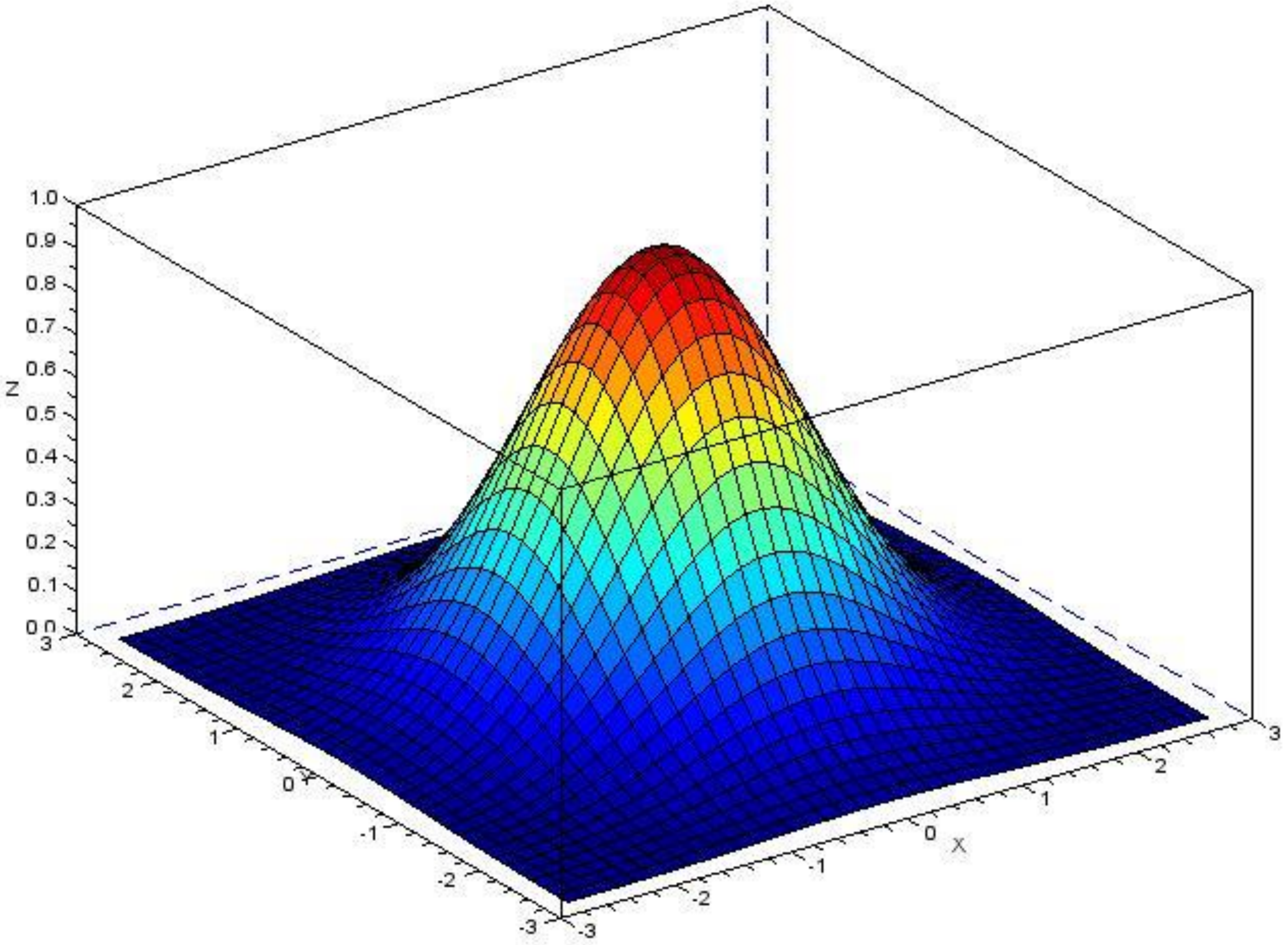


Image Manipulation & Processing

Afine transformations:

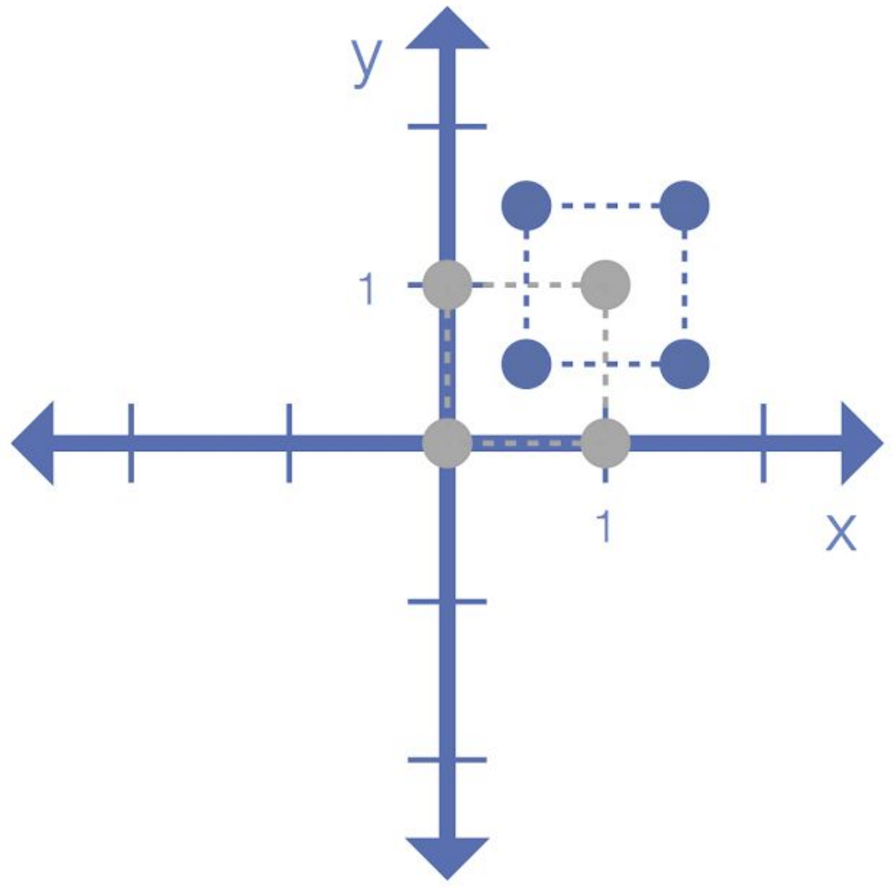
What is an Affine Transformation?

