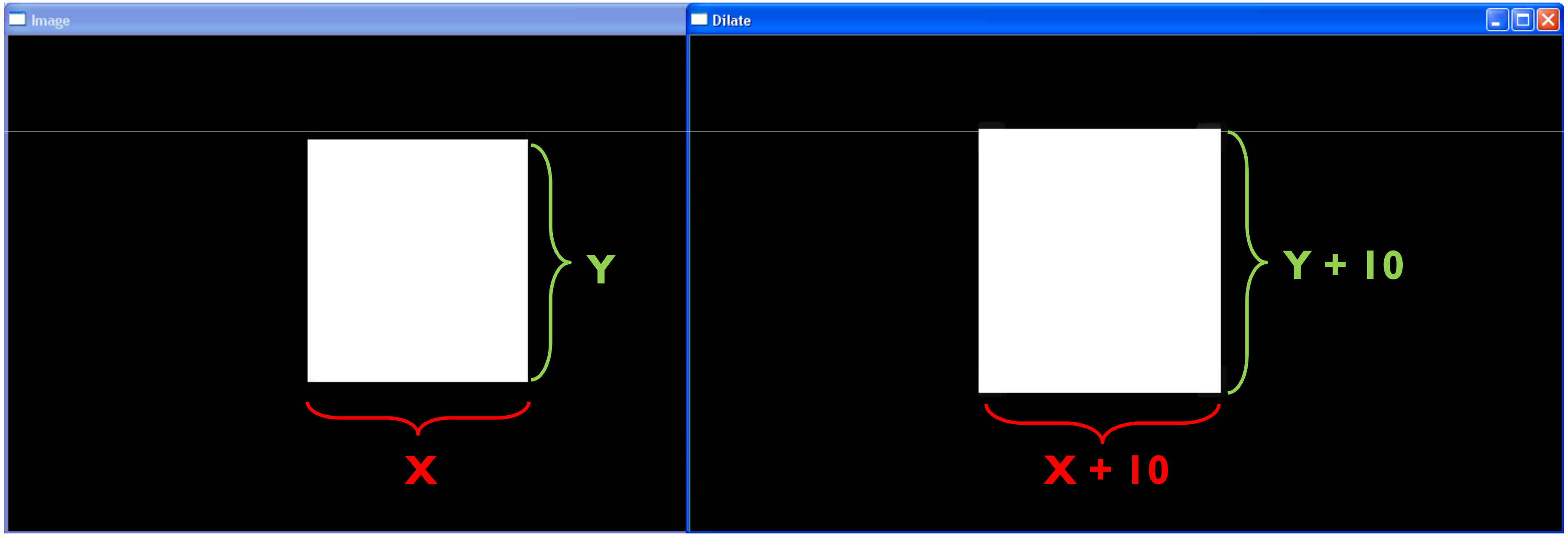
```
IplConvKernel* cvCreateStructuringElementEx(
    int          cols,
    int          rows,
    int          anchor_x,
    int          anchor_y,
    int          shape,
    int*         values=NULL
);
```

Shape value	Meaning
CV_SHAPE_RECT	The kernel is rectangular
CV_SHAPE_CROSS	The kernel is cross shaped
CV_SHAPE_ELLIPSE	The kernel is elliptical

