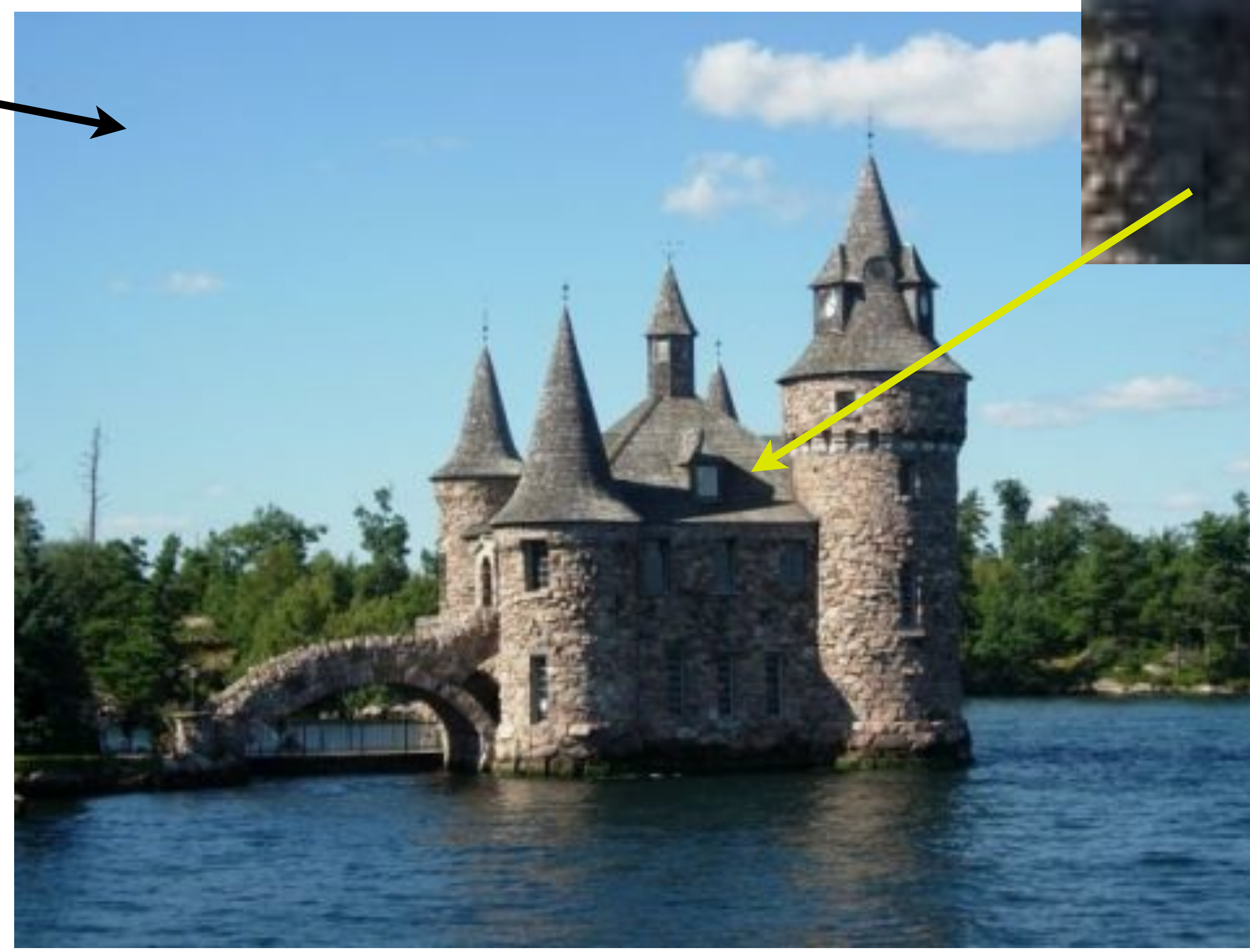
6

Filtering the Images

- Filtering images using low-pass filters
- Filtering images using a median filter
- Applying directional filters to detect edges
- Computing the Laplacian of an image