1. It is any transformation that can be expressed in the form of a *matrix multiplication* (linear transformation) followed by a *vector addition* (translation).
2. From the above, We can use an Affine Transformation to express:
   1. Rotations (linear transformation)
   2. Translations (vector addition)
   3. Scale operations (linear transformation)

Transformation matrices:

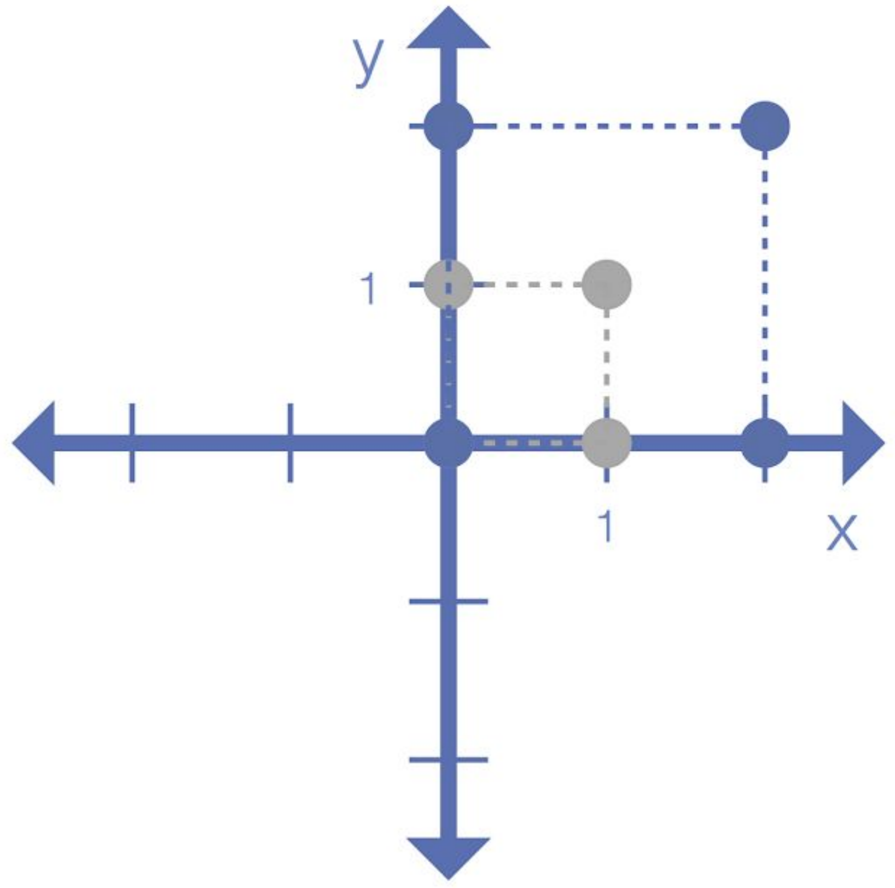
Translate

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



Scale

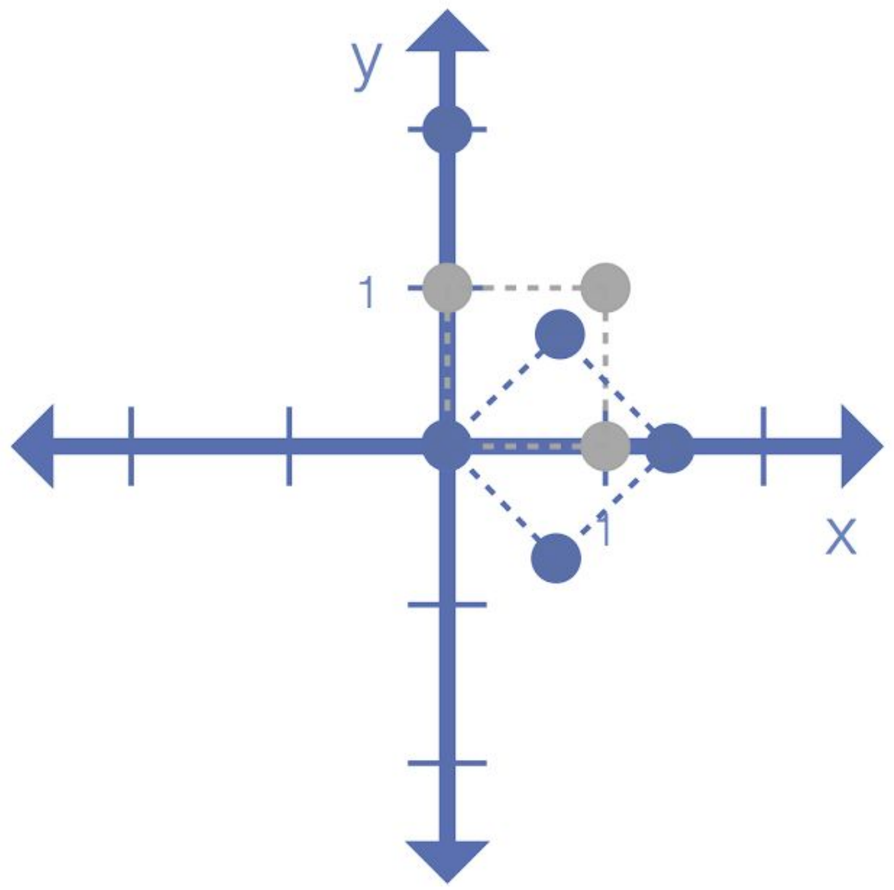
$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Rotate

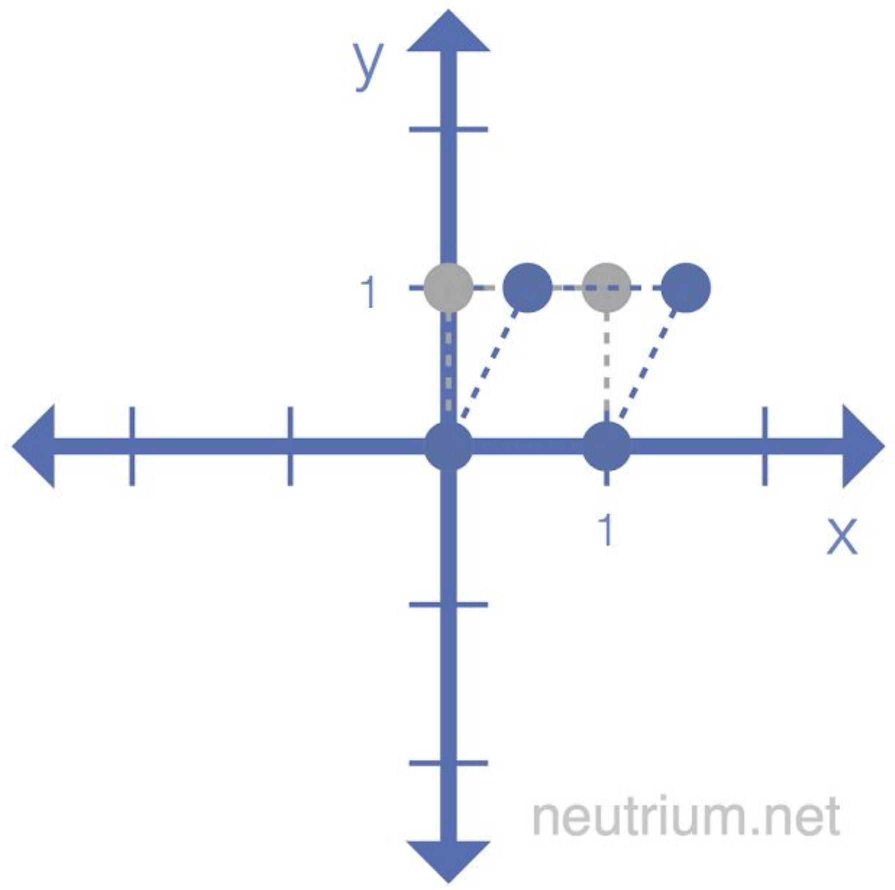
$$\begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$c = s = \sin(45^{\circ})$



Shear

$$\begin{bmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$



Translation

Translation is the shifting of object's location. If you know the shift in (x,y) direction, let it be (t_x,t_y), you can create the transformation matrix M as follows:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Rotation