Morphology

EXAMPLE

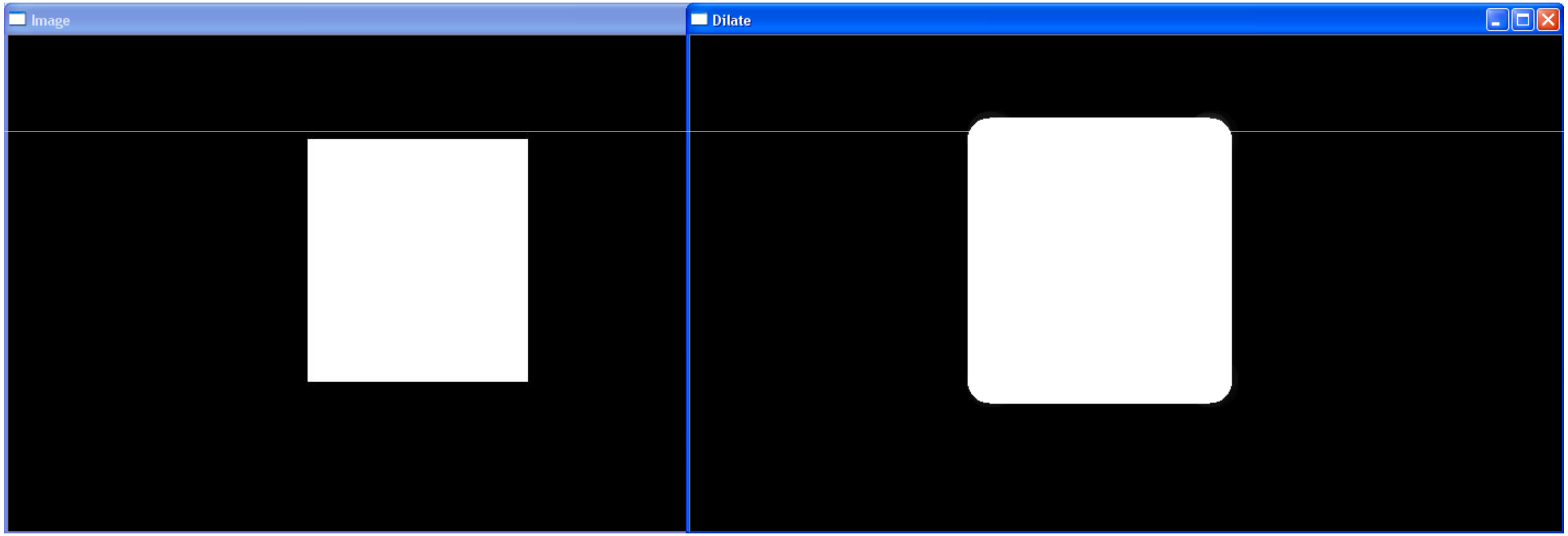
```
cvDilate( img2, img3, NULL, 10);
```



Morphology

EXAMPLE

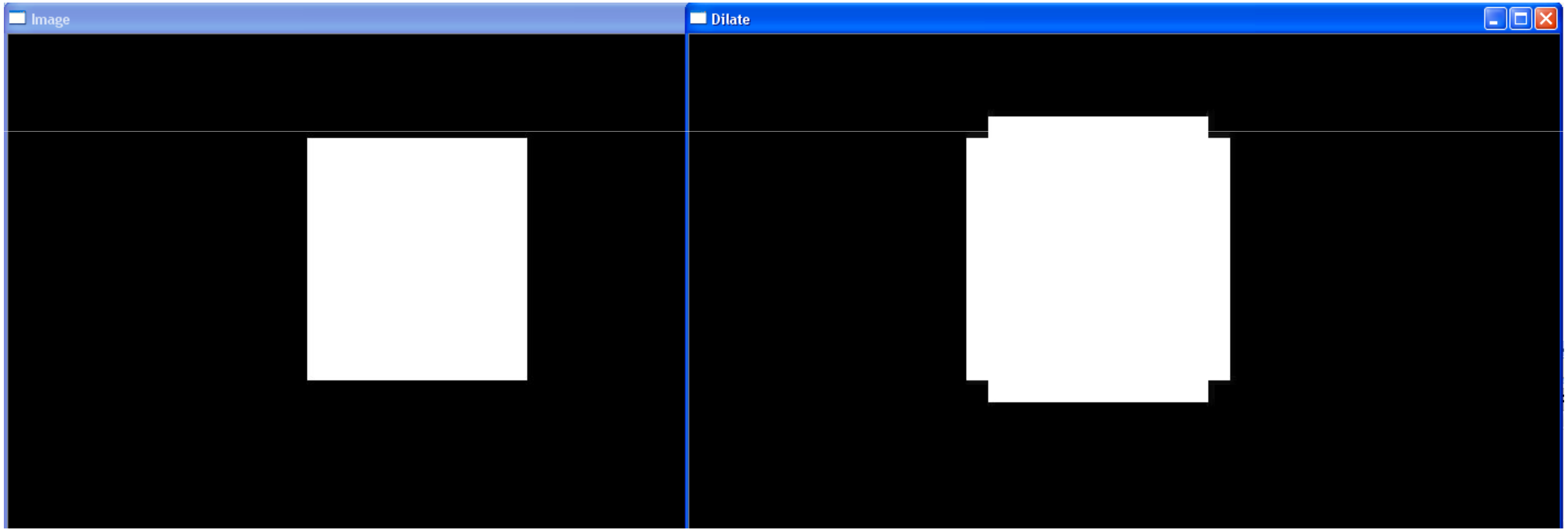
```
IplConvKernel* disk_21 = cvCreateStructuringElementEx(41, 41, 20, 20, CV_SHAPE_ELLIPSE);
cvDilate( img2, img3, disk_21, 1);
```



