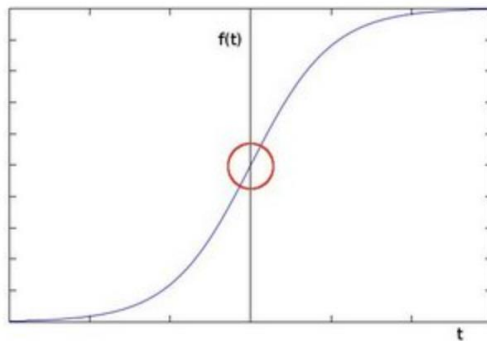


Edge Detection

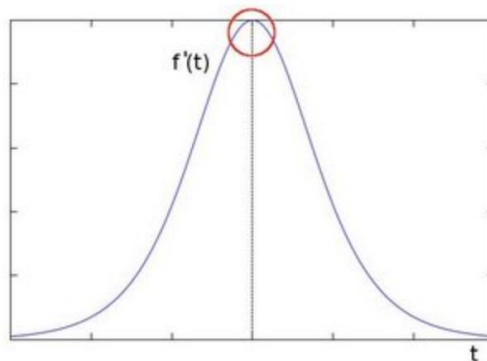
- What if we want to detect different objects regardless of colour?
- Edge detection is the process for finding structure and properties of the object (i.e. edges in an image).
- Edges are one of the most important features.
- Edges can be described by sudden changes in pixel intensity.
- Locations with extreme differences in brightness of pixels indicate an edge.
- We need to examine changes in the neighbouring pixels.
- In OpenCV, we have several options for edge detection.
- We will experiment with: **Sobel Edge Detection** and **Canny Edge Detection**
- **You can learn more about edge detection in the follow-up courses (Digital Image Processing and Image Analysis).**

Edge Detection

To be more graphical, let's assume we have a 1D-image. An edge is shown by the "jump" in intensity in the plot below:



The edge "jump" can be seen more easily if we take the first derivative (actually, here appears as a maximum)



Edge Detection

Assuming that the image to be operated is I :

1. We calculate two derivatives:

a. **Horizontal changes:** This is computed by convolving I with a kernel G_x with odd size. For example for a kernel size of 3, G_x would be computed as:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

a. **Vertical changes:** This is computed by convolving I with a kernel G_y with odd size. For example for a kernel size of 3, G_y would be computed as:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

2. At each point of the image we calculate an approximation of the *gradient* in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Although sometimes the following simpler equation is used:

$$G = |G_x| + |G_y|$$

Edge Detection

- Sobel edge detection
- We can use operation that is called convolution
- We need input image and kernel
- Multiply the image pixels by pixels of the filter, then sum the results
- In Sobel, we have two kernels