Rotation of an image for an angle \theta is achieved by the transformation matrix of the form:

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

But OpenCV provides scaled rotation with adjustable center of rotation so that you can rotate at any location you prefer. Modified transformation matrix is given by:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha) \cdot center.y \end{bmatrix}$$

Where:

$$\alpha = scale \cdot \cos\theta, \\ \beta = scale \cdot \sin\theta$$

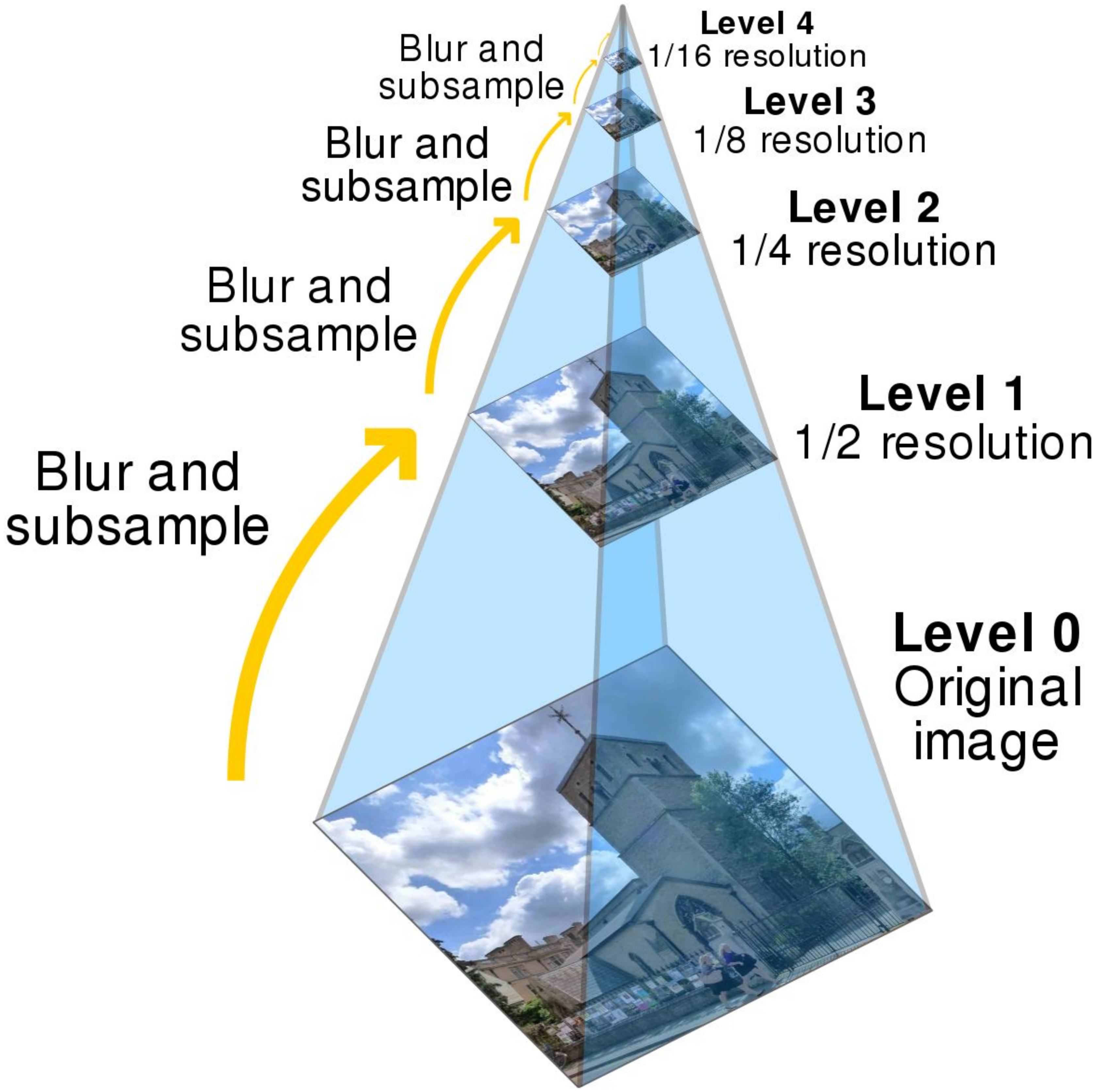
Scaling Images

Interpolation Methods

- INTER_NEAREST - a nearest-neighbor interpolation (Fastest)
- INTER_LINEAR - a bilinear interpolation (used by default) (good for upsampling)
- INTER_AREA - resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moire'-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method. (Good for downsampling)
- INTER_CUBIC - a bicubic interpolation over 4x4 pixel neighborhood (Better)
- INTER_LANCZOS4 - a Lanczos interpolation over 8x8 pixel neighborhood (Best one)

Image Pyramids

Normally, we used to work with an image of constant size. But on some occasions, we need to work with (the same) images in different resolution. For example, while searching for something in an image, like face, we are not sure at what size the object will be present in said image. In that case, we will need to create a set of the same image with different resolutions and search for object in all of them. These set of images with different resolutions are called Image Pyramids (because when they are kept in a stack with the highest resolution image at the bottom and the lowest resolution image at top, it looks like a pyramid).



There are two kinds of Image Pyramids.

1) Gaussian Pyramid and

2) Laplacian Pyramids

Cropping

For cropping an image all we need to do is to slice the matrix as we already know, example:

image[start_row : end_row ,start_col : end_col]

That is always the order to follow rows first then columns.