Frequency domain representation of the image

small variation
=
low frequency



large variation
=
high frequency

Frequency domain representation of the image

Low pass filter
= Low frequency pass filter
= image spacial smoothing

Image Smoothing = Low pass filtering

$$I'(x) = \frac{1}{3}I(x-1) + \frac{1}{3}I(x) + \frac{1}{3}I(x+1)$$

$$I' = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} * I$$

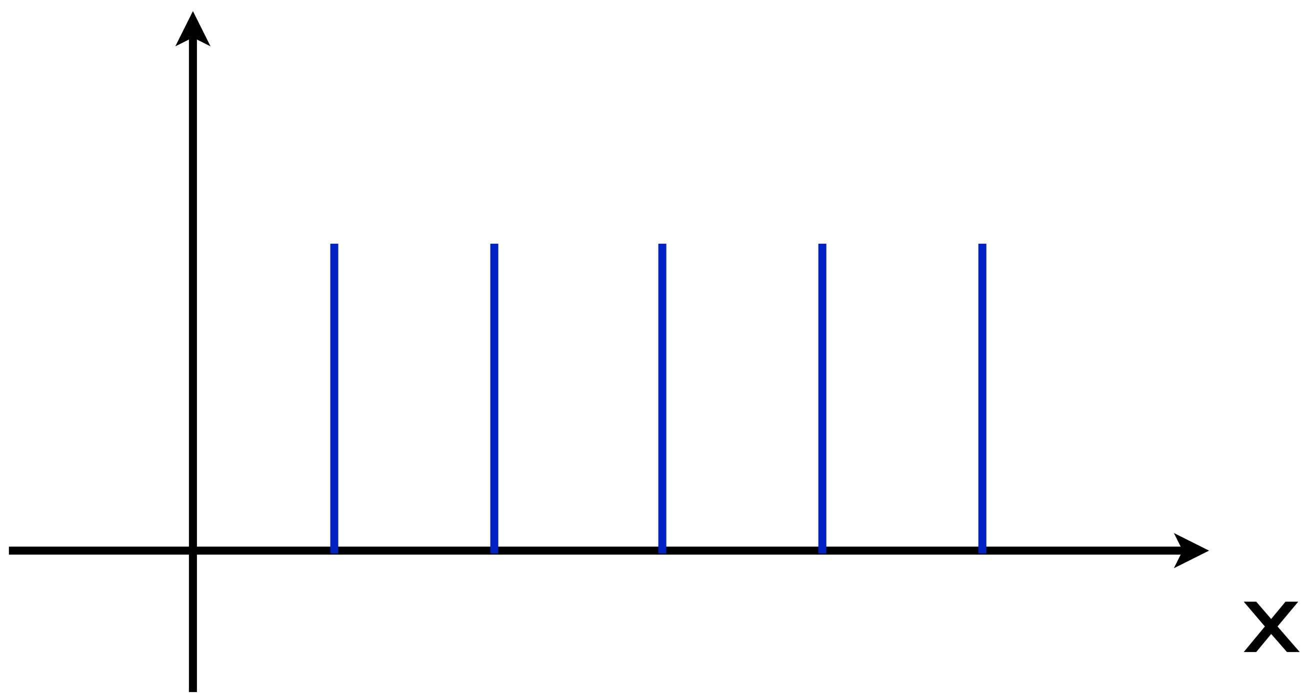
- Smoothing in the x-direction
- No change in the constant intensity area
- Big change on the edge (intensity discontinuity)
- Some simple examples

How to program in C/C++ with OpenCV

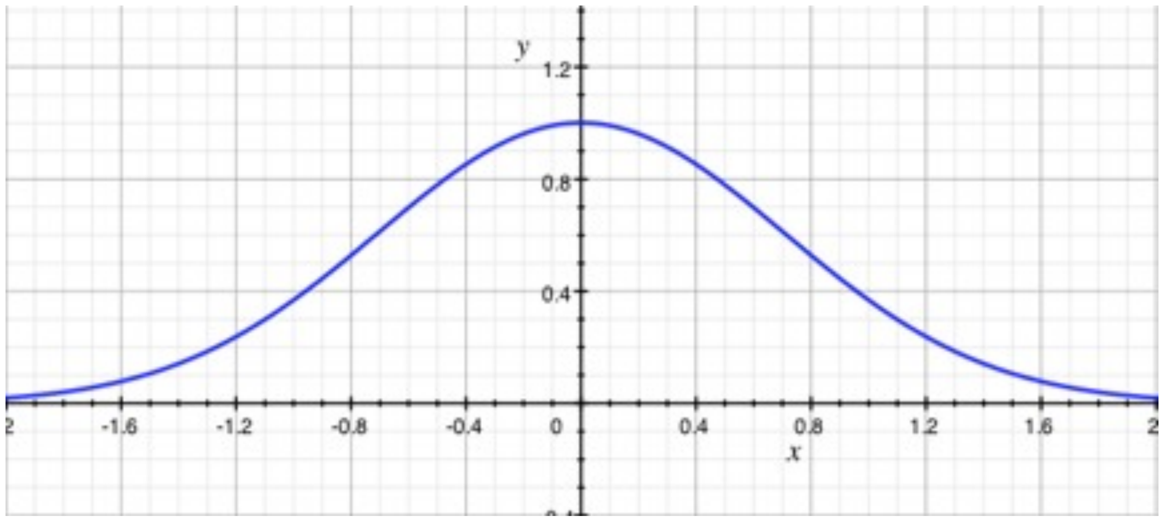
```
    for (int r=0; r<I.rows; r++)
        {
        uchar *ptr = I.ptr<uchar>(r);
        for (int c=1; c<I.cols-1; c++)
            J.at<uchar>(r,c) = (uchar)(ptr[c-1]/3.0+ptr[c]/3.0+ptr[c+1]/3.0);
        }
```

- The kernel size may be increased.
  - The sum of the kernel is 1. Why?
- The smoothing direction may be changed.

