



COMPUTER VISION AND INTELLIGENCE GROUP

In [86]:

```
from google.colab import files
uploaded = files.upload()

for fn in uploaded.keys():
    print('User uploaded file "{name}" with length {length} bytes'.format(
        name=fn, length=len(uploaded[fn])))
```

Choose Files No file chosen

Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.

Saving baby.png to baby.png
User uploaded file "baby.png" with length 22604 bytes

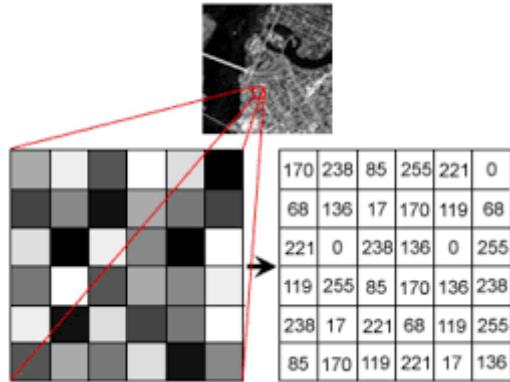
What is an Image?

Generally, we can consider an image as a matrix whose elements are numbers between 0 and 255. The size of this matrix is (image height) x (image width) x (# of image channels). A grayscale image has 1 channel where a color image has 3 channels (for RGB).

Each value mentioned above is a 8-bit number. i.e, it can take values between 0-255. These values correspond to the shade of that particular parameter. 0 for minimum and 255 for maximum.

Grayscale image

A pixel having the value 0 implies it is black and the pixel having value 255 implies that it is white. The values inbetween correspond to shades inbetween.



RGB image

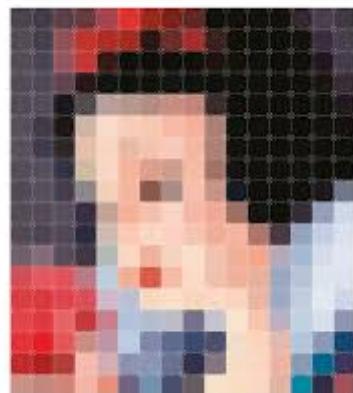
Each pixel has three channels, one for each colour.

R : 0-255.

G : 0-255.

B : 0-255.

Each number between 0-255 corresponds to a shade of that corresponding color. All 3 channels being 0 makes the pixel black while all 3 channels being 255 makes the pixel white.



69	R: 48	R: 10	R: 71	R: 79	R: 76	R: 74	R: 71	R: 72	R: 74	R:
44	G: 43	G: 43	G: 46	G: 41	G: 48	G: 41	G: 42	G: 44	G: 56	G:
68	B: 70	B: 72	B: 70	B: 66	B: 65	B: 69	B: 70	B: 67	B: 61	B:
70	B: 71	B: 71	B: 69	B: 70	B: 69	B: 72	B: 67	B: 110	B: 128	B: 1
45	G: 44	G: 44	G: 42	G: 43	G: 41	G: 46	G: 44	G: 98	G: 116	G: 1
73	B: 70	B: 67	B: 65	B: 67	B: 67	B: 62	B: 58	B: 51	B: 41	B:
74	R: 72	R: 76	R: 69	R: 74	R: 61	R: 102	R: 132	R: 148	R: 151	R:
42	G: 44	G: 44	G: 48	G: 49	G: 64	G: 98	G: 123	G: 138	G: 144	G:
75	B: 73	B: 70	B: 68	B: 64	B: 54	B: 49	B: 30	B: 25	B: 19	B:
76	R: 72	R: 76	R: 92	R: 115	R: 130	R: 143	R: 152	R: 161	R: 183	R:
44	G: 44	G: 47	G: 70	G: 94	G: 118	G: 133	G: 140	G: 143	G: 146	G:
69	B: 66	B: 62	B: 53	B: 44	B: 25	B: 11	B: 61	B: 13	B: 36	B:
68	B: 75	R: 103	R: 129	R: 121	R: 145	R: 151	R: 152	R: 157	R: 164	R: 1
44	G: 56	G: 89	G: 120	G: 122	G: 125	G: 141	G: 143	G: 148	G: 150	G:
62	B: 53	B: 47	B: 42	B: 27	B: 15	B: 7	B: 4	B: 15	B: 16	B:
76	R: 115	R: 136	R: 103	R: 140	R: 151	R: 153	R: 157	R: 160	R: 164	R:
59	G: 102	G: 128	G: 126	G: 133	G: 142	G: 158	G: 168	G: 159	G: 159	G:
48	B: 45	B: 37	B: 12	B: 11	B: 10	B: 12	B: 10	B: 18	B: 30	B:
38	B: 195	R: 128	R: 141	R: 161	R: 153	R: 159	R: 160	R: 163	R: 169	R:
105	G: 127	G: 124	G: 103	G: 141	G: 143	R: 145	G: 151	G: 154	G: 154	G:
46	B: 29	B: 14	B: 7	B: 6	B: 6	B: 7	B: 19	B: 24	B: 21	B:

There are other colour scales which are used to represent a coloured image which we won't discuss in detail. These include:

- HSV (Hue, Saturation and value)
- CMYK (Cyan Yellow Magenta Key) is a subtractive model used especially in printing.

Getting Started with Images

1) Reading an image

We use the function `cv2.imread()` to read an image. The image should be in the **working directory** or the full path of image should be given.

Python:

```
cv2.imread(src, flag)
```

Parameters:

- `src` – input image.
- `flag` – specifies the way image should be read.
 - `cv2.IMREAD_COLOR` or `1`: Loads a color image. Any transparency of image will be neglected. It is the default flag.
 - `cv2.IMREAD_GRAYSCALE` or `0` : Loads image in grayscale mode.

Warning: Even if the image path is wrong, it won't throw any error, but nothing gets stored in the image variable.

2) Displaying an image

We use the function `cv2.imshow()` to display an image in a window. The window automatically fits to the image size.

Python:

```
cv2.imshow(name, img)
```

Parameters:

- `name` – window name or title.
- `img` – image to be displayed.

Generally 2 more functions are used immediately after the `cv2.imshow()` function.

`cv2.waitKey()` is a keyboard binding function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard event. If you press any key in that time, the program continues. If `0` is passed, it waits indefinitely for a key stroke.

`cv2.destroyAllWindows()` simply destroys all the windows we created. If you want to destroy any specific window, use the function `cv2.destroyWindow()` where you pass the exact window name as the argument.

3) Saving an image

We use the function `cv2.imwrite()` to save an image. This will save the image in PNG format in the working directory.

Python:

```
cv2.imwrite(filename, img)
```

Parameters:

- filename – file name as which the image should be saved.
- img – image to be saved.

Make sure you understand what every line of the code below.

In [0]:

```
import cv2

img = cv2.imread('media/cube.jpg',0)
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imwrite('media/cube1.jpg',img)
```

Using Matplotlib

Matplotlib is a plotting library for Python which gives you a wide variety of plotting methods. Let's just learn how to display an image with Matplotlib.

Python:

```
plt.subplot(nrows, ncols, index)
```

Parameters:

- This command is used when we want to plot images in a tabular manner. Commas can be ignored as all inputs are constrained to a single digit number.
- nrows - number of rows of images you want.
- ncols - number of columns of images you want.
- index - index of the current image. (follows row-major)

Python:

```
plt.imshow(img, cmap)
```

Parameters:

- img - image to be plotted.
- cmap - colour map. Giving cmap='gray' means assigning white for 255 and black for 0. Default cmap depends on the compiler. Remove cmap and see what happens to the output.

Other commands used are self-explanatory.

In [80]:

```
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('media/messi.jpg',1)
img1 = cv2.imread('media/messi.jpg',0)
plt.subplot(121),plt.imshow(img,cmap='gray'),plt.title("Coloured"),plt.axis("off")
plt.subplot(122),plt.imshow(img1,cmap='gray'),plt.title("Grayscale"),plt.axis("off")
plt.show()
```

Coloured



Grayscale



Thresholding

Thresholding is the simplest method of image segmentation, which is the process of partitioning a digital image into multiple segments.

Simple Thresholding

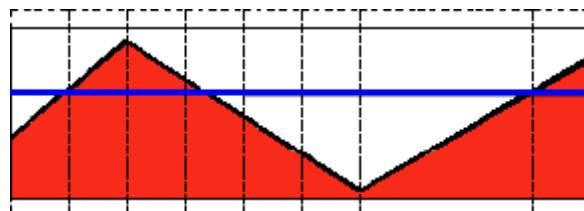
Every pixel of the image is checked and if the pixel value is greater than a constant threshold value T , it is assigned one value (may be white), else it is assigned another value (may be black).

- If $I(i, j) < T$, $I(i, j)$ is assigned the value given by the user. (Generally 0, black).
- If $I(i, j) > T$, $I(i, j)$ is assigned the value given by the user. (Generally 255, white).



Types of Simple Thresholding:

- OpenCV offers the function `cv2.threshold()` to perform thresholding operations.
- We can effectuate 5 types of Thresholding operations with this function. We will explain them in the following subsections.
- To illustrate how these thresholding processes work, let's consider that we have a source image with pixels with intensity values $\text{src}(x,y)$. The plot below depicts this. The horizontal blue line represents the threshold, thresh (fixed).

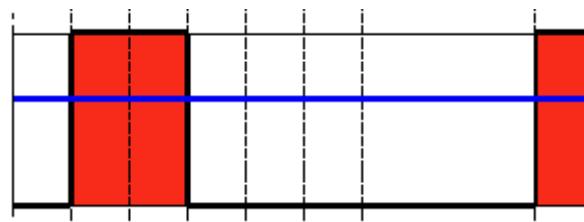


1) Threshold Binary

This thresholding operation can be expressed as:

$$\text{dst}(x, y) = \begin{cases} \text{maxVal} & \text{if } \text{src}(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

So, if the intensity of the pixel $\text{src}(x,y)$ is higher than thresh , then the new pixel intensity is set to a MaxVal . Otherwise, the pixels are set to 0.

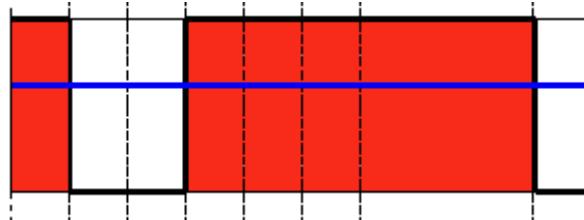


2) Threshold Binary, Inverted

This thresholding operation can be expressed as:

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > thresh \\ maxVal & \text{otherwise} \end{cases}$$

If the intensity of the pixel $src(x,y)$ is higher than $thresh$, then the new pixel intensity is set to a 0. Otherwise, it is set to MaxVal.

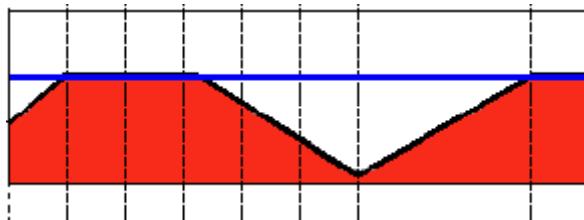


3) Truncate

This thresholding operation can be expressed as:

$$dst(x,y) = \begin{cases} threshold & \text{if } src(x,y) > thresh \\ src(x,y) & \text{otherwise} \end{cases}$$

The maximum intensity value for the pixels is $thresh$, if $src(x,y)$ is greater, then its value is truncated.

