Computer Vision: Open CV

Manual Template

Author: Layth Ayache

Contents

[1. Introduction 2](#_Toc200207996)

[2. OpenCV Installation 2](#_Toc200207997)

[2.1 Environment Setup 2](#_Toc200207998)

[2.2 Installation Steps 3](#_Toc200207999)

[2.3 Verification 3](#_Toc200208000)

[3. Getting Started with Images 4](#_Toc200208001)

[3.1 Loading & Display 4](#_Toc200208002)

[3.2 Image Properties 6](#_Toc200208003)

[4. Basic Image Manipulation 9](#_Toc200208004)

[4.1 Color Channels 9](#_Toc200208005)

[4.2 Color Conversions 10](#_Toc200208006)

[Conclusion 11](#_Toc200208007)

# 1. Introduction

The objective of this report is to showcase the full extent of the OpenCV library, and to guide any person that reads this report on how to use it and do the following :

1. OpenCV Installation
2. Getting Started with Images
3. Basic Image Manipulation
4. Image Annotation
5. Image Enhancement
6. Accessing the Camera
7. Video Writing

The content of this report were directly taken from the OpenCV bootcamp course :” <https://courses.opencv.org/courses/course-v1:OpenCV+Bootcamp+CV0/course/> “

# 2. OpenCV Installation

## 2.1 Environment Setup

A **Python environment** is an isolated workspace containing its own Python interpreter and libraries, so different projects don’t conflict. **Anaconda** is a popular distribution that includes the Conda package manager plus many scientific libraries, making it easy to create and manage these environments.

**Alternatives to Anaconda**

* Python’s built-in venv module
* virtualenv (stand-alone virtual environments)
* pyenv (manages multiple Python versions & environments)

the following will be the steps for anaconda in windowns:

PS: these steps differ depending on the OS being used (MAC OS is different than Linux and is different than Windows)

First step is to download the Anaconda installer and run it:

**Anaconda** or **Miniconda** installed (download from <https://docs.conda.io/>)

Then you want to create a Conda environment:  
**Open** your terminal (macOS/Linux) or Anaconda Prompt (Windows):

And type:   
conda create -n vision python=3.9 < - -this will create an environment called vision using python 3.9

Then type:  
conda activate vision < - - this will activate the environment

## 2.2 Installation Steps

The next step of our process is actually downloading the OpenCV library in the vision environment:  
conda install -c conda-forge opencv < - - this will install the latest installation of OpenCV

And believe it or not, this is enough to download opencv!!!

## 2.3 Verification

Now to verify that openCv was correctly downloaded  
open a python file and try to import cv2,   
if it works then OpenCV has been correctly downloaded on your environment

# 3. Getting Started with Images

## 3.1 Loading & Display

First we need to import the libraries, then we need to load and display the images,

In python:

*# Display any image.*

Image(filename**=**"NameofImage.formatofimage")

Now lets Use OpenCV:  
#display any image using cv2

cv2.imread( filename[, flags] )

you can add a variable

retval = cv2.imread( filename[, flags] )

retval: Is the image if it is successfully loaded. Otherwise it is None. This may happen if the filename is wrong or the file is corrupt.

The function has **1 required input argument** and one optional flag:

1. filename: This can be an **absolute** or **relative** path. This is a **mandatory argument**.
2. Flags: These flags are used to read an image in a particular format (for example, grayscale/color/with alpha channel). This is an **optional argument** with a default value of cv2.IMREAD\_COLOR or 1 which loads the image as a color image.

Before we proceed with some examples, let's also have a look at some of the flags available.

**Flags**

1. **cv2.IMREAD\_GRAYSCALE** or **0**: Loads image in grayscale mode
2. **cv2.IMREAD\_COLOR** or **1**: Loads a color image. Any transparency of image will be neglected. It is the default flag.
3. **cv2.IMREAD\_UNCHANGED** or **-1**: Loads image as such including alpha channel.

Example:  
 this is the picture we are using for this example, named “checkerboard\_18x18.png”

# Read image as gray scale.

cb\_img = cv2.imread("checkerboard\_18x18.png", 0) < - - - the 0 flag means gray scale

# Print the image data (pixel values), element of a 2D numpy array.

# Each pixel value is 8-bits [0,255] < - - - since it is in gray scale

print(cb\_img)

the output will be :  
[[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[255 255 255 255 255 255 0 0 0 0 0 0 255 255 255 255 255 255]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]

[ 0 0 0 0 0 0 255 255 255 255 255 255 0 0 0 0 0 0]]

## 3.2 Image Properties

Now after we learned how to load the image, lets study it’s properties

# print the size  of image  
print("Image size (H, W) is:", cb\_img.shape)

🡪Image size (H, W) is: (18, 18)