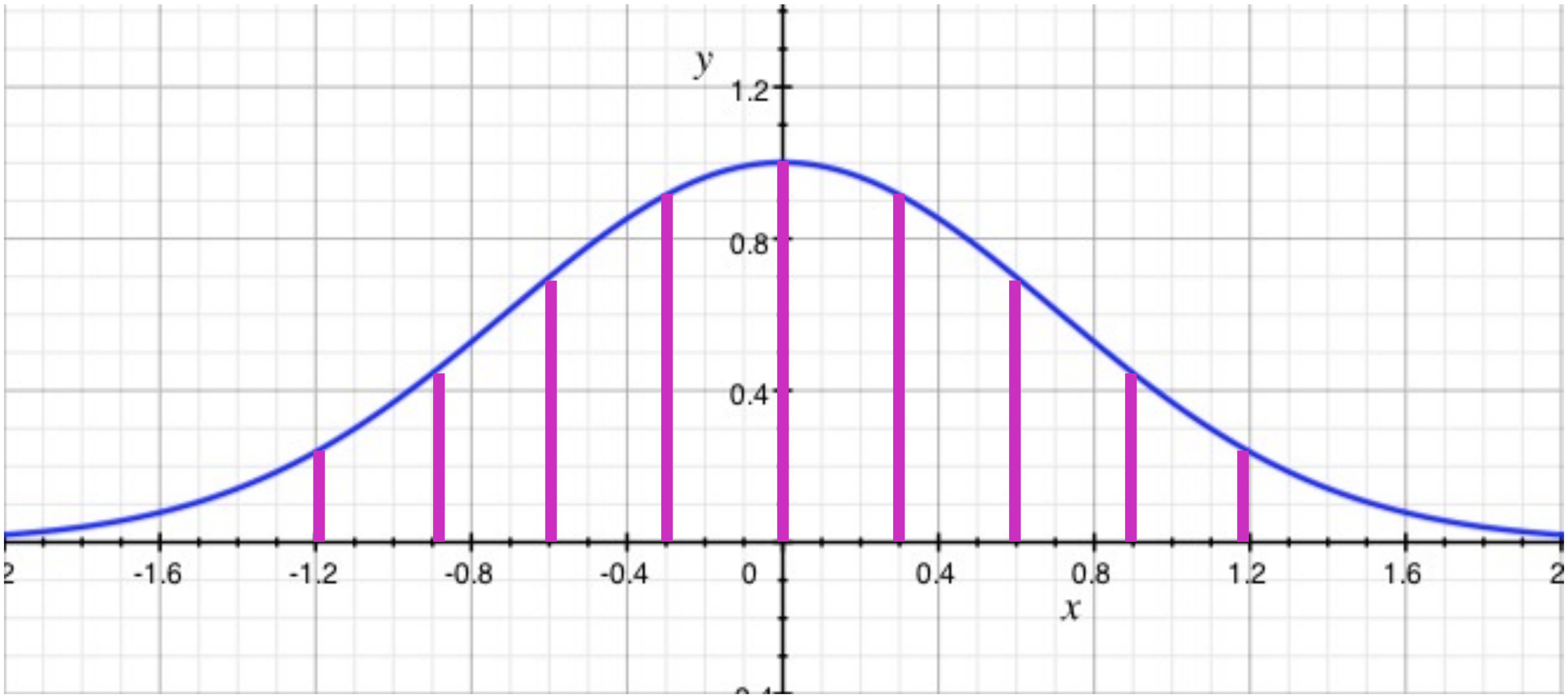
The shape of the filter kernel looks like a box.



- This filter is called a box filter (average/mean filter)
- Triangle filter.
- Gaussian filter