Brightening and darkening images

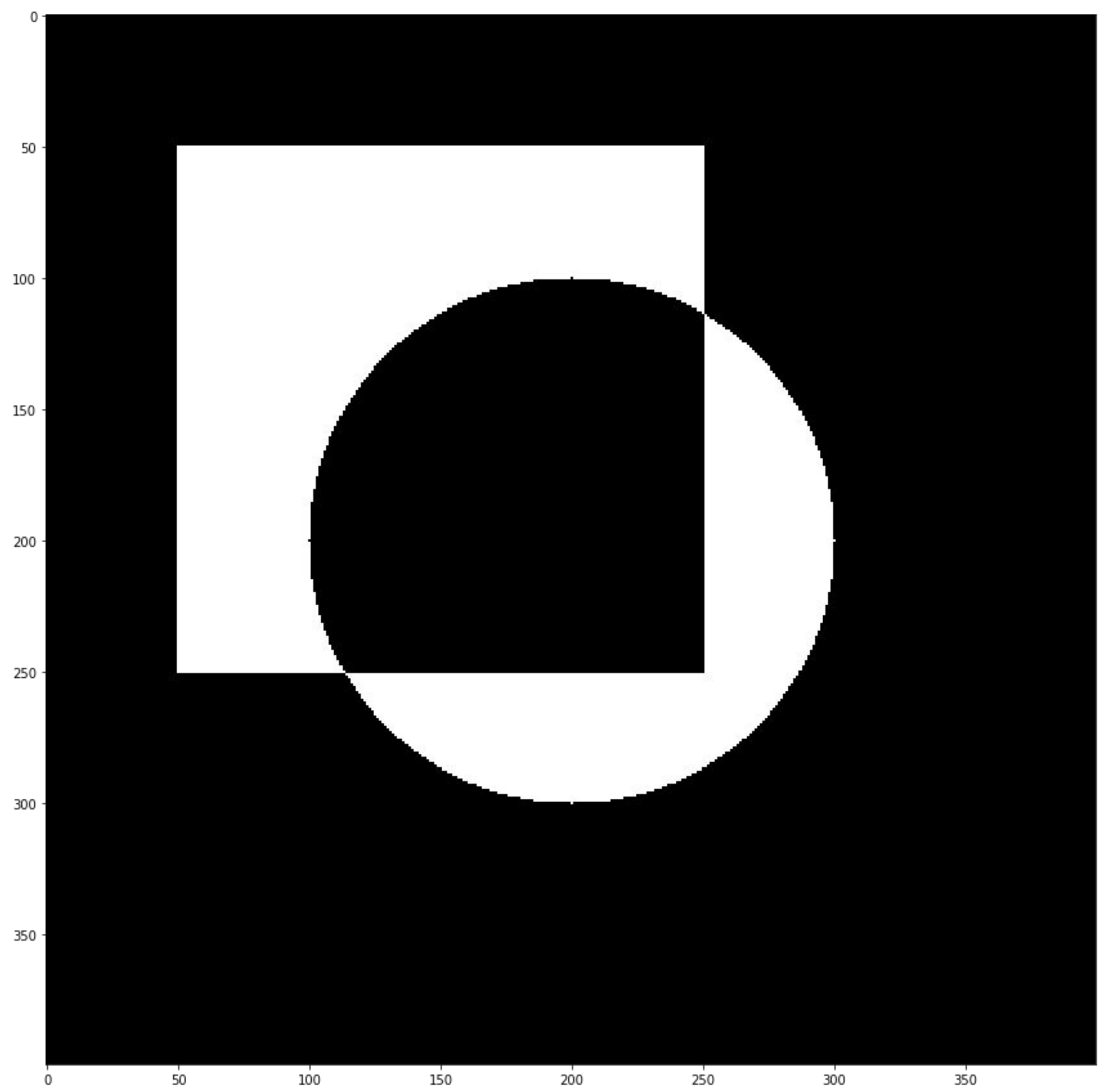
I’m using the example of adding bright or darkness to an image, but we can add or subtract values to any channel, remember the color spaces?

We can add to the R, G or B channels, or H, S or V channels, etc.

Once you understand how it will affect your image you can use it to process it in any way you could use to accomplish your desired task

Bitwise operations

- There are 4 bitwise operations we can use:
  - And
  - Or
  - Not
  - Xor

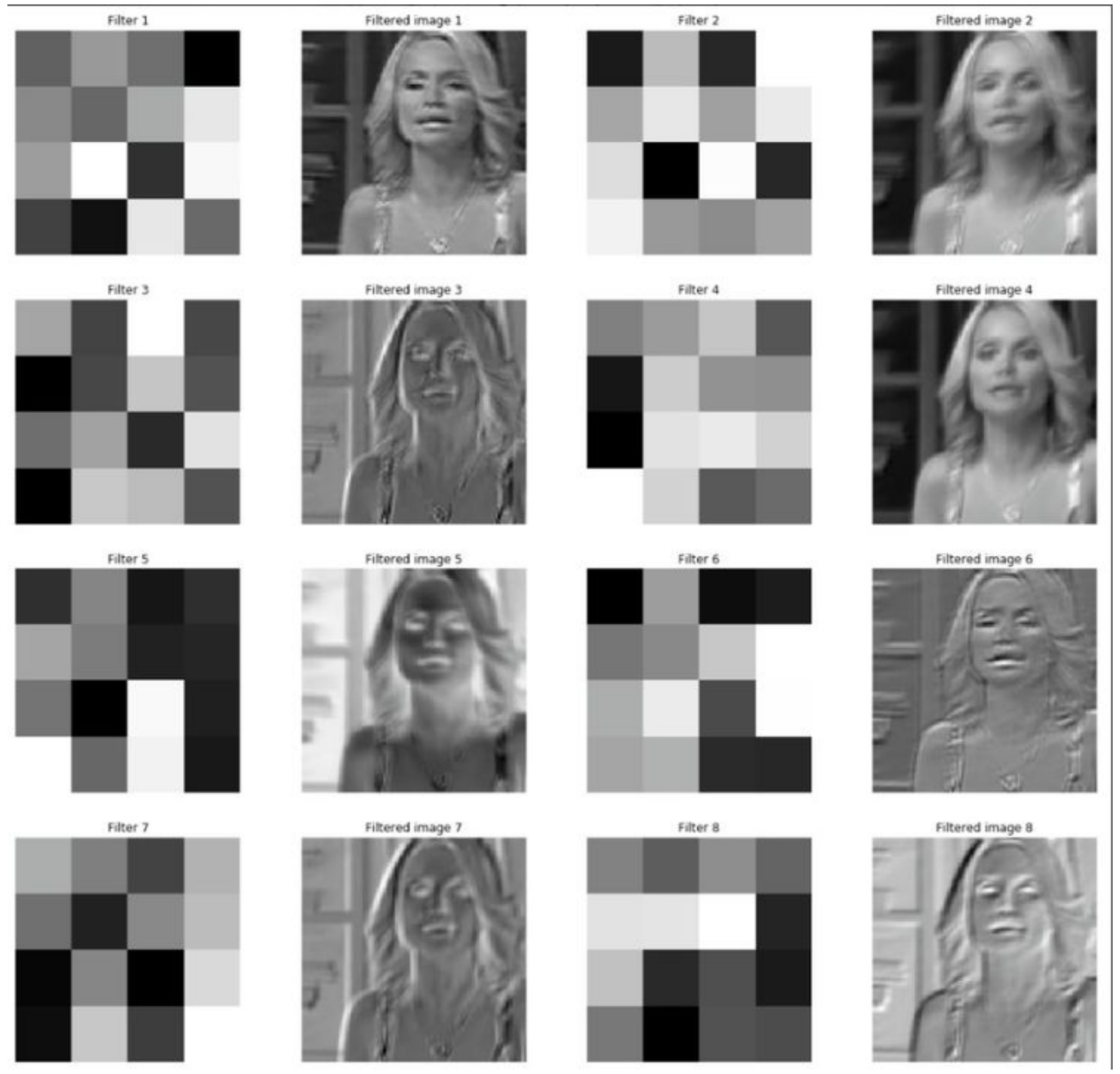


These operations are normally used to create masks to process your images further.