Morphology

EXAMPLE

```
IplConvKernel* disk_21 = cvCreateStructuringElementEx(41, 41, 20, 20, CV_SHAPE_CROSS);
cvDilate( img2, img3, disk_21, 1);
```



Practice 2/3

Write a program which:

- loads one of the binary image of the previous program (th_binary.jpg);
- shows the image on a window;
- apply “erode” and “dilate” operation for cleaning noise and connecting the object related to the flower;
- New binary image as “flower_binary.jpg”

TIP: use a 3x3 disk kernel and change the number of iterations

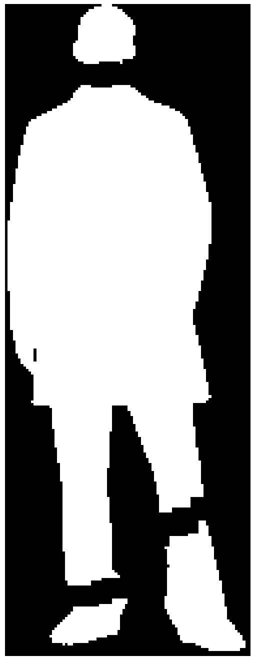
Blob analysis

So far we've learned:

- To smooth the image for reducing noise
- To threshold the image for segmenting the pixels of interest
- To erode the obtained binary image for deleting small objects due to residual noise
- To dilate the image to connect close object

BUT

We can still have binary images like this one:



So we need a method to label blobs belonging to the same object;

to get measures like area, perimeter…

Blob analysis

- OpenCV doesn't have any function for connecting and grouping pixels (blob).
- However, since lots of people work with OpenCV, the web is rich of useful libraries and codes to accomplish tasks like that.
- For example, *rickypetit1979* developped **CvBlobsLib** library, which is an OpenCV extension to find and manage connected components in binary images.
- You can find CvBlobsLib v8.3 in poliformat. Follow the instructions in the pdf file to add the library to your project.

Blob analysis

Relevant functions:

- blob type

```
CBlobResult blobs;
```

- object that will contain blobs of inputImage

```
blobs = CBlobResult( image, NULL, 0);
```

- get i^{th} blob

```
currentBlob = blobs.GetBlob(i);
```

- get blob pixels
  TIP: you need to create a black (0) image with white (255) pixels corresponding to the blob and then scan the obtained binary image getting white-pixels coordinates ☺

Blob analysis

- discard the blobs with less area than 5000 pixels (the criteria to filter can be any class derived from COperadorBlob)

```
blobs.Filter( blobs, B_INCLUDE, CBlobGetArea(), B_GREATER, 5000);
blobs.Filter( blobs, B_EXCLUDE, CBlobGetArea(), B_LESS, 5000);
```