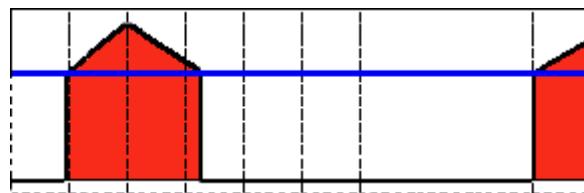


4) Threshold to Zero

This operation can be expressed as:

$$dst(x,y) = \begin{cases} src(x,y) & \text{if } src(x,y) > thresh \\ 0 & \text{otherwise} \end{cases}$$

If $src(x,y)$ is lower than $thresh$, the new pixel value will be set to 0.

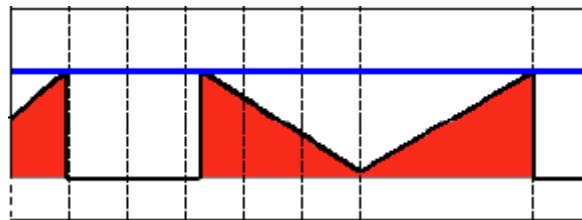


5) Threshold to Zero, Inverted

This operation can be expressed as:

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > thresh \\ src(x,y) & \text{otherwise} \end{cases}$$

If $src(x,y)$ is greater than $thresh$, the new pixel value will be set to 0.



Python:

```
cv2.threshold(src, thresh, maxVal, type)
```

Parameters:

- src - input image in **grayscale**.
- thresh - threshold value which is used to classify the pixel values.
- maxVal - depending on the type of threshold, it refers to various things.
- type - refers to the different styles of thresholding that OpenCV offers.
 - cv2.THRESH_BINARY
 - cv2.THRESH_BINARY_INV
 - cv2.THRESH_TRUNC
 - cv2.THRESH_TOZERO
 - cv2.THRESH_TOZERO_INV

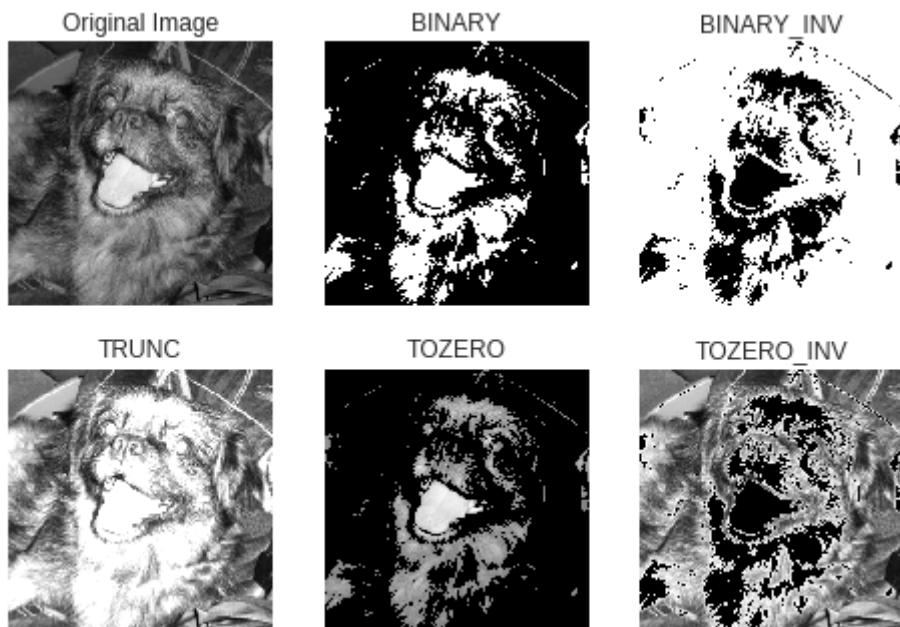
The function returns two outputs. First one is a retval which will be explained later. Second output is our thresholded image.

In [15]:

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

img = cv2.imread('media/dog.jpg',0)
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

plt.subplot(231),plt.imshow(img,cmap='gray'),plt.title("Original Image"),plt.axis("off")
plt.subplot(232),plt.imshow(thresh1,cmap='gray'),plt.title("BINARY"),plt.axis("off")
plt.subplot(233),plt.imshow(thresh2,cmap='gray'),plt.title("BINARY_INV"),plt.axis("off")
plt.subplot(234),plt.imshow(thresh3,cmap='gray'),plt.title("TRUNC"),plt.axis("off")
plt.subplot(235),plt.imshow(thresh4,cmap='gray'),plt.title("TOZERO"),plt.axis("off")
plt.subplot(236),plt.imshow(thresh5,cmap='gray'),plt.title("TOZERO_INV"),plt.axis("off")
plt.show()
```



Adaptive Thresholding

In the previous section, we used a global fixed value as our threshold value. But it may not be good in all the conditions where the image has different lighting conditions in different areas. In that case, we go for adaptive thresholding. In this, the algorithm calculates the threshold for 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.

Python:

```
cv2.adaptiveThreshold(src, maxVal, adapType, type, blockSize, c)
```

Parameters

- src - input image in grayscale.
- maxVal - same parameter as in simple thresholding.
- adapType - decides how threshold value is calculated.
 - cv2.ADAPTIVE_THRESH_MEAN_C : threshold value is the mean of neighbourhood area.
 - cv2.ADAPTIVE_THRESH_GAUSSIAN_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.
- type - refers to the different styles of thresholding that OpenCV offers. Same as in simple thresholding.
- blockSize - size of neighbourhood area.
- c - It is just a constant which is subtracted from the mean or weighted mean calculated.

This function returns only one output, the thresholded image. (Please note as to what happens when c changes!)

Below piece of code compares global thresholding and adaptive thresholding for an image with varying illumination.

In [19]:

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

img = cv2.imread('media/sudoku.jpg',0)

ret,th1 = cv2.threshold(img,127,255, cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY ,11,5)
th3 = cv2.adaptiveThreshold(img,255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,11,5)

plt.subplot(221),plt.imshow(img,cmap='gray'),plt.title("Original Image"),plt.axis("off")
plt.subplot(222),plt.imshow(th1,cmap='gray'),plt.title("Global Thresholding"),plt.axis("off")
plt.subplot(223),plt.imshow(th2,cmap='gray'),plt.title("Adaptive Mean Thresholding"),plt.axis("off")
plt.subplot(224),plt.imshow(th3,cmap='gray'),plt.title("Adaptive Gaussian Thresholding"),plt.axis("off")
plt.show()
```

Original Image



Global Thresholding



Adaptive Mean Thresholding



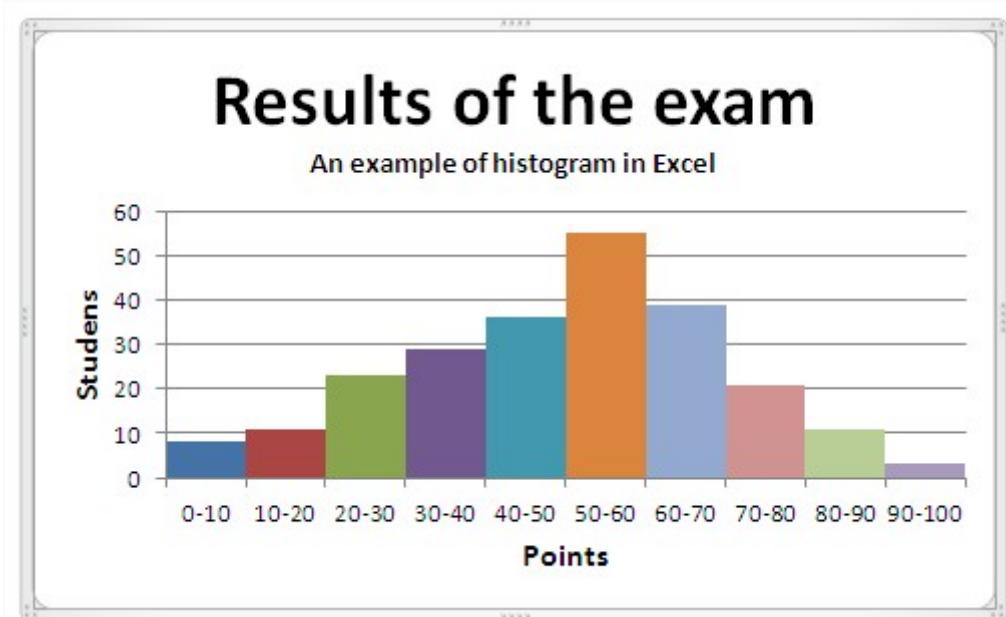
Adaptive Gaussian Thresholding



Histograms

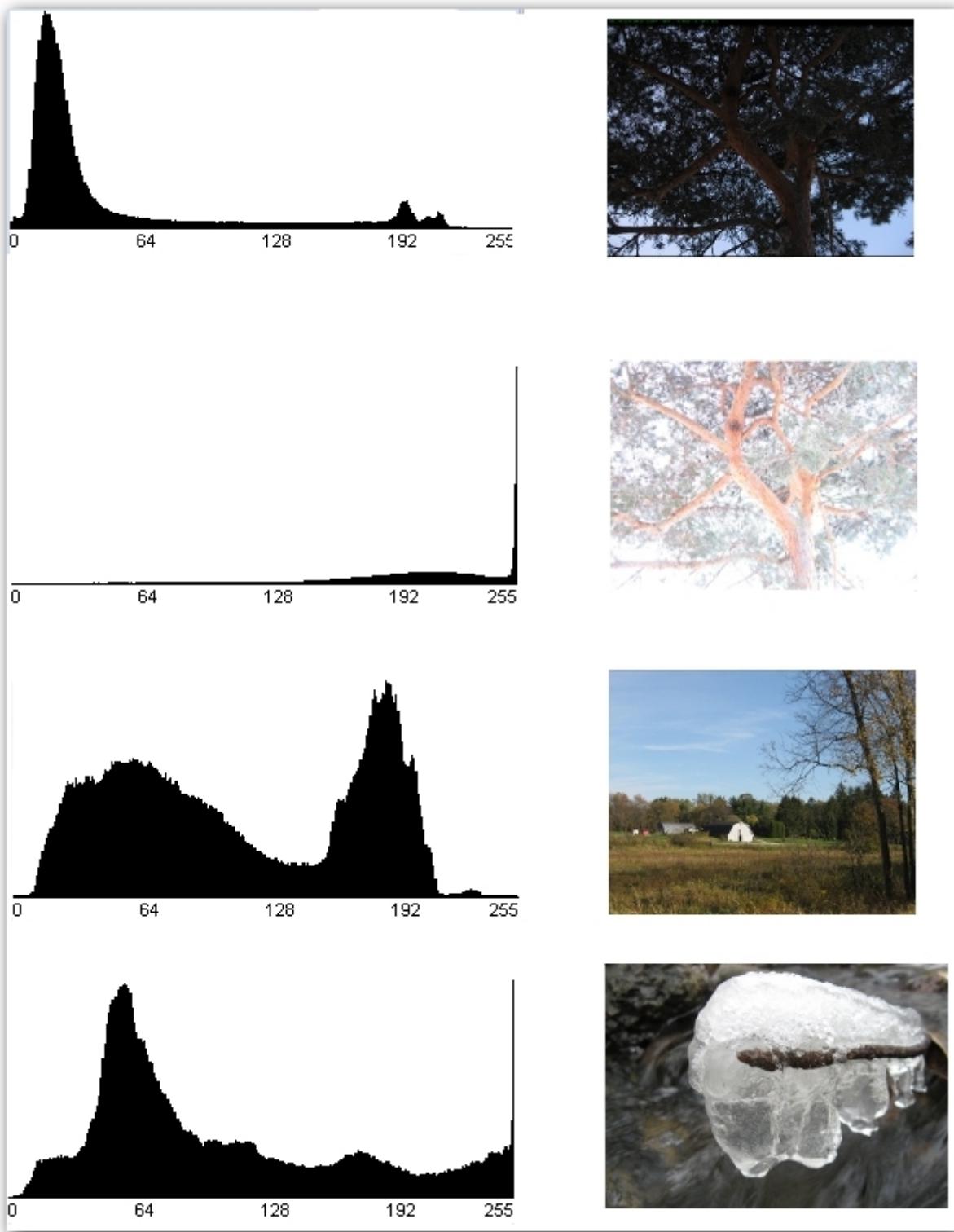
What is a Histogram?