Filters aka Kernels

We can modify images by applying different types of kernels:

1. Blur kernels
2. Sharpening kernels
3. Identity kernels
4. Sobel Kernels
5. Edge kernels
6. Emboss Kernel
7. Outline Kernels



http://setosa.io/ev/image-kernels/

Blur kernel

A blurring kernel has the following properties:

1. All the values in blurring masks are positive
2. The sum of all the values is equal to 1
3. The edge content is reduced by using a blurring mask
4. As the size of the mask grow, more smoothing effect will take place

Blur: Smoothing images

Blur is heavily used in computer vision, you can use a kernel and apply it to an image with filter2D or you can use predefined Blur functions like:

- Blur (you define the kernel size)
- GaussianBlur (Gaussian Kernel)
- medianBlur (median of all the pixels)
- bilateralFilter (effective in noise removal while keeping edges sharp)

1/16

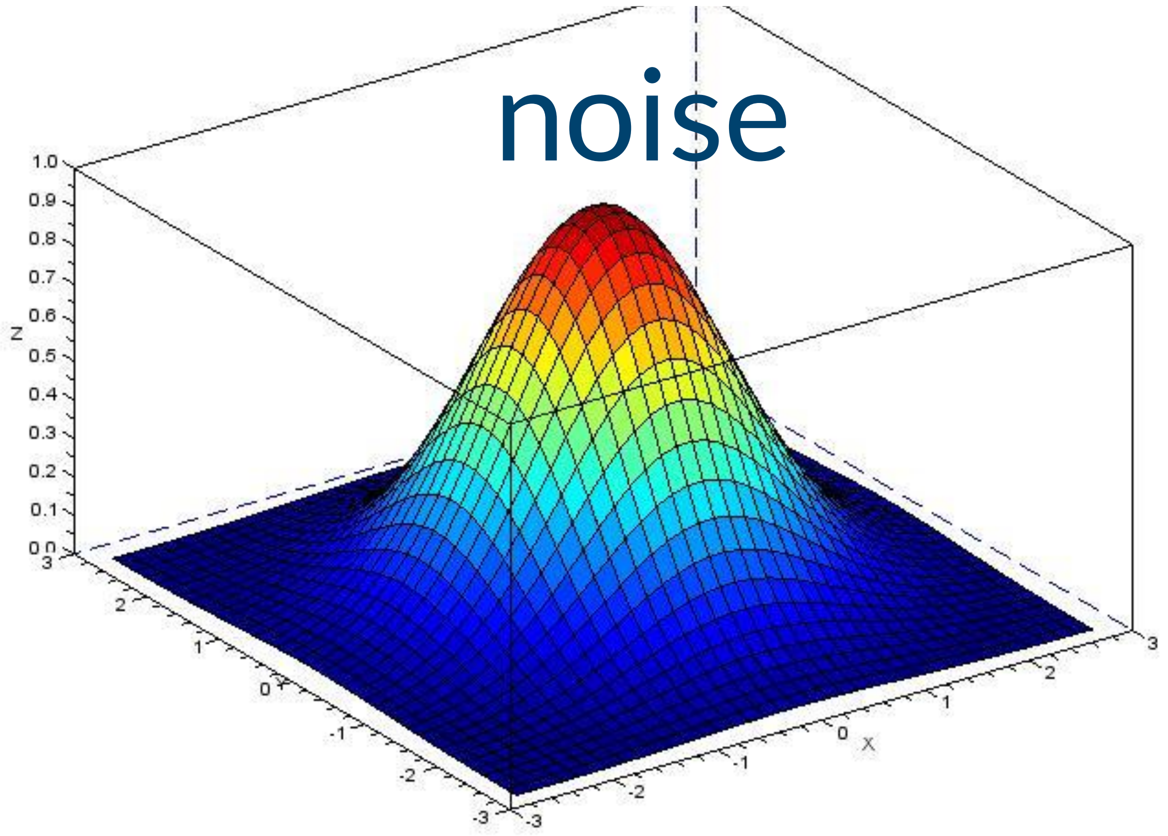
1	2	1
2	4	2
1	2	1

1/273

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0



Sharpening

The details of an image can be emphasized by using a high-pass filter

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

It will sum up to 1 with the highest value in the center of the kernel.

```
[-1., -1., -1.]
[-1.,  9., -1.]
[-1., -1., -1.]
```

```
[-1., -1., -1., -1., -1.]
[-1., -1., -1., -1., -1.]
[-1., -1., 25., -1., -1.]
[-1., -1., -1., -1., -1.]
[-1., -1., -1., -1., -1.]
```

Edge detection kernels

Is very similar to the sharpening kernels but the values will sum to 0

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Sharpening kernel

```
[-1., -1., -1.]
[-1.,  9., -1.]
[-1., -1., -1.]
```

Edge kernel

```
[-1., -1., -1.]
[-1.,  8., -1.]
[-1., -1., -1.]
```

Image Thresholding

If pixel value is **greater** than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black).

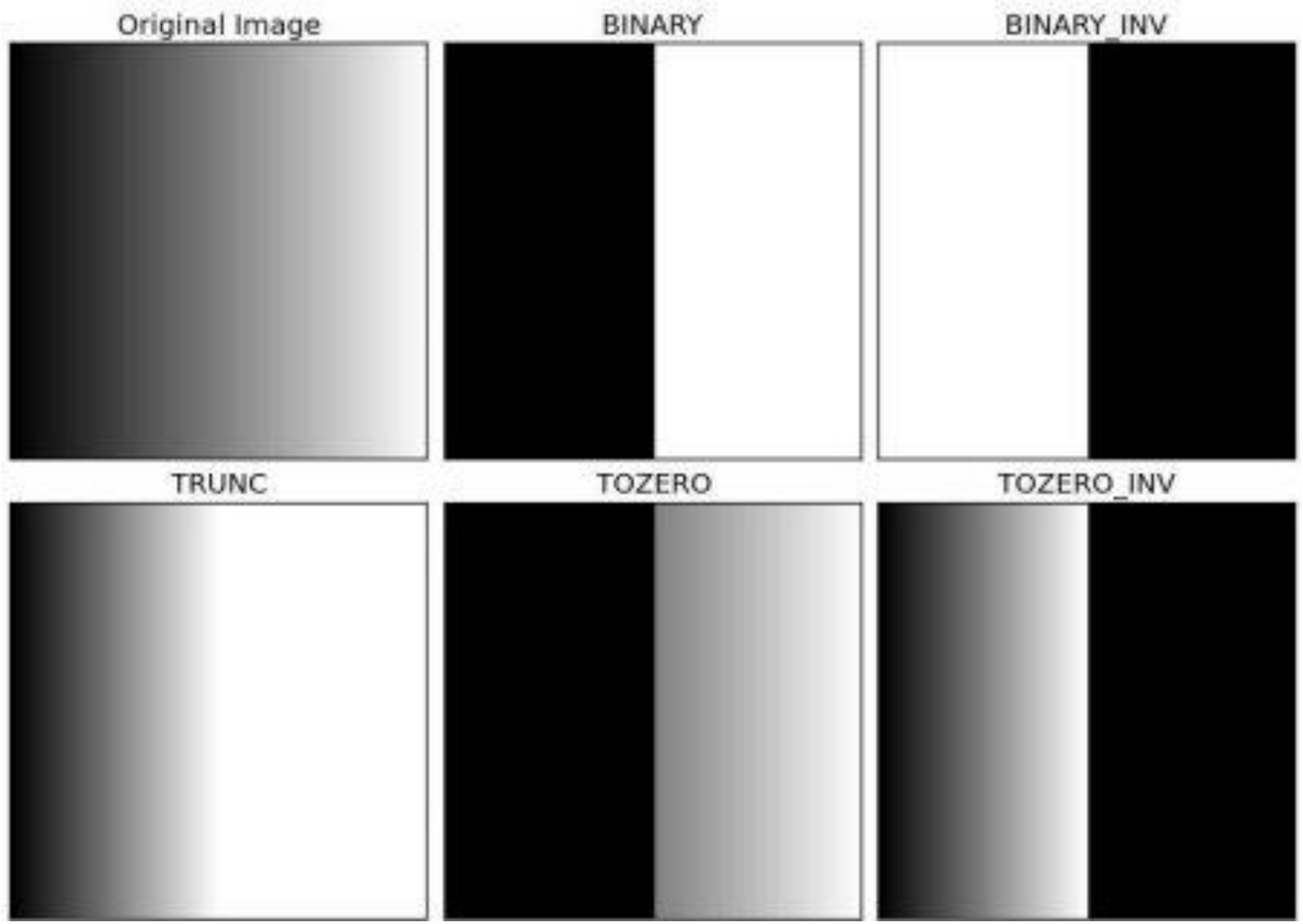


Image Thresholding

The function used is **cv2.threshold**.