X – Direction Kernel

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

Y – Direction Kernel

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

Edge Detection

In Sobel, we have two kernels

X – Direction Kernel

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

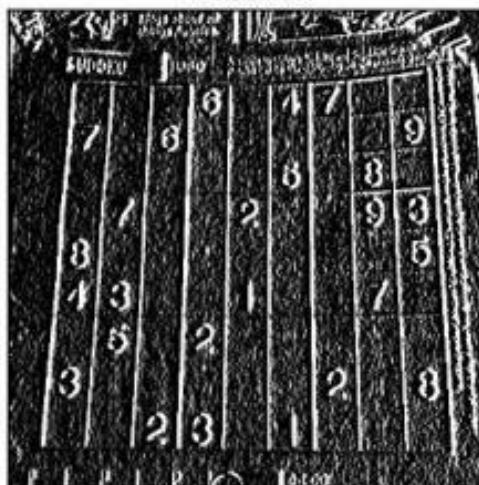
Y – Direction Kernel

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

Original



Sobel X



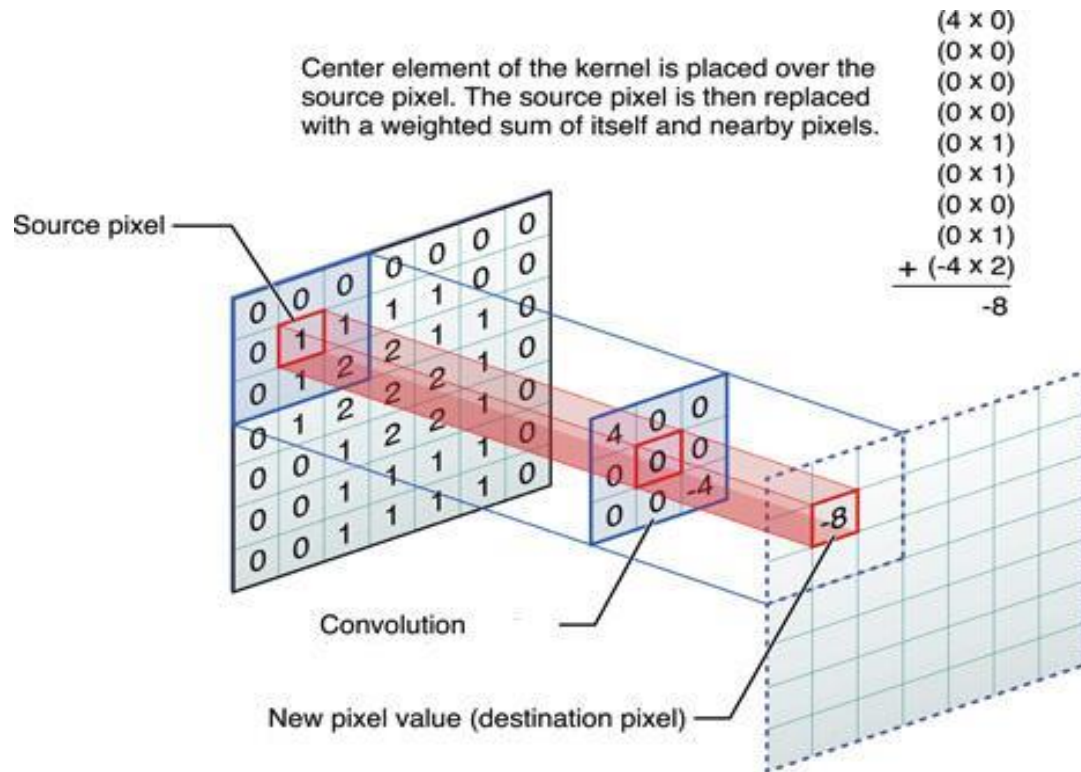
Sobel Y



Edge Detection

Simple (naive) explanation of convolution steps:

1. Center of the kernel is positioned over a specific pixel in an input image.
2. Each element in the kernel is multiplied with the corresponding pixel element in the input image.
3. Sum the result of multiplications
4. This result can be stored in our new image (edge map)



Edge Detection

Simple (naive) explanation of convolution steps:

1. Center of the kernel is positioned over a specific pixel in an input image.
2. Each element in the kernel is multiplied with the corresponding pixel element in the input image.
3. Sum the result of multiplications
4. This result can be stored in our new image (edge map)