In Statistics, Histogram is a graphical representation showing a visual impression of the distribution of data.



However in image processing.... Histograms are associated with the intensity values of pixels. You can consider histogram as a graph or plot, which gives you an overall idea about the intensity distribution of an image. It is a plot with pixel values (ranging from 0 to 255, not always) in X-axis and corresponding number of pixels with that intensity in the image on Y-axis.

In general, we have 256 intensity bins (0-255). Given below are a few examples of Histograms of an image.

**Python:**

```
cv2.calcHist(img, channel, mask, histSize, ranges)
```

Parameters:

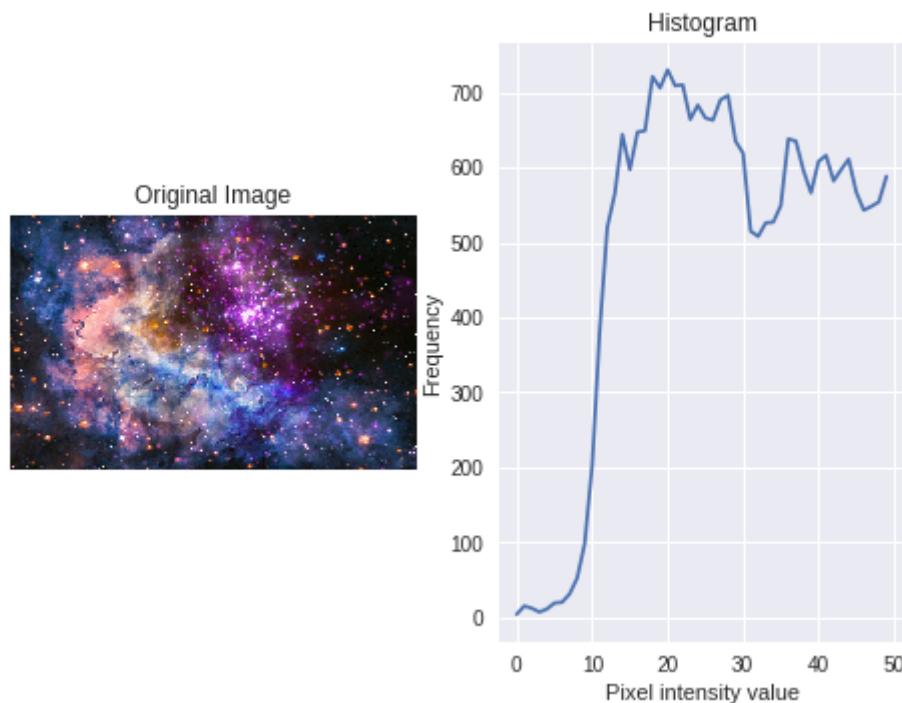
- img - input image. It should be given in square brackets, ie, “[img]”.
- channel - index of channel for which we calculate histogram. For example, if input is grayscale image, its value is [0]. For color image, you can pass [0], [1] or [2] to calculate histogram of blue, green or red channel respectively.
- mask - mask image. To find histogram of full image, it is given as “None”. But if you want to find histogram of particular region of image, you have to create a mask image for that and give it as mask (discussed later)

- histSize - this represents our BIN count. Need to be given in square brackets. For full scale, we pass [256].
- ranges - this is our RANGE. Normally, it is [0,256].

In [30]:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('media/space.jpg')
plt.subplot(121),plt.axis("off"),plt.title("Original Image"),plt.imshow(img,cmap ='gray')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
hist = cv2.calcHist([img],[0], None, [256], [0,256])
plt.subplot(122),plt.plot(hist),plt.xlabel('Pixel intensity value'),plt.ylabel('Frequency'),plt.title('Histogram')
plt.show()
```

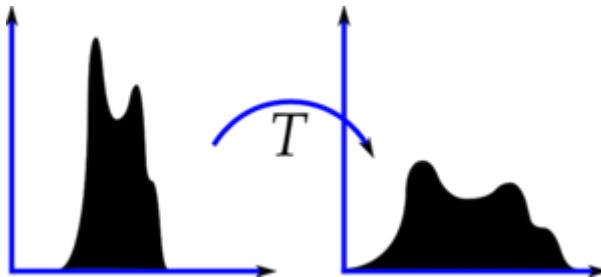


Histogram Equalization

The most commonly used Histogram feature in image processing is known as Histogram Equalization.

It's usually advised to normalize a histogram by dividing each of its value by total number of pixels in that image, thus forming a normalised histogram.

This normalised histogram can be interpreted probability functions that denote the probability of occurrence of a gray scale intensity r_k (just a variable) in the image. But it goes without mentioning that sum of all components of a normalized histogram is 1.



Why is histogram equalisation used??

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. It is not necessary that contrast will always be increase in this. There may be some cases where histogram equalization can be worse. In that cases the contrast is decreased.

Histogram equalization employs a monotonic, non-linear mapping which re-assigns the intensity values of pixels in the input image such that the output image contains a uniform distribution of intensities (i.e. a flat histogram).

Python:

```
cv2.equalizeHist(img)
```

Parameter:

- img - input image in grayscale.

In [89]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('lawn.jpg',0)
equ = cv2.equalizeHist(img)
plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(equ,cmap='gray'),plt.axis('off'),plt.title('Histogram Equalized')
plt.show()
```



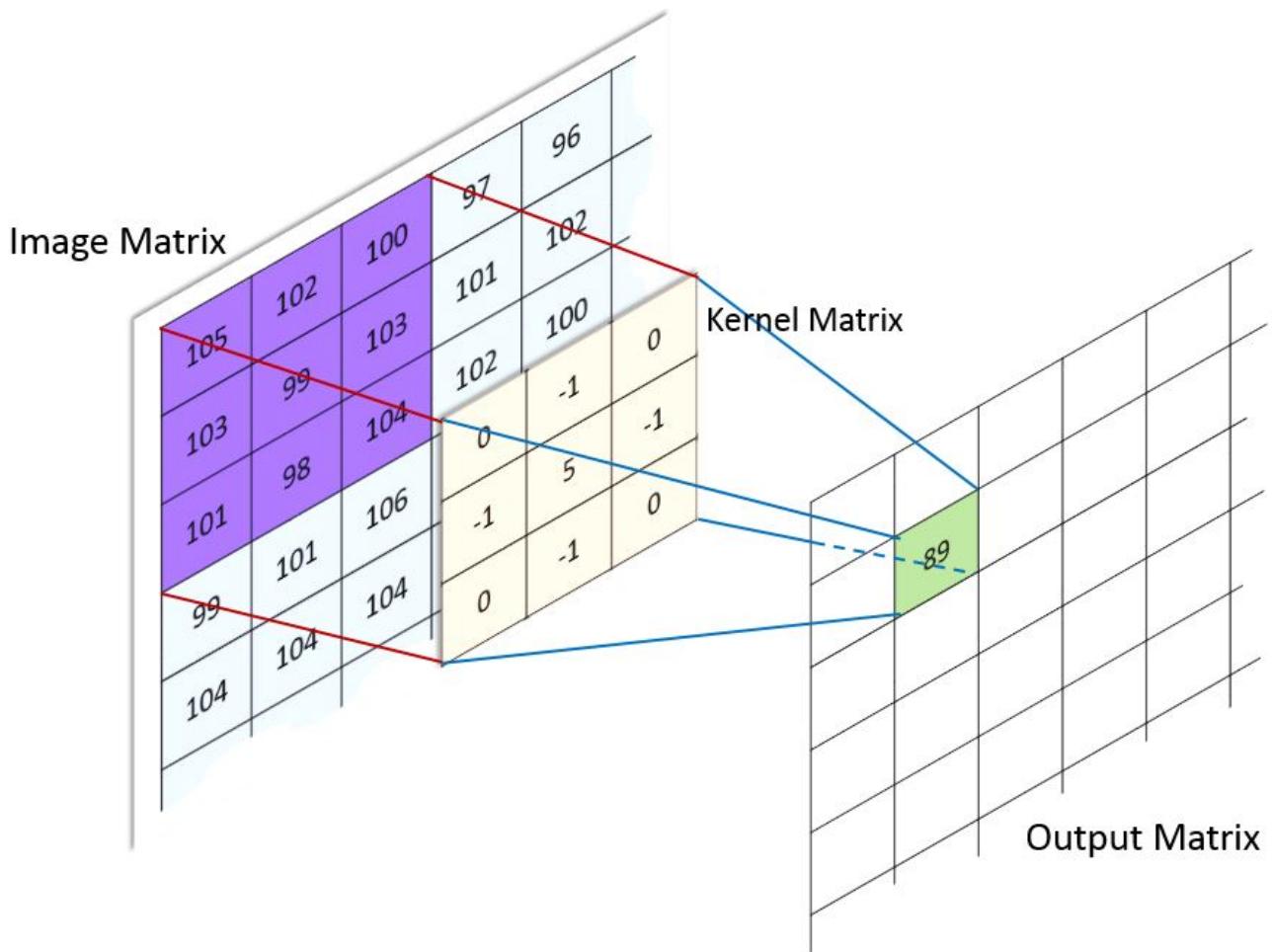
Image Filtering - 2D Convolution

Each convolution operation has a kernel which could be any matrix smaller than the original image in height and width. Each kernel is useful for a specific task, such as sharpening, blurring, edge detection, and more. Let's start with the sharpening kernel which is defined as:

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

The output of image convolution is calculated as follows:

- Flip the kernel both horizontally and vertically. As our selected kernel is symmetric, the flipped kernel is equal to the original.
- Put the first element of the kernel at every pixel of the image (element of the image matrix). Then each element of the kernel will stand on top of an element of the image matrix.