1. First argument: is the source image, which should be a grayscale image.
2. Second argumen:t is the threshold value which is used to classify the pixel values.
3. Third argument: is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value
4. Fourth argument: is the styles of thresholding :
    a. **THRESH_BINARY**
    b. **THRESH_BINARY_INV**
    c. **THRESH_TRUNC**
    d. **THRESH_TOZERO**
    e. **THRESH_TOZERO_INV**

Adaptive Thresholding

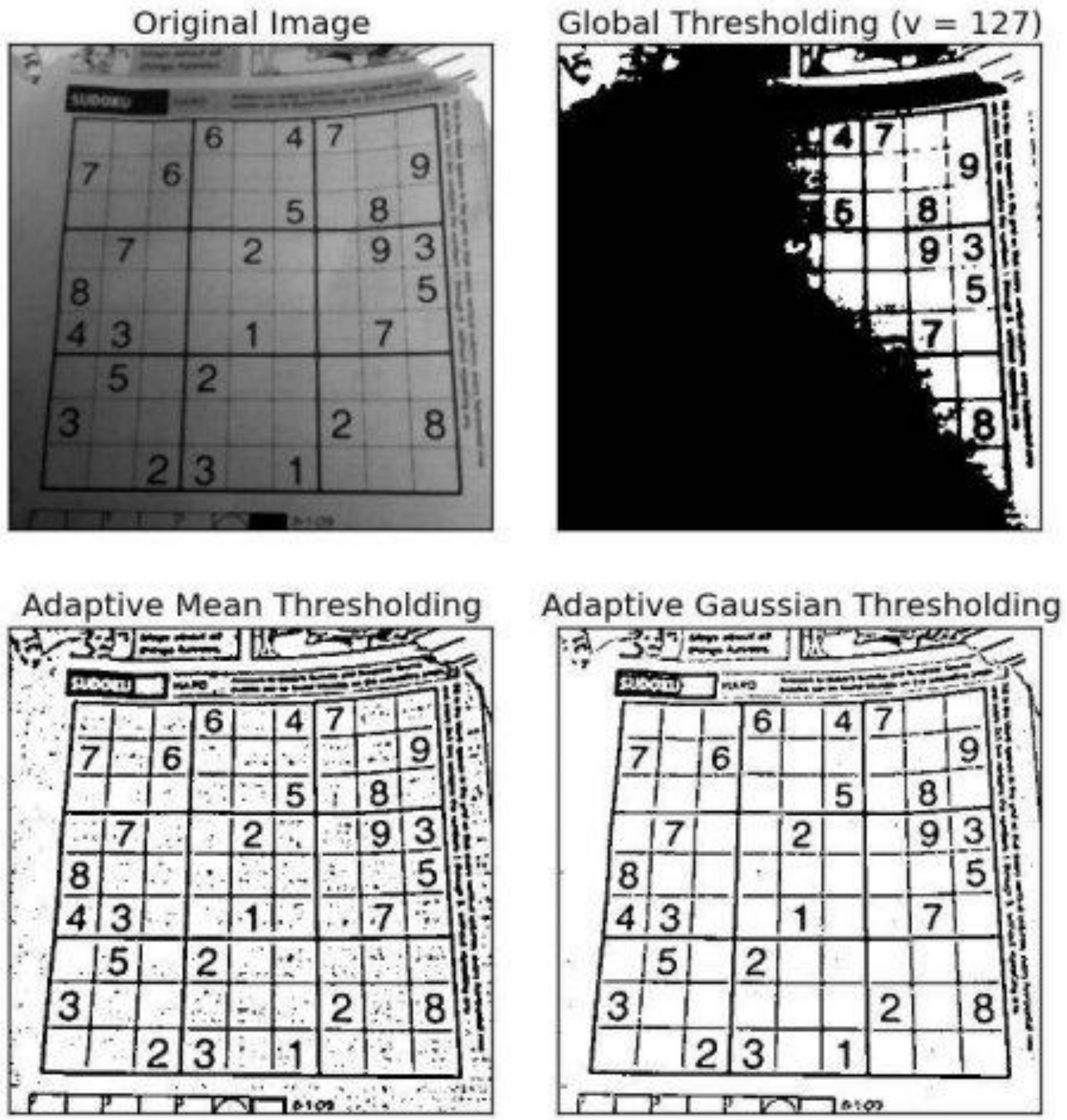
Before we used a global value as threshold value. But it **may not be good** in all the conditions **where image has different lighting conditions in different areas**.

In that case, we go for **adaptive thresholding**. In this, **the algorithm calculate the threshold for a small regions of the image**. So we get **different thresholds for different regions** of the same image and it gives us better results for images with varying illumination.

`cv.adaptiveThreshold`

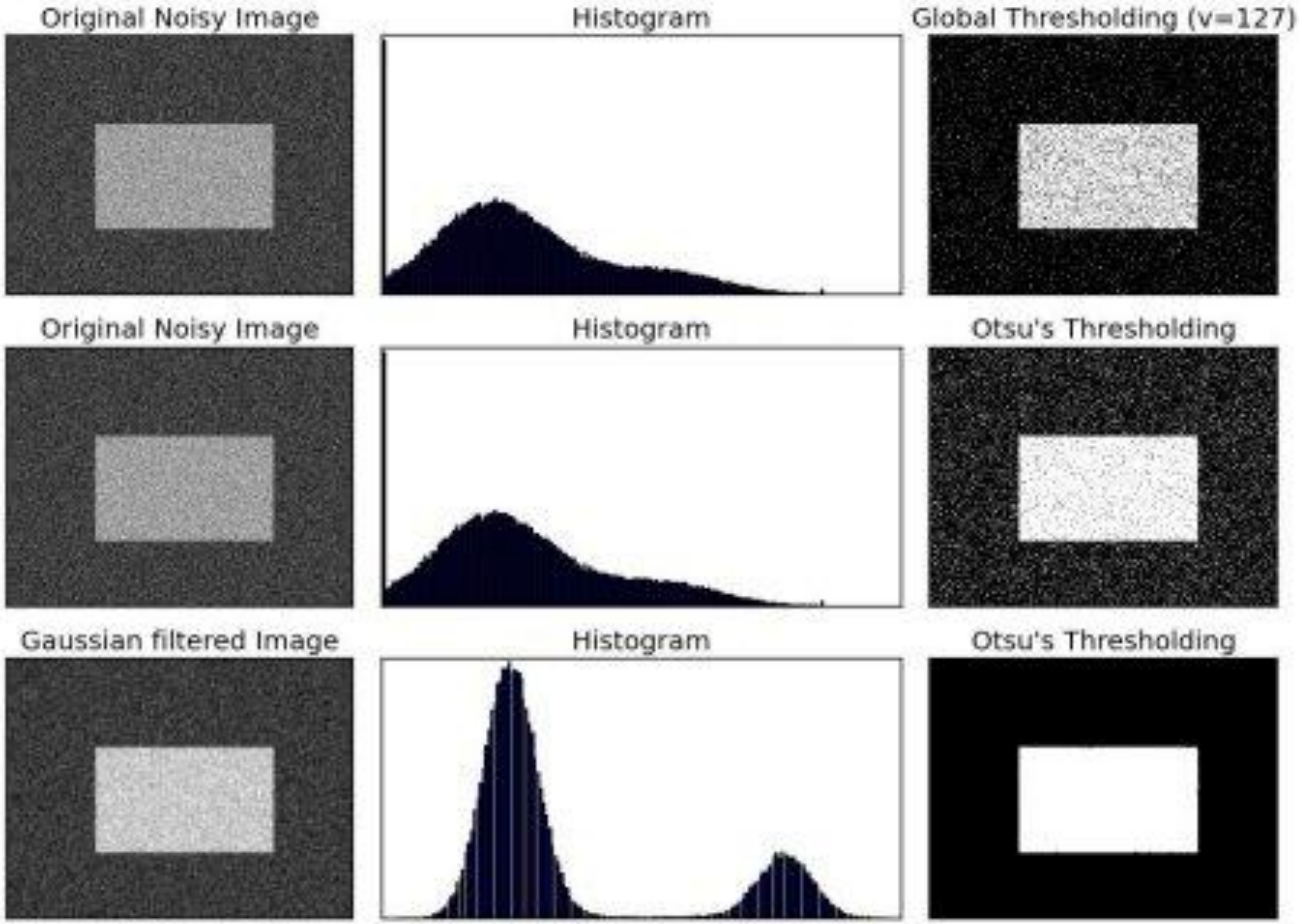
Special input parameters are:

1. **Adaptive Method** - It decides how thresholding value is calculated.:
   a. ADAPTIVE_THRESH_MEAN_C
   b. ADAPTIVE_THRESH_GAUSSIAN_C
2. **Block Size** - It decides the size of neighbourhood area.
3. **C** - It is just a constant which is subtracted from the mean or weighted mean calculated.



Otsu’s Binarization

Can be used for bimodal images (an image whose histogram has two peaks)



Otsu’s Binarization

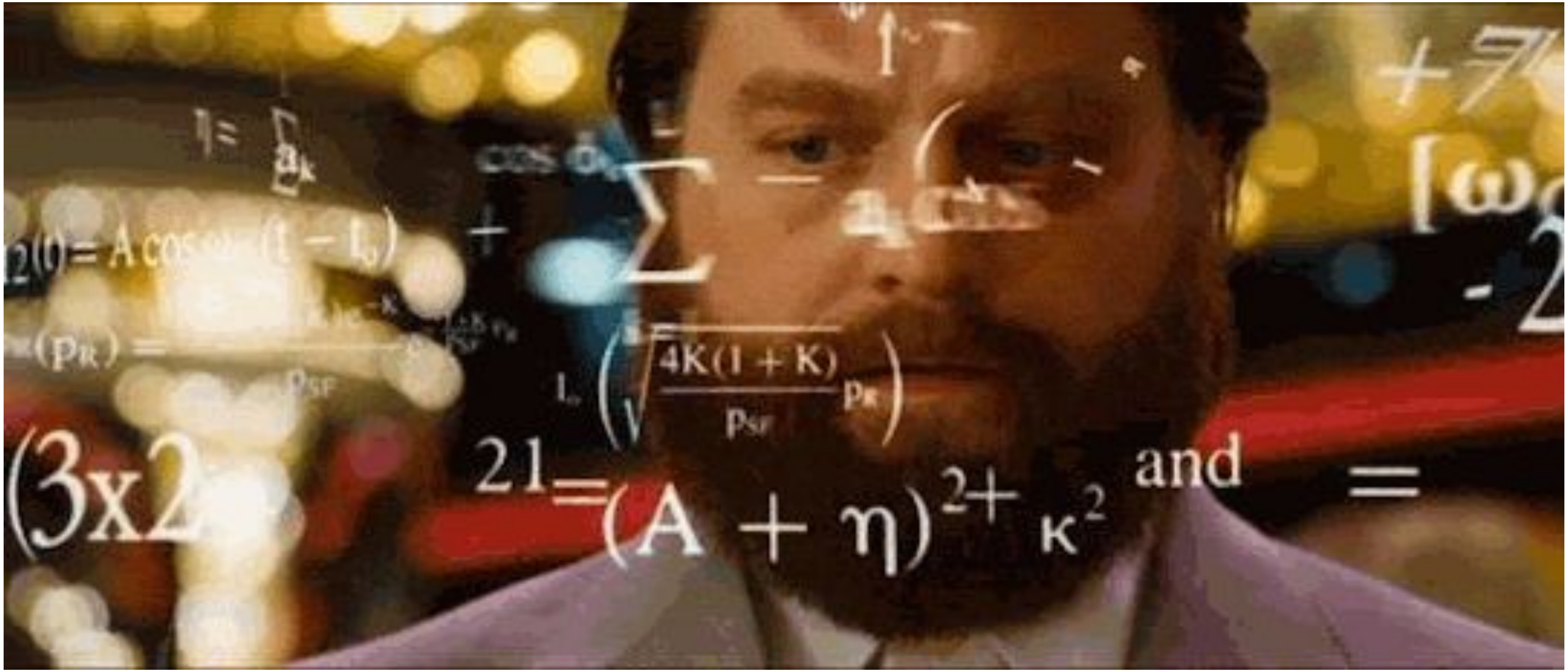
For this, our cv.threshold() function is used, but pass an extra flag, cv2.THRESH_OTSU. For threshold value, simply pass zero.

Then the algorithm finds the optimal threshold value and returns you as the second output, retVal.

If Otsu thresholding is not used, retVal is same as the threshold value you used.

```
# Otsu's thresholding after Gaussian filtering
blur = cv2.GaussianBlur(img,(5,5),0)
ret,th = cv2.threshold(blur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
```

How Otsu works?



How Otsu works?

Since we are working with bimodal images, Otsu's algorithm tries to find a threshold value (t) which minimizes the weighted within-class variance given by the relation :

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

$$q_1(t) = \sum_{i=1}^{t} P(i) \quad \& \quad q_2(t) = \sum_{i=t+1}^{I} P(i)$$

$$\mu_1(t) = \sum_{i=1}^{t} \frac{iP(i)}{q_1(t)} \quad \& \quad \mu_2(t) = \sum_{i=t+1}^{I} \frac{iP(i)}{q_2(t)}$$

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} \quad \& \quad \sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 \frac{P(i)}{q_2(t)}$$

It actually finds a value of t which lies in between two peaks such that variances to both classes are minimum

Edge detection

- Sobel: To emphasize vertical or horizontal edges
- Laplacian: Gets all orientations
- Canny: optimal due to low error rate, well defined edges and accurate detection
  - Applies gaussian blurring
  - Finds intensity gradient of the image
  - Applies non-maximum suppresion (removing pixels that are not edges)
  - Applies thresholds