100	100	200	200
100	100	200	200
100	100	200	200
100	100	200	200

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

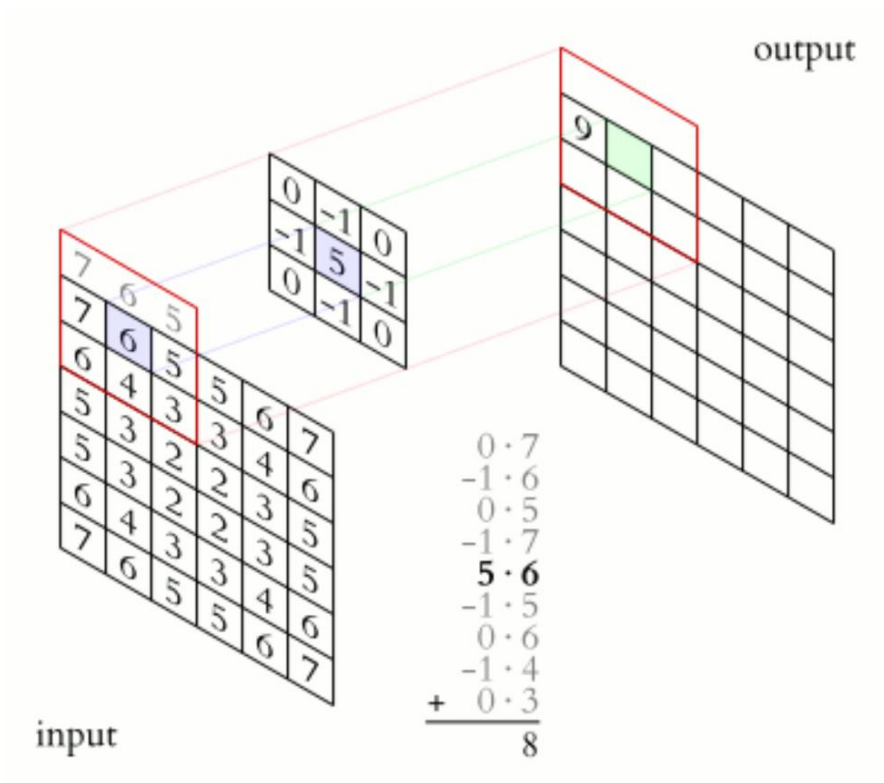
-100
-200
-100
200
400
<u>+200</u>
=400

Kernel Convolution: The bigger the value at the end, the more noticeable the edge will be.

Edge Detection

Simple (naive) explanation of convolution steps:

1. Center of the kernel is positioned over a specific pixel in an input image.
2. Each element in the kernel is multiplied with the corresponding pixel element in the input image.
3. Sum the result of multiplications
4. This result can be stored in our new image (edge map)



Edge Detection

```
img_input = cv.imread('Morphology_1_Tutorial_Original_Image.jpg', 0)
img_input = cv.medianBlur(img_input, 3)

sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1] ])

sobel_y = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1] ])

dx_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_x)
dy_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_y)

#dx_img = cv.Sobel(img_input, cv.CV_32FC1, 1, 0)
#dy_img = cv.Sobel(img_input, cv.CV_32FC1, 0, 1)

dx_img = cv.convertScaleAbs(dx_img)
dy_img = cv.convertScaleAbs(dy_img)

grad = cv.addWeighted(dx_img, 0.5, dy_img, 0.5, 0)

ret, Gm_th = cv.threshold(grad, 120, 255, cv.THRESH_BINARY)

cv.imshow("dx_img", dx_img)
cv.imshow("dy_img", dy_img)
cv.imshow("grad", grad)
cv.imshow("Gm_th", Gm_th)
cv.waitKey()
```

Edge Detection

```
img_input = cv.imread('Morphology_1_Tutorial_Original_Image.jpg', 0)
img_input = cv.medianBlur(img_input, 3)
```

```
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1] ])
```

```
sobel_y = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1] ])
```

```
dx_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_x)
dy_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_y)
```

```
#dx_img = cv.Sobel(img_input, cv.CV_32FC1, 1, 0)
#dy_img = cv.Sobel(img_input, cv.CV_32FC1, 0, 1)
```

```
dx_img = cv.convertScaleAbs(dx_img)
dy_img = cv.convertScaleAbs(dy_img)
```

```
grad = cv.addWeighted(dx_img, 0.5, dy_img, 0.5, 0)
```

```
ret, Gm_th = cv.threshold(grad, 120, 255, cv.THRESH_BINARY)
```

```
cv.imshow("dx_img", dx_img)
cv.imshow("dy_img", dy_img)
cv.imshow("grad", grad)
cv.imshow("Gm_th", Gm_th)
cv.waitKey()
```

filter2D()

```
void cv::filter2D ( InputArray  src,
                  OutputArray dst,
                  int          ddepth,
                  InputArray  kernel,
                  Point        anchor = Point(-1, -1) ,
                  double       delta = 0 ,
                  int          borderType = BORDER_DEFAULT
                )
```

Python:

```
cv.filter2D( src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]) -> dst
```

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

Convolves an image with the kernel.