- To calculate the value of convolution output at pixel (2,2), center the kernel at that position on the image matrix.
- Multiply each element of the kernel with its corresponding element of the image matrix (the one which is overlapped with it)
- Sum up all product outputs and put the result at the same position in the output matrix as the center of kernel in image matrix.

Kernel Matrix

105	102	100	97	96	
103	99	103	101	102	
101	98	104	102	100	
99	101	106	104	99	
104	104	104	100	98	

0	-1	0
-1	5	-1
0	-1	0

		89	111		

Image Matrix

$$\begin{aligned}
 & 102 * 0 + 100 * -1 + 97 * 0 \\
 & + 99 * -1 + 103 * 5 + 101 * -1 \\
 & + 98 * 0 + 104 * -1 + 102 * 0 = 111
 \end{aligned}$$

Output Matrix

105	102	100	97	96	
103	99	103	101	102	
101	98	104	102	100	
99	101	106	104	99	
104	104	104	100	98	

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

		89	111		

Image Matrix

$$\begin{aligned}
 & 102 * 0 + 100 * -1 + 97 * 0 \\
 & + 99 * -1 + 103 * 5 + 101 * -1 \\
 & + 98 * 0 + 104 * -1 + 102 * 0 = 111
 \end{aligned}$$

Output Matrix

- For the pixels on the border of image matrix, some elements of the kernel will stand out of the image matrix and therefore does not have any corresponding element from the image matrix. In this case, we apply padding to the input matrix (based on the size of the kernel we might need one or more pixels padding, in our example we just need 1 pixel padding).

0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

210	89	111			

Image Matrix

$$\begin{aligned}
 & 0 * 0 + 105 * -1 + 102 * 0 \\
 & + 0 * -1 + 103 * 5 + 99 * -1 \\
 & + 0 * 0 + 101 * -1 + 98 * 0 = 210
 \end{aligned}$$

Output Matrix

0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

320					
210	89	111			

Image Matrix

$$\begin{aligned}
 & 0 * 0 + 0 * -1 + 0 * 0 \\
 & + 0 * -1 + 105 * 5 + 102 * -1 \\
 & + 0 * 0 + 103 * -1 + 99 * 0 = 320
 \end{aligned}$$

Output Matrix

As you can see, the output of convolution might violate the input range of [0-255]. Hence, any value above 255 is taken as 255 and the pixel is made white in colour.

Python:

```
cv2.filter2D(img, ddepth, kernel)
```

Parameters:

- img – input image
- ddepth – output image depth; the following combinations of ddepth are supported:
 - CV_8U/CV_8S (8-bit unsigned and signed integer).
 - CV_16U/CV_16S (16-bit unsigned and signed integer).
 - CV_32F (32-bit floating point).
 - CV_64F (32-bit floating point).
 - -1 to retain the same depth as input image.
- kernel – convolution matrix.

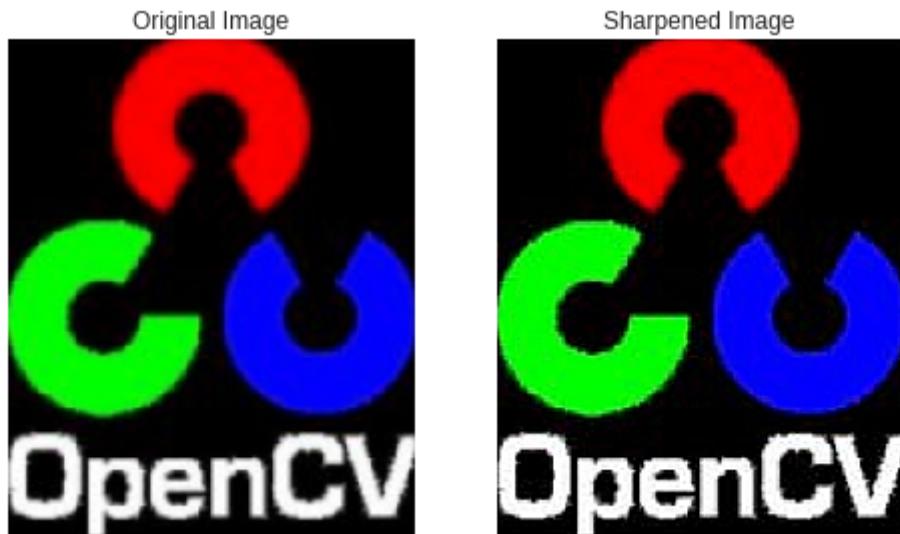
In [33]:

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

img = cv2.imread('opencv-logo.png')

kernel = np.matrix('0 -1 0 ; -1 5 -1 ; 0 -1 0')
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.axis("off"),plt.title("Original Image"),plt.imshow(img,cmap ='gray')
plt.subplot(122),plt.axis("off"),plt.title("Sharpened Image"),plt.imshow(dst,cmap='gray')
plt.show()
```



Smoothing Images - Image Blurring

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noises from the image. We will discuss the 3 majorly used image smoothing kernels here.

1) Box Filter

This is done by convolving image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replace the central element. This is done by the function cv2.blur().

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Python:

```
cv2.blur(img, ksize)
```

Parameters:

- img - input image.
- ksize - kernel size to be used for box filter.

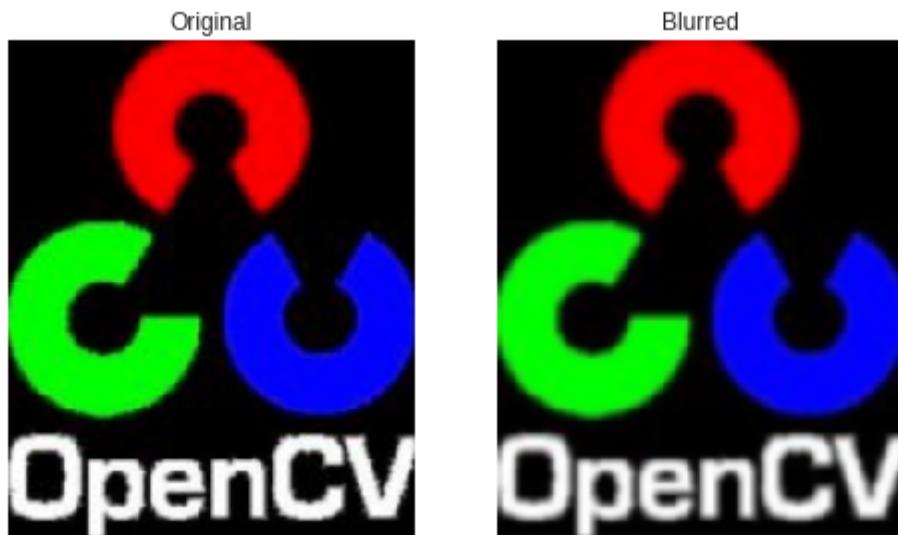
In [51]:

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

img = cv2.imread('opencv-logo.png',1)

blur = cv2.blur(img,(5,5))

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(blur,cmap='gray'),plt.axis('off'),plt.title('Blurred')
plt.show()
```



2) Gaussian Blurring

In this, instead of box filter, gaussian kernel is used. It is done with the function, cv2.GaussianBlur(). We should specify the width and height of kernel which should be positive and odd. We also should specify the standard deviation in X and Y direction, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as same as sigmaX. If both are given as zeros, default values are used depending on the kernel size. Gaussian blurring is highly effective in removing gaussian noise from the image.

Given below is the default 5×5 Gaussian filter used with a standard deviation of 1.4.

The asterisk below denotes a convolution operation.

$$\mathbf{B} = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}.$$

Python:

```
cv2.GaussianBlur(img, ksize, sigmaX, sigmaY)
```

Parameters:

- img - input image.
- ksize - kernel size to be used for box filter.
- sigmaX - standard deviation in x-direction.
- sigmaY - standard deviation in y-direction.

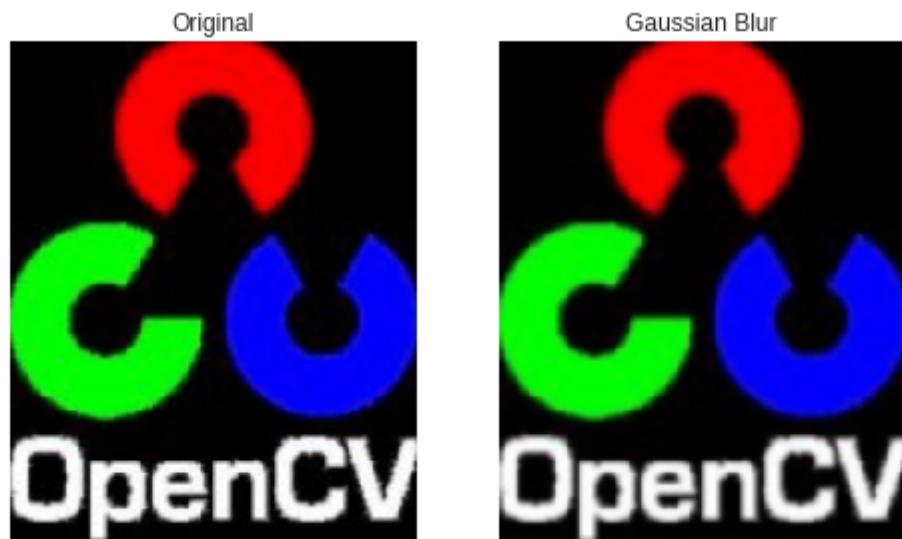
In [41]:

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

img = cv2.imread('opencv-logo.png')

blur = cv2.GaussianBlur(img,(5,5),0)

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(blur,cmap='gray'),plt.axis('off'),plt.title('Gaussian Blur')
plt.show()
```



3) Median Blurring

the function cv2.medianBlur() takes median of all the pixels under kernel area and central element is replaced with this median value. This is highly effective against **salt-and-pepper noise** in the images. Interesting thing is that, in the above filters, central element is a newly calculated value which may be a pixel value in the image or a new value. But in median blurring, 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.

Python:

```
cv2.medianBlur(img, ksize)
```

Parameters:

- img - input image.
- ksize - kernel size.

In [49] :

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

img = cv2.imread('saltnpepper.png')

median = cv2.medianBlur(img,5)

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(median,cmap='gray'),plt.axis('off'),plt.title('Median Blur')
plt.show()
```



Morphological Transformations

Morphological operations are a set of operations that process images based on shapes. They apply a structuring element to an input image and generate an output image.

The most basic morphological operations are two: Erosion and Dilation.

1) Erosion

Basics of Erosion:

- Erodes away the boundaries of foreground object.
- Used to diminish the features of an image.

Working of erosion:

- A kernel of odd size (3, 5 or 7) is convolved with the image.
- A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero).
- Thus all the pixels near boundary will be discarded depending upon the size of kernel.
- So the thickness or size of the foreground object decreases or simply white region decreases in the image.