- get the blob with biggest perimeter

```
blobs.GetNthBlob( CBlobGetPerimeter(), 0, blobWithBiggestPerimeter );
```

- get the blob with less area

```
blobs.GetNthBlob( CBlobGetArea(), blobs.GetNumBlobs() - 1, blobWithLessArea );
```

Blob analysis

- build an output image equal to the input but with 3 channels (to draw the coloured blobs)

```
IplImage *outputImage;
outputImage = cvCreateImage( cvSize( inputImage->width,
inputImage->height ), IPL_DEPTH_8U, 3 );
cvMerge( inputImage, inputImage, inputImage, NULL,
outputImage );
```

- plot the selected blobs in a output image

```
blobWithBiggestPerimeter.FillBlob( outputImage, CV_RGB( 255,
0, 0 ));
blobWithLessArea.FillBlob( outputImage, CV_RGB( 0, 255, 0 ));
```

Practice 2/4

1. Add the cvblobslib_OpenCV_v8_3 library to your project and try the following example: …\cvblobslib_OpenCV_v8_3\testBlobs\main.cpp (copying and pastying into your project, if you prefer).

- Which intensity value is the background? How can you easily find it?
- Which size does the triangle have?
- In A is the position of the second trackbar, change the code so that areas <u>higher</u> than A are black colored

Practice 2/5

- Read and show the image "flower_binary.jpg" from practice 2/3
- Apply the blob analysis
- How many object can you label?

TIP:

- Use CV_THRESH_BINARY_INV for thresholding the image since CBlobResult looks for dark objects.

Template Matching

- **OpenCV**

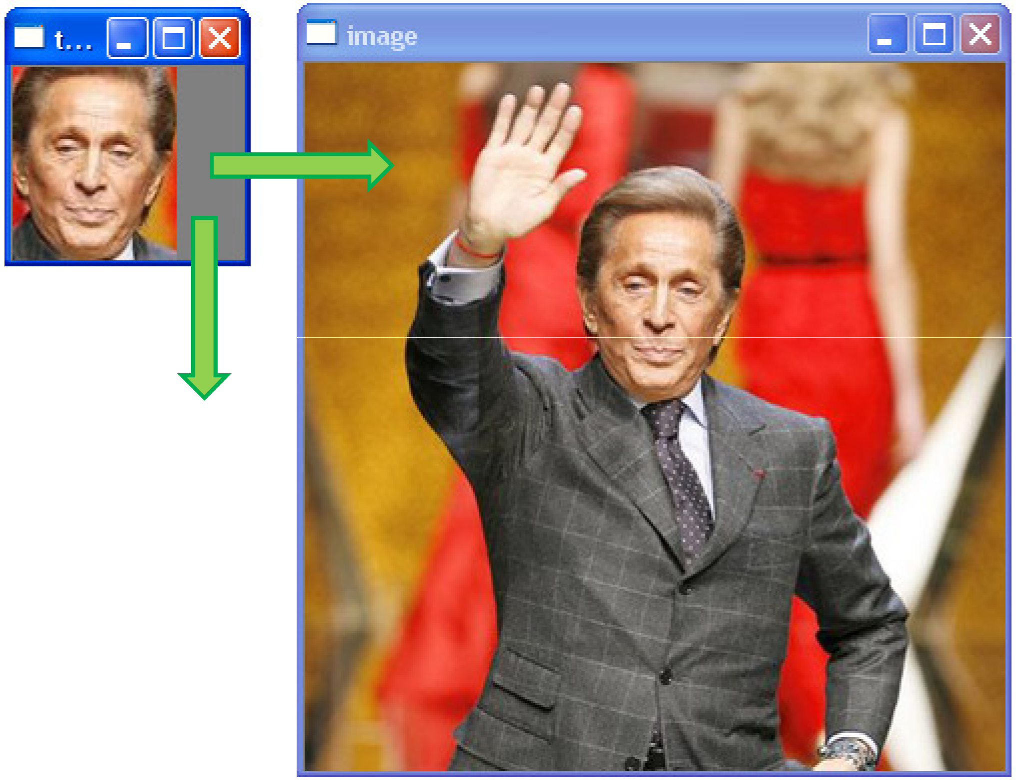
```
void cvMatchTemplate(
    const CvArr* image,
    const CvArr* templ,
    CvArr*       result,
    int          method
);
```

`cvMatchTemplate` matches an actual image patch against an input image by “sliding” the patch over the input image using one of the matching methods described in this section.