The function applies an arbitrary linear filter to an image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

The function does actually compute correlation, not the convolution:

$$dst(x, y) = \sum_{\substack{0 \leq x' < \text{kernel.cols} \\ 0 \leq y' < \text{kernel.rows}}} kernel(x', y') * src(x + x' - \text{anchor.x}, y + y' - \text{anchor.y})$$

Edge Detection

```
img_input = cv.imread('Morphology_1_Tutorial_Original_Image.jpg', 0)
img_input = cv.medianBlur(img_input, 3)
```

```
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1] ])
```

```
sobel_y = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1] ])
```

```
dx_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_x)
dy_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_y)
```

Alternatively, you can use cv.Sobel()

```
#dx_img = cv.Sobel(img_input, cv.CV_32FC1, 1, 0)
#dy_img = cv.Sobel(img_input, cv.CV_32FC1, 0, 1)
```

```
dx_img = cv.convertScaleAbs(dx_img)
dy_img = cv.convertScaleAbs(dy_img)
```

```
grad = cv.addWeighted(dx_img, 0.5, dy_img, 0.5, 0)
```

```
ret, Gm_th = cv.threshold(grad, 120, 255, cv.THRESH_BINARY)
```

```
cv.imshow("dx_img", dx_img)
cv.imshow("dy_img", dy_img)
cv.imshow("grad", grad)
cv.imshow("Gm_th", Gm_th)
cv.waitKey()
```

◆ Sobel()

```
void cv::Sobel ( InputArray  src,
                OutputArray dst,
                int         ddepth,
                int         dx,
                int         dy,
                int         ksize = 3 ,
                double      scale = 1 ,
                double      delta = 0 ,
                int         borderType = BORDER_DEFAULT
              )
```

Python:

```
cv.Sobel( src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]] ] -> dst
```

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

Calculates the first, second, third, or mixed image derivatives using an extended Sobel operator.

The Sobel operators combine Gaussian smoothing and differentiation, so the result is more or less resistant to the noise. Most often, the function is called with (xorder = 1, yorder = 0, ksize = 3) or (xorder = 0, yorder = 1, ksize = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The second case corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Edge Detection

```
img_input = cv.imread('Morphology_1_Tutorial_Original_Image.jpg', 0)
img_input = cv.medianBlur(img_input, 3)
```

```
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1] ])
```

```
sobel_y = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1] ])
```

```
dx_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_x)
dy_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_y)
```

```
#dx_img = cv.Sobel(img_input, cv.CV_32FC1, 1, 0)
#dy_img = cv.Sobel(img_input, cv.CV_32FC1, 0, 1)
```

```
dx_img = cv.convertScaleAbs(dx_img)
dy_img = cv.convertScaleAbs(dy_img)
```

calculates absolute values, and converts the result to 8-bit.

```
grad = cv.addWeighted(dx_img, 0.5, dy_img, 0.5, 0)
```

```
ret, Gm_th = cv.threshold(grad, 120, 255, cv.THRESH_BINARY)
```

```
cv.imshow("dx_img", dx_img)
cv.imshow("dy_img", dy_img)
cv.imshow("grad", grad)
cv.imshow("Gm_th", Gm_th)
cv.waitKey()
```

Edge Detection

```
img_input = cv.imread('Morphology_1_Tutorial_Original_Image.jpg', 0)
img_input = cv.medianBlur(img_input, 3)
```

```
sobel_x = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1] ])
```

```
sobel_y = np.array([
    [-1, -2, -1],
    [ 0,  0,  0],
    [ 1,  2,  1] ])
```

```
dx_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_x)
dy_img = cv.filter2D(img_input, cv.CV_32FC1, sobel_y)
```

```
#dx_img = cv.Sobel(img_input, cv.CV_32FC1, 1, 0)
#dy_img = cv.Sobel(img_input, cv.CV_32FC1, 0, 1)
```

```
dx_img = cv.convertScaleAbs(dx_img)
dy_img = cv.convertScaleAbs(dy_img)
```

```
grad = cv.addWeighted(dx_img, 0.5, dy_img, 0.5, 0)
```

```
ret, Gm_th = cv.threshold(grad, 120, 255, cv.THRESH_BINARY)
```

```
cv.imshow("dx_img", dx_img)
cv.imshow("dy_img", dy_img)
cv.imshow("grad", grad)
cv.imshow("Gm_th", Gm_th)
cv.waitKey()
```