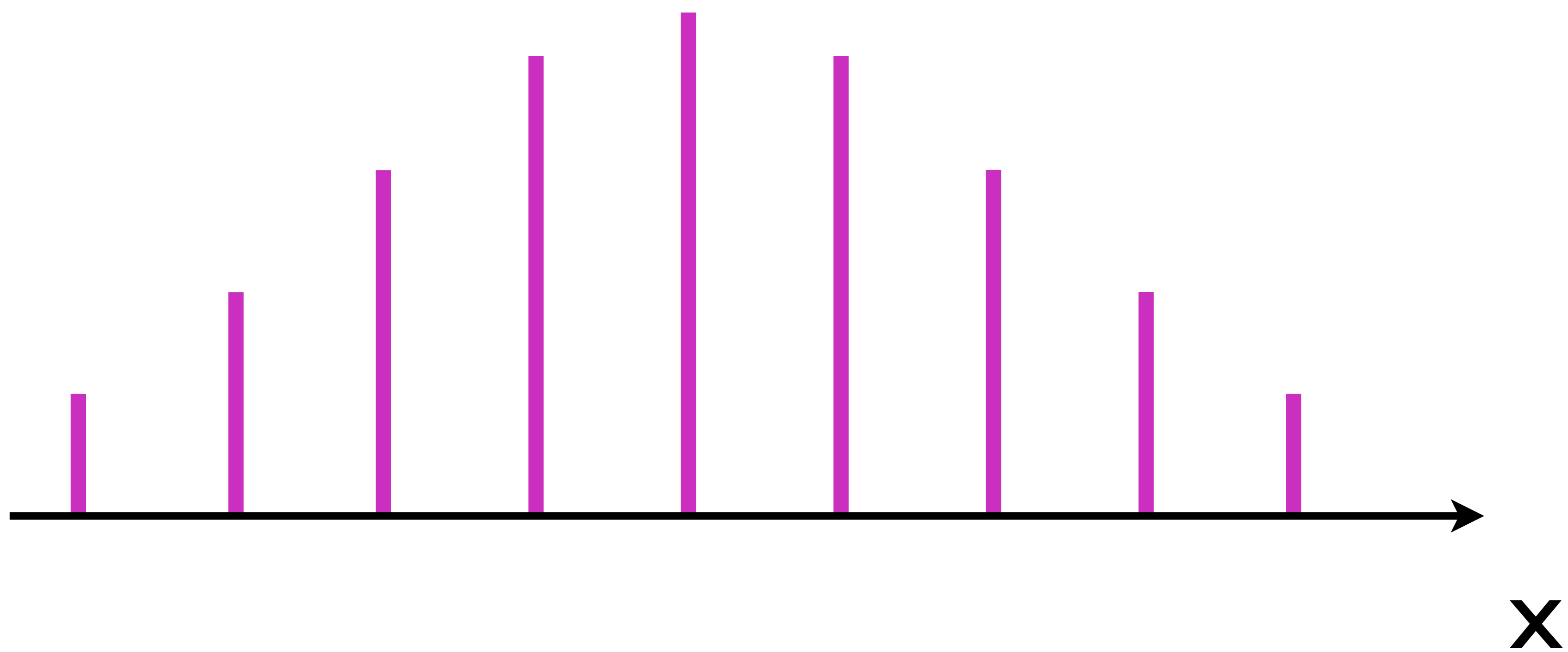


Gaussian Filter



$$G(x, \sigma) = A \exp\left\{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2\right\}$$

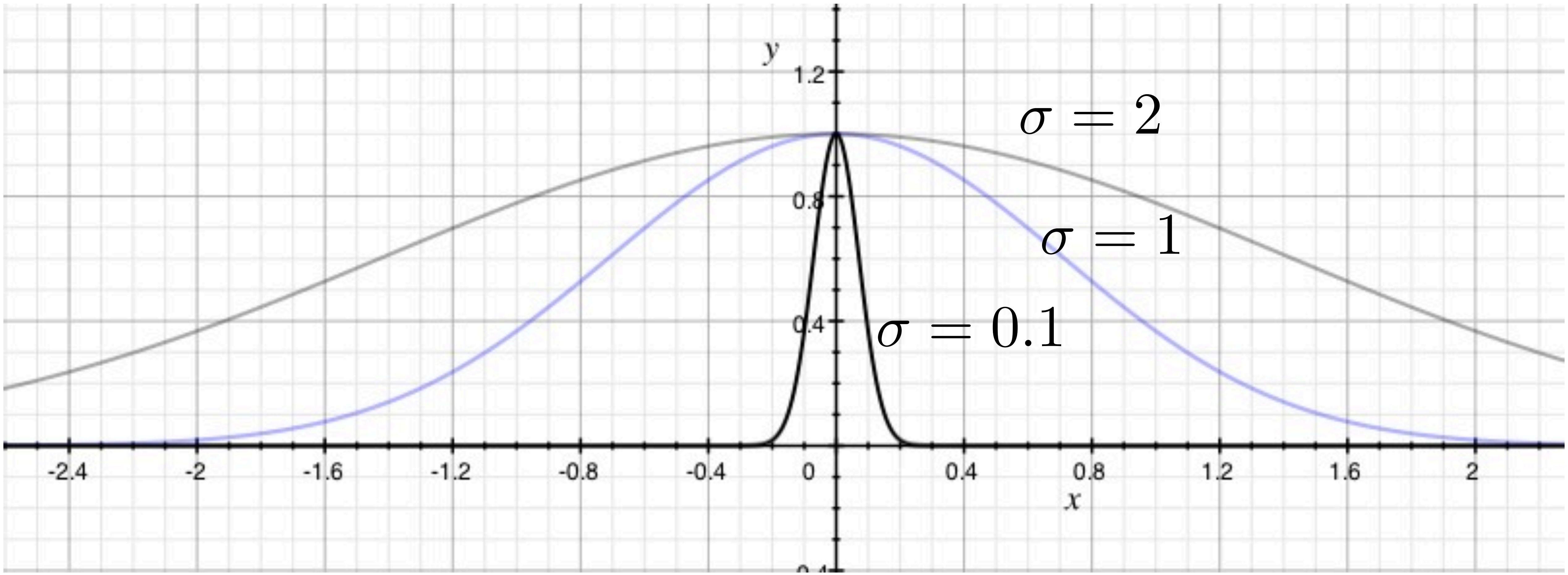
Gaussian Filter



$$G(x,\sigma) = A\exp\left\{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2\right\}$$

Gaussian Filter and Sigma

$$\exp\left\{-\left(\frac{x}{\sigma}\right)^2\right\}$$



For example, if we compute the coefficients of the 1D Gaussian filter for the interval `[-4,…,0,…4]` with σ`=0.5`, we obtain:

```
[0.0 0.0 0.00026 0.10645 0.78657 0.10645 0.00026 0.0 0.0]
```

While for σ`=1.5` these coefficients are:

```
[0.00761 0.036075 0.10959 0.21345 0.26666
 0.21345 0.10959 0.03608 0.00761 ]
```

Note that these values were obtained by calling the `cv::getGaussianKernel` function with the appropriate σ value:

```
cv::Mat gauss= cv::getGaussianKernel(9,sigma,CV_32F);
```

getGaussianKernel

Returns Gaussian filter coefficients.

C++: Mat **getGaussianKernel**(int **ksize**, double **sigma**, int **ktype**=CV_64F)

Parameters:

1. **ksize** – Aperture size. It should be odd ( ) and positive.
2. **sigma** – Gaussian standard deviation. If it is non-positive, it is computed from `ksize` as `sigma = 0.3*((ksize-1)*0.5 - 1)+ 0.8` .
3. **ktype** – Type of filter coefficients. It can be `CV_32f` or `CV_64F` .

The function computes and returns the matrix of Gaussian filter coefficients:

$$G_i = \alpha * e^{-(i-(\mathtt{ksize}-1)/2)^2/(2*\mathtt{sigma})^2},$$

$$i = 0..\mathtt{ksize} - 1$$

Two of such generated kernels can be passed to **sepFilter2D()** or to **createSeparableLinearFilter()**. Those functions automatically recognize smoothing kernels (a symmetrical kernel with sum of weights equal to 1) and handle them accordingly. You may also use the higher-level **GaussianBlur()**.

See also **sepFilter2D()**, **createSeparableLinearFilter()**,**getDerivKernels()**, **getStructuringElement()**, **GaussianBlur()**

Blurring & Gaussian Smoothing

```
cv::Mat image= cv::imread("../images/boldt.jpg",0);
if (!image.data) return 0;

 // Display the image
cv::namedWindow("Original Image");
cv::imshow("Original Image",image);

// Blur the image
cv::Mat result;
cv::GaussianBlur(image,result,cv::Size(5,5),1.5);

 // Display the blurred image
cv::namedWindow("Gaussian filtered Image");
cv::imshow("Gaussian filtered Image",result);

// Get the gaussian kernel (1.5)
cv::Mat gauss= cv::getGaussianKernel(9,1.5,CV_32F);

// Display kernel values
std::cout << "GaussianKernel(9,15)=[";
for (int i=0; i<gauss.rows; i++) {
  std::cout << gauss.at<float>(i) << " ";
}
std::cout << "]" << std::endl;
```

Filter: Separable or Non-Separable

Separable: 2D filtering = two 1D filtering

Non-Separable: 2D filtering = 2D filtering, no other option!

Median filtering

In probability theory and statistics, a **median** is described as the numerical value separating the higher half of a sample, a population, or a probability distribution, from the lower half.

To demonstrate, using a window size of three with one entry immediately preceding and following each entry, a median filter will be applied to the following simple 1D signal:

x = [2 80 6 3]

So, the median filtered output signal y will be:
y[1] = Median[2 2 80] = 2
y[2] = Median[2 80 6] = Median[2 6 80] = 6
y[3] = Median[80 6 3] = Median[3 6 80] = 6
y[4] = Median[6 3 3] = Median[3 3 6] = 3

i.e. y = [2 6 6 3].

http://en.wikipedia.org/wiki/Median_filter

How to do it...

The call to the median filtering function is done in a way similar to the other filters:

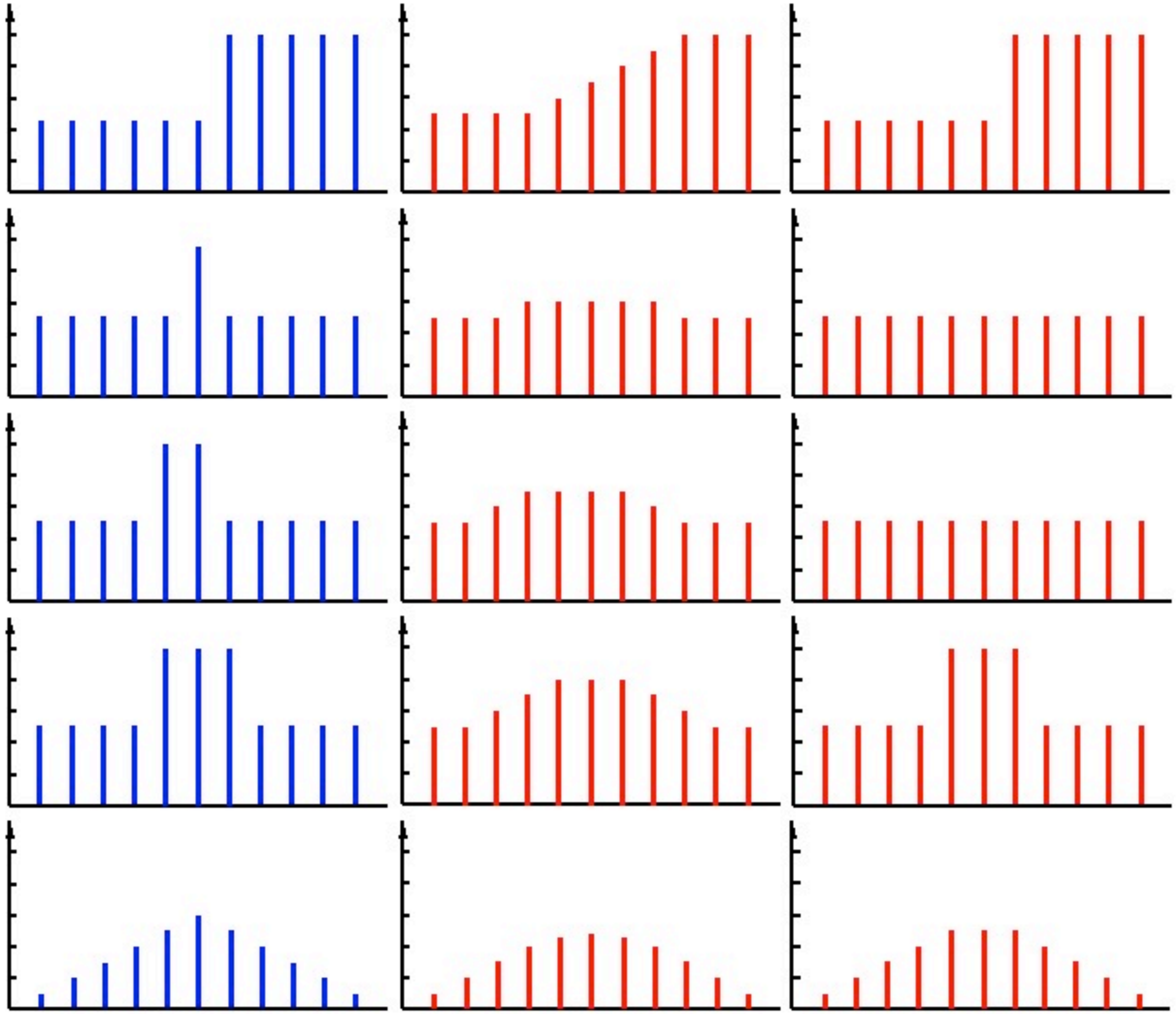
```
cv::medianBlur(image,result,5);
```

What is this value?

Pseudo-code for median filtering

```
For each pixel (x,y) in Image
{
 W := collect_values_in_the_window(s);
 W := sort (W);
 median := W(s/2);
 J(x,y) = median;
}
```

src mean(5) median(5)



http://fourier.eng.hmc.edu/e161/lectures/smooth_sharpen/node3.html

High pass filtering = edge finding = image enhancement

Computing a gradient: $G(x) = \frac{1}{2}(I(x+1) - I(x-1))$

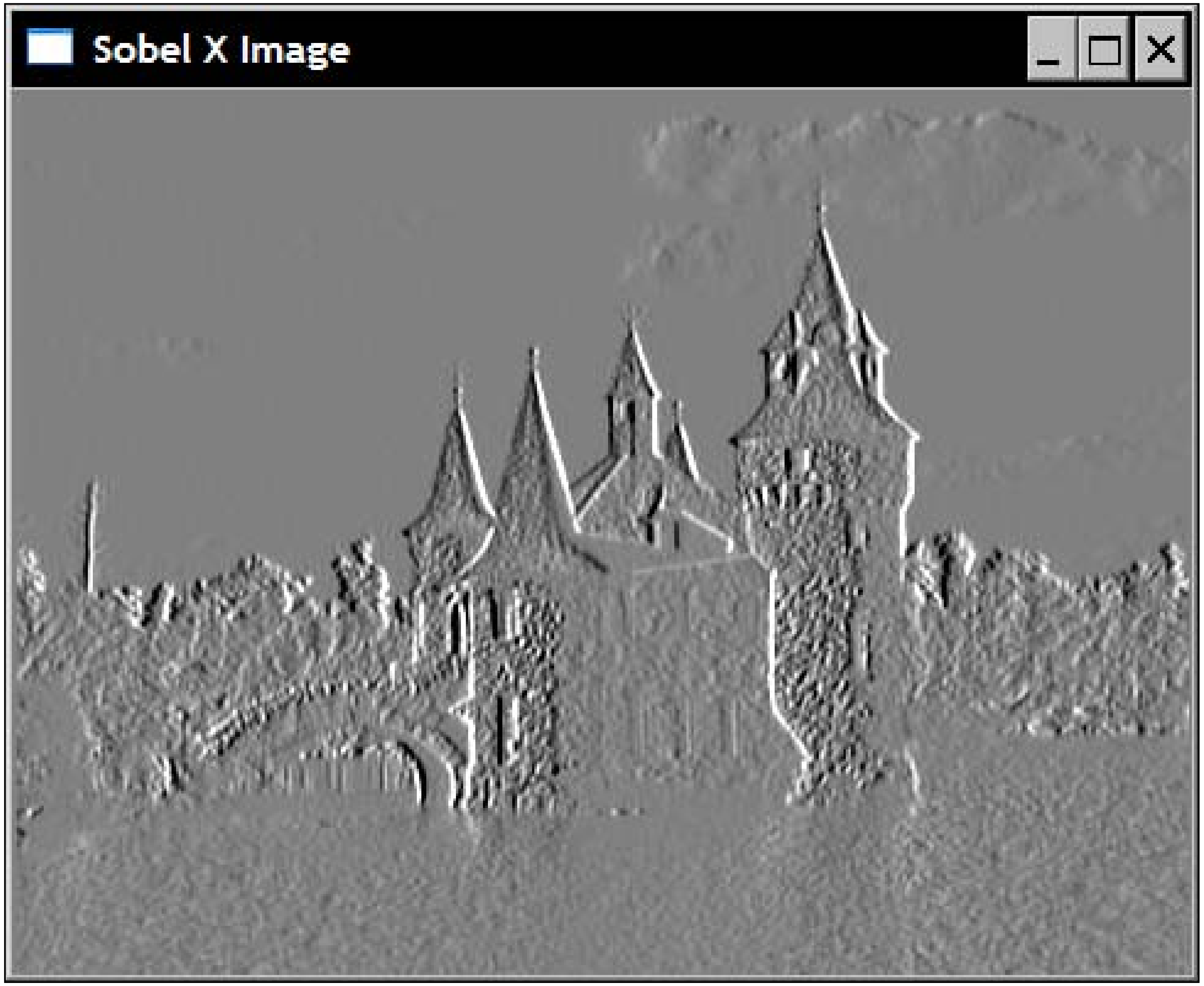
$$G_x(x,y) = \frac{1}{2}(I(x+1,y) - I(x-1,y))$$

$$G_y(x,y) = \frac{1}{2}(I(x,y+1) - I(x,y-1))$$

conversion to magnitude/angle

$$\begin{aligned} M(x,y) &= \|[G_x, G_y]\| \\ D(x,y) &= \angle[G_x, G_y] \end{aligned}$$

The result of the horizontal Sobel operator is as follows:




```
cv::Sobel(image,sobelX,CV_8U,1,0,3,0.4,128);
```

Sobel operators: a digital approximation of the gradient.

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

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

$$grad\,(I) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]^{\mathbf{T}}$$

```
cv::Sobel(image,  // input
          sobel,  // output
          image_depth,   // image type
          xorder,yorder, // kernel specification
          kernel_size,   // size of the square kernel
          alpha, beta);  // scale and offset
```

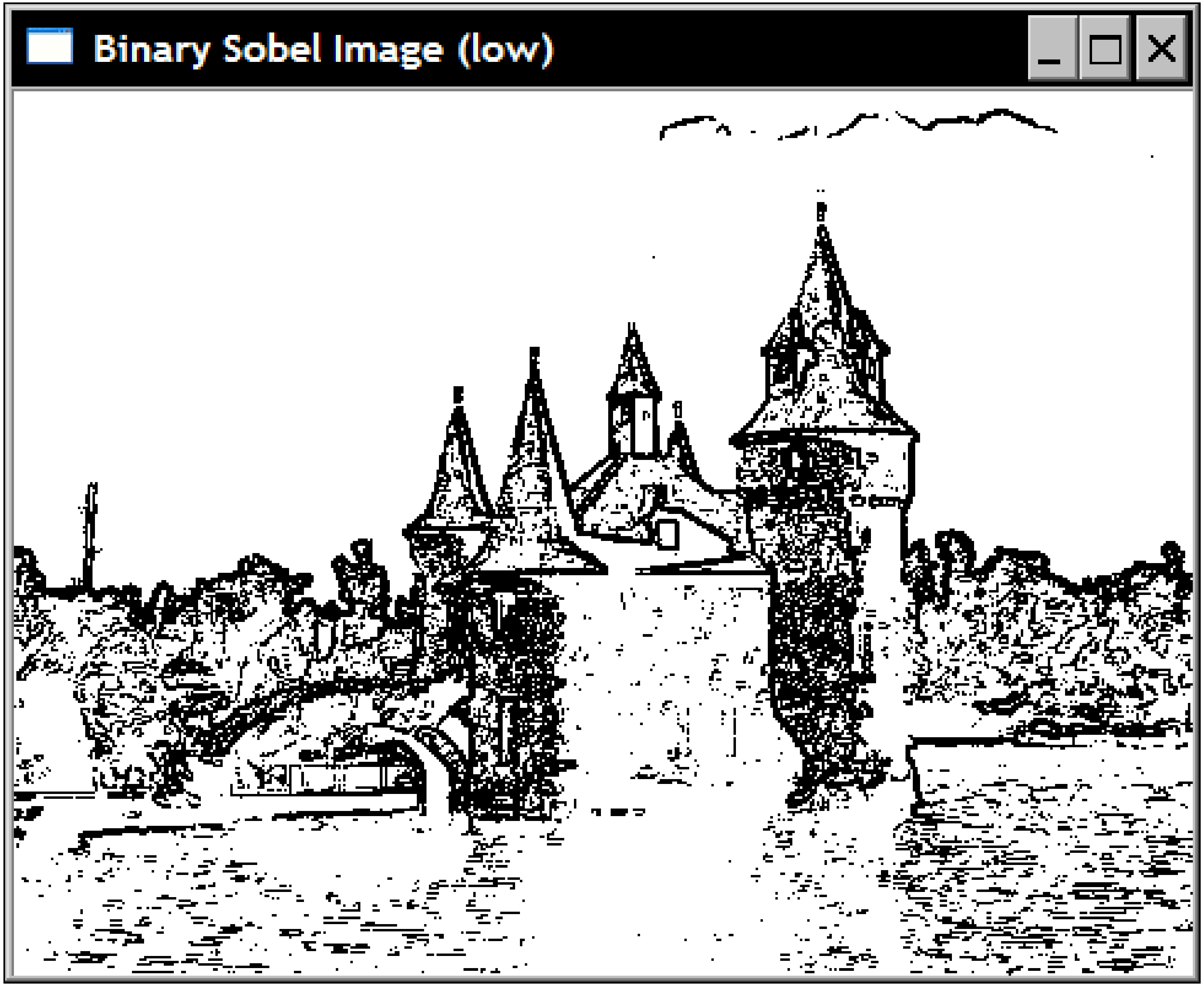
C++: void **Sobel** (InputArray **src**, OutputArray **dst**, int **ddepth**, int **xorder**, int **yorder**, int **ksize**=3, double **scale**=1, double **delta**=0, int **borderType**=BORDER_DEFAULT)

```
// Compute norm of Sobel
cv::Sobel(image,sobelX,CV_16S,1,0);
cv::Sobel(image,sobelY,CV_16S,0,1);
cv::Mat sobel;
//compute the L1 norm
sobel= abs(sobelX)+abs(sobelY);

// Find Sobel max value
double sobmin, sobmax;
cv::minMaxLoc(sobel,&sobmin,&sobmax);
// Conversion to 8-bit image
// sobelImage = -alpha*sobel + 255
cv::Mat sobelImage;
sobel.convertTo(sobelImage,CV_8U,-255./sobmax,255);
```


Sobel Image

```
cv::threshold(sobelImage, sobelThresholded,
                         threshold, 255, cv::THRESH_BINARY);
```



```
// Apply threshold to Sobel norm (low threshold value)
cv::Mat sobelThresholded, otsuThresholded;
cv::threshold(sobelImage, sobelThresholded, 225, 255, cv::THRESH_BINARY);

float otsuTh = cv::threshold(sobelImage, otsuThresholded, 225, 255, cv::THRESH_OTSU);
std::cerr << "Otsu_TH = " << otsuTh << std::endl;

// Display the image
cv::namedWindow("Binary Sobel Image (low)");
cv::imshow("Binary Sobel Image (low)",sobelThresholded);

cv::namedWindow("Otsu Output");
cv::imshow("Otsu Output",otsuThresholded);
```

Otsu_TH = 211