Python:

```
cv2.erode(img, kernel, iterations)
```

Parameters:

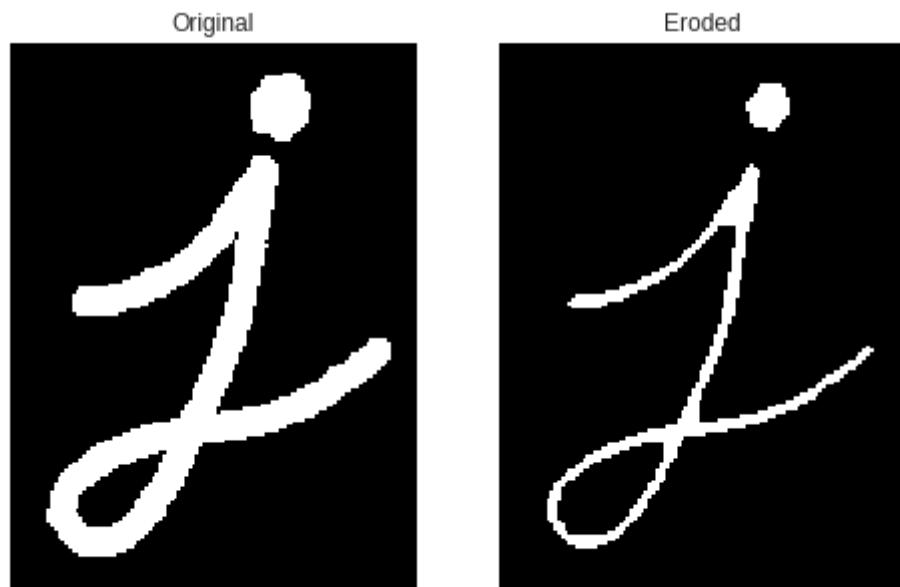
- img - Input image in **Binary**
 - kernel - The kernel used for erosion.
 - iterations - The number of times you want the convolution to happen. Multiple erosions can be combined into a single function using this argument. By default iterations = 1.
-

In [53]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('j.png')
kernel = np.ones((5,5))
erosion = cv2.erode(img, kernel, iterations = 1)

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(erosion,cmap='gray'),plt.axis('off'),plt.title('Eroded')
plt.show()
```



2) Dilation

Basics of dilation:

- Increases the object area.
- Used to accentuate features.

Working of dilation:

- A kernel of odd size (3, 5 or 7) is convolved with the image.
- A pixel element in the original image is '1' if atleast one pixel under the kernel is '1'.
- It increases the white region in the image or in other words, the size of the foreground object increases.

Python:

```
cv2.dilate(img, kernel, iterations)
```

Parameters:

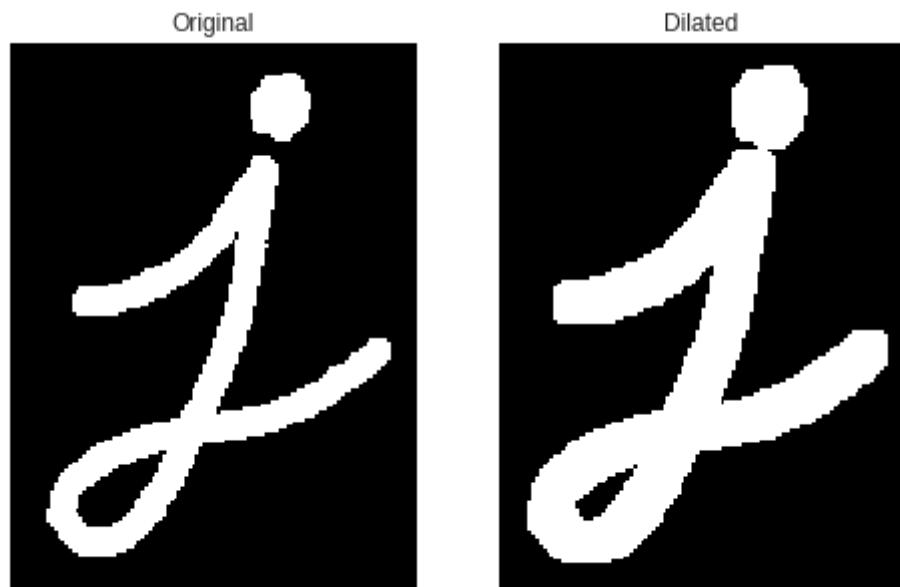
- img - Input image in **Binary**
 - kernel - The kernel used for dilation.
 - iterations - The number of times you want the convolution to happen. Multiple dilations can be combined into a single function using this argument. By default iterations = 1.
-

In [54]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('j.png')
kernel = np.ones((5,5))
dilation = cv2.dilate(img, kernel, iterations = 1)

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(dilation,cmap='gray'),plt.axis('off'),plt.title('Dilated')
plt.show()
```



3) Other Morphological Transformations

Opening

Opening is just erosion followed by dilation. It is useful in removing noise as seen in the image below.



Closing

Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small black points on the object.



Morphological Gradient

It is just the difference between dilation and erosion of an image. The result will look like the outline of the object.



Top Hat

It is the difference between input image and opening of the image. Below example is done for a 9x9 kernel.



Black Hat

It is the difference between the closing of the input image and input image.



Python:

```
cv2.morphologyEx(img, operation, kernel)
```

Parameters:

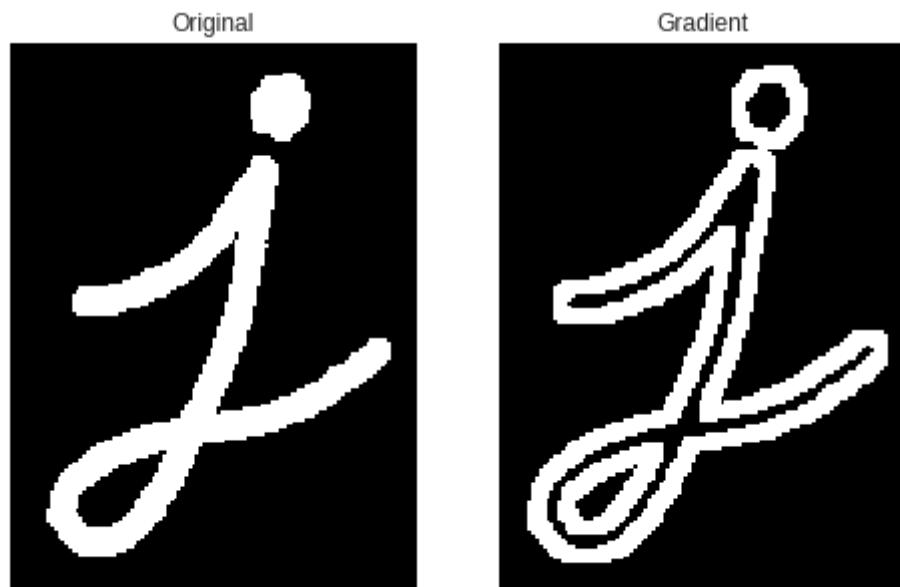
- img - Input image in **Binary**
- transform - The type of operation can take the following arguments:
 - cv2.MORPH_OPEN to implement opening.
 - cv2.MORPH_CLOSE to implement closing.
 - cv2.MORPH_GRADIENT to implement morphological gradient.
 - cv2.MORPH_TOPHAT to implement top hat.

In [55]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('j.png')
kernel = np.ones((5,5))
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)

plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(gradient,cmap='gray'),plt.axis('off'),plt.title('Gradient')
plt.show()
```



What are Features?

In computer vision and image processing, a feature is a piece of information which is relevant for solving the computational task related to a certain application.

There is no universal or exact definition of what constitutes a feature, and the exact definition often depends on the problem or the type of application. Given that, a feature is defined as an "interesting" part of an image, and features are used as a starting point for many computer vision algorithms.

Features maybe specific structures in the image such as points, edges or objects. Features may also be the result of a general neighbourhood operation or feature detection applied to the image.

Feature detection includes methods for computing abstractions of image information and making local decisions at every image point whether there is an image feature of a given type at that point or not. The resulting features will be subsets of the image domain, often in the form of isolated points, continuous curves or connected regions.

Feature detection is a low-level image processing operation. That is, it is usually performed as the first operation on an image, and examines every pixel to see if there is a feature present at that pixel. If this is part of a larger algorithm, then the algorithm will typically only examine the image in the region of the features. As a built-in pre-requisite to feature detection, the input image is usually smoothed by a Gaussian kernel in a scale-space representation and one or several feature images are computed, often expressed in terms of local image derivatives operations.

The feature concept is very general and the choice of features in a particular computer vision system may be highly dependent on the specific problem at hand.

Basic Image Features

Edges

Edges are points where there is a boundary (or an edge) between two image regions. In general, an edge can be of almost arbitrary shape, and may include junctions. In practice, edges are usually defined as sets of points in the image which have a strong gradient magnitude. Furthermore, some common algorithms will then chain high gradient points together to form a more complete description of an edge. These algorithms usually place some constraints on the properties of an edge, such as shape, smoothness, and gradient value.

Locally, edges have a one-dimensional structure.

Corners/Interest Points

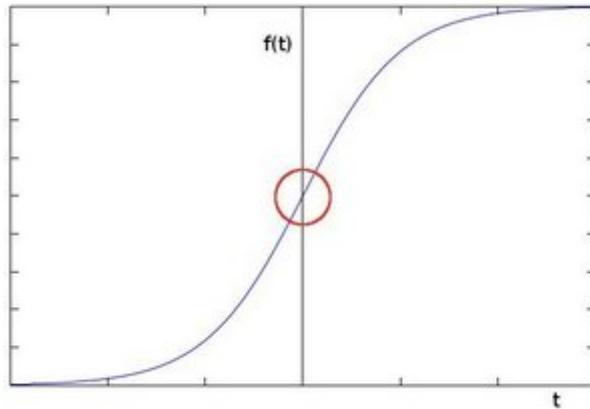
The terms corners and interest points are used somewhat interchangeably and refer to point-like features in an image, which have a local two dimensional structure. The name "Corner" arose since early algorithms first performed edge detection, and then analysed the edges to find rapid changes in direction (corners). These algorithms were then developed so that explicit edge detection was no longer required, for instance by looking for high levels of curvature in the image gradient. It was then noticed that the so-called corners were also being detected on parts of the image which were not corners in the traditional sense (for instance a small bright spot on a dark background may be detected). These points are frequently known as interest points, but the term "corner" is used by tradition.

Image Gradients

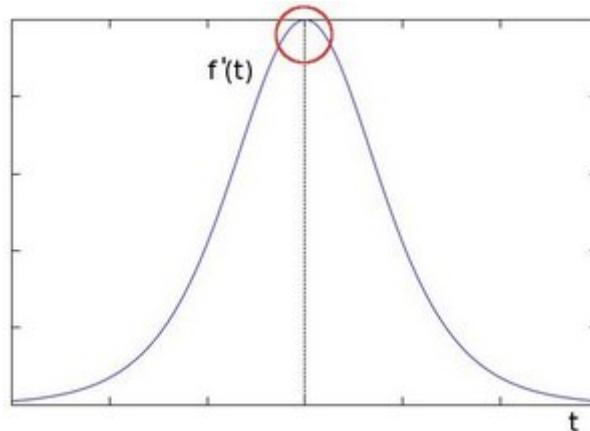
One of the most basic and important convolutions is the computation of derivatives (or approximations to them). There are many ways to do this, but only a few are well suited to a given situation.

Let's imagine we want to detect the edges present in the image. You can easily notice that in an edge, the pixel intensity changes in a notorious way. A good way to express changes is by using derivatives. A high change in gradient indicates a major change in the image.

Let's assume we have a 1D-image. An edge is shown by the "jump" in intensity as shown in the plot below:



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



So, from the explanation above, we can deduce that a method to detect edges in an image can be performed by locating pixel locations where the gradient is higher than its neighbors (or to generalize, higher than a threshold).

OpenCV provides three types of gradient filters or High-pass filters.

- Sobel
- Scharr
- Laplacian

We will see each one of them.

Sobel

The Sobel Operator is a discrete differentiation operator. It computes an approximation of the gradient of an image intensity function. The Sobel Operator combines Gaussian smoothing and differentiation.

Assuming that the image to be operated is I :

We calculate two derivatives:

Horizontal changes: This is computed by convolving I with a kernel of odd size as given below. For a kernel size of 3, G_x would be computed as the convolution of the following matrix with I .

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

Vertical changes: This is computed by convolving I with a kernel of odd size as given below. For a kernel size of 3, G_y would be computed as the convolution of the following matrix with I .

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

At each point of the image we calculate an approximation of the gradient at that point by combining both the results above:

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

Although sometimes the following simpler equation can also be used:

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

Python:

```
cv2.Sobel(src, ddepth, xorder, yorder, ksize)
```

Parameters:

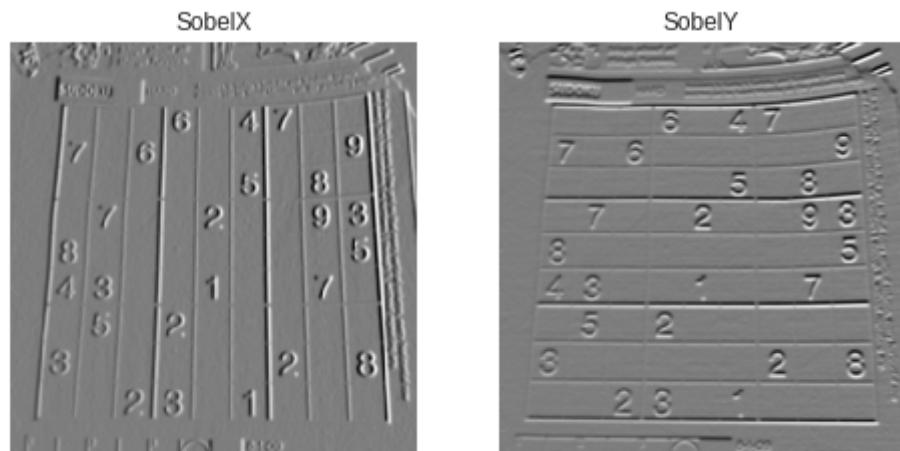
- src – input image in grayscale.
- ddepth – output image depth; the following combinations of ddepth are supported:
 - CV_8U
 - CV_16U/CV_16S
 - CV_32F
 - CV_64F
- xorder – order of the derivative x.
- yorder – order of the derivative y.
- ksize – size of the extended Sobel kernel; it must be 1, 3, 5, or 7.

In [56]:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('media/sudoku.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.GaussianBlur(img, (5, 5), 0)
sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)

plt.subplot(121), plt.imshow(sobelx, cmap='gray'), plt.axis('off'), plt.title('Sobel X')
plt.subplot(122), plt.imshow(sobely, cmap='gray'), plt.axis('off'), plt.title('Sobel Y')
plt.show()
```



Scharr

When the size of the kernel is 3, the Sobel kernel shown above may produce noticeable inaccuracies (after all, Sobel is only an approximation of the derivative). OpenCV addresses this inaccuracy for kernels of size 3 by using the Scharr function. This is as fast but more accurate than the standard Sobel function. It implements the following kernels:

Horizontal changes: G_x would be computed as the convolution of the following matrix with I.

$$\begin{bmatrix} -3 & 0 & +3 \\ -10 & 0 & +10 \\ -3 & 0 & +3 \end{bmatrix}$$

Vertical changes: G_y would be computed as the convolution of the following matrix with I.

$$\begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ +3 & +10 & +3 \end{bmatrix}$$

Python:

```
cv2.Scharr(src, ddepth, xorder, yorder)
```

Parameters:

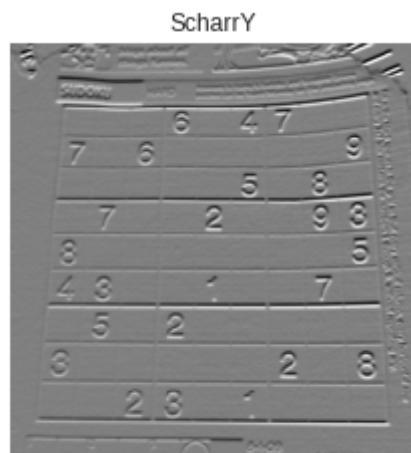
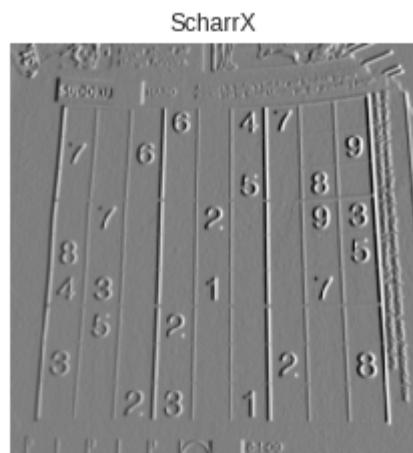
- src – input image in grayscale.
 - ddepth – output image depth; the following combinations of ddepth are supported:
 - CV_8U
 - CV_16U/CV_16S
 - CV_32F
 - CV_64F
 - xorder – order of the derivative x.
 - yorder – order of the derivative y.
-

In [57]:

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

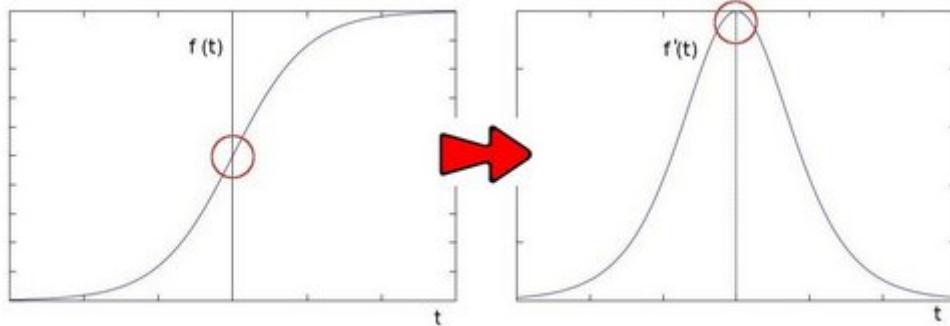
img = cv2.imread('media/sudoku.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.GaussianBlur(img, (5, 5), 0)
scharrx = cv2.Scharr(img, cv2.CV_64F, 1, 0)
scharry = cv2.Scharr(img, cv2.CV_64F, 0, 1)

plt.subplot(121), plt.imshow(scharrx, cmap='gray'), plt.axis('off'), plt.title('ScharrX')
plt.subplot(122), plt.imshow(scharry, cmap='gray'), plt.axis('off'), plt.title('ScharrY')
plt.show()
```

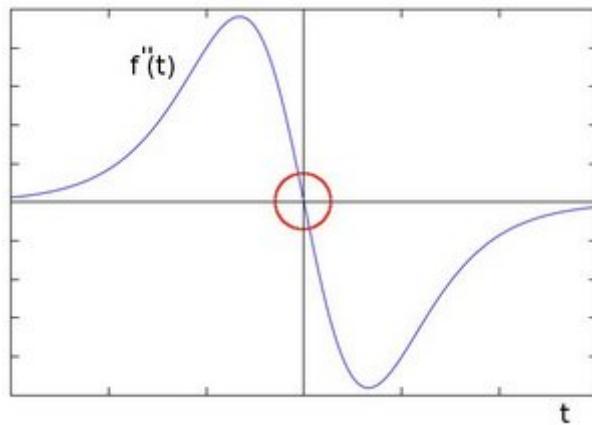


Laplacian

In the previous section we learned how to use the Sobel Operator. It was based on the fact that in the edge area, the pixel intensity shows a sudden jump. By getting the first derivative of the intensity, we observed that an edge is characterized by a maximum, as it can be seen in the figure:



What happens if we take the second derivative?



You can observe that the second derivative is zero! So, we can also use this criterion to attempt to detect edges in an image.

However, note that zeros will not only appear in edges, they can appear in other meaningless locations too due to noise. This can be easily solved by applying filters.

From the explanation above, we deduce that the second derivative can be used to detect edges. Since images are "2D", we would need to take the derivative in both dimensions.

The Laplacian operator is defined by:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Since the Laplacian uses the gradient of images, it calls internally the Sobel operator to perform its computation.

But when `ksize==1`, the Laplacian is computed by convolving the image with the following 3X3 kernel, as double gradient for a `ksize` of 1 returns a zero everywhere.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Python:

```
cv2.Laplacian(src, ddepth)
```

Parameters:

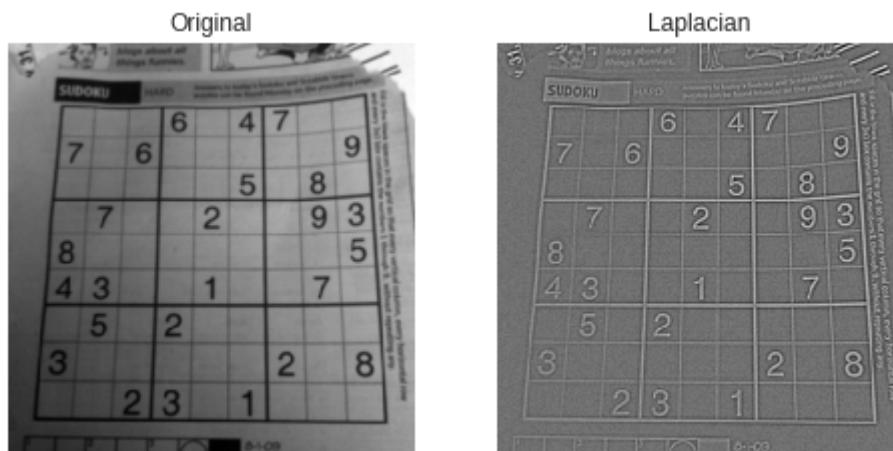
- src – input image in grayscale.
- ddepth – output image depth; the following combinations of ddepth are supported:
 - CV_8U
 - CV_16U/CV_16S
 - CV_32F
 - CV_64F

In [59]:

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

img = cv2.imread('media/sudoku.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.GaussianBlur(img, (5, 5), 0)
laplacian = cv2.Laplacian(img, cv2.CV_64F)

plt.subplot(121), plt.imshow(img, cmap='gray'), plt.axis('off'), plt.title('Original')
plt.subplot(122), plt.imshow(laplacian, cmap='gray'), plt.axis('off'), plt.title('Laplacian')
plt.show()
```



Edge Detection

The general criteria for edge detection include:

- **Low error rate:** This means that the detection should accurately catch as many edges in the image as possible.
- **Good localization:** The edge point detected from the operator should accurately localize on the center of the edge.
- **Minimal response:** A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

There are many methods for edge detection, but most of them can be grouped into two categories:

- **Search based methods:** These detect edges by first computing a measure of edge strength, usually the gradient magnitude, and then searching for local directional maximas of the gradient magnitude.
- **Zero-crossing based methods:** These search for zero crossings in a second-order derivative expression computed from the image, usually the zero-crossings of the Laplacian operator.