◆ addWeighted()

```
void cv::addWeighted ( InputArray  src1,
                     double      alpha,
                     InputArray  src2,
                     double      beta,
                     double      gamma,
                     OutputArray  dst,
                     int          dtype = -1
                     )
```

Python:

```
cv.addWeighted( src1, alpha, src2, beta, gamma[, dst[, dtype]] )-> dst
```

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



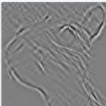




Calculates the weighted sum of two arrays.

The function `addWeighted` calculates the weighted sum of two arrays as follows:

$$dst(I) = \text{saturnate}(src1(I) * \alpha + src2(I) * \beta + \gamma)$$

We try to approximate the *gradient* by adding both directional gradients (note that this is not an exact calculation at all! but it is good for our purposes).

Depending on the element values, a kernel can cause a wide range of effects:

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 x 3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Gaussian blur 5 x 5 (approximation)	$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

<https://setosa.io/ev/image-kernels/>

Canny edge detection (the main steps in a simplified form):

- **Noise Reduction:** Smooth the image using a Gaussian filter to minimize noise and avoid false edges.
- **Gradient Calculation:** Compute intensity gradients using operators like Sobel to determine edge magnitude and direction.
- **Non-Maximum Suppression:** Thin edges by keeping only pixels that are local maxima along the gradient direction.
- **Double Thresholding and Hysteresis:** Classify edges using two thresholds (high and low), retaining strong edges and connecting weak edges only if linked to strong ones, discarding isolated weak edges.
- You can learn more about Canny edge detection in the follow-up courses (Digital Image Processing)

Canny Edge Detection in OpenCV

```
43 def edge_detection_canny():
44     in_mat = cv2.imread("test2.jpg")
45     in_mat_gray = cv2.cvtColor(in_mat, cv2.COLOR_BGR2GRAY)
46
47     # img, minVal, maxVal
48     canny_edges = cv2.Canny(in_mat_gray, 100, 200)
49
50     cv2.imshow('canny_edges', canny_edges)
51     cv2.imwrite("canny_edges_100_200.png", canny_edges)
52     cv2.waitKey(0)
```

- You can learn more about Canny edge detection in the follow-up course (Digital Image Processing)