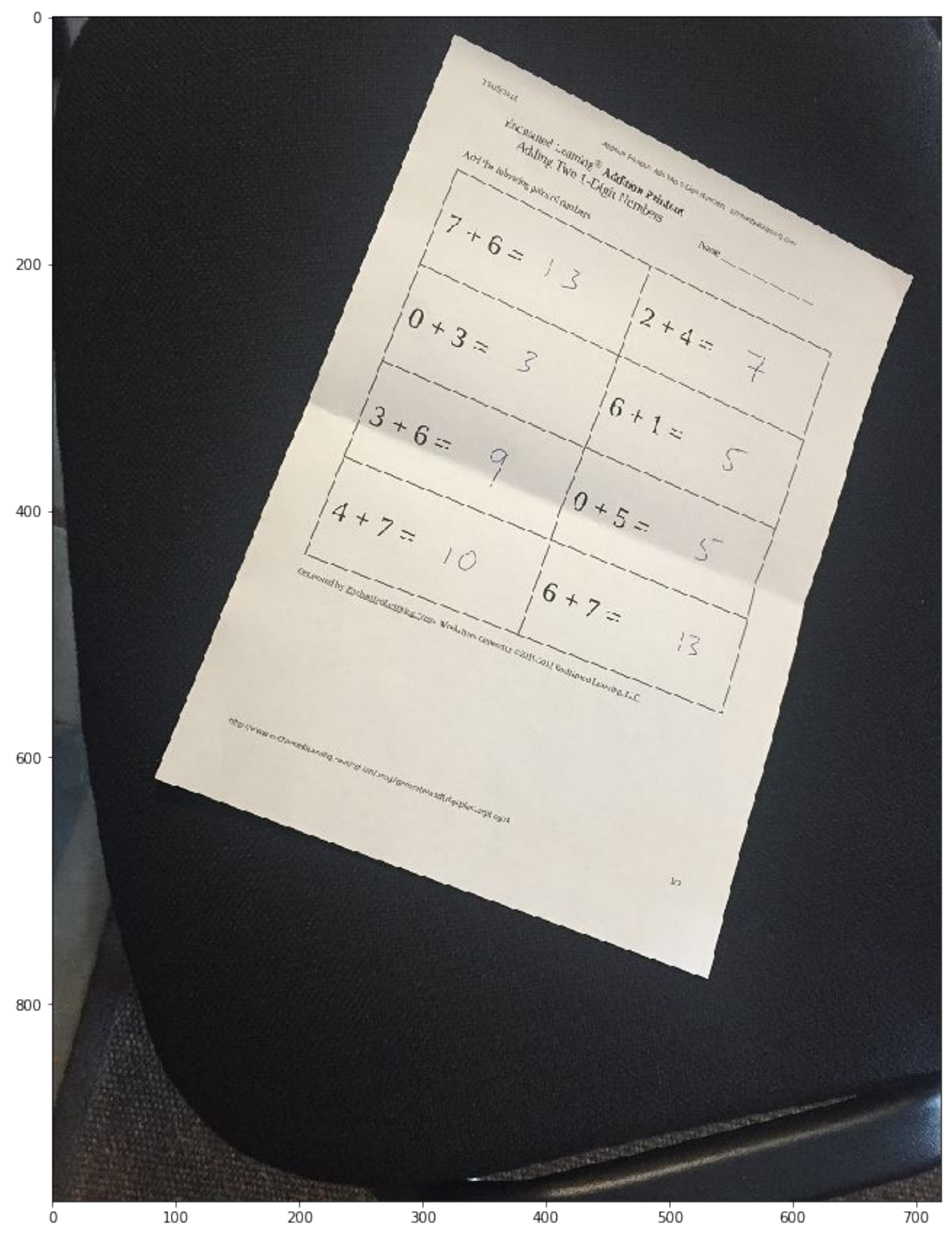
Perspective Transform

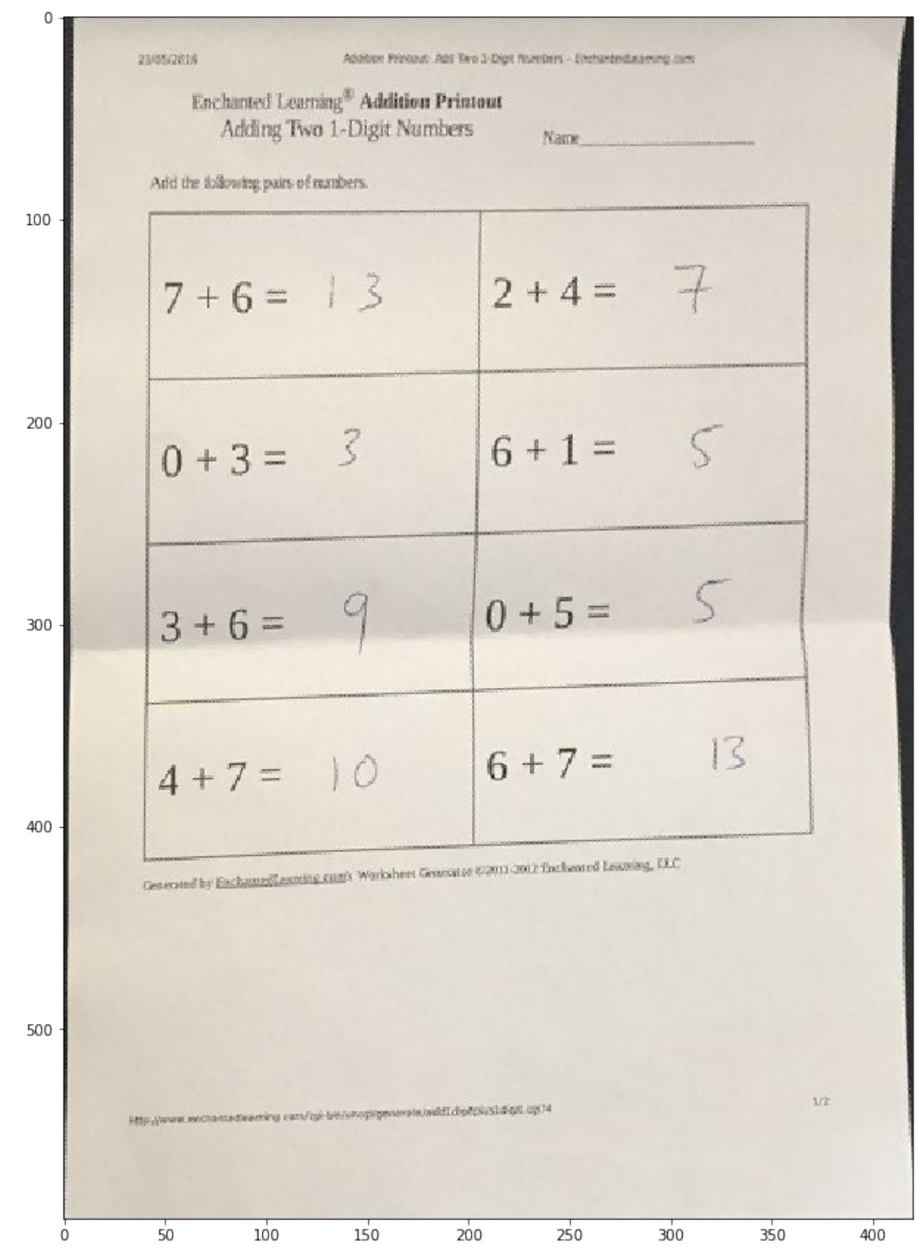
- We can change the perspective of an image with:

  cv2.warpPerspective

- First we need to get the transformation matrix, we can do it with:

  cv2.getPerspectiveTransform(points_A, points_B)





About me…

George Studenko

Software Engineer and AI Enthusiast

georgestudenko georgestudenko george-studenko