As a pre-processing step to edge detection, a smoothing stage, typically **Gaussian smoothing**, is almost always applied for noise reduction.

The various edge detection methods that have been published mainly differ in the types of smoothing filters applied and the way the measures of edge strength are computed.

The most popular and widely implemented algorithm for edge detection in Computer Vision is the Canny Edge Detection.

Canny Edge Detector

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986.

The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
 2. Find the intensity gradients of the image.
 3. Apply non-maximum suppression to get rid of spurious response to edge detection.
 4. Apply double threshold to determine potential edges.
 5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.
-

1) Gaussian filter

Since edge detection is susceptible to noise in the image, first step is to remove it with a 5x5 Gaussian filter. The Gaussian filter is applied to convolve with the image. This step will slightly smoothen the image to reduce the effects of obvious noise on the edge detector.

Given below is the 5x5 Gaussian filter generally used before implementing the Canny Edge Detection with a standard deviation of 1.4.

The asterisk below denotes a convolution operation.

$$\mathbf{B} = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}.$$

Note: The gaussian filter won't be implicitly implemented on a call to the cv2.Canny() function. Hence, make sure that the image is filtered before passing it on to the Canny function.

2) Finding intensity gradient of the image

Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical directions to get first derivative in horizontal direction (G_x) and vertical direction (G_y). From these two images, we can find edge gradient and direction for each pixel as follows:

$$\text{Edge_Gradient } (G) = \sqrt{G_x^2 + G_y^2}$$

$$\text{Angle } (\theta) = \tan^{-1} \left(\frac{G_y}{G_x} \right)$$

The edge direction angle is then rounded to one of four angles representing vertical, horizontal and the two diagonals (0° , 45° , 90° and 135°). For instance θ in $[0^\circ, 22.5^\circ]$ or $[157.5^\circ, 180^\circ]$ is mapped to 0° .

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. A threshold for the image gradients is set and only those pixels whose gradient magnitudes fall above this threshold are taken as edges.

3) Non-maximum suppression

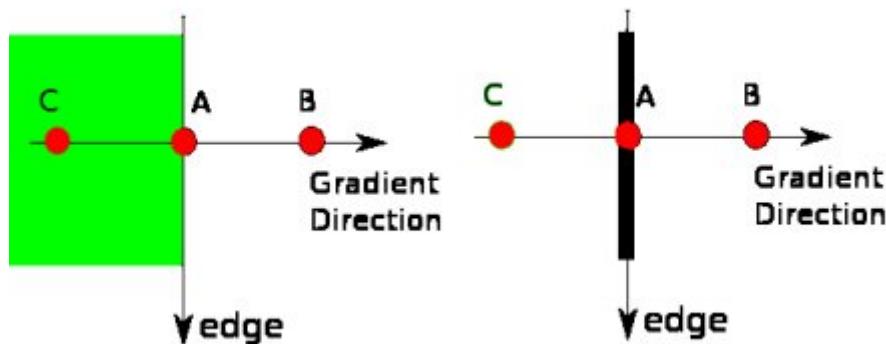
Non-maximum suppression is an **edge thinning technique**. After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. As minimal response is one of the important criterions for edge detection, there should only be one accurate response to the edge. Thus non-maximum suppression can help to suppress all the gradient values (by setting them to 0) except the local maxima, which indicate locations with the sharpest change of intensity value.

For this, every pixel is checked if it is a local maximum in the gradient direction.

The following algorithm is followed for each pixel in the gradient image.

- Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
- If the edge strength of the current pixel is the largest compared to both, the value will be preserved. Otherwise, the value will be suppressed.

It is by logic, we observe that the **gradient direction is always normal to the edge**.

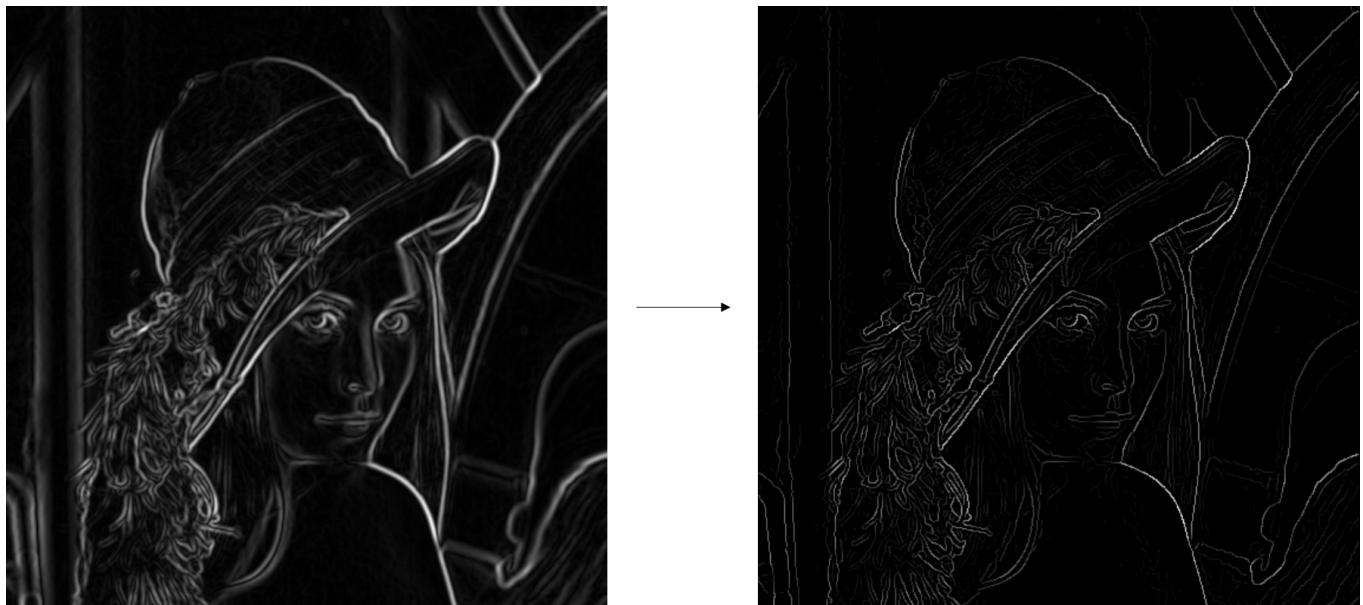


Point A is on the edge (in vertical direction). Point B and C are in gradient directions. So point A is checked with point B and C to see if it forms a local maximum. If so, it is considered for next stage, otherwise, it is suppressed (put to zero).

- If the rounded gradient angle is 0° (i.e. the edge is in the north-south direction), the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the east and west directions.
- If the rounded gradient angle is 90° (i.e. the edge is in the east-west direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north and south directions.
- If the rounded gradient angle is 135° (i.e. the edge is in the northeast-southwest direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north west and south-east directions.
- If the rounded gradient angle is 45° (i.e. the edge is in the north west–south east direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north east and south west directions.

Note that the sign of the direction is irrelevant, i.e. north–south is the same as south–north and so on.

In short, the result you get is a binary image with “thin edges” as shown below.



4) Double thresholding

After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value.

This is accomplished by selecting upper and lower threshold values.

- If the pixel gradient is higher than the upper threshold, it is marked as a strong edge pixel.
- If the pixel gradient is between the two thresholds, it is marked as a weak edge pixel.
- If the pixel gradient value is below the lower threshold, it will be suppressed.

The two threshold values are passed on as arguments while calling the Canny Edge Detector function.

It is recommended that the upper:lower ratio is maintained between 2:1 and 3:1 for good results.

5) Hysteresis

The strong edge pixels obtained from the above thresholding are accepted as "true-edges".

The weak edge pixels obtained from the above thresholding are classified as edges or non-edges based on their connectivity. If they are somehow connected to a strong edge pixel moving along the edge, they are considered to be "true-edges". Otherwise, they are also discarded.

So what we finally get are only the strong edges in the image.



Note: The input image passed on to the Canny function must be a grayscale image. The output image of the Canny function will be a Boolean (binary) grayscale image.

Python:

```
cv2.Canny(src, minValue, maxValue, ksize, L2gradient)
```

Parameters:

- **src** – input image in **grayscale**.
- **minVal** – lower threshold value for double thresholding.
- **maxVal** – upper threshold value for double thresholding.
- **ksize** – size of the Sobel kernel for finding gradients; default **ksize==3**.
- **L2gradient** – specifies the equation for finding gradient magnitude; default **L2gradient==False**.

If **L2gradient** is *True*, it uses $G = \sqrt{G_x^2 + G_y^2}$ for finding gradient magnitude.

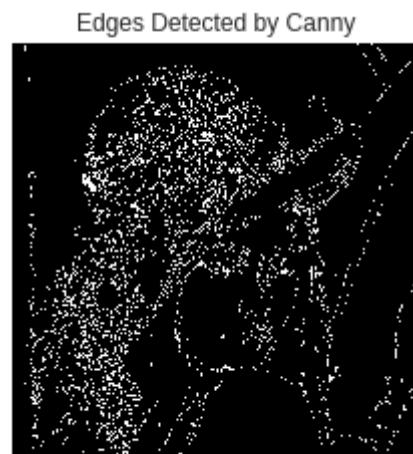
If **L2gradient** is *False*, it uses $G = |G_x| + |G_y|$ for finding gradient magnitude.