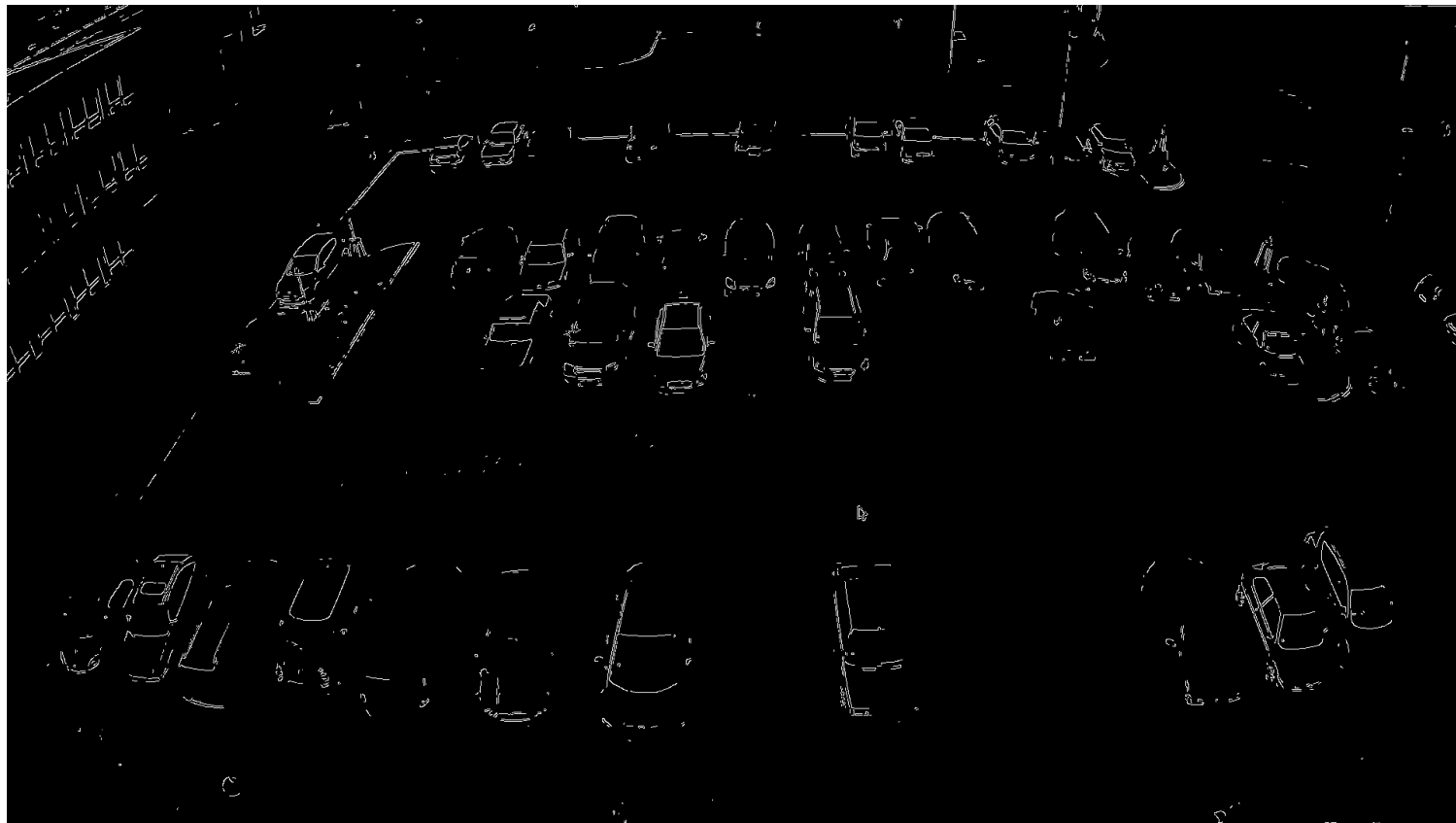
Canny Edge Detection

10, 100



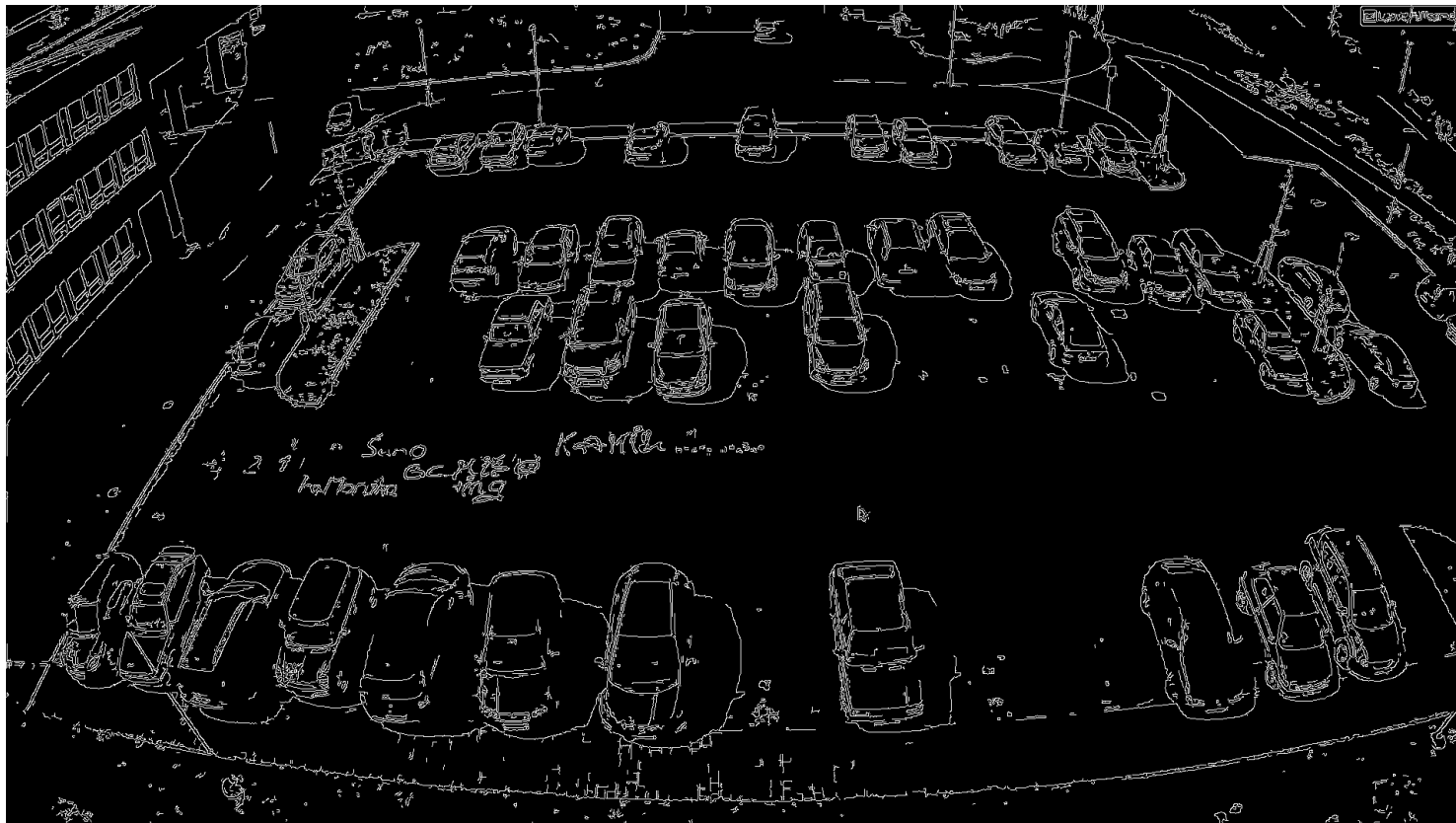
Canny Edge Detection

400, 500



Canny Edge Detection

100, 200



Canny Edge Detection

- **Thresholding - we need to set appropriate thresholds**
- At the top of the graph, we can see that **A** is a sure edge, since $A > T_{upper}$.
- **B** is also an edge, even though $B < T_{upper}$ since it is connected to a strong edge, **A**.
- **C** is not an edge since $C < T_{upper}$ and is not connected to a strong edge.
- Finally, **D** is not an edge since $D < T_{lower}$ and is automatically discarded.

Setting these threshold ranges is not always a trivial process.