- `image`: single 8-bit or floating-point plane or color image (input.)
- `templ`: patch from a similar image containing the object for which you are searching
- `result`: single-channel byte or floating-point image of size

  images->width – patch_size.x + 1, images->height – patch_size.y + 1

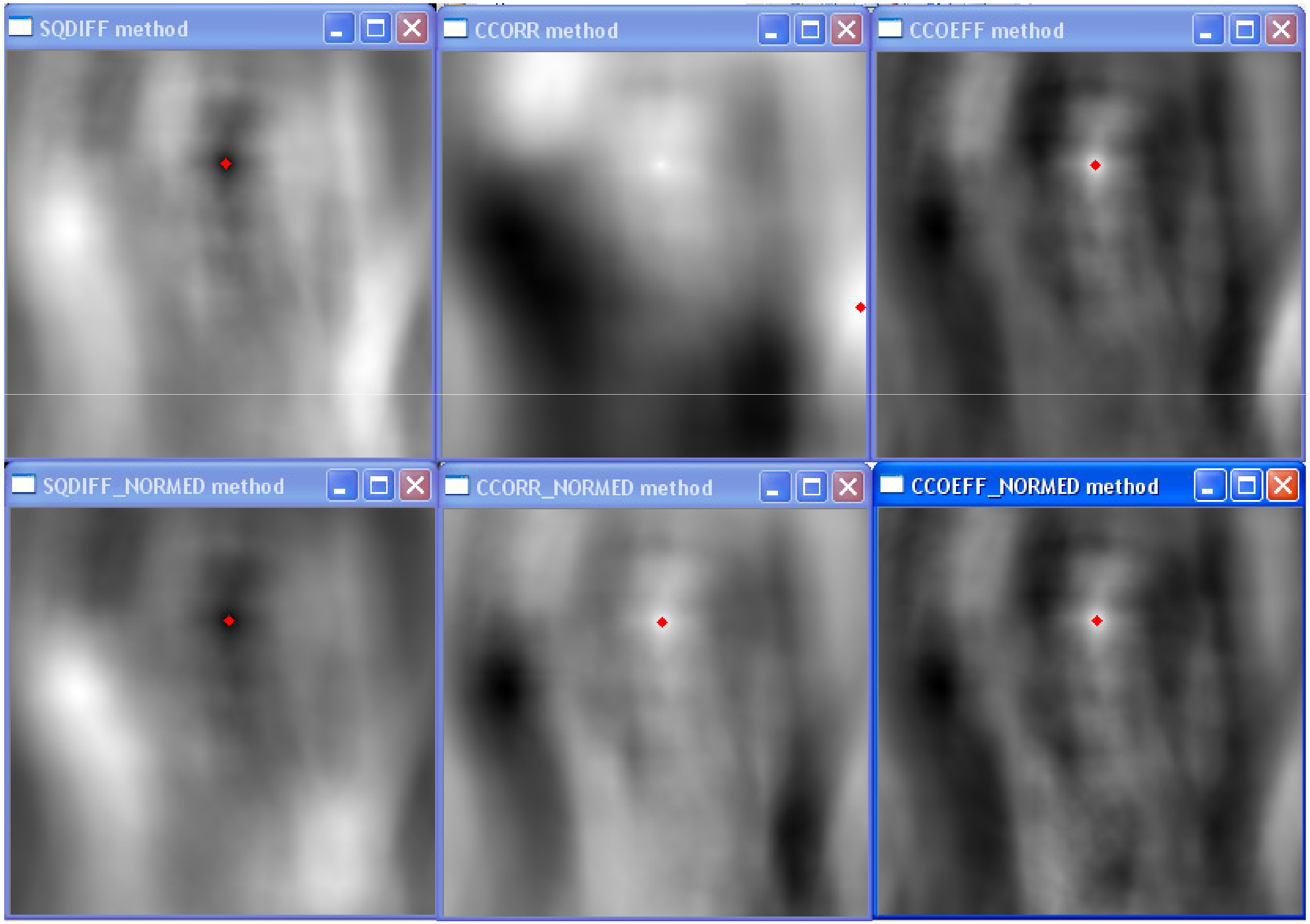
Example 8: Template matching



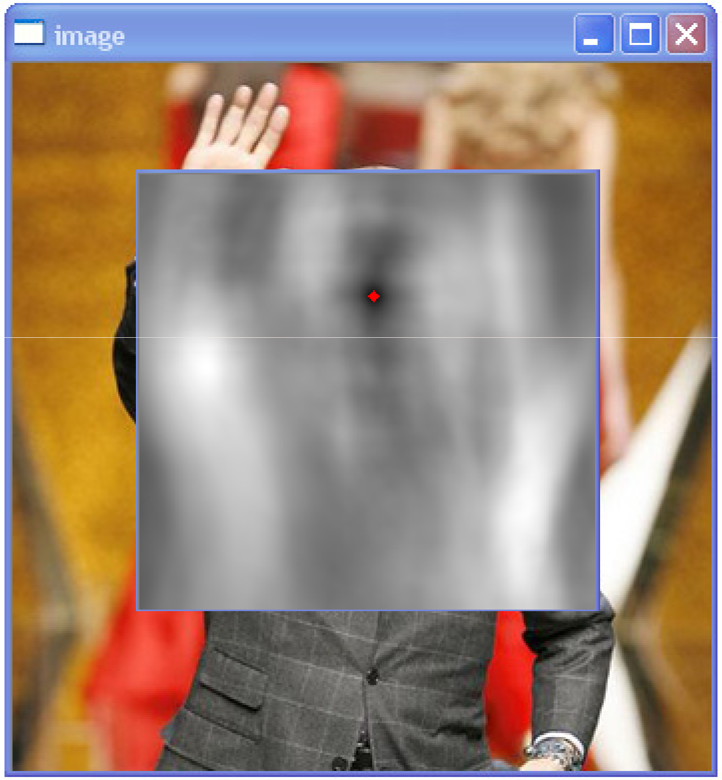
Example 8: Template matching

```
#include "cv.h"
#include "highgui.h"

int main()
{
    char* path_image="valentino.jpg";
    char* path_templ="valentino_face.jpg";
    IplImage* image = cvLoadImage(path_image);
    IplImage* templ = cvLoadImage(path_templ);
    IplImage* result_RGB[6];
    IplImage* result[6]; //array of 6 images -> results

    //allocate output images
    int iwidth = image->width - templ->width + 1;
    int iheight = image->height - templ->height + 1;
    for(int i=0; i<6; ++i){
        result[i] = cvCreateImage(cvSize(iwidth,iheight),32,1);
        result_RGB[i] = cvCreateImage(cvSize(iwidth,iheight),32,3);
    }

    //template matching with 6 methods
    for(int i=0; i<6; ++i){
        cvMatchTemplate( image, templ, result[i], i);
        cvNormalize(result[i], result[i], 0, 1, CV_MINMAX);

        cvMinMaxLoc(result[i], &minVal, &maxVal, &minLoc, &maxLoc, NULL);

        cvCvtColor(result[i] , result_RGB[i], CV_GRAY2RGB);
        cvCircle(result_RGB[i], maxLoc, 1, CV_RGB(255, 0, 0), 3);
    }
```

Example 8: Template matching



Example 8: Template matching



Practice 2/6

- Load image "erasmusip.jpg"
- Try template matching algorithms with your own templates…

Who do you want to detect? ☺

The projects…