In [71]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('media/portrait.jpg')
imggray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(imggray, (5, 5), 0)
canny = cv2.Canny(blurred,20,50)

plt.subplot(121),plt.imshow(blurred,cmap='gray'),plt.axis('off'),plt.title('Original')
plt.subplot(122),plt.imshow(canny,cmap='gray'),plt.axis('off'),plt.title('Edges Detected by Canny')
plt.show()
```



Corner Detection

We already saw that corners are point-like features in an image, which have a local two dimensional structure. In other words, corners are regions in the image with large intensity variations in two perpendicular directions. Corners are generally invariant to translation, rotation and illumination.

The most widely implemented methods for corner detection in Computer Vision are the Harris and Shi-Tomasi algorithms.

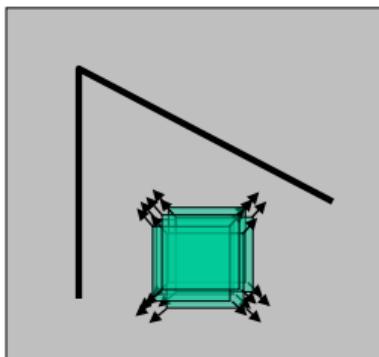
Let's discuss the basic algorithm used in both the methods.

Algorithm

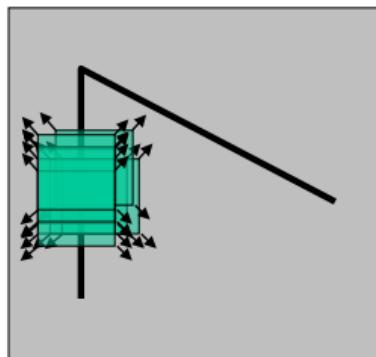
Each pixel in the image is checked to see if a corner is present by considering how similar a patch centered on the pixel is to nearby, largely overlapping patches.

The 3 basic spaces in an image can be distinguished as follows:

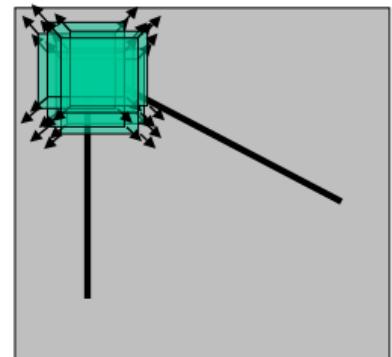
- If the pixel is in a flat region of uniform intensity, then the nearby patches will look similar.
- If the pixel is on an edge, then nearby patches in a direction perpendicular to the edge will look quite different, but nearby patches in a direction parallel to the edge will result only in a small change.
- If the pixel is on a corner/feature with variation in all directions, then none of the nearby patches will look similar.



“flat” region:
no change in
all directions



“edge”:
no change along
the edge direction



“corner”:
significant change
in all directions

The similarity between adjacent patches is measured by taking the sum of squared differences (SSD) between the corresponding pixels of two patches. A lower number indicates more similarity.

Without loss of generality, we will assume that a grayscale 2-dimensional image is used. Let the intensity of the pixel at (x, y) be given by $I(x, y)$. Consider taking two image patches; one centred around the pixel at (x, y) and the other shifted from the first patch by (u, v) . The weighted sum of squared differences (SSD) between these two patches is given by:

$$E(u, v) = \sum_x \sum_y w(x, y) (I(x + u, y + v) - I(x, y))^2$$

$w(a, b)$ is a window function and is either gaussian or rectangular(equal weight for all pixels in the window). It gives weights to pixels underneath the window. The size of the window function must be given while calling the corner detector and must be odd.

In the above expression, $I(x + u, y + v)$ can be approximated by Taylor expansion. Let I_x and I_y be the image intensity gradients in the x and y directions respectively found using the Sobel operator. This produces the following approximation.

$$\begin{aligned} I(x + u, y + v) &\approx I(x, y) + I_x(x, y)u + I_y(x, y)v \\ E(u, v) &\approx \sum_x \sum_y w(x, y) (I_x(x, y)u + I_y(x, y)v)^2 \end{aligned}$$

This can be written in matrix form as follows:

$$E(u, v) \approx [u \ v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

where,

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

From the Spectral theorem in linear algebra, we know that any symmetric matrix can be represented as follows

$$M = Q \cdot D \cdot Q^T$$

where Q is a rotation and D is a diagonal matrix of the eigen values λ_1 and λ_2 of the matrix M.

As $Q^T \cdot [u \ v]^T = [u' \ v']^T$ and $Q \cdot [u \ v] = [u' \ v']$,

$$E(u, v) = [u' \ v'] D \begin{bmatrix} u' \\ v' \end{bmatrix} = \lambda_1(u')^2 + \lambda_2(v')^2$$

Hence, $E(u, v)$ is large in both the directions u' and v' if both the eigen values λ_1 and λ_2 are large.

The following inferences can be made based on this argument:

- If $\lambda_1 \approx 0$ and $\lambda_2 \approx 0$ then this pixel (x, y) has no features of interest.
- If $\lambda_1 \approx 0$ and λ_2 has some large positive value or vice-versa, then an edge is found.
- If λ_1 and λ_2 both have large positive values, then a corner is found.

Hence, for detecting a Corner, both the eigen values must be large enough.Harris and Shi-Tomasi Corner Detectors maximize these eigen values in 2 different ways.

Harris Corner Detector

The exact computation of the eigenvalues is computationally expensive, since it requires the computation of a square root while solving a quadratic equation. The Harris Corner Detector uses a score to maximize these eigen values without calculating them individually.

As λ_1 and λ_2 are the eigen-values of M , we have the following properties.

$$\det(M) = \lambda_1 \lambda_2$$

$$\text{trace}(M) = \lambda_1 + \lambda_2$$

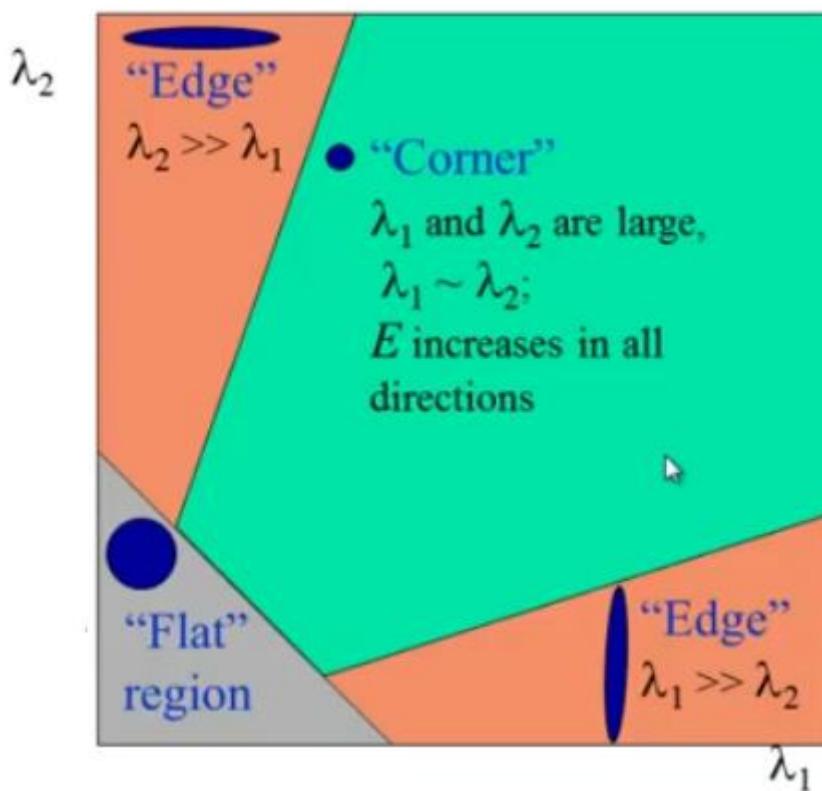
Let's define a score R as follows:

$$R = \det(M) - k (\text{trace}(M))^2$$

where k is an empirically determined constant between $0.04 - 0.06$.

- When $|R|$ is small, which happens when λ_1 and λ_2 are small, the region is flat.
- When $R < 0$, which happens when $\lambda_1 \gg \lambda_2$ or vice-versa, the region is an edge
- When R is large, which happens when λ_1 and λ_2 are large and $\lambda_1 \sim \lambda_2$, the region is a corner.

The image below illustrates this with λ_1 and λ_2 as axes.



The result of Harris Corner Detection is a grayscale image with this score as the intensity of that particular pixel. **Thresholding** for a suitable value of R gives the corners in the image.

Python:

```
cv2.cornerHarris(img, blockSize, ksize, k)
```

Parameters:

- img - Input image in grayscale.
- blockSize - Size of neighbourhood patch or window size considered for corner detection.

- ksize - Kernel size for taking the Sobel derivative.
- k - Empirical constant to be used in the Harris detector; k should lie between 0.04 and 0.06 for good results.

In [75]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('chess.jpg')
plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
img1 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ch = cv2.cornerHarris(img1,3,3,0.05)
img[ch>0.01*ch.max()]=[0,0,0] #Thresholding

plt.subplot(122),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Harris Corners')
plt.show()
```

Original



Harris Corners

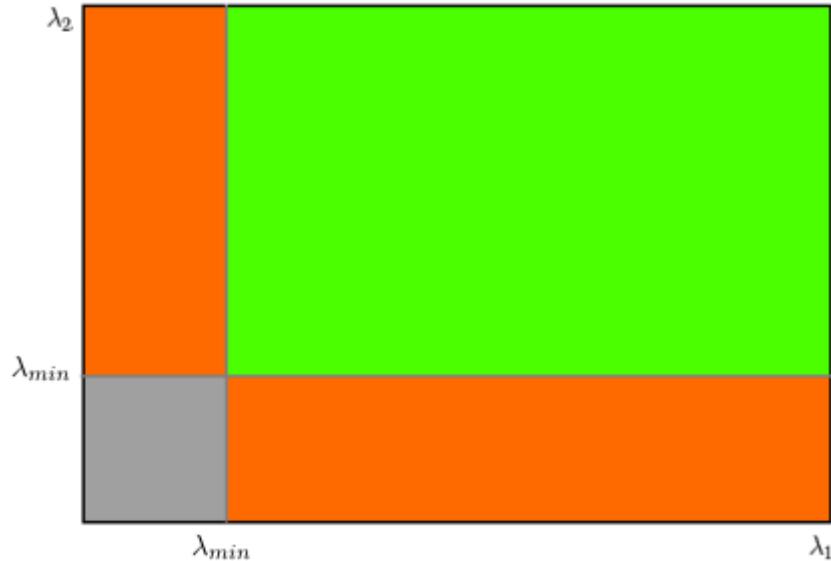


Shi-Tomasi Corner Detector

The Shi-Tomasi Corner Detector uses a different score to find the corners.

$$R = \min(\lambda_1, \lambda_2)$$

The value of this score is thresholded to find required corners in the image.



From the figure, we can see that only when both λ_1 and λ_2 are above a minimum value, λ_{min} , it is considered as a corner(green region).

Python:

```
cv2.goodFeaturesToTrack(img, maxCorners, qualityLevel, minDistance)
```

It finds N(maxCorners) strongest corners in the image by Shi-Tomasi method.

Parameters:

- img - Input image in grayscale.
- maxCorners – Maximum number of corners to return.
- qualityLevel – A value between 0-1, which denotes the minimum quality of corner below which everything is rejected. This value is multiplied with the maximum value of R in the image to get the quality measure. Any corner with R less than this quality measure is rejected as a corner.
- minDistance – Minimum possible Euclidean distance between the returned corners.

With all these information, the function finds corners in the image. All corners below quality measure are rejected. Then it sorts the remaining corners based on quality in the descending order. Then the function takes the most strongest corner, throws away all the nearby corners in the range of minimum distance and finally returns the N strongest corners in the image.

In [77]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('cube.jpg')
plt.subplot(121),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Original')
img1 = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(img1,30,0.01,30)

for i in corners:
    x,y = i.ravel()
    cv2.circle(img,(x,y),5,(0,255,0),-1)

plt.subplot(122),plt.imshow(img,cmap='gray'),plt.axis('off'),plt.title('Shi-Tomasi Corners')
plt.show()
```