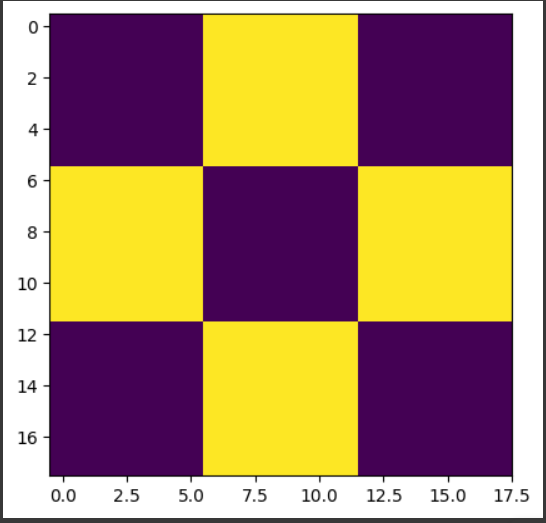
# print data-type of image  
print("Data type of image is:", cb\_img.dtype)

🡪Data type of image is: uint8

Display Images using Matplotlib

# Display image.

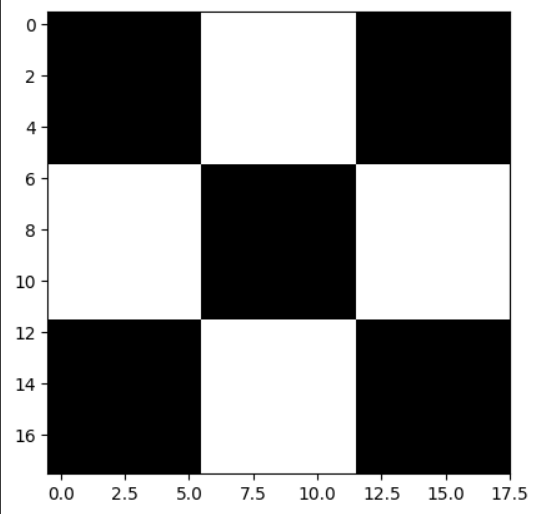
plt.imshow(cb\_img)



Even though the image was read in as a gray scale image, it won't necessarily display in gray scale when using imshow(). matplotlib uses different color maps and it's possible that the gray scale color map is not set.

# Set color map to gray scale for proper rendering.

plt.imshow(cb\_img, cmap="gray")



Until now, we have been using gray scale images in our discussion. Let us now discuss color images:  
# Read and display Coca-Cola logo.

Image("coca-cola-logo.png")



Let us read a color image and check the parameters. Note the image dimension.  
# Read in image

coke\_img = cv2.imread("coca-cola-logo.png", 1)

# print the size  of image

print("Image size (H, W, C) is:", coke\_img.shape)

# print data-type of image

print("Data type of image is:", coke\_img.dtype)

🡪Image size (H, W, C) is: (700, 700, 3)

🡪Data type of image is: uint8

PS: note that opencv expect the numbers as BGR and matplotlib expects them in RGB

# 4. Basic Image Manipulation

## 4.1 Color Channels

* **cv2.split()** Divides a multi-channel array into several single-channel arrays.
* **cv2.merge()** Merges several arrays to make a single multi-channel array. All the input matrices must have the same size.

# Split the image into the B,G,R components

img\_NZ\_bgr = cv2.imread("New\_Zealand\_Lake.jpg", cv2.IMREAD\_COLOR)

b, g, r = cv2.split(img\_NZ\_bgr)

# Show the channels

plt.figure(figsize=[20, 5])

plt.subplot(141);plt.imshow(r, cmap="gray");plt.title("Red Channel")

plt.subplot(142);plt.imshow(g, cmap="gray");plt.title("Green Channel")

plt.subplot(143);plt.imshow(b, cmap="gray");plt.title("Blue Channel")

# Merge the individual channels into a BGR image

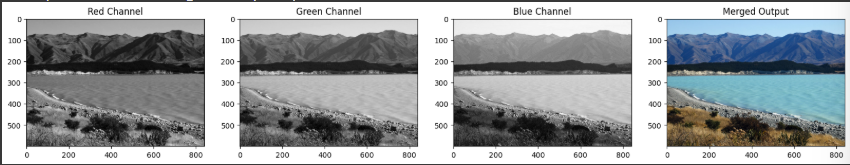
imgMerged = cv2.merge((b, g, r))

# Show the merged output

plt.subplot(144)

plt.imshow(imgMerged[:, :, ::-1])

plt.title("Merged Output")



## 4.2 Color Conversions

**cv2.cvtColor()** Converts an image from one color space to another. The function converts an input image from one color space to another. In case of a transformation to-from RGB color space, the order of the channels should be specified explicitly (RGB or BGR). Note that the default color format in OpenCV is often referred to as RGB but it is actually BGR (the bytes are reversed). So the first byte in a standard (24-bit) color image will be an 8-bit Blue component, the second byte will be Green, and the third byte will be Red. The fourth, fifth, and sixth bytes would then be the second pixel (Blue, then Green, then Red), and so on.

dst = cv2.cvtColor( src, code )

dst: Is the output image of the same size and depth as src.

The function has **2 required arguments**:

1. src input image: 8-bit unsigned, 16-bit unsigned ( CV\_16UC... ), or single-precision floating-point.
2. code color space conversion code (see ColorConversionCodes).

# OpenCV stores color channels in a differnet order than most other applications (BGR vs RGB).

img\_NZ\_rgb = cv2.cvtColor(img\_NZ\_bgr, cv2.COLOR\_BGR2RGB)

plt.imshow(img\_NZ\_rgb)

# Image manipulation

## Accessing Individual Pixel

Let us see how to access a pixel in the image.

For accessing any pixel in a numpy matrix, you have to use matrix notation such as matrix[r,c], where the r is the row number and c is the column number. Also note that the matrix is 0-indexed.

For example, if you want to access the first pixel, you need to specify matrix[0,0]. Let us see with some examples. We will print one black pixel from top-left and one white pixel from top-center.

# print the first pixel of the first black box  
print(cb\_img[0, 0])

# print the first white pixel to the right of the first black box  
print(cb\_img[0, 6])

## Modifying Image Pixels

cb\_img\_copy = cb\_img.copy()

cb\_img\_copy[2, 2] = 200

cb\_img\_copy[2, 3] = 200

cb\_img\_copy[3, 2] = 200

cb\_img\_copy[3, 3] = 200

# Same as above

# cb\_img\_copy[2:3,2:3] = 200

plt.imshow(cb\_img\_copy, cmap="gray")

print(cb\_img\_copy)

what this code does is change the values of the 3rd(index 2) and 4th(index 3) column of the pixel arrays and the respective rows as well. From their previous value to 200 (in the gray scale)

## 5.3 Cropping images

Cropping an image is simply achieved by selecting a specific (pixel) region of the image

img\_NZ\_bgr = cv2.imread("New\_Zealand\_Boat.jpg", cv2.IMREAD\_COLOR)

img\_NZ\_rgb = img\_NZ\_bgr[:, :, ::-1]

plt.imshow(img\_NZ\_rgb)

Crop out the middle region of the image:

cropped\_region = img\_NZ\_rgb[200:400, 300:600]

plt.imshow(cropped\_region)

## Resizing Images

The function **resize** resizes the image src down to or up to the specified size. The size and type are derived from the src,dsize,fx, and fy.

dst = resize( src, dsize[, dst[, fx[, fy[, interpolation]]]] )

dst: output image; it has the size dsize (when it is non-zero) or the size computed from src.size(), fx, and fy; the type of dst is the same as of src.

The function has **2 required arguments**:

1. src: input image
2. dsize: output image size

Optional arguments that are often used include:

1. fx: Scale factor along the horizontal axis; when it equals 0, it is computed as (𝚍𝚘𝚞𝚋𝚕𝚎)𝚍𝚜𝚒𝚣𝚎.𝚠𝚒𝚍𝚝𝚑/𝚜𝚛𝚌.𝚌𝚘𝚕𝚜
2. fy: Scale factor along the vertical axis; when it equals 0, it is computed as (𝚍𝚘𝚞𝚋𝚕𝚎)𝚍𝚜𝚒𝚣𝚎.𝚑𝚎𝚒𝚐𝚑𝚝/𝚜𝚛𝚌.𝚛𝚘𝚠𝚜

The output image has the size dsize (when it is non-zero) or the size computed from src.size(), fx, and fy; the type of dst is the same as of src.

We can resize an image using multiple methods, a simple method is by specifying the scaling factor of both fx and fy:

resized\_cropped\_region\_2x = cv2.resize(cropped\_region, None, fx=2, fy=2)

plt.imshow(resized\_cropped\_region\_2x)

Another method is by specifying the exact size of the output image:  
desired\_width = 100

desired\_height = 200

dim = (desired\_width, desired\_height)

# Resize background image to save size as logo image

resized\_cropped\_region = cv2.resize(cropped\_region, dsize=dim, interpolation=cv2.INTER\_AREA)

plt.imshow(resized\_cropped\_region)

now if we want to maintain the same aspect ratio we just add these lines  
# Method 2: Using 'dsize'

desired\_width = 100

aspect\_ratio = desired\_width / cropped\_region.shape[1] < - - this determines the aspect\_ratio

desired\_height = int(cropped\_region.shape[0] \* aspect\_ratio)

dim = (desired\_width, desired\_height)

# Resize image

resized\_cropped\_region = cv2.resize(cropped\_region, dsize=dim, interpolation=cv2.INTER\_AREA)

plt.imshow(resized\_cropped\_region)

## 5.4 Flipping Images

The function **flip** flips the array in one of three different ways (row and column indices are 0-based):

Function Syntax

dst = cv.flip( src, flipCode )

dst: output array of the same size and type as src.

The function has **2 required arguments**:

1. src: input image
2. flipCode: a flag to specify how to flip the array; 0 means flipping around the x-axis and positive value (for example, 1) means flipping around y-axis. Negative value (for example, -1) means flipping around both axes.

# Image Annotation

## Drawing a Line

Let's start off by drawing a line on an image. We will use cv2.line function for this.

img = cv2.line(img, pt1, pt2, color[, thickness[, lineType[, shift]]])

img: The output image that has been annotated.

The function has **4 required arguments**:

1. img: Image on which we will draw a line
2. pt1: First point(x,y location) of the line segment
3. pt2: Second point of the line segment
4. color: Color of the line which will be drawn < - - remember that it is in BGR format

Other optional arguments that are important for us to know include:

1. thickness: Integer specifying the line thickness. Default value is 1.
2. lineType: Type of the line. Default value is 8 which stands for an 8-connected line. Usually, cv2.LINE\_AA (antialiased or smooth line) is used for the lineType

## 6.2 Drawing a Circle

Let's start off by drawing a circle on an image. We will use cv2.circle function for this.

Functional syntx

img = cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]])

img: The output image that has been annotated.

The function has **4 required arguments**:

1. img: Image on which we will draw a line
2. center: Center of the circle
3. radius: Radius of the circle
4. color: Color of the circle which will be drawn

Next, let's check out the (optional) arguments which we are going to use quite extensively.

1. thickness: Thickness of the circle outline (if positive). If a negative value is supplied for this argument, it will result in a filled circle.
2. lineType: Type of the circle boundary. This is exact same as lineType argument in **cv2.line**

## Drawing a Rectangle

We will use **cv2.rectangle** function to draw a rectangle on an image. The function syntax is as follows.

Functional syntx

img = cv2.rectangle(img, pt1, pt2, color[, thickness[, lineType[, shift]]])

img: The output image that has been annotated.

The function has **4 required arguments**:

1. img: Image on which the rectangle is to be drawn.
2. pt1: Vertex of the rectangle. Usually we use the **top-left vertex** here.
3. pt2: Vertex of the rectangle opposite to pt1. Usually we use the **bottom-right** vertex here.
4. color: Rectangle color

Next, let's check out the (optional) arguments which we are going to use quite extensively.

1. thickness: Thickness of the rectangle outline (if positive). If a negative value is supplied for this argument, it will result in a filled rectangle.
2. lineType: Type of the rectangle boundary. This is exact same as lineType argument in **cv2.line**

## 6.4 Adding Text

Finally, let's see how we can write some text on an image using **cv2.putText** function.

Functional syntx

img = cv2.putText(img, text, org, fontFace, fontScale, color[, thickness[, lineType[, bottomLeftOrigin]]])

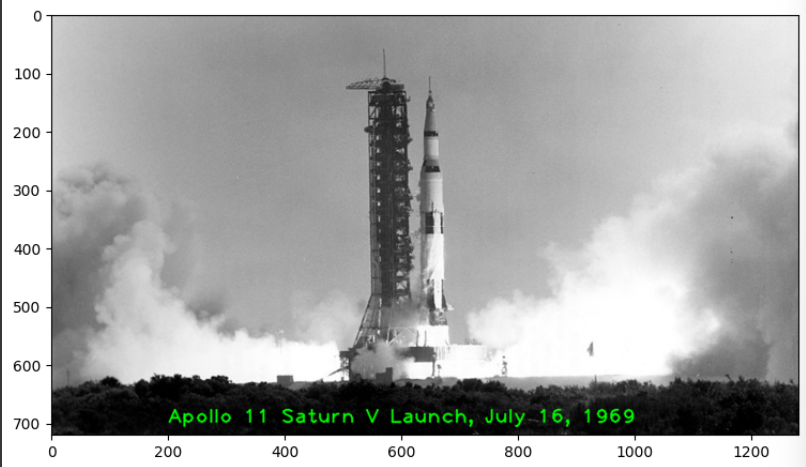
img: The output image that has been annotated.

The function has **6 required arguments**:

1. img: Image on which the text has to be written.
2. text: Text string to be written.
3. org: Bottom-left corner of the text string in the image.
4. fontFace: Font type
5. fontScale: Font scale factor that is multiplied by the font-specific base size.
6. color: Font color

Other optional arguments that are important for us to know include:

1. thickness: Integer specifying the line thickness for the text. Default value is 1.
2. lineType: Type of the line. Default value is 8 which stands for an 8-connected line. Usually, cv2.LINE\_AA (antialiased or smooth line) is used for the lineType.



# Image Enhancement

## 7.1 Brightness

The first operation we discuss is simple addition of images. This results in increasing or decreasing the brightness of the image since we are eventually increasing or decreasing the intensity values of each pixel by the same amount. So, this will result in a global increase/decrease in brightness.

So to make it brighter we add the images, and to make it darker we substract



## Contrast

Now contrast is basically multiplication, just like addition can result in brightness change, multiplication can be used to improve the contrast of the image.

Contrast is the difference in the intensity values of the pixels of an image. Multiplying the intensity values with a constant can make the difference larger or smaller ( if multiplying factor is < 1 ).



Notice we had to transfer the values from 8 bits to floats then back to int(8bits), now you can also see some dark cloud when the contrast is higher, the reason is simple, the clouds in the original photo were probably close to 255 so when we turned them into floats and multiplied the number by our matrix, we probably got very high numbers, then when we returned to 8 bits (int) we exceeded the 255 limit so the numbers relooped to low numbers, basically close to zero and thus creating black clouds.  
now to fix this we use the np.clip overflow function  


## Image Thresholding

Binary Images have a lot of use cases in Image Processing. One of the most common use cases is that of creating masks. Image Masks allow us to process on specific parts of an image keeping the other parts intact. Image Thresholding is used to create Binary Images from grayscale images. You can use different thresholds to create different binary images from the same original image.

Function Syntax

retval, dst = cv2.threshold( src, thresh, maxval, type[, dst] )

dst: The output array of the same size and type and the same number of channels as src.

The function has **4 required arguments**:

1. src: input array (multiple-channel, 8-bit or 32-bit floating point).
2. thresh: threshold value.
3. maxval: maximum value to use with the THRESH\_BINARY and THRESH\_BINARY\_INV thresholding types.
4. type: thresholding type (see ThresholdTypes).

Basically what it’s trying to do, is to isolate certain outlines or colors depending on how dark they are (above a certain threshold/number of intensity)

But this can lead to over thresholding and under thresholding, either the image will not have a consistent brightness along the page or maybe there isn’t a global threshold that works, so we move over to an adaptive threshold,   
Suppose you wanted to build an application that could read (decode) sheet music. This is similar to Optical Character Recognigition (OCR) for text documents where the goal is to recognize text characters. In either application, one of the first steps in the processing pipeline is to isolate the important information in the image of a document (separating it from the background). This task can be accomplished with thresholding techniques. Let's take a look at an example.  
# Read the original image

img\_read = cv2.imread("Piano\_Sheet\_Music.png", cv2.IMREAD\_GRAYSCALE)

# Perform global thresholding

retval, img\_thresh\_gbl\_1 = cv2.threshold(img\_read, 50, 255, cv2.THRESH\_BINARY)

# Perform global thresholding

retval, img\_thresh\_gbl\_2 = cv2.threshold(img\_read, 130, 255, cv2.THRESH\_BINARY)

# Perform adaptive thresholding

img\_thresh\_adp = cv2.adaptiveThreshold(img\_read, 255, cv2.ADAPTIVE\_THRESH\_MEAN\_C, cv2.THRESH\_BINARY, 11, 7)

# Show the images

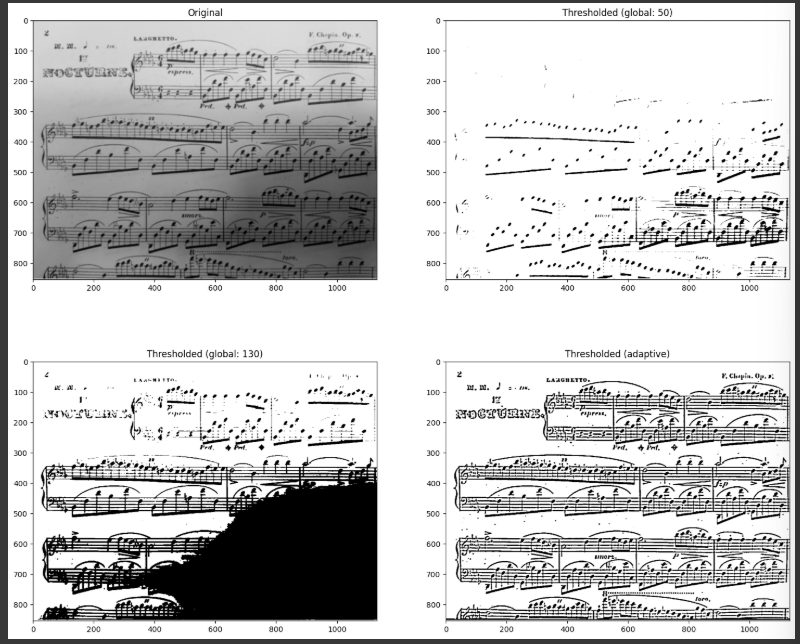
plt.figure(figsize=[18,15])

plt.subplot(221); plt.imshow(img\_read,        cmap="gray");  plt.title("Original");

plt.subplot(222); plt.imshow(img\_thresh\_gbl\_1,cmap="gray");  plt.title("Thresholded (global: 50)");

plt.subplot(223); plt.imshow(img\_thresh\_gbl\_2,cmap="gray");  plt.title("Thresholded (global: 130)");

plt.subplot(224); plt.imshow(img\_thresh\_adp,  cmap="gray");  plt.title("Thresholded (adaptive)");



## bitwise Operations

Function Syntax

Example API for cv2.bitwise\_and(). Others include: cv2.bitwise\_or(), cv2.bitwise\_xor(), cv2.bitwise\_not()

dst = cv2.bitwise\_and( src1, src2[, dst[, mask]] )

dst: Output array that has the same size and type as the input arrays.

The function has **2 required arguments**:

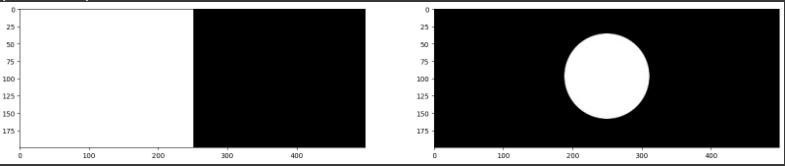
1. src1: first input array or a scalar.
2. src2: second input array or a scalar.

An important optional argument is:

1. mask: optional operation mask, 8-bit single channel array, that specifies elements of the output array to be changed.

Basically the logic gates approach of pixels,

These are some examples



The two original images (binary images),

Lets try to play around with the operators  
this is an AND operator  
A black and white image of a half moon

AI-generated content may be incorrect.

This is an OR operator,  
A black and white graph

AI-generated content may be incorrect.

this is a XOR operator:

A black and white circle with numbers

AI-generated content may be incorrect.

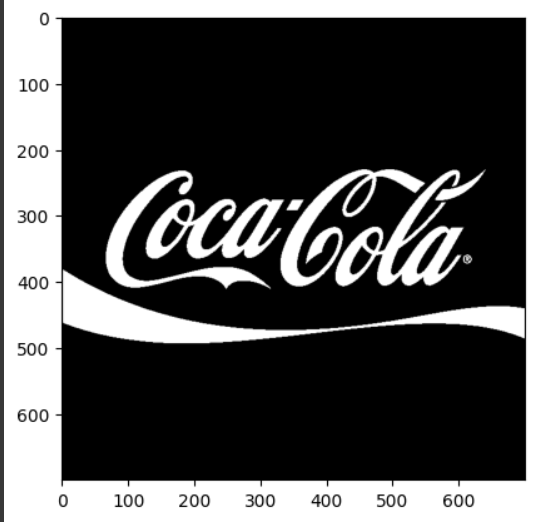
Lets try for example to show you how this works in real life setting,

Application: Logo Manipulation

In this section we will show you how to fill in the white lettering of the Coca-Cola logo below with a background image.

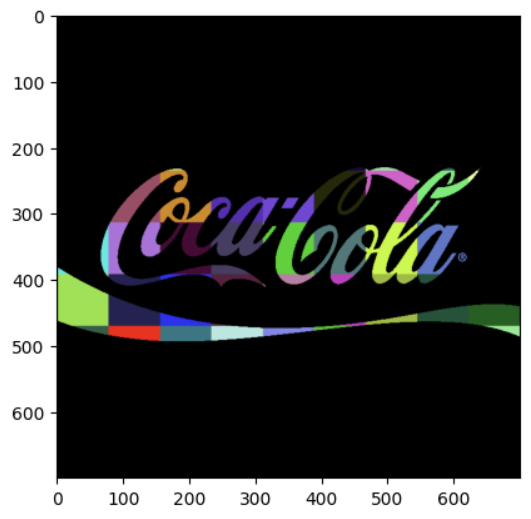
A colorful squares with black text

AI-generated content may be incorrect.

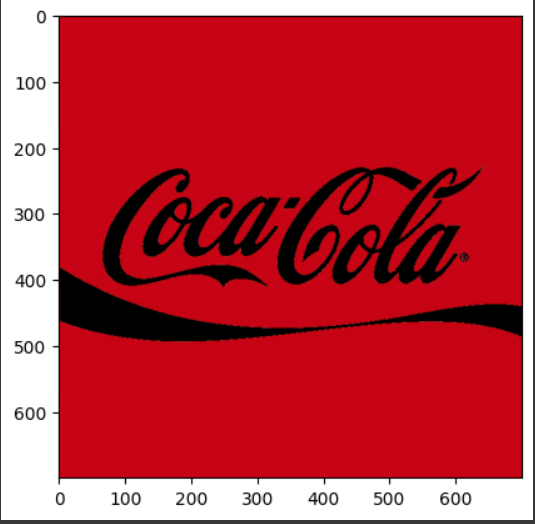
The first step is to read both images, next we need to determine a “Mask” for the original image, we do this by turning it into a binary image using thresholding after turning the original image into Grayscale.  


Then we invert the mask using the NOT bit operator, since we want the goal to change the black parts of the image( we could have just changed the white bits but I wanted to showcase the not operator)

Then we apply the “background” on the mask by using the AND bit operator,



Then we must isolate the foreground from the image, what it means is to get the original image and to make sure the areas of the mask are equal to 0(black) since we want to add the new merged section on it and we don’t want it to over flow



Now we add the two picture (remember the add function when we discussed brightness)

A screenshot of a computer

AI-generated content may be incorrect.

And done, we have our new logo.

# Accessing the Camera

This part is very straightforward, we just put the syntax required,

import cv2

import sys

s = 0

if len(sys.argv) > 1:

s = sys.argv[1]

source = cv2.VideoCapture(s)

win\_name = 'Camera Preview'

cv2.namedWindow(win\_name, cv2.WINDOW\_NORMAL)

while cv2.waitKey(1) != 27: # Escape

has\_frame, frame = source.read()

if not has\_frame:

break

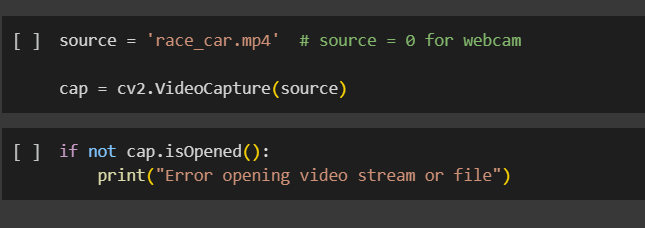
cv2.imshow(win\_name, frame)

source.release()

cv2.destroyWindow(win\_name)

# Video Writing

## 9.1 Read Video from Source



## Read and display one frame

A screenshot of a computer program

AI-generated content may be incorrect.



## Write Video

For writing the video, you need to create a videowriter object with the right parameters.

Function Syntax

VideoWriter object = cv.VideoWriter(filename, fourcc, fps, frameSize )

where, **Parameters**

1. filename: Name of the output video file.
2. fourcc: 4-character code of codec used to compress the frames. For example, VideoWriter::fourcc('P','I','M','1') is a MPEG-1 codec, VideoWriter::fourcc('M','J','P','G') is a motion-jpeg codec etc. List of codes can be obtained at Video Codecs by FOURCC page. FFMPEG backend with MP4 container natively uses other values as fourcc code: see ObjectType, so you may receive a warning message from OpenCV about fourcc code conversion.
3. fps: Framerate of the created video stream.
4. frameSize: Size of the video frames.

# Default resolutions of the frame are obtained.

# Convert the resolutions from float to integer.

frame\_width = int(cap.get(3))

frame\_height = int(cap.get(4))

# Define the codec and create VideoWriter object.

out\_avi = cv2.VideoWriter("race\_car\_out.avi", cv2.VideoWriter\_fourcc("M", "J", "P", "G"), 10, (frame\_width, frame\_height))

out\_mp4 = cv2.VideoWriter("race\_car\_out.mp4", cv2.VideoWriter\_fourcc(\*"XVID"), 10, (frame\_width, frame\_height))

# Image Filtering (Edge Detection)

The syntax is simple:  
# camera\_filters.py

# A simple OpenCV script to apply and switch between real-time camera filters

import cv2 # OpenCV library for image/video processing

import sys # For command-line argument handling

import numpy # Numerical operations (used for reshaping corner arrays)

# ------------------------------

# Filter Mode Constants

# ------------------------------

PREVIEW = 0 # Display the raw camera feed (no processing)

BLUR = 1 # Apply a simple box blur filter to smooth the image

FEATURES = 2 # Detect and draw corner features on the image

CANNY = 3 # Perform Canny edge detection to highlight edges

# ------------------------------

# Parameters for Corner Detection

# ------------------------------

feature\_params = dict(

maxCorners=500, # Max number of corners to detect

qualityLevel=0.2, # Minimum accepted quality of corners (0 -> 1)

minDistance=15, # Minimum distance between detected corners

blockSize=9 # Size of the neighborhood considered for corner detection

)

# ------------------------------

# Select Video Source

# ------------------------------

# Default to webcam (device index 0)

s = 0

# If a file path or alternate index is provided as a command-line argument, use it

if len(sys.argv) > 1:

s = sys.argv[1]

# ------------------------------

# Initialization

# ------------------------------

image\_filter = PREVIEW # Start in preview mode

alive = True # Control flag for the main loop

win\_name = "Camera Filters"

# Create a resizable window to display results\ ncv2.namedWindow(win\_name, cv2.WINDOW\_NORMAL)

result = None # Placeholder for the processed frame

# Open the video capture (camera or file)

source = cv2.VideoCapture(s)

# ------------------------------

# Main Processing Loop

# ------------------------------

while alive:

# Read a frame from the source

has\_frame, frame = source.read()

if not has\_frame:

break # Exit loop if no frame is returned

# Flip horizontally to create a mirror-like effect

frame = cv2.flip(frame, 1)

# Apply the currently selected filter

if image\_filter == PREVIEW:

# No processing, show raw feed

result = frame

elif image\_filter == CANNY:

# Edge detection: thresholds at 80 and 150

result = cv2.Canny(frame, 80, 150)

elif image\_filter == BLUR:

# Box blur with a 13x13 kernel

result = cv2.blur(frame, (13, 13))

elif image\_filter == FEATURES:

# Corner detection:

# 1. Copy the frame for drawing (to preserve original)

result = frame.copy()

# 2. Convert to grayscale for corner detection

frame\_gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

# 3. Detect good corners to track

corners = cv2.goodFeaturesToTrack(frame\_gray, \*\*feature\_params)

# 4. If corners found, draw them

if corners is not None:

for x, y in numpy.float32(corners).reshape(-1, 2):

# Draw a small circle at each corner (radius=10, color=green, thickness=1)

cv2.circle(result, (int(x), int(y)), 10, (0, 255, 0), 1)

# Display the resulting frame

cv2.imshow(win\_name, result)

# Handle keyboard input (wait 1ms)

key = cv2.waitKey(1)

if key in (ord("Q"), ord("q"), 27):

# Quit on 'Q', 'q', or ESC

alive = False

elif key in (ord("C"), ord("c")):

image\_filter = CANNY # Switch to Canny mode

elif key in (ord("B"), ord("b")):

image\_filter = BLUR # Switch to Blur mode

elif key in (ord("F"), ord("f")):

image\_filter = FEATURES # Switch to Corner Features mode

elif key in (ord("P"), ord("p")):

image\_filter = PREVIEW # Switch back to Preview mode

# ------------------------------

# Cleanup

# ------------------------------

source.release() # Release the video capture object

cv2.destroyWindow(win\_name) # Close the display window



The orbs represent edges found by the algorithm, what it does is it searches for sudden drop of pixel values.  
Corner detection via cv2.goodFeaturesToTrack (Shi–Tomasi) relies purely on intensity gradients, not absolute color. If the “corner” patch has nearly the same grayscale value as its background, there’s not enough gradient change in any direction, so the algorithm won’t flag it as a corner. You’d need to boost contrast (e.g. via histogram equalization) or switch to a color‐gradient–based method if you really need to pick up low-contrast corners

# Conclusion

Throughout this manual, we’ve walked step-by-step through the core capabilities of OpenCV: from setting up a clean Python environment and installing the library, to loading, displaying, and probing image properties. We then explored fundamental pixel-level operations—cropping, resizing, flipping—and moved on to more advanced image-processing techniques such as color-space conversions, blurring (average, Gaussian, bilateral), edge detection (Canny, Laplacian), and feature extraction (Shi–Tomasi corner detector). We also covered practical image annotation routines—drawing shapes and text—and looked at basic enhancement methods for brightness, contrast, thresholding, and bitwise masking. Finally, we demonstrated how to interface with live camera feeds and how to encode processed frames into video files.

By mastering these building blocks, you now have a solid foundation for virtually any computer-vision task: preprocessing data, visualizing results, prototyping real-time applications, or preparing inputs for machine-learning pipelines. From here, you can readily extend into deeper topics—object detection and tracking, machine-learning integration, advanced morphological operations, or GPU acceleration. OpenCV’s modular design and extensive documentation make it straightforward to layer in new functionality, so keep experimenting, reference the official API, and leverage the vibrant OpenCV community as you develop ever more sophisticated vision systems.