If the threshold range is *too wide*, then we'll get many false edges instead of being about to find *just* the structure and outline of an object in an image.

Similarly, if the threshold range is *too tight*, we won't find many edges at all and could be at risk of missing the structure/outline of the object entirely!

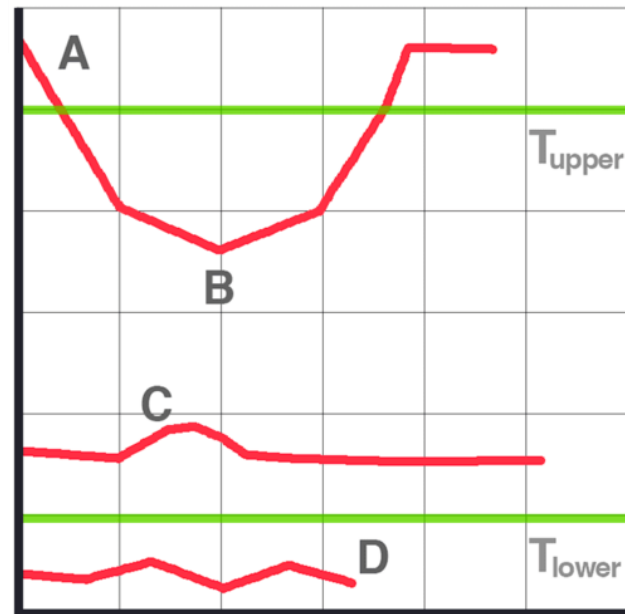


Image Filtering

2D Convolution (Image Filtering)

As in one-dimensional signals, images also can be filtered with various low-pass filters (LPF), high-pass filters (HPF), etc. LPF helps in removing noise, blurring images, etc. HPF filters help in finding edges in images.

OpenCV provides a function `cv.filter2D()` to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel will look like the below:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The operation works like this: keep this kernel above a pixel, add all the 25 pixels below this kernel, take the average, and replace the central pixel with the new average value. This operation is continued for all the pixels in the image. Try this code and check the result:

Image Filtering

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

img = cv.imread('opencv_logo.png')
assert img is not None, "file could not be read, check with os.path.exists()"

kernel = np.ones((5,5),np.float32)/25
dst = cv.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([], plt.yticks([]))
plt.show()
```

Result:

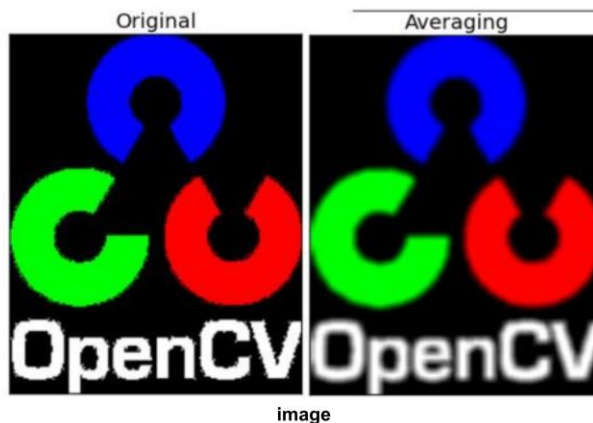


Image Filtering

Alternatively, you can use `cv.blur()`

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

img = cv.imread('opencv-logo-white.png')
assert img is not None, "file could not be read, check with os.path.exists()"

blur = cv.blur(img, (5,5))

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(blur),plt.title('Blurred')
plt.xticks([], plt.yticks([]))
plt.show()
```

◆ blur()

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

Python:

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

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

Blurs an image using the normalized box filter.

The function smooths an image using the kernel:

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

Gaussian Blurring

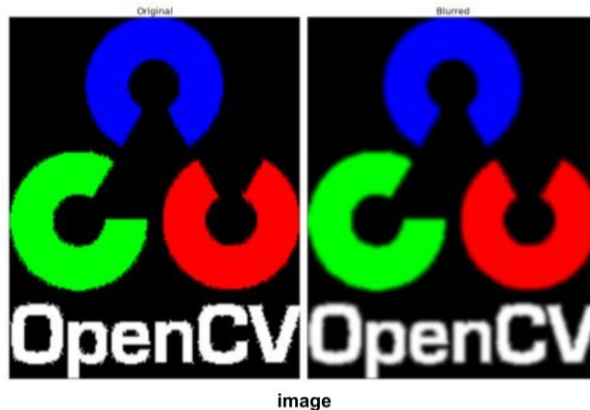
Gaussian blur works by convolving an image with a Gaussian kernel. Unlike box filter (`cv.blur`), which gives equal weight to all pixels in the neighborhood, Gaussian filter assigns greater weight to pixels closer to the center and less weight to more distant pixels.

If you want, you can create a Gaussian kernel with the function, `cv.getGaussianKernel()`.

The above code can be modified for Gaussian blurring:

```
blur = cv.GaussianBlur(img, (5,5), 0)
```

Result:



Median Blurring

Here, the function `cv.medianBlur()` takes the median of all the pixels under the kernel area and the central element is replaced with this median value. This is highly effective against salt-and-pepper noise in an image. Interestingly, in the above filters, the central element is a newly calculated value which may be a pixel value in the image or a new value. But in median blurring, the central element is always replaced by some pixel value in the image. It reduces the noise effectively. Its kernel size should be a positive odd integer.

In this demo, I added a 50% noise to our original image and applied median blurring. Check the result:

```
median = cv.medianBlur(img, 5)
```

Result:

