BMDS2133
Image Processing

My Practical Handbook

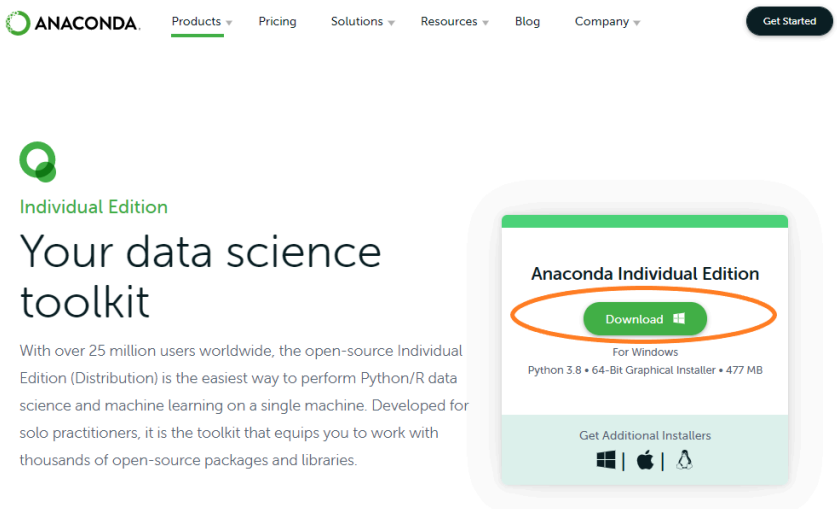
Faculty of Computing and Information Technology (FOCS)
Tunku Abdul Rahman University of Management and Technology

Table of Contents

- OpenCV was started at Intel in 1999 by **Gary Bradsky**, and the first release came out in 2000. **Vadim Pisarevsky** joined Gary Bradsky to manage Intel's Russian software OpenCV team.
- OpenCV supports a wide variety of programming languages such as C++, Python, Java, etc., and is available on different platforms including Windows, Linux, OS X, Android, and iOS. Interfaces for high-speed GPU operations based on CUDA and OpenCL are also under active development.
- OpenCV-Python is the Python API for OpenCV, combining the best qualities of the OpenCV C++ API and the Python language.
- OpenCV-Python makes use of **Numpy**, which is a highly optimised library for numerical operations with a MATLAB-style syntax. All the OpenCV array structures are converted to and from Numpy arrays. This also makes it easier to integrate with other libraries that use Numpy such as SciPy and Matplotlib.

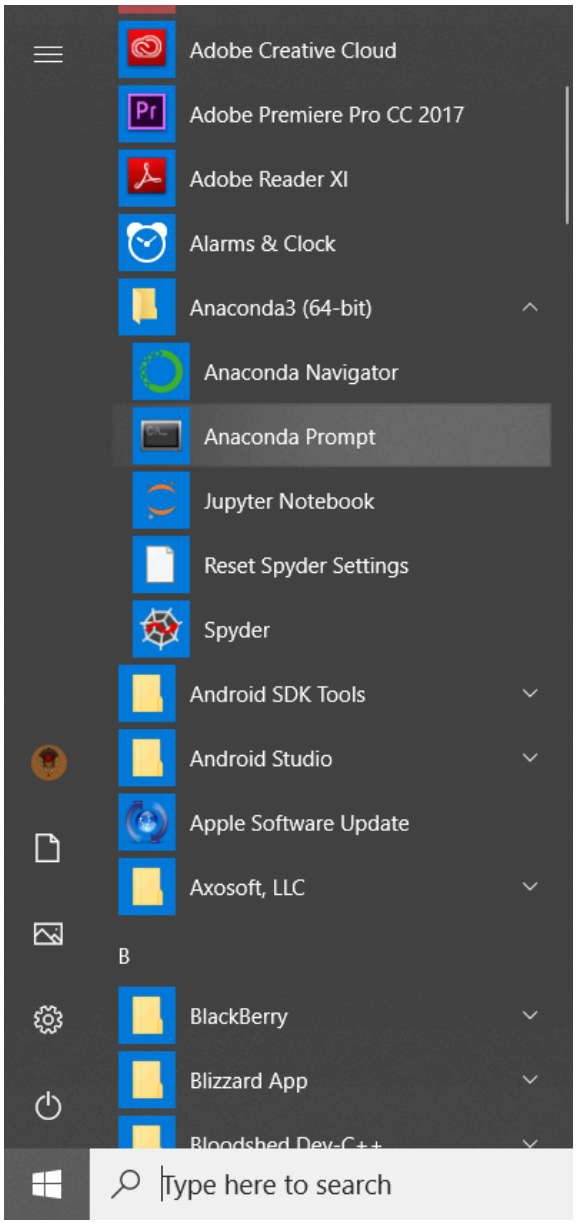
Install OpenCV in Windows

1. **Download and install anaconda environment Python X.X version:**
2. Download: https://www.anaconda.com/download/#windows
3. Install: http://docs.anaconda.com/anaconda/install/windows/



4. **Open Anaconda Prompt**

5. Start Menu / Anaconda3 / Anaconda Prompt



6.

7. **In Anaconda Prompt, type commands to install necessary libraries:**

   - pip install opencv-python
   - pip install opencv-contrib-python (if you having trouble to install the package) - contributed by YUNG JUN YONG
   - 🚨If you facing issue of numpy, consider downgrade the numpy version to <2 with the following code in anaconda prompt.

Python

```
pip uninstall numpy
pip install "numpy<2.0"
```

Practical 1 Getting Started with OpenCV

Getting Started with Images

Learn to load an image, display it, and save it back

In this practical you will learn how to:

- Read an image from file (using **cv::imread**)
- Display an image in an OpenCV window (using **cv::imshow**)
- Write an image to a file (using **cv::imwrite**)

To deal with Image: Tiga Sekawan

Python

```
cv.imread()

cv.imshow()

cv.waitKey(0)
```

To deal with Video: Empat Sekawan

Python

```
cv.VideoCapture()

cv.imshow()

cap.release()

cv.waitKey(1)
```

Source Code

```python
import cv2 as cv

img = cv.imread("starry_night.jpg")
cv.imshow("Display window", img)
cv.waitKey(0)
```

Flip the image/frame

```python
cv.flip()
```

Image too large to display entirely in the monitor? Let's make it resizable.

```python
cv.namedWindow('Original Image', cv.WINDOW_NORMAL)
```

Save the Frame

```python
if cv.waitKey(1) == ord('s'):
        cv.imwrite("pic.jpg",frame)
```

Capture Video from Camera

Often, we have to capture a live stream with a camera. OpenCV provides a very simple interface to do this. Let's capture a video from the camera (I am using the built-in webcam on my laptop), convert it into grayscale video and display it. Just a simple task to get started.

To capture a video, you need to create a **VideoCapture** object. Its argument can be either the device index or the name of a video file. A device index is just the number to specify which camera.

Normally one camera will be connected (as in my case). So I simply pass 0 (or -1). You can select the second camera by passing 1 and so on. After that, you can capture it frame-by-frame. But at the end, don't forget to release the capture.

Python

```python
import cv2 as cv
cap = cv.VideoCapture(0)

while True:
    ret, frame = cap.read()
    cv.imshow('Video Input', frame)
    if cv.waitKey(1) == ord('q'):
        break
cap.release()
cv.destroyAllWindows()
```

Convert into grayscale for faster processing

Python

```python
import numpy as np
import cv2 as cv
cap = cv.VideoCapture(0)

while True:
   # Capture frame-by-frame
   ret, frame = cap.read()
   gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
   cv.imshow('Video Input', gray)
   if cv.waitKey(1) == ord('q'):
       break
# When everything done, release the capture
cap.release()
cv.destroyAllWindows()
```

Note

If you are getting an error, make sure your camera is working fine using any other camera application (like Cheese in Linux).

Playing Video from file

Playing video from a file is the same as capturing it from a camera, just change the camera index to a video file name. Also while displaying the frame, use appropriate time for `cv.waitKey()`. If it is too little, video will be very fast and if it is too high, video will be slow (Well, that is how you can display videos in slow motion). 25 milliseconds will be OK in normal cases.

Video Link: https://github.com/opencv/opencv/blob/master/samples/data/vtest.avi

```python
import numpy as np
import cv2 as cv
cap = cv.VideoCapture('vtest.avi')
while True:
    ret, frame = cap.read()

    if not ret:
        print("Video ends...Bye")
        break

    #gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    cv.imshow('frame', frame)
    if cv.waitKey(1) == ord('q'):
        break
cap.release()
cv.destroyAllWindows()
```

Attention 🚨🚨:

Make sure a proper version of ffmpeg or gstreamer is installed. Sometimes it is a headache to work with video capture, mostly due to wrong installation of ffmpeg/gstreamer.

Saving a Video

So we capture a video and process it frame-by-frame, and we want to save that video. For images, it is very simple: just use `cv.imwrite()`. Here, a little more work is required.

This time we create a **VideoWriter** object. We should specify the output file name (eg: output.avi). Then we should specify the **FourCC** code (details in next paragraph). Then number of frames per second (fps) and frame size should be passed. And the last one is the **isColor** flag. If it is `True`, the encoder expects a color frame, otherwise it works with a grayscale frame.

FourCC is a 4-byte code used to specify the video codec. The list of available codes can be found in fourcc.org. It is platform dependent. The following codecs work fine for me.

- In Fedora: DIVX, XVID, MJPG, X264, WMV1, WMV2. (XVID is more preferable. MJPG results in high size video. X264 gives very small size video)
- In Windows: DIVX (More to be tested and added)
- In OSX: MJPG (.mp4), DIVX (.avi), X264 (.mkv).

FourCC code is passed as `cv.VideoWriter_fourcc('M','J','P','G')or cv.VideoWriter_fourcc(*'MJPG')` for MJPG.

The below code captures from a camera and saves the video.

```python
import cv2 as cv

cap = cv.VideoCapture(0)

fourcc = cv.VideoWriter_fourcc(*'XVID')
out = cv.VideoWriter('output.avi', fourcc, 20.0, (640,  480))

while True:
    ret,frame = cap.read()
    if ret == True:

        out.write(frame)

        cv.imshow('My First Video', frame)

#     cv.waitKey(1)
```

```
        if cv.waitKey(1) & 0xFF == ord('q'):
                break

cap.release()
out.release()
cv.destroyAllWindows()
```

Drawing Functions in OpenCV

- Learn to draw different geometric shapes with OpenCV
- Useful for: Segmentation, Bounding Box
- You will learn these functions : **cv.line()**, **cv.circle()** , **cv.rectangle()**, **cv.ellipse()**, **cv.putText()** etc.
- Reminder: The origin (0,0) is located at the top left corner of entire frame/canva.

Code

In all the above functions, you will see some common arguments as given below:

- img : The image where you want to draw the shapes
- color : Color of the shape. for BGR, pass it as a tuple, eg: (255,0,0) for blue. For grayscale, just pass the scalar value.
- thickness : Thickness of the line or circle etc. If **-1** is passed for closed figures like circles, it will fill the shape. *default thickness = 1*
- lineType : Type of line, whether 8-connected, anti-aliased line etc. *By default, it is 8-connected.* **cv.LINE_AA** gives an anti-aliased line which looks great for curves.

Drawing Line

To draw a line, you need to pass the starting and ending coordinates of the line. We will create a black image and draw a blue line on it from top-left to bottom-right corners.

```
import numpy as np
```

```
import cv2 as cv
# Create a black image
img = np.zeros((512,512,3), np.uint8)
# Draw a diagonal blue line with thickness of 5 px
cv.line(img,(0,0),(511,511),(255,0,0),5)
```

Drawing Rectangle

To draw a rectangle, you need the top-left corner and bottom-right corner of the rectangle. This time we will draw a green rectangle at the top-right corner of the image.

```
cv.rectangle(img,(384,0),(510,128),(0,255,0),3)
```

Drawing Circle

To draw a circle, you need its center coordinates and radius. We will draw a circle inside the rectangle drawn above.

```
cv.circle(img,(447,63), 63, (0,0,255), -1)
                  ^       ^      ^      ^
              Location  size   colour   mode of fill
```

Drawing Ellipse

To draw the ellipse, we need to pass several arguments. One argument is the center location (x,y). Next argument is axis lengths (major axis length, minor axis length). angle is the angle of rotation of an ellipse in anti-clockwise direction. startAngle and endAngle denotes the starting and ending of ellipse arc measured in clockwise direction from the major axis. i.e. giving values 0 and 360 gives the full ellipse. For more details, check the documentation of **cv.ellipse()**. Below example draws a half ellipse at the centre of the image.

```
cv.ellipse(img,(256,256),(100,50),0,0,180,255,-1)
```

Drawing Polygon

To draw a polygon, first you need coordinates of vertices. Make those points into an array of shape ROWSx1x2 where ROWS are the number of vertices and it should be of type int32. Here we draw a small polygon with four vertices in yellow colour.

```
pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
pts = pts.reshape((-1,1,2))
cv.polylines(img,[pts],True,(0,255,255))
```

Adding Text to Images:

To put text in images, you need to specify the following things.

- Text data that you want to write
- Position coordinates of where you want to put it.
- Support font:

cv.FONT_HERSHEY_SIMPLEX	normal size sans-serif font
cv.FONT_HERSHEY_PLAIN	small size sans-serif font
cv.FONT_HERSHEY_DUPLEX	normal size sans-serif font (more complex than FONT_HERSHEY_SIMPLEX)
cv.FONT_HERSHEY_COMPLEX	normal size serif font
cv.FONT_HERSHEY_TRIPLEX	normal size serif font (more complex than FONT_HERSHEY_COMPLEX)
cv.FONT_HERSHEY_COMPLEX_SMALL	smaller version of FONT_HERSHEY_COMPLEX
cv.FONT_HERSHEY_SCRIPT_SIMPLEX	hand-writing style font
cv.FONT_HERSHEY_SCRIPT_COMPLEX	more complex variant of FONT_HERSHEY_SCRIPT_SIMPLEX
cv.FONT_ITALIC	flag for italic font

- Font Scale (specifies the size of font)
- regular things like color, thickness, lineType etc. For better look, lineType = **cv.LINE_AA** is recommended.

We will write **OpenCV** on our image in white colour.

```
font = cv.FONT_HERSHEY_SIMPLEX
cv.putText(img,'Sean',(50, 50), font, 4,(255,255,255),2,cv.LINE_AA)
```

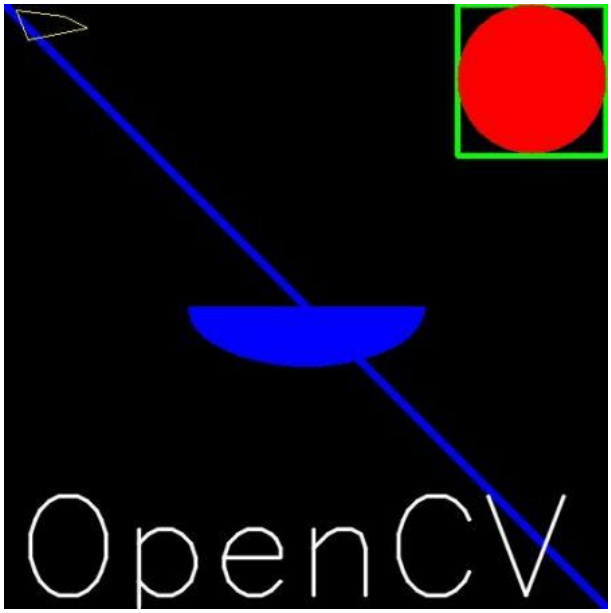
Make your display windows resizable?

```
Python
cv.namedWindow('Result', cv.WINDOW_NORMAL)
```

Result

So it is time to see the final result of our drawing. As you studied in previous articles, display the image to see it.

```
##Show the result
cv.namedWindow('Result', cv.WINDOW_NORMAL)
cv.imshow('Result',img)
cv.waitKey(0)
cv.destroyAllWindows()
```



image

Exercises

1. Try to create the logo of OpenCV/Your favoured icon/any other object using drawing functions available in OpenCV.
2. Post it to Padlet and name your masterpiece.

Practical 2 Concept of Region of Interest (ROI) and Region of Non Interest (RONI)

In the context of image processing, ROI stands for Region of Interest, and RONI stands for Region of Non-interest.

1. Region of Interest (ROI):
A Region of Interest refers to a specific portion or area within an image that is selected or defined to focus on a particular object, feature, or region for further analysis, processing, or manipulation. The ROI can be a rectangular area, a polygon, or any other shape that encapsulates the desired region. By isolating the ROI, it becomes possible to extract relevant information or apply specific algorithms to enhance or analyse that specific region, while disregarding the rest of the image.

2. Region of Non-interest (RONI):
A Region of Non-interest, or RONi, is the opposite of ROI. It refers to the areas or regions within an image that are considered unimportant or irrelevant for the particular analysis or processing task at hand. These regions are typically excluded or ignored during the image processing workflow to focus computational resources and algorithms on the regions of interest. RONI helps to reduce unnecessary computations and enhance the efficiency of image processing tasks by excluding non-relevant information.

In summary, ROI and RONI are concepts used in image processing to define the areas of interest and non-interest, respectively, within an image for the purpose of targeted analysis, processing, or manipulation.

Haar Cascade Classifier: [Haar Cascade - Github]

```
import cv2 as cv

# Load the Haar cascade XML file for face detection
face_cascade = cv.CascadeClassifier(cv.data.haarcascades +
'haarcascade_frontalface_default.xml')

# Create a VideoCapture object to read from the camera
cap = cv.VideoCapture(0)

while True:
    # Read the video frame by frame
    ret, frame = cap.read()

    # Convert the frame to gray scale for face detection
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # Detect faces in the frame
```

```
    faces = face_cascade.detectMultiScale(gray,
scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

    # Blur the faces in the frame
    for (x, y, w, h) in faces:
        # Extract the region of interest (face)
        face_roi = frame[y:y+h, x:x+w]

        # Apply Gaussian blur to the face region
        blurred_face = cv.GaussianBlur(face_roi, (99, 99), 30)

        # Replace the original face with the blurred face
        frame[y:y+h, x:x+w] = blurred_face

    # Display the resulting frame
    cv.imshow('Video', frame)

    # Check for the 'q' key to exit
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

# Release the VideoCapture and close the windows
cap.release()
cv.destroyAllWindows()
```

Basic Operations on Images

Goal

Learn to:

- Access pixel values and modify them
- Access image properties
- Set a Region of Interest (ROI) vs RONI (Region of non-interest)
- Split and merge images

*Almost all the operations in this section are mainly related to Numpy rather than OpenCV. A good knowledge of Numpy is required to write better optimised code with OpenCV.

Accessing and Modifying pixel values

Let's load a colour image first:

Download: messi.jpg

```
import numpy as np
import cv2 as cv
img = cv.imread('messi.jpg')
```

You can access a pixel value by its row and column coordinates. For BGR image, it returns an array of Blue, Green, Red values. For grayscale image, just corresponding intensity is returned.

```
px = img[100,100]
print( px )

# accessing only blue pixel
blue = img[100,100,0]
print( blue )
cv.imshow("Pic", img)
cv.waitKey(0)
cv.destroyAllWindows()
```

You can modify the pixel values the same way.

```
>>> img[100,100] = [0,0,0]
>>> print( img[100,100] )
[255 255 255]
```

Accessing Image Properties

Image properties include number of rows, columns, and channels; type of image data; number of pixels; etc.

The shape of an image is accessed by img.shape. It returns a tuple of the number of rows, columns, and channels (if the image is color):

```
>>> print( img.shape )
(342, 548, 3)
```

Total number of pixels is accessed by `img.size`:

```
>>> print( img.size )
562248
```

Image datatype is obtained by `img.dtype`:

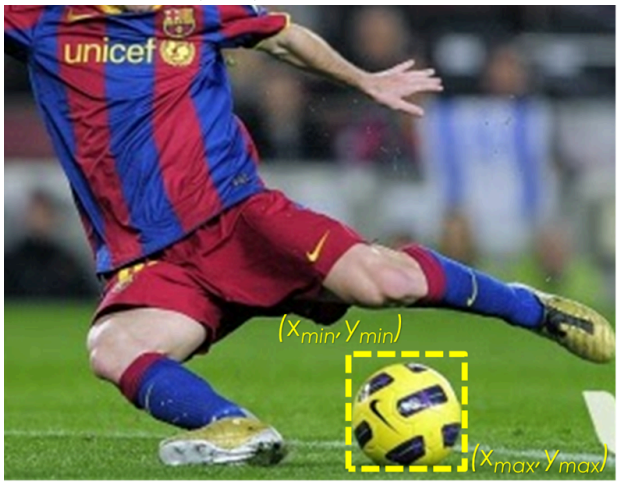
```
>>> print( img.dtype )
uint8
```

Image ROI

Sometimes, you will have to play with certain regions of images. For eye detection in images, first face detection is done over the entire image. When a face is obtained, we select the face region alone and search for eyes inside it instead of searching the whole image. It improves accuracy (because eyes are always on faces :D) and performance (because we search in a small area).

ROI is again obtained using Numpy indexing. Here I am selecting the ball and copying it to another region in the image:

```
#1st value = y-coordinate, 2nd value = x-coordinate
>>> ball = img[280:340, 330:390]
>>> img[273:333, 100:160] = ball

#in this case, 280=ymin; 340=ymax; 330=xmin; 390=xmax
# img[ymin:ymax,xmin,xmax)
```




```
Theory involved: Copy Move Attack

import cv2 as cv
img = cv.imread('messi.jpg')

px = img[100,100]
print(px)

ball = img[280:340, 330:390]
img[273:333, 100:160] = ball
img[200:260, 100:160] = ball
img[130:190, 50:110] = ball



cv.imshow("Messi",img)
cv.waitKey(0)
cv.destroyAllWindows()
```

Splitting and Merging Image Channels

Sometimes you will need to work separately on the B,G,R channels of an image. In this case, you need to split the BGR image into single channels. In other cases, you may need to join these individual channels to create a BGR image. You can do this simply by:

```
>>> b,g,r = cv.split(img)
>>> img = cv.merge((b,g,r))
```

Making Borders for Images (Padding)

If you want to create a border around an image, something like a photo frame, you can use **cv.copyMakeBorder()**. But it has more applications for convolution operation, zero padding etc. This function takes following arguments:

- **src** - input image
- **top**, **bottom**, **left**, **right** - border width in number of pixels in corresponding directions
- **borderType** - Flag defining what kind of border to be added. It can be following types:
    - **cv.BORDER_CONSTANT** - Adds a constant colored border. The value should be given as next argument.
    - **cv.BORDER_REFLECT** - Border will be mirror reflection of the border elements, like this : *fedcba|abcdefgh|hgfedcb*
    - **cv.BORDER_REFLECT_101** or **cv.BORDER_DEFAULT** - Same as above, but with a slight change, like this : *gfedcb|**a**bcdefg**h**|gfedcba*
    - **cv.BORDER_REPLICATE** - Last element is replicated throughout, like this: *aaaaaa|abcdefgh|hhhhhhh*
    - **cv.BORDER_WRAP** - Can't explain, it will look like this : *cdefgh|abcdefgh|abcdefg*
- **value** - Color of border if border type is **cv.BORDER_CONSTANT**

Below is a sample code demonstrating all these border types for better understanding:

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
BLUE = [255,0,0]
img1 = cv.imread('opencv-logo.png')
replicate = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REPLICATE)
reflect = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REFLECT)
reflect101 = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_REFLECT_101)
wrap = cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_WRAP)
```

```
constant=
       cv.copyMakeBorder(img1,10,10,10,10,cv.BORDER_CONSTANT,value=BLUE)
plt.subplot(231),plt.imshow(img1,'gray'),plt.title('ORIGINAL')
plt.subplot(232),plt.imshow(replicate,'gray'),plt.title('REPLICATE')
plt.subplot(233),plt.imshow(reflect,'gray'),plt.title('REFLECT')
plt.subplot(234),plt.imshow(reflect101,'gray'),plt.title('REFLECT_101')
plt.subplot(235),plt.imshow(wrap,'gray'),plt.title('WRAP')
plt.subplot(236),plt.imshow(constant,'gray'),plt.title('CONSTANT')
plt.show()
```

Arithmetic Operations on Images

Goal

- Learn several arithmetic operations on images, like addition, subtraction, bitwise operations, and etc.
- Learn these functions: **cv.add()**, **cv.addWeighted()**, etc.

Image Addition

You can add two images with the OpenCV function, **cv.add()**, or simply by the numpy operation res = img1 + img2. Both images should be of same depth and type, or the second image can just be a scalar value.

For example, consider the below sample:

```
x = np.uint8([250])
y = np.uint8([10])
print( cv.add(x,y) ) # 250+10 = 260 (overflow) => 255

[[255]]
------------------
>>> print( x+y )          # 250+10 = 260 % 256 = 4
[4]
```

Image Blending

This is also image addition, but different weights are given to images in order to give a feeling of blending or transparency. Images are added as per the equation below:

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

```
import cv2 as cv
import numpy as np

img1 = cv.imread('formula.png')
height1, width1, channels1 = img1.shape

dim = (400, 400)
resized = cv.resize(img1, dim, interpolation = cv.INTER_AREA)

img2 = cv.imread('najib_blur.jpg')
height2, width2, channels2 = img1.shape
dim = (400, 400)
resized2 = cv.resize(img2, dim, interpolation = cv.INTER_AREA)

dst = cv.addWeighted(resized,0.3,resized2,0.7,0)  #← Modify the weightage for
       each picture here
cv.imshow('dst',dst)
cv.waitKey(0)
cv.destroyAllWindows()

—--------------------------------
```

Enhance with subplot

```
JavaScript
from matplotlib import pyplot as plt
plt.subplot(131),plt.imshow(resized),plt.title('Image 1')
plt.subplot(132),plt.imshow(resized2),plt.title('Image 2')
plt.subplot(138),plt.imshow(dst),plt.title('Result')
plt.show()
```

Modify the code and use your creativity to create image blending. Post it to Padlet and name your masterpiece.

Practical 3 Image Processing in OpenCV

- In this tutorial, you will learn how to convert images from one color-space to another, like BGR ↔ Gray, BGR↔ HSV, etc.
- In addition to that, we will create an application to extract a colored object in a video
- You will learn the following functions: **cv.cvtColor()**, **cv.inRange()**, etc.

Changing Color-space

There are about 346 color-space conversion methods available in OpenCV. But we will look into only two, which are most widely used ones: BGR ↔ Gray and BRG ↔ HSV.

For color conversion, we use the function cv.cvtColor(input_image, flag) where flag determines the type of conversion.

For BGR → Gray conversion, we use the flag **cv.COLOR_BGR2GRAY**. Similarly for BGR → HSV, we use the flag **cv.COLOR_BGR2HSV**. To get other flags, just run following commands in your Python terminal:

```
>>> import cv2 as cv
>>> flags = [i for i in dir(cv) if i.startswith('COLOR_')]
>>> print( flags )
```

Object Tracking

Now that we know how to convert a BGR image to HSV, we can use this to extract a coloured object. In HSV, it is easier to represent a colour than in BGR color-space. In our application, we will try to extract a blue coloured object. So here is the method:

- Take each frame of the video
- Convert from BGR to H(Hue) SV color-space
- We threshold the HSV image for a range of blue color
- Now extract the blue object alone, we can do whatever we want on that image.

Below is the code which is commented in detail:

```
```

Python

```python
import cv2 as cv
import numpy as np

# Initialize webcam
cap = cv.VideoCapture(0)

if not cap.isOpened():
    print("Error: Could not open webcam.")
    exit()

while True:
    # Read frame
    ret, frame = cap.read()
    if not ret:
        print("Error: Failed to capture frame.")
        break

    # Convert BGR to HSV
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)

    # Define range for blue color in HSV
    lower_blue = np.array([110, 50, 50])
    upper_blue = np.array([130, 255, 255])  # Ensure upper > lower

    # Threshold to extract blue objects
    mask = cv.inRange(hsv, lower_blue, upper_blue)

    # Bitwise-AND mask with original image
    res = cv.bitwise_and(frame, frame, mask=mask)

    # Display windows
    cv.imshow('Original Frame', frame)
    cv.imshow('Mask', mask)
    cv.imshow('Filtered Output', res)

    # Press 'Esc' to exit
    if cv.waitKey(1) == 27:
        break

# Release resources
cap.release()
cv.destroyAllWindows()
```

😋 Tips:

- Hue (H): Determined by the dominant color (converted from 0–360 to 0–179 for OpenCV).

- Saturation (S): Measures color intensity, scaled to 0–255.
- Value (V): Represents brightness, scaled to 0–255.

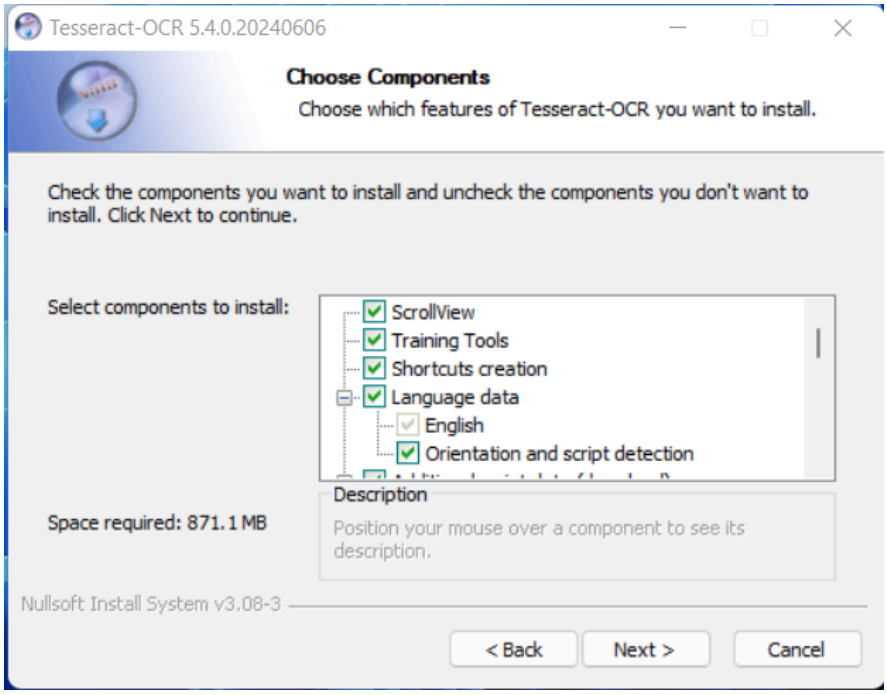
Adding Bounding Circle

❓ *The key question now is: At which point should the code be inserted to enable the display on the output console? How to add in the text once the object is being detected?*

```python
    contours, _ = cv.findContours(mask, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

    for contour in contours:
        if cv.contourArea(contour) > 500:  # Filter out small areas
            (x, y), radius = cv.minEnclosingCircle(contour)
            center = (int(x), int(y))
            radius = int(radius)
            cv.circle(frame, center, radius, (0, 255, 0), 2)  # Draw bounding
circle
```

Optical Character Recognition (OCR) using Tesseract

1. Download & Install Tesseract from OCR (Search UB Mannheim Tesseract)
    a. Ensure we download the additional language and script support



2. Install wrapper (`pip install pytesseract`)

```python
import cv2 as cv
import pytesseract

# If you don't have tesseract executable in your PATH,
include the following:
pytesseract.pytesseract.tesseract_cmd =
r'<full_path_to_your_tesseract_executable>'
# Example tesseract_cmd = r'C:\Program Files
(x86)\Tesseract-OCR\tesseract'

img = cv.imread('Your image here.jpg')

# Simple image to string
print(pytesseract.image_to_string(img))
```

Testing Image: For OCR

ROI Selection

```
import cv2
import numpy as np


# Read image
image = cv2.imread("image.png")

# Select ROI
r = cv2.selectROI("select the area", image)

# Crop image
cropped_image = image[int(r[1]):int(r[1]+r[3]),
                      int(r[0]):int(r[0]+r[2])]
# Display cropped image
cv2.imshow("Cropped image", cropped_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Image Thresholding

Goal

- In this tutorial, you will learn simple thresholding, adaptive thresholding and Otsu's thresholding.
- You will learn the functions **cv.threshold** and **cv.adaptiveThreshold**.

Simple Thresholding

Here, the matter is straight-forward. For every pixel, the same threshold value is applied. If the pixel value is smaller than the threshold, it is set to 0, otherwise it is set to a maximum value. The function **cv.threshold** is used to apply the thresholding. The first argument is the source image, which **should be a grayscale image**. The second argument is the threshold value which is used to classify the pixel values. The third argument is the maximum value which is assigned to pixel values exceeding the threshold. OpenCV provides different types of thresholding which is given by the

fourth parameter of the function. Basic thresholding as described above is done by using the type **cv.THRESH_BINARY**. All simple thresholding types are:

- **cv.THRESH_BINARY**
- **cv.THRESH_BINARY_INV**
- **cv.THRESH_TRUNC**
- **cv.THRESH_TOZERO**
- **cv.THRESH_TOZERO_INV**

See the documentation of the types for the differences.

The method returns two outputs. The first is the threshold that was used and the second output is the **thresholded image**.

This code compares the different simple thresholding types:

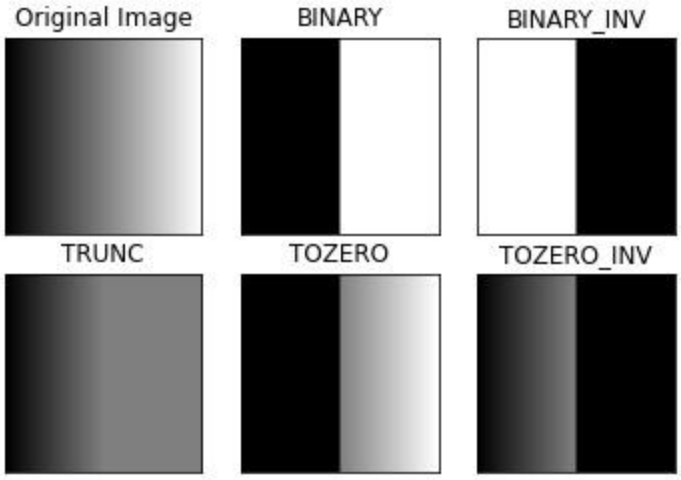
gradient.png

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('gradient.png',0)
ret,thresh1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
ret,thresh2 = cv.threshold(img,127,255,cv.THRESH_BINARY_INV)
ret,thresh3 = cv.threshold(img,127,255,cv.THRESH_TRUNC)
ret,thresh4 = cv.threshold(img,127,255,cv.THRESH_TOZERO)
ret,thresh5 = cv.threshold(img,127,255,cv.THRESH_TOZERO_INV)
titles = ['Original
       Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
for i in range(6):
   plt.subplot(2,3,i+1),plt.imshow(images[i],'gray',vmin=0,vmax=255)
   plt.title(titles[i])
   plt.xticks([]),plt.yticks([])
plt.show()
```

Note

To plot multiple images, we have used the plt.subplot() function. Please checkout the matplotlib docs for more details.

The code yields this result:



image

Use case of Thresholding

1. Segmentation
2. Increase the contrast
3. Enhance the image details
4. Line detection
5. Good for feature extraction (to be used in machine learning)

Adaptive Thresholding

In the previous section, we used one global value as a threshold. But this might not be good in all cases, e.g. if an image has different lighting conditions in different areas. In that case, adaptive thresholding can help. Here, the algorithm determines the threshold for a pixel based on a small region around it. So we get different thresholds for different regions of the same image which gives better results for images with varying illumination.

In addition to the parameters described above, the method **cv.adaptiveThreshold** takes three input parameters:

The **adaptiveMethod** decides how the threshold value is calculated:

- **cv.ADAPTIVE_THRESH_MEAN_C**: The threshold value is the mean of the neighbourhood area minus the constant **C**.

- **cv.ADAPTIVE_THRESH_GAUSSIAN_C**: The threshold value is a gaussian-weighted sum of the neighbourhood values minus the constant **C**.

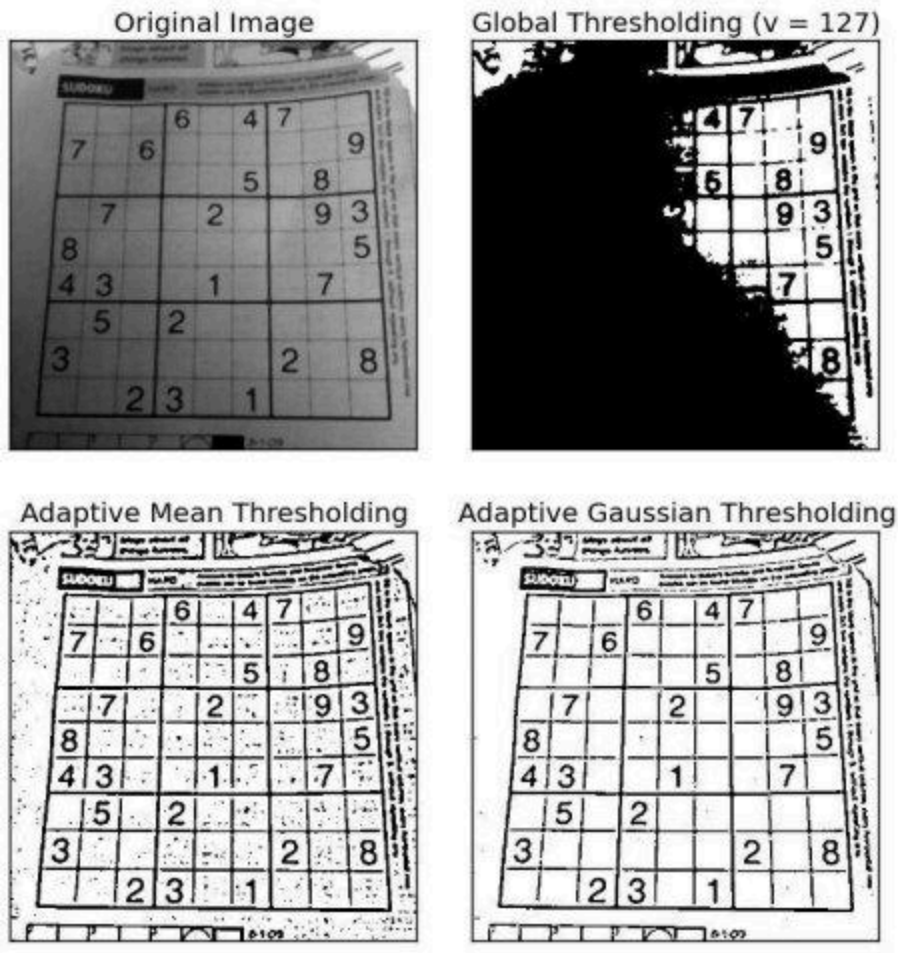
The **blockSize** determines the size of the neighbourhood area and **C** is a constant that is subtracted from the mean or weighted sum of the neighbourhood pixels.

The code below compares global thresholding and adaptive thresholding for an image with varying illumination:

sudoku.jpg

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('sudoku.jpg',0)
img = cv.medianBlur(img,5)
ret,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
th2 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_MEAN_C,\
            cv.THRESH_BINARY,11,2)
th3 = cv.adaptiveThreshold(img,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,\
            cv.THRESH_BINARY,11,2)
titles = ['Original Image', 'Global Thresholding (v = 127)',
            'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]
plt.figure(figsize=(10,10))
for i in range(4):
   plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
   plt.title(titles[i])
   plt.xticks([]),plt.yticks([])
plt.show()
```

Result:



image

Otsu's Binarization

In global thresholding, we used an arbitrary chosen value as a threshold. In contrast, Otsu's method avoids having to choose a value and determines it automatically.

Consider an image with only two distinct image values (*bimodal image*), where the histogram would only consist of two peaks. A good threshold would be in the middle of those two values. Similarly, Otsu's method determines an optimal global threshold value from the image histogram.

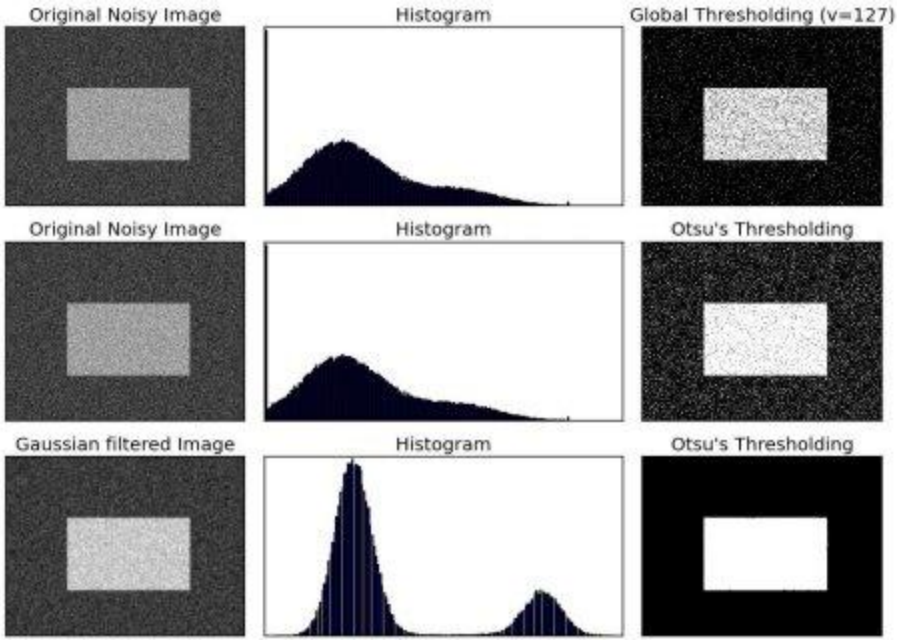
In order to do so, the **cv.threshold()** function is used, where **cv.THRESH_OTSU** is passed as an extra flag. The threshold value can be chosen arbitrarily. The algorithm then finds the optimal threshold value which is returned as the first output.

Check out the example below. The input image is a noisy image. In the first case, global thresholding with a value of 127 is applied. In the second case, Otsu's thresholding is applied directly. In the third case, the image is first filtered with a 5x5 gaussian kernel to remove the noise, then Otsu thresholding is applied. See how noise filtering improves the result.

noisy2.png

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('noisy2.png',0)
# global thresholding
ret1,th1 = cv.threshold(img,127,255,cv.THRESH_BINARY)
# Otsu's thresholding
ret2,th2 = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
# Otsu's thresholding after Gaussian filtering
blur = cv.GaussianBlur(img,(5,5),0)
ret3,th3 = cv.threshold(blur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
# plot all the images and their histograms
images = [img, 0, th1,
          img, 0, th2,
          blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
          'Original Noisy Image','Histogram',"Otsu's Thresholding",
          'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]
for i in range(3):
   plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
   plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
   plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
   plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
   plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2],'gray')
   plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()
```

Result:



image

Smoothing Images

Goals

Learn to:

- Blur images with various **low pass filters**
- Apply custom-made filters to images (2D convolution)

2D Convolution (Image Filtering)

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

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

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

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

taruc.jpg pimple.jpg

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('taruc.jpg')
kernel = np.ones((5,5),np.float32)/25
dst = cv.filter2D(img,-1,kernel)
plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([]), plt.yticks([])
plt.show()
```

Exercise: Try to remove the pimple *(Hint: Kernel size).* Find a “pimple” image from the internet and try to remove it. Post your before and after photo on Padlet. 😏

Image Blurring (Image Smoothing)

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (eg: noise, edges) from the image. So edges are blurred a little bit in this operation (there are also blurring techniques which don't blur the edges). OpenCV provides four main types of blurring techniques.

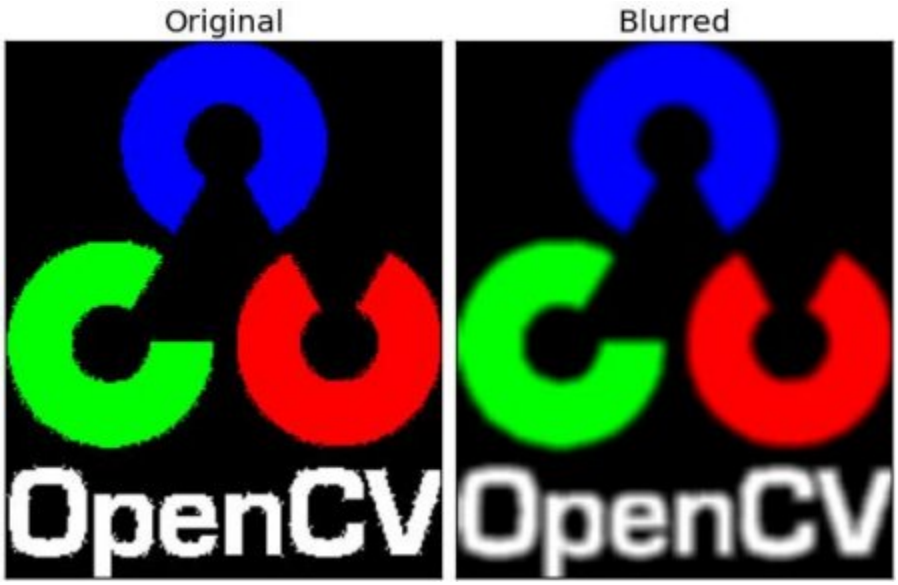
1. Averaging

This is done by convolving an image with a normalised box filter. It simply takes the average of all the pixels under the kernel area and **replaces the central element**. This is done by the function **cv.blur()** or **cv.boxFilter()**. Check the docs for more details about the kernel. We

should specify the width and height of the kernel. A 3x3 normalized box filter would look like the below:

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

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('your image here')
blur = cv.blur(img,(5,5))
plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.show()
```



2. Gaussian Blurring

In this method, instead of a box filter, a Gaussian kernel is used. It is done with the function, **cv.GaussianBlur()**. We should specify the width and height of the kernel which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as the same as

sigmaX. If both are given as zeros, they are calculated from the kernel size. Gaussian blurring is highly effective in removing Gaussian noise from an image.

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

The above code can be modified for Gaussian blurring:

gaussian.jpg

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

Result:

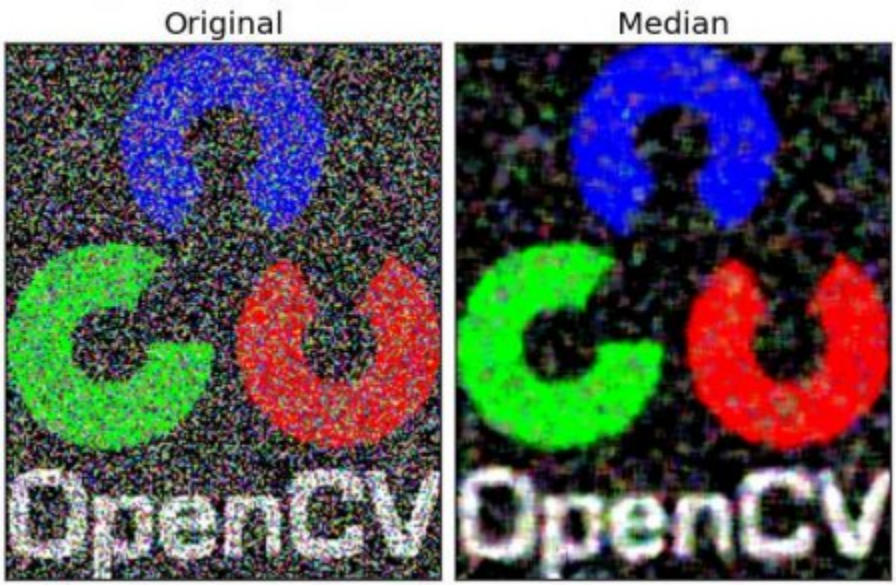
3. Median Blurring

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

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

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

Result:



Exercise: Find a “salt and pepper” image from the internet and try to remove the noise. Post your before and after photo on Padlet.

4. Bilateral Filtering

cv.bilateralFilter() is highly effective in noise removal while keeping edges sharp. But the operation is slower compared to other filters. We already saw that a Gaussian filter takes the neighbourhood around the pixel and finds its Gaussian weighted average. This Gaussian filter is a function of space alone, that is, nearby pixels are considered while filtering. It doesn't consider whether pixels have almost the same intensity. It doesn't consider whether a pixel is an edge pixel or not. So it blurs the edges also, which we don't want to do.

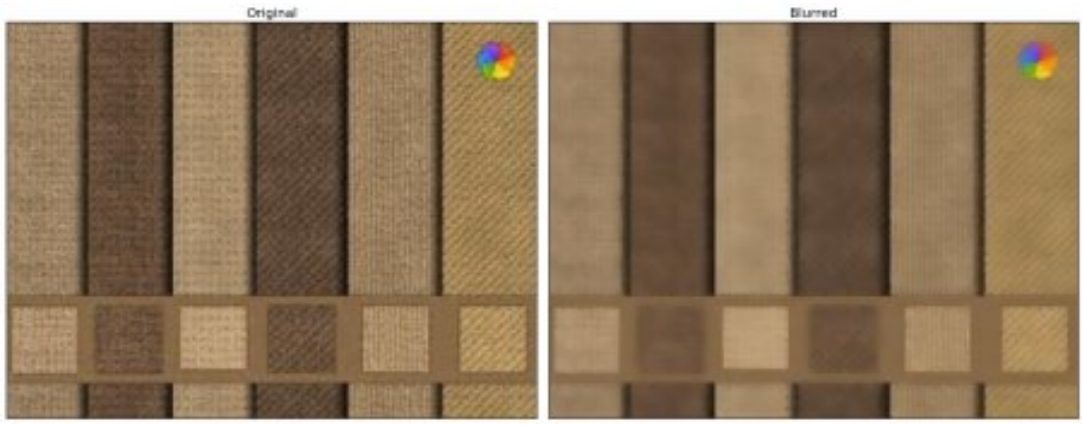
Bilateral filtering also takes a Gaussian filter in space, but one more Gaussian filter which is a function of pixel difference. The Gaussian function of space makes sure that only nearby pixels are considered for blurring, while the Gaussian function of intensity difference makes sure that only those pixels with similar intensities to the central pixel are considered for blurring. So it preserves the edges since pixels at edges will have large intensity variation.

The below sample shows use of a bilateral filter (For details on arguments, visit docs).

lake.jpg taj.jpg

```
blur = cv.bilateralFilter(img,9,75,75)
```

Result:



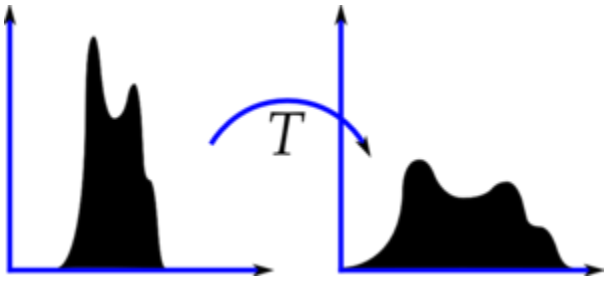
See, the texture on the surface is gone, but the edges are still preserved.

Histogram Equalisation

We will learn the concepts of histogram equalisation and use it to improve the contrast of our images.

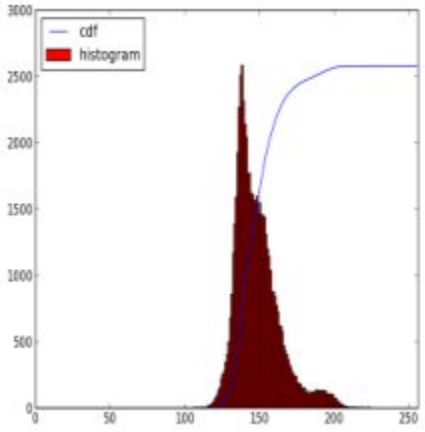
Histogram Equalisation X Captain America = captain_america.gif

Consider an image whose pixel values are confined to some specific range of values only. For example, a brighter image will have all pixels confined to high values. But a good image will have pixels from all regions of the image. So you need to stretch this histogram to either end (as given in below image, from wikipedia) and that is what Histogram Equalization does (in simple words). This normally improves the contrast of the image.



mountain.jpg

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('mountain.jpg',0)
hist,bins = np.histogram(img.flatten(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf * float(hist.max()) / cdf.max()
plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.show()
```



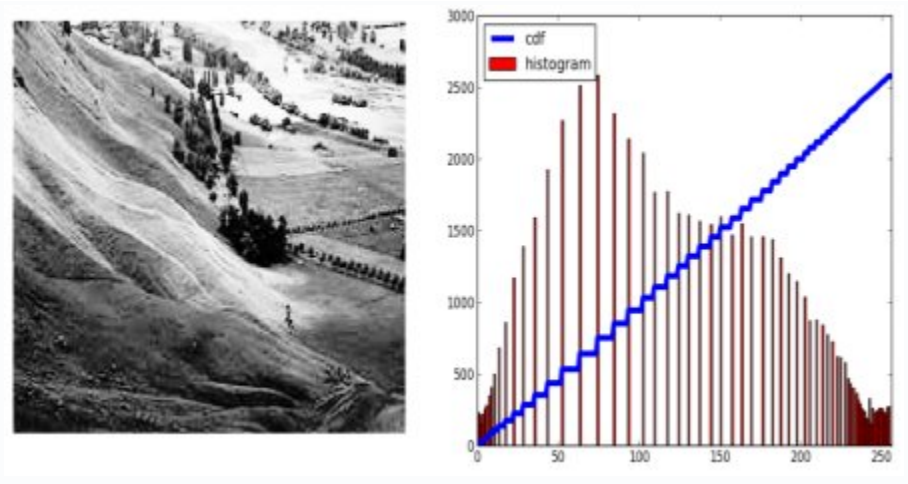
You can see histogram lies in a brighter region. We need the full spectrum. For that, we need a transformation function which maps the input pixels in a brighter region to output pixels in the full region. That is what histogram equalization does.

Now we find the minimum histogram value (excluding 0) and apply the histogram equalisation equation.

```
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m,0).astype('uint8')
```

Now we have the look-up table that gives us the information on what is the output pixel value for every input pixel value. So we just apply the transform.

```
img2 = cdf[img]
```

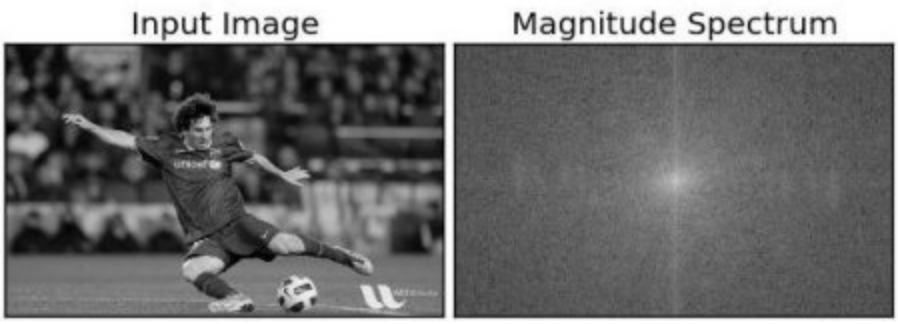


Fourier Transform

First we will see how to find Fourier Transform using Numpy. Numpy has an FFT package to do this. **np.fft.fft2()** provides us the frequency transform which will be a complex array. Its first argument is the input image, which is grayscale. Second argument is optional which decides the size of the output array. If it is greater than the size of the input image, the input image is padded with zeros before calculation of FFT. If it is less than input image, input image will be cropped. If no arguments passed, Output array size will be the same as input.

Now once you get the result, zero frequency component (DC component) will be at the top left corner. If you want to bring it to center, you need to shift the result by N/2 in both the directions. This is simply done by the function, **np.fft.fftshift()**. (It is easier to analyze). Once you found the frequency transform, you can find the magnitude spectrum.

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
img = cv.imread('messi.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))
plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

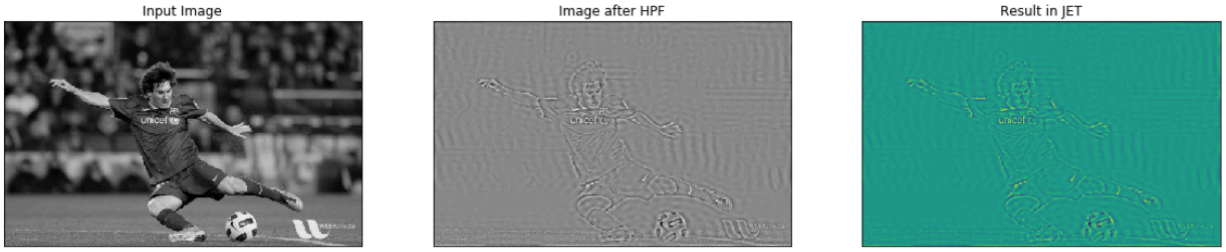


See, You can see more whiter region at the center showing low frequency content.

So you found the frequency transform. Now you can do some operations in the frequency domain, like high pass filtering and reconstruct the image, ie find inverse DFT. For that you simply remove the low frequencies by masking with a rectangular window of size 60x60. Then apply the inverse shift using **np.fft.ifftshift()** so that DC component again come at the top-left corner. Then find inverse FFT using **np.ifft2()** function. The result, again, will be a complex number. You can take its absolute value.

OpenCV: ColorMaps in OpenCV

```
rows, cols = img.shape
crow,ccol = rows//2 , cols//2
fshift[crow-30:crow+31, ccol-30:ccol+31] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.real(img_back)

plt.figure(figsize=(20,20))

plt.subplot(131),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])

plt.subplot(132),plt.imshow(img_back, cmap = 'gray')
plt.title('Image after HPF'), plt.xticks([]), plt.yticks([])

plt.subplot(133),plt.imshow(img_back, cmap = 'pink')
plt.title('Result in JET'), plt.xticks([]), plt.yticks([])
plt.show()
```



The result shows High Pass Filtering is an edge detection operation. This is what we have seen in Image Gradients chapter. This also shows that most of the image data is present in the Low frequency region of the spectrum. Anyway we have seen how to find DFT, IDFT etc in Numpy.

If you closely watch the result, especially the last image in JET color, you can see some artifacts . It shows some ripple like structures there, and it is called **ringing effects**. It is caused by the rectangular window we used for masking. This mask is converted to sinc shape which causes this problem.

Practical 4 Computational Photography

Image Denoising

Goal

In this chapter,

- You will learn about Non-local Means Denoising algorithm to remove noise in the image.
- You will see different functions like **cv.fastNlMeansDenoising()**

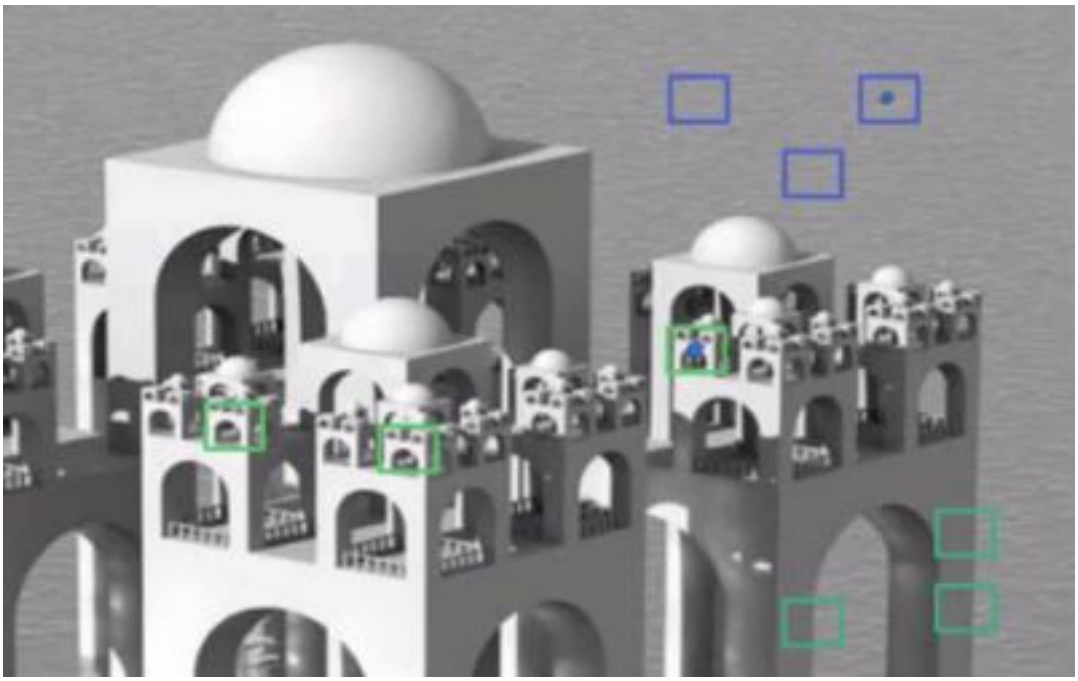
Theory

In earlier chapters, we have seen many image smoothing techniques like Gaussian Blurring, Median Blurring etc and they were good to some extent in removing small quantities of noise. In those techniques, we took a small neighbourhood around a pixel and did some operations like gaussian weighted average, median of the values etc to replace the central element. In short, noise removal at a pixel was local to its neighbourhood.

There is a property of noise. Noise is generally considered to be a random variable with zero mean. Consider a noisy pixel, $p=p_0+n$ where p_0 is the true value of the pixel and n is the noise in that pixel. You can take a large number of the same pixels (say N) from different images and compute their average. Ideally, you should get $p=p_0$ since the mean of noise is zero.

You can verify it yourself by a simple setup. Hold a static camera to a certain location for a couple of seconds. This will give you plenty of frames, or a lot of images of the same scene. Then write a piece of code to find the average of all the frames in the video (This should be too simple for you now). Compare the final result and first frame. You can see a reduction in noise. Unfortunately this simple method is not robust to camera and scene motions. Also often there is only one noisy image available.

So the idea is simple, we need a set of similar images to average out the noise. Consider a small window (say 5x5 window) in the image. Chance is large that the same patch may be somewhere else in the image. Sometimes in a small neighbourhood around it. What about using these similar patches together and finding their average? For that particular window, that is fine. See an example image below:



The blue patches in the image look similar. Green patches look similar. So we take a pixel, take a small window around it, search for similar windows in the image, average all the windows and replace the pixel with the result we got. This method is Non-Local Means Denoising. It takes more time compared to blurring techniques we saw earlier, but its result is very good.

Image Denoising in OpenCV

OpenCV provides four variations of this technique.

1. **cv.fastNlMeansDenoising()** - works with a single grayscale images
2. **cv.fastNlMeansDenoisingColored()** - works with a colour image.
3. **cv.fastNlMeansDenoisingMulti()** - works with image sequence captured in short period of time (grayscale images)
4. **cv.fastNlMeansDenoisingColoredMulti()** - same as above, but for colour images.

Common arguments are:

- h : parameter deciding filter strength. Higher h value removes noise better, but removes details of image also. (10 is ok)
- hForColorComponents : same as h, but for color images only. (normally same as h)
- templateWindowSize : should be odd. (recommended 7)
- searchWindowSize : should be odd. (recommended 21)

Please visit the first link in additional resources for more details on these parameters.

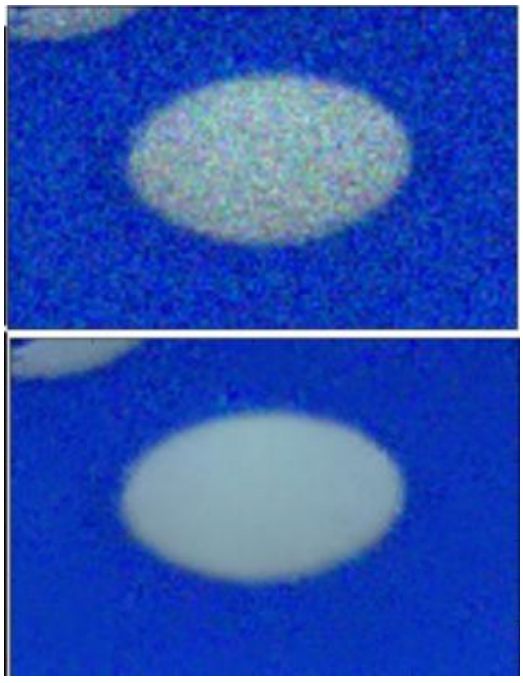
We will demonstrate 1 here. Rest is left for you.

1. cv.fastNlMeansDenoisingColored()

As mentioned above it is used to remove noise from color images. (Noise is expected to be gaussian). See the example below:

die.jpg

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('die.jpg')
dst = cv.fastNlMeansDenoisingColored(img,None,10,10,7,21)
dst2 = cv.fastNlMeansDenoisingColored(img,None,7,21,7,10)
dst3 = cv.fastNlMeansDenoisingColored(img,None,15,21,7,10)
plt.figure(figsize=(20,20))
plt.subplot(141),plt.imshow(img)
plt.subplot(142),plt.imshow(dst2)
plt.subplot(143),plt.imshow(dst2)
plt.subplot(144),plt.imshow(dst3)
plt.show()
```



Practical 5 Image Transforms

Geometric Transformations of Images

Goals

- Learn to apply different geometric transformations to images, like translation, rotation, affine transformation etc.
- You will see these functions: **cv.getPerspectiveTransform**

OpenCV provides two transformation functions, **cv.warpAffine** and **cv.warpPerspective**, with which you can perform all kinds of transformations. **cv.warpAffine** takes a 2x3 transformation matrix while **cv.warpPerspective** takes a 3x3 transformation matrix as input.

Scaling

Scaling is just resizing the image. OpenCV comes with a function **cv.resize()** for this purpose. The size of the image can be specified manually, or you can specify the scaling factor. Different interpolation methods are used. Preferable interpolation methods are **cv.INTER_AREA** for shrinking and **cv.INTER_CUBIC** (slow) & **cv.INTER_LINEAR** for zooming. By default, the interpolation method **cv.INTER_LINEAR** is used for all resizing purposes. You can resize an input image with either of following methods:

```
import numpy as np
import cv2 as cv
img = cv.imread('messi5.jpg')
res = cv.resize(img,None,fx=0.5, fy=0.5, interpolation = cv.INTER_CUBIC)
```

Interpolation: In image processing, interpolation is a technique that estimates unknown pixel values, often used when resizing or transforming images, by creating smooth transitions between known pixel values to maintain image quality.

Translation

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

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

You can take make it into a Numpy array of type np.float32 and pass it into the **cv.warpAffine()** function. See the below example for a shift of (100,50):

```
import numpy as np
import cv2 as cv
img = cv.imread('messi.jpg',0)
rows,cols = img.shape
M = np.float32([[1,0,100],[0,1,50]])
dst = cv.warpAffine(img,M,(cols,rows))
```

```
cv.imshow('img',dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

Rotation

Rotation of an image for an angle θ is achieved by the transformation matrix of the form

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

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

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

$$\begin{aligned} \alpha &= scale \cdot \cos\theta, \\ \beta &= scale \cdot \sin\theta \end{aligned}$$

To find this transformation matrix, OpenCV provides a function, **cv.getRotationMatrix2D**. Check out the below example which rotates the image by 90 degree with respect to center without any scaling.

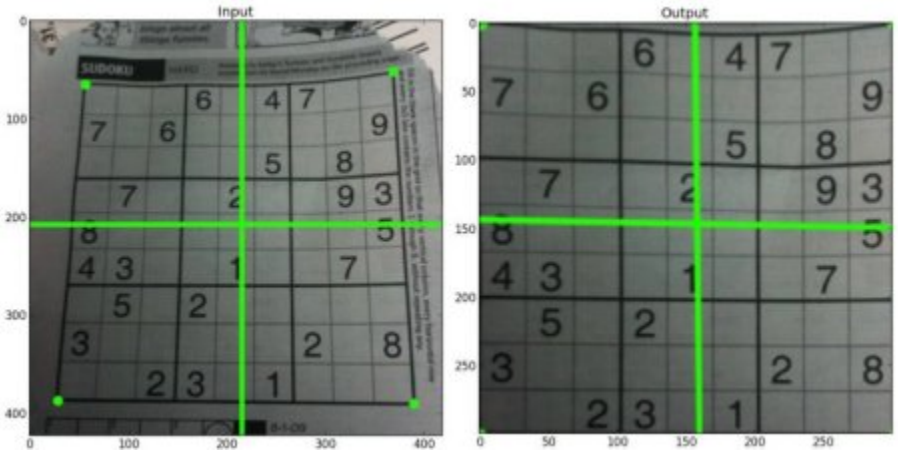
```
img = cv.imread('messi.jpg',0)
rows,cols = img.shape
# cols-1 and rows-1 are the coordinate limits.
M = cv.getRotationMatrix2D(((cols-1)/2.0,(rows-1)/2.0),90,1) ## counterclock wise
dst = cv.warpAffine(img,M,(cols,rows))
```

Perspective Transformation

For perspective transformation, you need a 3x3 transformation matrix. Straight lines will remain straight even after the transformation. To find this transformation matrix, you need 4 points on the input image and corresponding points on the output image. Among these 4 points, 3 of them should not be collinear. Then the transformation matrix can be found by the function **cv.getPerspectiveTransform**. Then apply **cv.warpPerspective** with this 3x3 transformation matrix.

See the code below:

- sudoku.jpg



Python

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('sudoku.jpg')
rows,cols,ch = img.shape
pts1 = np.float32([[56,65],[368,52],[28,387],[389,390]])
pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])
M = cv.getPerspectiveTransform(pts1,pts2)
dst = cv.warpPerspective(img,M,(300,300))
plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

Add in the Mouse Event Listener

Python

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

# Read the image
img = cv.imread('sudoku.jpg')
if img is None:
    print("Error: Image not found.")
    exit()

rows, cols, ch = img.shape

# List to store points
pts1 = []
temp_img = img.copy()

def select_points(event, x, y, flags, param):
    """Callback function to store clicked points."""
    global pts1, temp_img
    if event == cv.EVENT_LBUTTONDOWN:
        pts1.append((x, y))
        cv.circle(temp_img, (x, y), 5, (0, 0, 255), -1)  # Mark point
        cv.imshow('Select 4 Points', temp_img)

        if len(pts1) == 4:
            cv.destroyAllWindows()

# Show the image and set the mouse callback
```

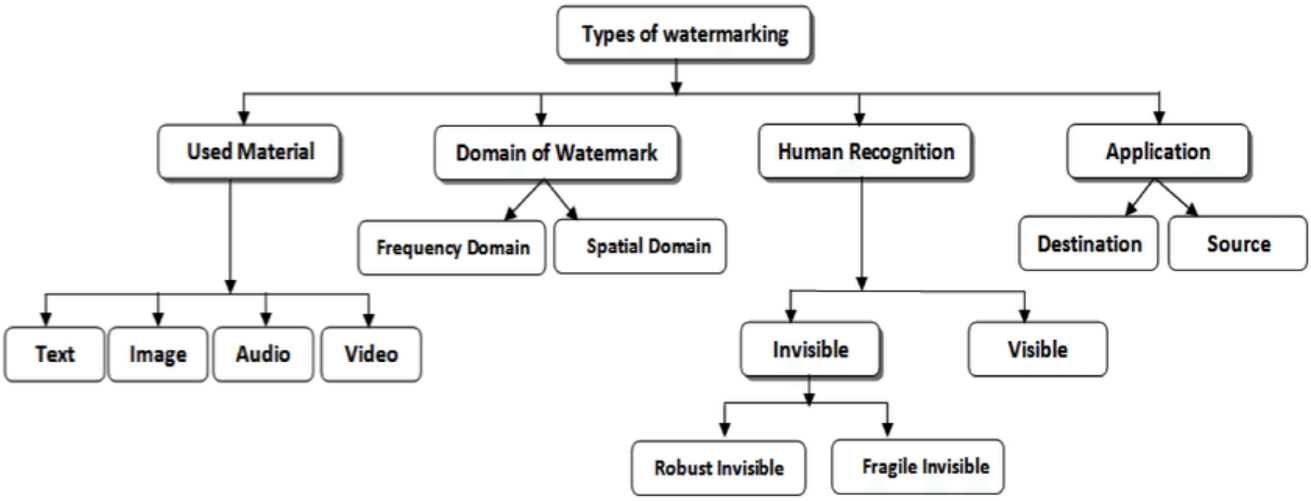
```
cv.namedWindow('Select 4 Points', cv.WINDOW_NORMAL) ## for resizeable
cv.imshow('Select 4 Points', temp_img)
cv.setMouseCallback('Select 4 Points', select_points)
cv.waitKey(0)  # Wait for user to select 4 points

if len(pts1) == 4:
    pts1 = np.float32(pts1)
    pts2 = np.float32([[0, 0], [300, 0], [0, 300], [300, 300]])  # Target
points

    # Compute the perspective transform matrix
    M = cv.getPerspectiveTransform(pts1, pts2)
    dst = cv.warpPerspective(img, M, (300, 300))

    # Display results
    plt.subplot(121), plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB)),
plt.title('Input')
    plt.subplot(122), plt.imshow(cv.cvtColor(dst, cv.COLOR_BGR2RGB)),
plt.title('Output')
    plt.show()
else:
    print("You need to select exactly 4 points.")
```

Practical 6 Watermarking

Watermarking is the process of hiding a secret message within a larger one in such a way that someone can not know the presence or contents of the hidden message. The purpose of watermarking is to maintain **secret communication** between two parties.

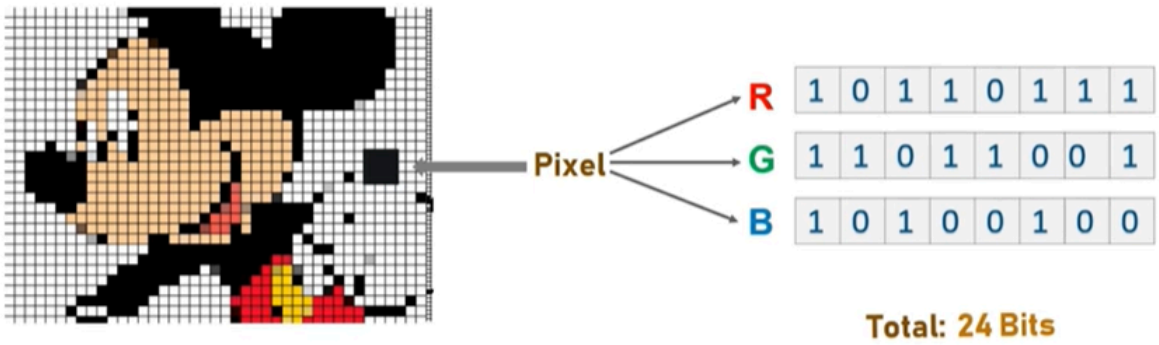
Types of Watermarking

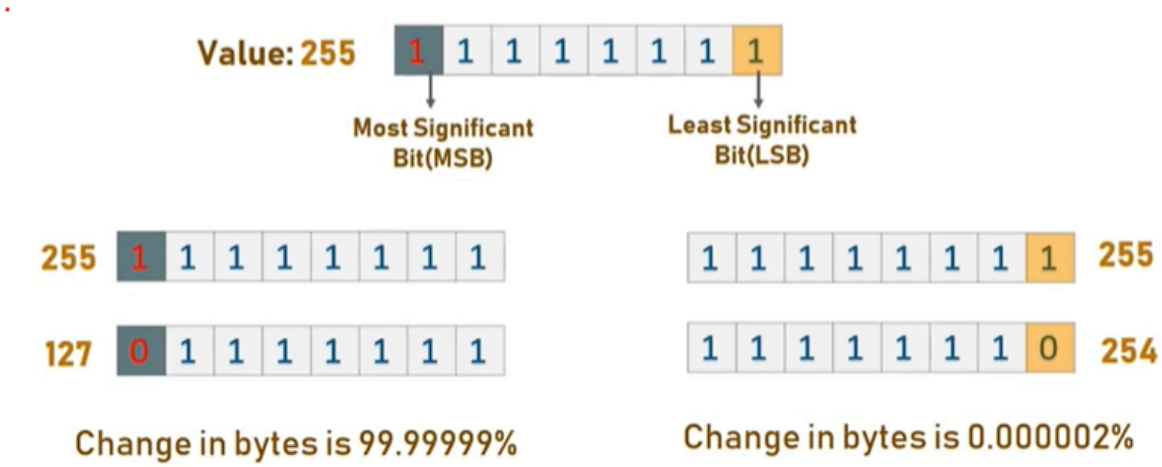
Watermarking works have been carried out on different transmission media like images, video, text, or audio.



Spatial Domain (Least Significant Bit Watermarking)

Least Significant Bit (LSB) is a technique in which the last bit of each pixel is modified and replaced with the secret message's data bit.





🔊 From the above image it is clear that, if we change MSB it will have a larger impact on the final value but if we change the LSB the impact on the final value is minimal, thus we use least significant bit watermarking.

How does LSB technique work?

Each pixel contains three values which are Red, Green, Blue, these values range from 0 to 255, in other words, they are 8-bit values.

Let's take an example of how this technique works, suppose you want to hide the message "**hi**" into a 4x3 image which has the following pixel values:

[(225, 12, 99), (155, 2, 50), (99, 51, 15), (15, 55, 22),(155, 61, 87), (63, 30, 17), (1, 55, 19), (99, 81, 66),(219, 77, 91), (69, 39, 50), (18, 200, 33), (25, 54, 190)]

Using the ASCII Table, we can convert the secret message from ASCII into binary:

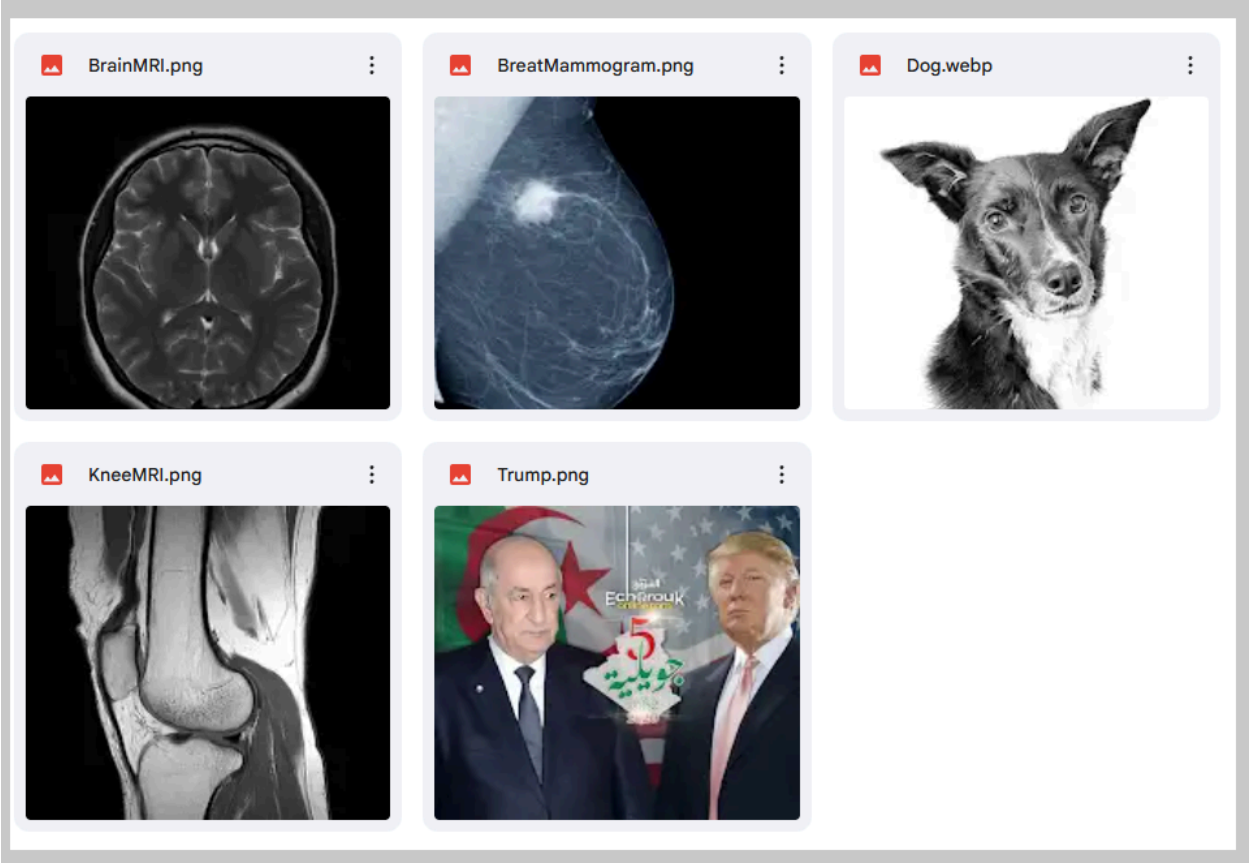
"h" (ASCII) > 01101000 (binary)
"i" (ASCII) > 01101001 (binary)

Now, we iterate over the pixel values one by one, after converting them to binary, we replace each least significant bit with that message bits sequentially (e.g 225 is ,11100001 we replace the last bit, the bit in the right (1) with the first data bit (0) and so on).This will only modify the pixel values by +1 or -1 which is not noticeable at all. The resulting pixel values after performing LSBs is as shown below:

Before:
[(225, 12, 99**), (155, 2,** 50**), (99, 51,** 15**), (**15**, 55, 22),(155,** 61, 87), (63, 30, 17), (1, 55, 19), (99, 81, 66),(219, 77, 91), (69, 39, 50), (18, 200, 33), (25, 54, 190)]

After:
[(224, 13, 99**),(154, 3,** 50**),(98, 50, 14),(**15**,** 55**, 23),(154,** 61, 87),(63, 30, 17),(1, 55, 19),(99, 81, 66),(219, 77, 91),(69, 39, 50),(18, 200, 33),(25, 54, 190)]

Testing Image: Sample Image⏬



Step 1: Import all the required python libraries

```python
import cv2
import numpy as np
import types
```

Step 2: Define a function to convert any type of data into binary, we will use this to convert the secret data and pixel values to binary in the encoding and decoding phase.

```python
def messageToBinary(message):
  if type(message) == str:
    return ''.join([ format(ord(i), "08b") for i in message ])
  elif type(message) == bytes or type(message) == np.ndarray:
    return [ format(i, "08b") for i in message ]
  elif type(message) == int or type(message) == np.uint8:
    return format(message, "08b")
  else:
    raise TypeError("Input type not supported")
```

Step 3: Write a function to hide secret message into the image by altering the LSB

```python
# Function to hide the secret message into the image
def hideData(image, secret_message):

  # calculate the maximum bytes to encode
  n_bytes = image.shape[0] * image.shape[1] * 3 // 8
  print("Maximum bytes to encode:", n_bytes)

  #Check if the number of bytes to encode is less than the maximum bytes in the
image
  if len(secret_message) > n_bytes:
      raise ValueError("Error encountered insufficient bytes, need bigger image
or less data !!")
```

```
  secret_message += "#####" # you can use any string as the delimiter

  data_index = 0
  # convert input data to binary format using messageToBinary() fucntion
  binary_secret_msg = messageToBinary(secret_message)

  data_len = len(binary_secret_msg) #Find the length of data that needs to be
hidden
  for values in image:
      for pixel in values:
          # convert RGB values to binary format
          r, g, b = messageToBinary(pixel)
          # modify the least significant bit only if there is still data to
store
          if data_index < data_len:
              # hide the data into least significant bit of red pixel
              pixel[0] = int(r[:-1] + binary_secret_msg[data_index], 2)
              data_index += 1
          if data_index < data_len:
              # hide the data into least significant bit of green pixel
              pixel[1] = int(g[:-1] + binary_secret_msg[data_index], 2)
              data_index += 1
          if data_index < data_len:
              # hide the data into least significant bit of  blue pixel
              pixel[2] = int(b[:-1] + binary_secret_msg[data_index], 2)
              data_index += 1
          # if data is encoded, just break out of the loop
          if data_index >= data_len:
              break

  return image
```

Step 4: Define a function to decode the hidden message from the watermarked image

Python

```
# Encode data into image
def encode_text():
image_name = input("Enter image name(with extension): ")
image = cv2.imread(image_name) # Read the input image using OpenCV-Python.
```

```
    #It is a library of Python bindings designed to solve computer vision
problems.

    #details of the image
    print("The shape of the image is: ",image.shape) #check the shape of image
to calculate the number of bytes in it
    print("The original image is as shown below: ")
    resized_image = cv2.resize(image, (500, 500)) #resize the image as per your
requirement
    cv2.imshow('',resized_image) #display the image
    cv2.waitKey(0)


    data = input("Enter data to be encoded : ")
    if (len(data) == 0):
        raise ValueError('Data is empty')

    filename = input("Enter the name of new encoded image(with extension): ")
    encoded_image = hideData(image, data) # call the hideData function to hide
the secret message into the selected image
    cv2.imwrite(filename, encoded_image)
```

Step 5: Function that takes the input image name and secret message as input from user and calls hideData() to encode the message

Python

```
# Encode data into image
def encode_text():
    image_name = input("Enter image name(with extension): ")
    image = cv2.imread(image_name) # Read the input image using OpenCV-Python.
    #It is a library of Python bindings designed to solve computer vision
problems.

    #details of the image
    print("The shape of the image is: ",image.shape) #check the shape of image
to calculate the number of bytes in it
    print("The original image is as shown below: ")
    resized_image = cv2.resize(image, (500, 500)) #resize the image as per your
requirement
```

```
    cv2.imshow('',resized_image) #display the image
    cv2.waitKey(0)


    data = input("Enter data to be encoded : ")
    if (len(data) == 0):
        raise ValueError('Data is empty')

    filename = input("Enter the name of new encoded image(with extension): ")
    encoded_image = hideData(image, data) # call the hideData function to hide
the secret message into the selected image
    cv2.imwrite(filename, encoded_image)
```

Step 6: Create a function to ask the user to enter the name of the image that needs to be decoded and call the showData() function to return the decoded message.

Python

```
def showData(image):

  binary_data = ""
  for values in image:
      for pixel in values:
          r, g, b = messageToBinary(pixel) #convert the red,green and blue
values into binary format
          binary_data += r[-1] #extracting data from the least significant bit
of red pixel
          binary_data += g[-1] #extracting data from the least significant bit
of red pixel
          binary_data += b[-1] #extracting data from the least significant bit
of red pixel
  # split by 8-bits
  all_bytes = [ binary_data[i: i+8] for i in range(0, len(binary_data), 8) ]
  # convert from bits to characters
  decoded_data = ""
  for byte in all_bytes:
      decoded_data += chr(int(byte, 2))
      if decoded_data[-5:] == "#####": #check if we have reached the delimeter
which is "#####"
```

```
            break
    #print(decoded_data)
    return decoded_data[:-5] #remove the delimeter to show the original hidden
message


# Decode the data in the image
def decode_text():
    # read the image that contains the hidden image
    image_name = input("Enter the name of the watermarked image that you want
to decode (with extension) :")
    image = cv2.imread(image_name) #read the image using cv2.imread()

    print("The Watermarked image is as shown below: ")
    resized_image = cv2.resize(image, (500, 500))  #resize the original image
as per your requirement
    cv2.imshow('',resized_image) #display the Watermarked image
    cv2.waitKey(0)

    text = showData(image)
    return text
```

Step 7: Main Function()

Python

```
# Image Watermarking
def Watermarking():
    a = input("Image Watermarking \n 1. Encode the data \n 2. Decode the data
\n Your input is: ")
    userinput = int(a)
    if (userinput == 1):
        print("\nEncoding....")
        encode_text()

    elif (userinput == 2):
        print("\nDecoding....")
        print("Decoded message is " + decode_text())
    else:
        raise Exception("Enter correct input")
```

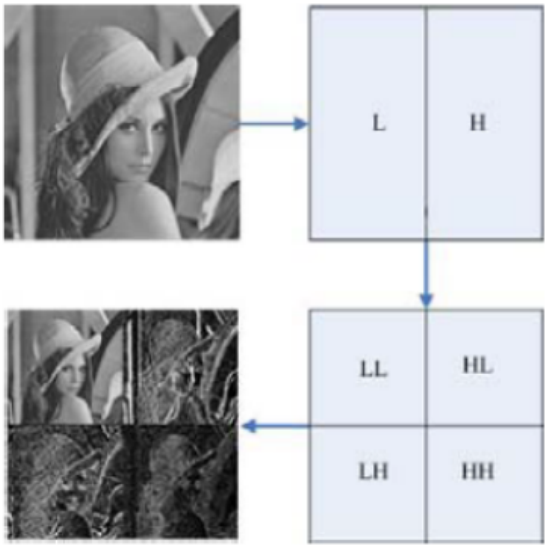
Final results: Image Processing Practical 7A - Spatial Domain.ipynb (In case, you didn't get it through the steps above.)

Measure the visual quality: Image Processing Practical 7A - Spatial Domain with PSNR.ipynb

Frequency Domain (DWT+DCT)

Discrete Wavelet transform (DWT) for image processing is one of the most popular methods in the frequency domain, the information of images could be represented as a group based on this method.

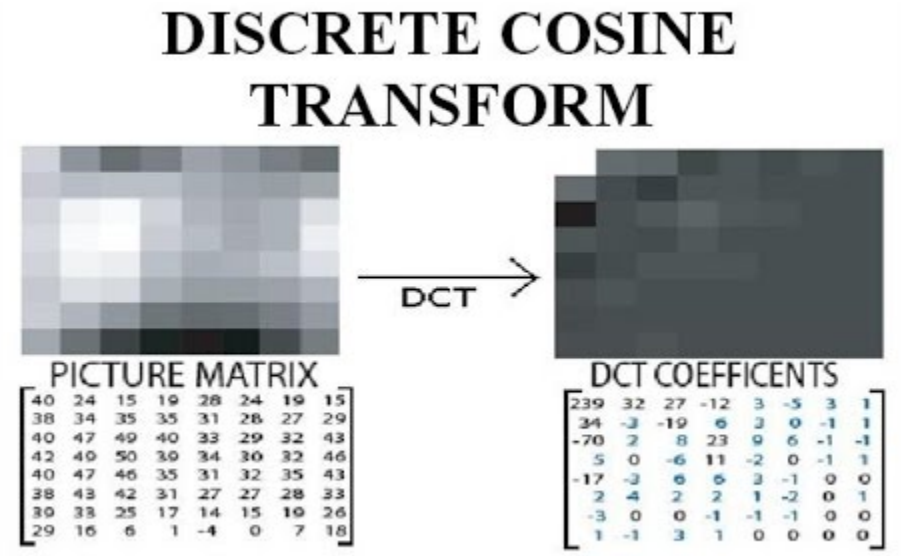
We can use to pywt.**dwt2** decompose our image to the DWT domain.

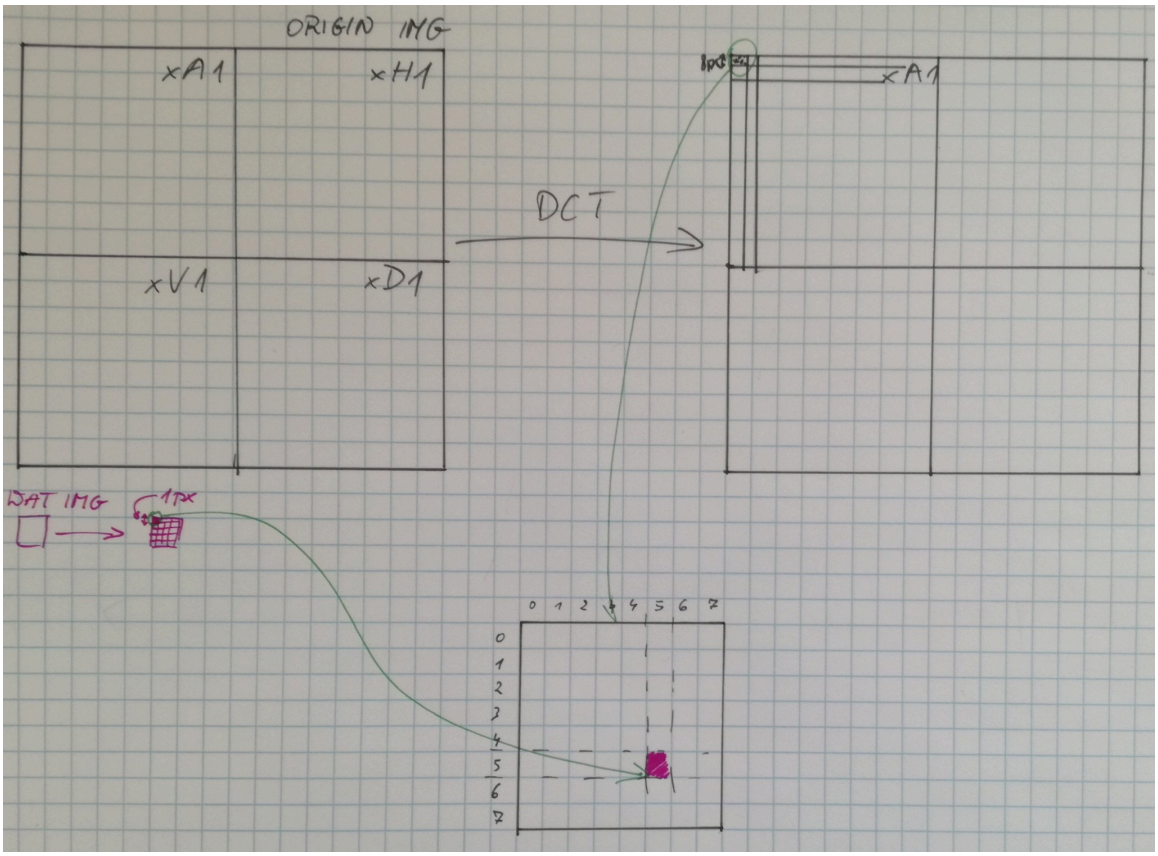


- LL is the **approximate** image of the input image; it is a low frequency subband so it is used for further decomposition process.
- LH subbands extract the horizontal features of the original image.
- HL subband gives vertical features .
- HH subband gives diagonal features = High in vertical & High in horizontal

Discrete Cosine Transform (DCT)

We can use **scipy.fftpack** to decompose our image to the DCT domain.


DISCRETE COSINE TRANSFORM
DCT
PICTURE MATRIX
40 24 15 19 28 24 19 15
38 34 35 35 31 28 27 29
40 47 49 40 33 29 32 43
42 49 50 39 34 30 32 46
40 47 46 35 31 32 35 43
38 43 42 31 27 27 28 33
30 33 25 17 14 15 19 26
29 16 6 1 -4 0 7 18
DCT COEFFICENTS
239 32 27 -12 3 -5 3 1
34 -3 -19 6 3 0 -1 1
-70 2 8 23 9 6 -1 -1
5 0 -6 11 -2 0 -1 1
-17 -3 6 6 3 -1 0 0
2 4 2 2 1 -2 0 1
-3 0 0 -1 -1 -1 0 0
1 -1 3 1 0 0 0 0

- 
ORIGIN IMG
xA1
xH1
xV1
xD1
DCT
xA1
DAT IMG
1px
0 1 2 3 4 5 6 7

qrcode.png mis1.jpg

Peak Signal-to-Noise Ratio (PSNR)

(higher value, higher similarity= better for watermarking)

Peak signal-to-noise ratio (PSNR) is the ratio between the maximum possible power of an image and the power of corrupting noise that affects the quality of its representation. To estimate the PSNR of an image, it is necessary to compare that image to an ideal clean image with the maximum possible power.

PSNR is most commonly used to estimate the efficiency of compressors, filters, etc. The larger the value of PSNR, the more efficient a corresponding compression or filter method.

PSNR is defined as follows:

$$PSNR = 10log_{10}(\frac{(L-1)^2}{MSE}) = 20log_{10}(\frac{L-1}{RMSE})$$

Here, L is the number of maximum possible intensity levels (minimum intensity level supposed to be 0) in an image.

MSE is the mean squared error & it is defined as:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(O(i,j) - D(i,j))^2$$

Where, O represents the matrix data of the original image. D represents the matrix data of a degraded image. m represents the numbers of rows of pixels and i represents the index of that row of the image. n represents the number of columns of pixels and j represents the index of that column of the image.

RMSE is the root mean squared error.

Let's move to this file for Invisible Watermarking in DWT+DCT domain: Image Processing Practical 7B - Frequency Domain Watermarking.ipynb

Practical 7 Morphological Image Processing

Goal

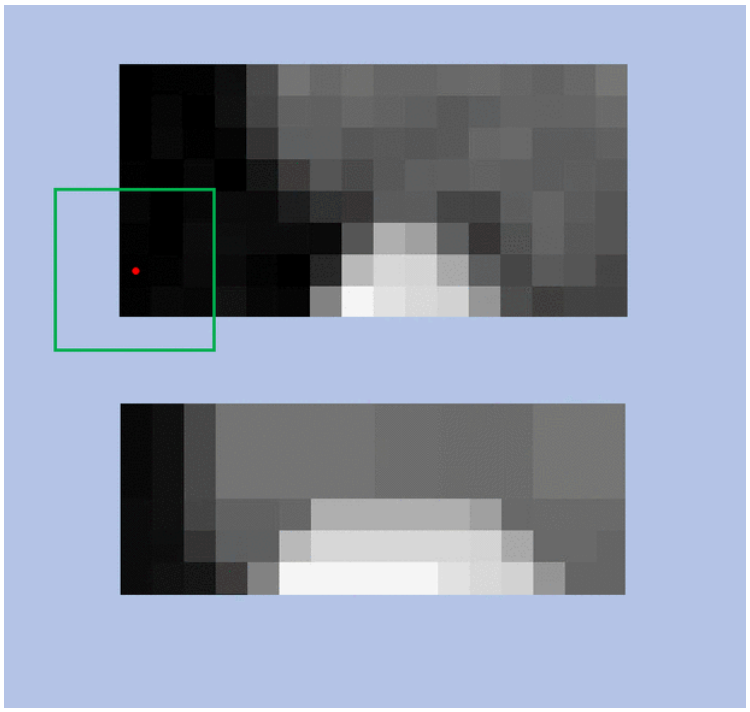
In this tutorial you will learn how to:

- Apply two very common morphological operators: Erosion and Dilation. For this purpose, you will use the following OpenCV functions:
    - **cv::erode**
    - **cv::dilate**

Morphological Operations

- In short: A set of operations that process images based on shapes. Morphological operations apply a structuring element to an input image and generate an output image.
- The most basic morphological operations are: Erosion and Dilation. They have a wide array of uses, i.e. :
    - Removing noise
    - Isolation of individual elements and joining disparate elements in an image.
    - Finding of intensity bumps or holes in an image
- We will explain dilation and erosion briefly, using the following image as an example:

Dilation

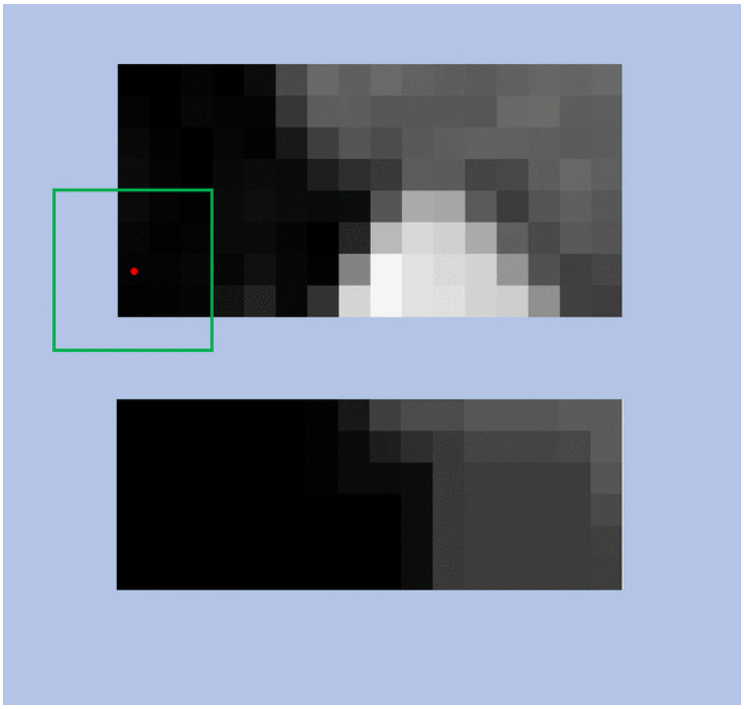


- This operation consists of convolving an image A with some kernel (B), which can have any shape or size, usually a square or circle.
- The kernel B has a defined *anchor point*, usually being the center of the kernel.
- As the kernel B is scanned over the image, we compute the **maximal pixel value** overlapped by B and replace the image pixel in the anchor point position with that maximal value. As you can deduce, this maximizing operation causes bright regions within an image to "grow" (therefore the name *dilation*).
- The dilatation operation is: $\texttt{dst}(x,y)=\max_{(x',y')\,\texttt{element}(x',y')\neq 0}\texttt{src}(x+x',y+y')$
- Take the above image as an example. Applying dilation we can get:

Before After

- Increases the object area
- Used to accentuate (强调) features

Erosion



- This operation is the sister of dilation. It computes a local minimum over the area of a given kernel.
- As the kernel B is scanned over the image, we compute the **minimal pixel value** overlapped by B and replace the image pixel under the anchor point with that minimal value.
- The erosion operation is: $dst(x,y)=\min_{(x',y):element(x',y')\neq 0} src(x+x',y+y')$
- Analogously to the example for dilation, we can apply the erosion operator to the original image (shown above). You can see in the result below that the bright areas of the image get thinner, whereas the dark zones gets bigger.

Before

After

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

signature.png LinuxLogo.jpg watermark41.jpg

Method 1:

```
# Python program to demonstrate erosion and
# dilation of images.
import cv2
import numpy as np

# Reading the input image
img = cv2.imread('signature.png', 0)

# Taking a matrix of size 5 as the kernel
kernel = np.ones((5,5), np.uint8)

# The first parameter is the original image,
# kernel is the matrix with which image is
# convolved and third parameter is the number
# of iterations, which will determine how much
# you want to erode/dilate a given image.
img_erosion = cv2.erode(img, kernel, iterations=10)
img_dilation = cv2.dilate(img, kernel, iterations=1)

cv2.imshow('Input', img)
cv2.imshow('Erosion', img_erosion)
cv2.imshow('Dilation', img_dilation)

cv2.waitKey(0)
```

Closing (CDE > C = D + E)

Closing = Dilation + Erosion

- As the name suggests, a closing is used to **close holes inside** of objects or for **connecting components together**.
- Useful to remove small holes (dark regions).

```
closing = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)
```

Opening (OED)

Opening = Erosion + Dilation

- Useful for **removing small objects** (it is assumed that the objects are bright on a dark foreground)

```
opening = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)
```

Morphological Gradient

MG = Dilation - Erosion

- The result will look like the outline of the object.
  character_j.png

```
gradient = cv.morphologyEx(img, cv.MORPH_GRADIENT, kernel)
```

Top Hat

TH = Image - Opening

character_j.png

```
tophat = cv.morphologyEx(img, cv.MORPH_TOPHAT, kernel)
```

Black Hat

BH = Closing - Input

character_j.png

```
blackhat = cv.morphologyEx(img, cv.MORPH_BLACKHAT, kernel)
```

🔎Practical Assessment

Objective:

Implement a program that performs morphological operations on an image. The user should be able to choose which operation to perform using the keyboard and also select a region of interest (ROI) using the mouse.

Morphological Operations (ONE marks for each completed and workable morphological operation):

1. Dilation
2. Erosion

3. Opening
4. Closing
5. Morphological Gradient
6. Top Hat
7. Black Hat

Requirements:

1. **User Interface:**
    - Display the original image.
    - Allow the user to select a morphological operation using keyboard inputs.
    - Allow the user to select a region of interest (ROI) using the mouse.
    - Apply the chosen operation on the selected ROI.
2. **Mouse Interaction:**
    - Use the mouse to draw a rectangle around the region of interest (ROI) in the image.
3. **Keyboard Inputs:**
    - Use specific keys to select the morphological operation:
        - 'd' for Dilation
        - 'e' for Erosion
        - 'o' for Opening
        - 'c' for Closing
        - 'g' for Morphological Gradient
        - 't' for Top Hat
        - 'b' for Black Hat
4. **Display Results:**
    - Display the resulting image after applying the selected morphological operation on the selected ROI.

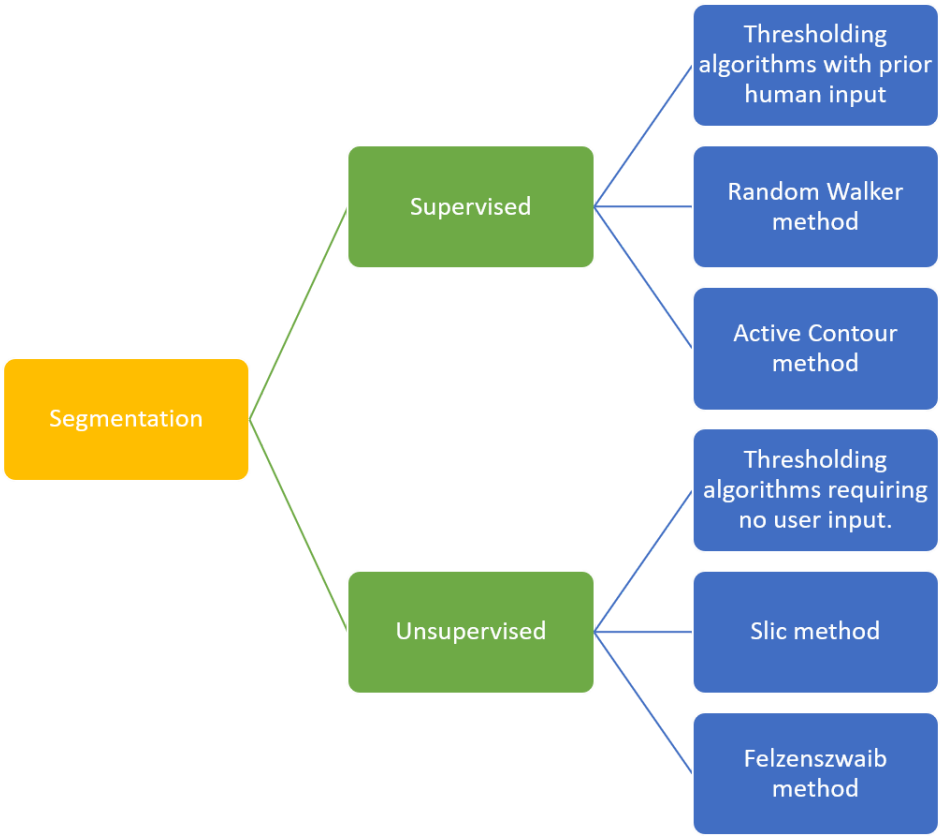
Instructions:

1. **Set Up:**
    - Load an image.
    - Implement the functionality to select ROI using the mouse.
2. **Implement Morphological Operations:**
    - Implement each of the morphological operations.
3. **User Interaction:**
    - Implement keyboard interaction to choose the morphological operation.
    - Implement mouse interaction to select the ROI.
4. **Apply Operations:**
    - Apply the selected operation on the chosen ROI and display the result.

Practical 8 Image Segmentation

Image Segmentation is essentially the process of partitioning a digital image into multiple segments to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse.

ROI = Region of interest
RONI = Region of Non-interest



Some of the Segmentation Algorithms available in the scikit-image library

- Supervised segmentation: Some prior knowledge, possibly from human input, is used to guide the algorithm.

- Unsupervised segmentation: No prior knowledge is required. These algorithms attempt to subdivide images into meaningful regions automatically. The user may still be able to tweak certain settings to obtain desired outputs.

  Let's begin with the simplest algorithm called **Thresholding**.

Thresholding

It is the simplest way to segment objects from the background by choosing pixels above or below a certain threshold. This is generally helpful when we intend to segment objects from their background.

Let's try this on an image of a textbook that comes preloaded with the scikit-image dataset.

Basic Imports

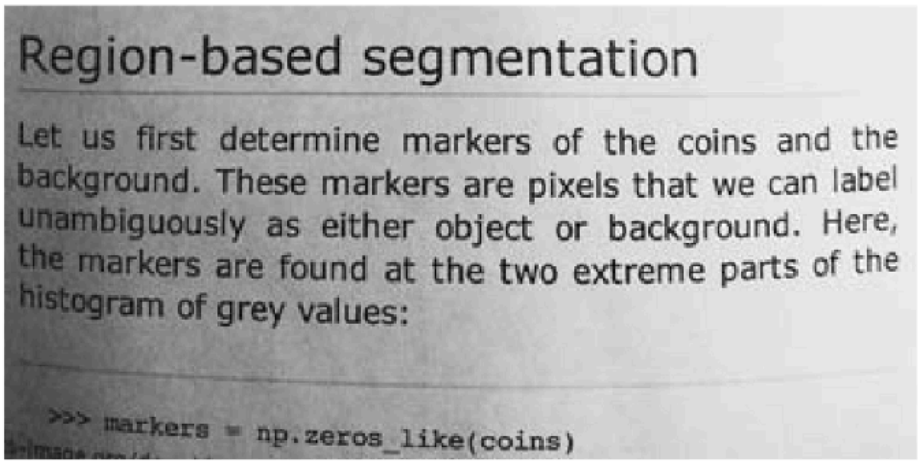
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.data as data
import skimage.segmentation as seg
import skimage.filters as filters
import skimage.draw as draw
import skimage.color as color
```

A simple function to plot the images

```
def image_show(image, nrows=1, ncols=1, cmap='gray'):
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(14, 14))
    ax.imshow(image, cmap='gray')
    ax.axis('off')
    return fig, ax
```

Image (Module: data — skimage v0.19.0.dev0 docs)

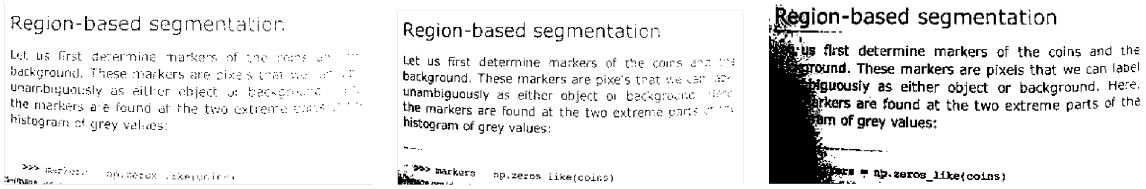
```
text = data.page()#change your image here
image_show(text)
```



Supervised thresholding

Since we will be choosing the thresholding value ourselves, we call it supervised thresholding.

```
text_segmented = text > (value concluded from histogram i.e 50,70,120 )
image_show(text_segmented);
```



Left: text>50 | Middle : text > 70 | Right : text >120

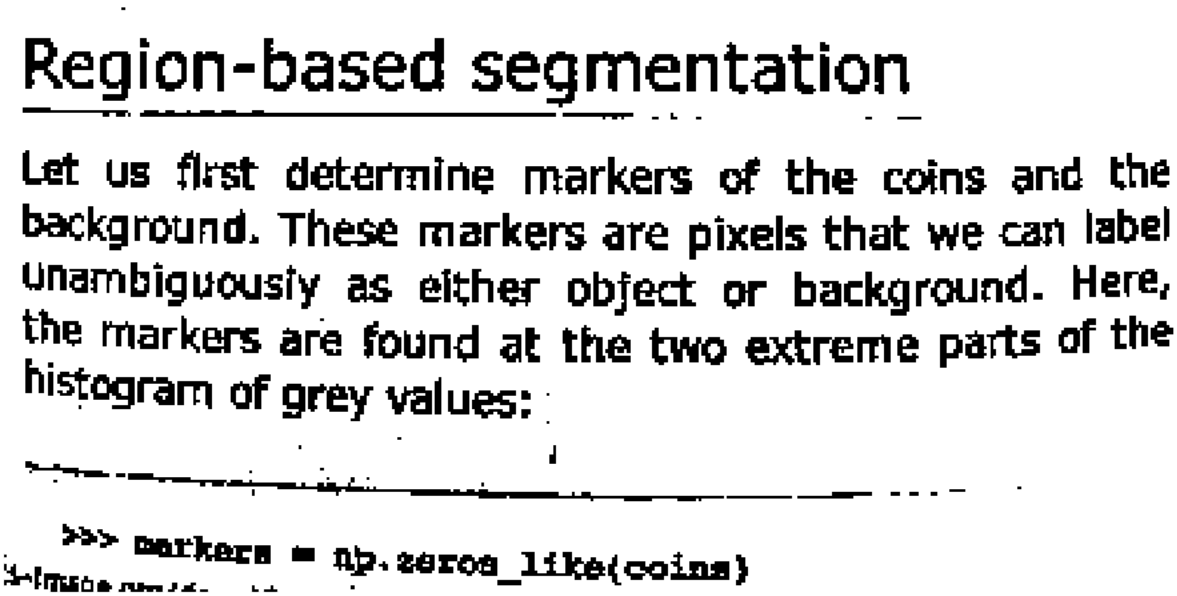
👀 We didn't get any **ideal results** since the shadow on the left creates problems. Let's try with unsupervised thresholding now.

Unsupervised thresholding

Scikit-image has a number of automatic thresholding methods, which require no input in choosing an optimal threshold. Some of the methods are `otsu, li, local`

In the case of local, we also need to specify the block_size . Offset helps to tune the image for better results.

```
text_threshold =
filters.threshold_local(text,block_size=51, offset=10)
image_show(text > text_threshold);
```



local thresholding

👀 This is pretty good and has got rid of the noisy regions to a large extent.

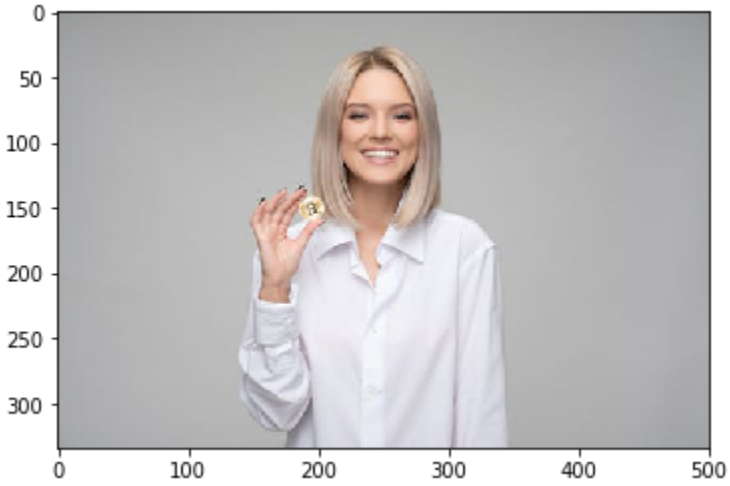
Thresholding is a very basic segmentation process and will not work properly in a high-contrast image for which we will be needing more advanced tools.

Active Contour Segmentation
For this section, we will use an example image that is freely available and attempt to segment the head portion using supervised segmentation techniques.

girl.jpg

```
# import the image
from skimage import io
from skimage import io, color
import numpy as np
import matplotlib.pyplot as plt
import skimage.data as data
import skimage.segmentation as seg
import skimage.filters as filters
import skimage.draw as draw
import skimage.color as color
```

```
image = io.imread('girl.jpg')
plt.imshow(image);
```



Before doing any segmentation on an image, it is a good idea to de-noise it using some filters. However, in our case, the image is not very noisy, so we will take it as it is. Next step would be to convert the image to grayscale with `rgb2gray`.

```
image_gray = color.rgb2gray(image)
image_show(image_gray);
```

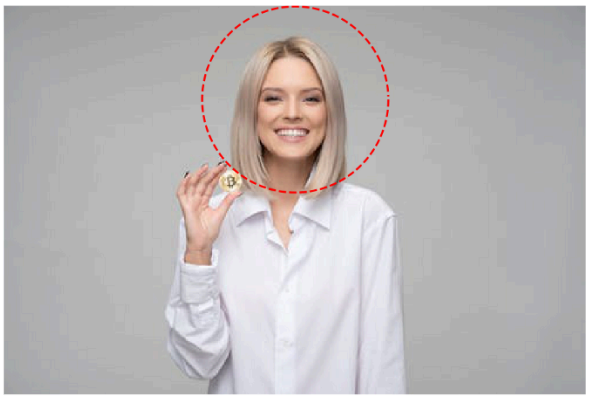
We will use two segmentation methods that work on entirely different principles.

- Action Contour Segmentation
    - Active Contour segmentation is also called **snakes** and is initialized using a user-defined contour or line, around the area of interest, and this contour then slowly contracts and is attracted or repelled from light and edges.
    - Step 1: Draw a circle around the person's head to initialize the snake.

```
def circle_points(resolution, center, radius):
    """
    Generate points which define a circle on an image.Centre refers
to the centre of the circle
    """
    radians = np.linspace(0, 2*np.pi, resolution)
    c = center[1] + radius*np.cos(radians)#polar coordinates
    r = center[0] + radius*np.sin(radians)

    return np.array([c, r]).T
#  Exclude  last  point  because  a  closed  path  should  not  have
duplicate points
points  =  circle_points(200,  [80,  250],  80)[:-1]  #adjust  circle
position here
```

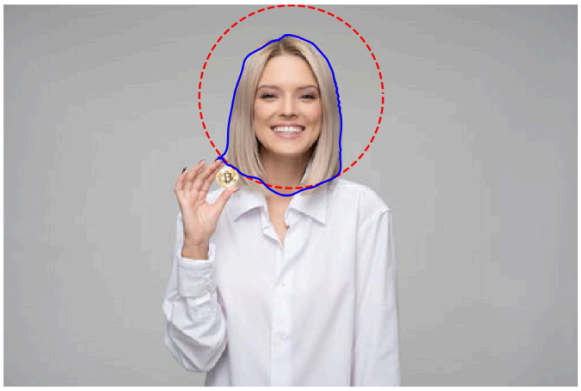
- Step 2: Show the mapping.

```
fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
```



- Step 3: Apply the segmentation.(The algorithm then segments the face of a person from the rest of an image by fitting a closed curve to the edges of the face.)

```
snake = seg.active_contour(image_gray, points)
fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
ax.plot(snake[:, 0], snake[:, 1], '-b', lw=3);
```



- We can tweak the parameters called `alpha` and `beta`.
  - Higher values of alpha will make this snake contract faster
  - Higher beta makes the snake smoother.

```
snake = seg.active_contour(image_gray, points,alpha=0.06,beta=0.3)
fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
ax.plot(snake[:, 0], snake[:, 1], '-b', lw=3);
```

Exercise:

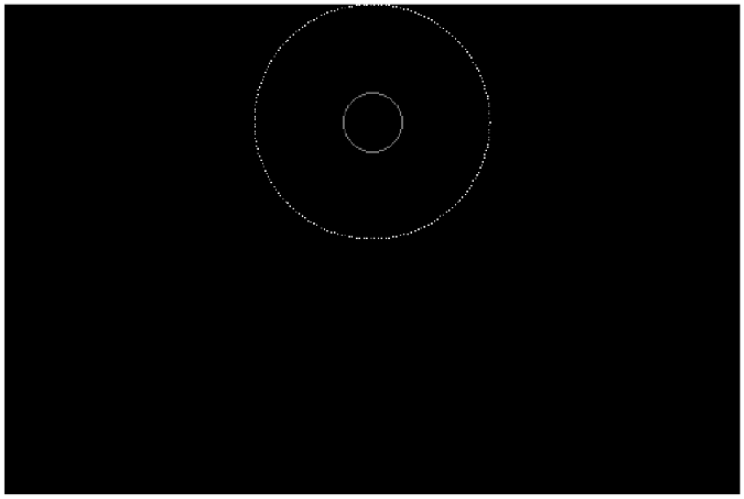
- Try it with your own image.
- Post your results in Padlet.

- Random Walker segmentation
  - In this method, an user interactively labels a small number of pixels which are known as labels. Each unlabeled pixel is then imagined to release a random walker and one can then determine the probability of a random walker starting at each unlabeled pixel and reaching one of the prelabeled pixels. By assigning each pixel to the label for which the greatest probability is calculated, high-quality image segmentation may be obtained. Read the Reference paper here.
  - Step 1: Initialize the image labels

```
image_labels = np.zeros(image_gray.shape, dtype=np.uint8)
```

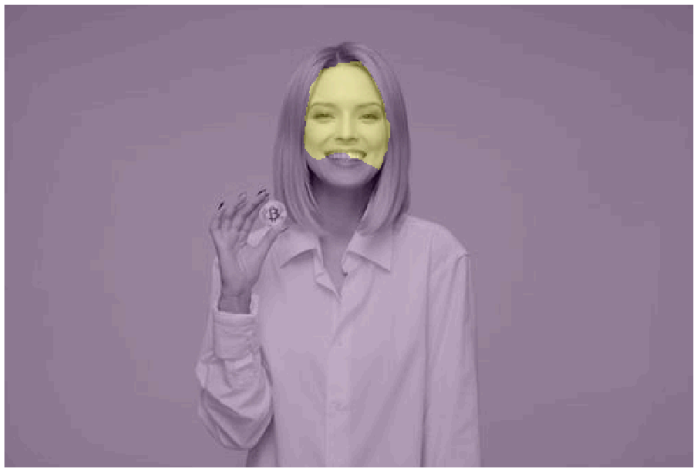
  - Step 2: Create a circle

```
indices = draw.circle_perimeter(180,450,20)
image_labels[indices] = 1
image_labels[points[:,1].astype(int),points[:,0].astype(int)] = 2
image_show(image_labels);
```



- Step 3: Apply Random Walker

```
image_segmented = seg.random_walker(image_gray, image_labels)

# Check our results
fig, ax = image_show(image_gray)
ax.imshow(image_segmented == 1, alpha=0.3);
```



It doesn't look like it's grabbing edges as we wanted. To resolve this situation we can tune in the beta parameter until we get the desired results. After several attempts, a value of 3000 works reasonably well.

```
image_segmented = seg.random_walker(image_gray, image_labels, beta
= 3000)
# Check our results
fig, ax = image_show(image_gray)
ax.imshow(image_segmented == 1, alpha=0.3);
```

Exercise:

- Try to tune the parameters to get the best result that can segment the face only.
- Try with other pictures.
- Post your results in Padlet.

Takeaway:

💔 That's all for Supervised Segmentation where we had to provide certain inputs and also had to tweak certain parameters. However, it is not always possible to have a human looking at an image and then deciding what inputs to give or where to start from.

Practical 9 Image Segmentation with Watershed Algorithm

Goal

In this chapter,

- We will learn to use marker-based image segmentation using watershed algorithm
- We will see: **cv.watershed()**

Theory

Any grayscale image can be viewed as a topographic surface where high intensity denotes peaks and hills while low intensity denotes valleys. You start filling every isolated valley (local minima) with different coloured water (labels). As the water rises, depending on the peaks (gradients) nearby, water from different valleys, obviously with different colours will start to merge. To avoid that, you build barriers in the locations where water merges. You continue the work of filling water and building barriers until all the peaks are under water. Then the barriers you created give you the segmentation result. This is the "philosophy" behind the watershed. You can visit the CMM webpage on watershed to understand it with the help of some animations.

But this approach gives you oversegmented result due to noise or any other irregularities in the image. So OpenCV implemented a marker-based watershed algorithm where you specify which are all valley points are to be merged and which are not. It is an interactive image segmentation. What we do is to give different labels for our object we know. Label the region which we are sure of being the foreground or object with one color (or intensity), label the region which we are sure of being background or non-object with another color and finally the region which we are not sure of anything, label it with 0. That is our marker. Then apply watershed algorithm. Then our marker will be updated with the labels we gave, and the boundaries of objects will have a value of -1.

Code

Below we will see an example on how to use the Distance Transform along with watershed to segment mutually touching objects.

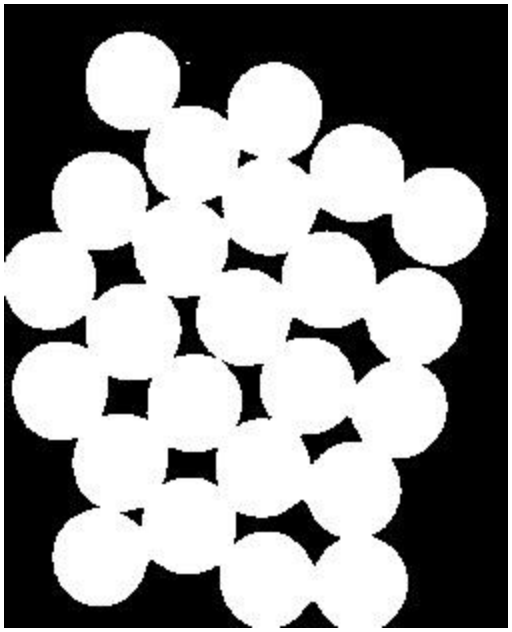
Consider the coins image below, the coins are touching each other. Even if you threshold it, it will be touching each other.

coins

We start with finding an approximate estimate of the coins. For that, we can use the Otsu's binarization.

- coins.jpg

```
import numpy as np
import cv2 as cv
%from matplotlib import pyplot as plt
img = cv.imread('coins.jpg')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(gray,0,255,cv.THRESH_BINARY_INV+cv.THRESH_OTSU)
```

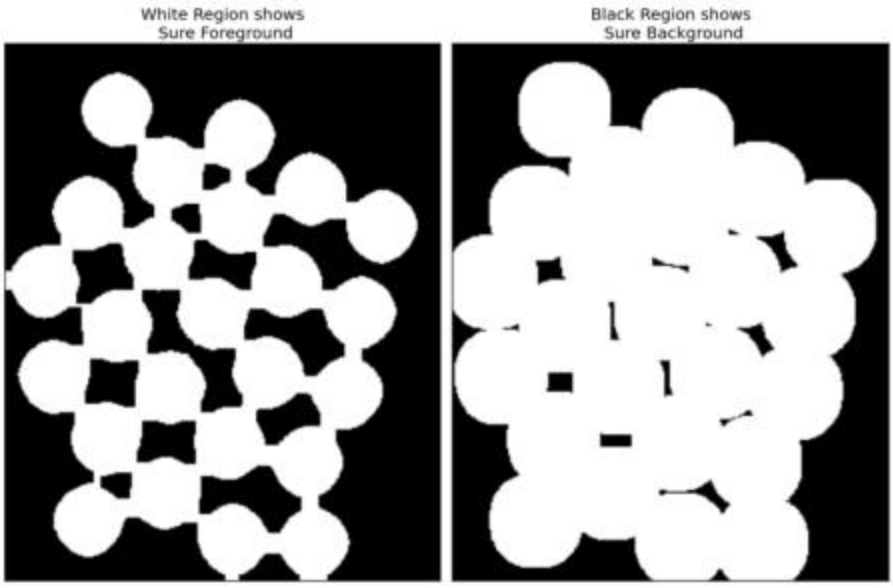


Result after Otsu's binarization

- Now we need to remove any small white noises in the image. For that we can use morphological opening.
- To remove any small holes in the object, we can use morphological closing.

So, now we know for sure that regions near to the centre of objects are foreground and regions much away from the object are background. Only region we are not sure of is the boundary region of coins.

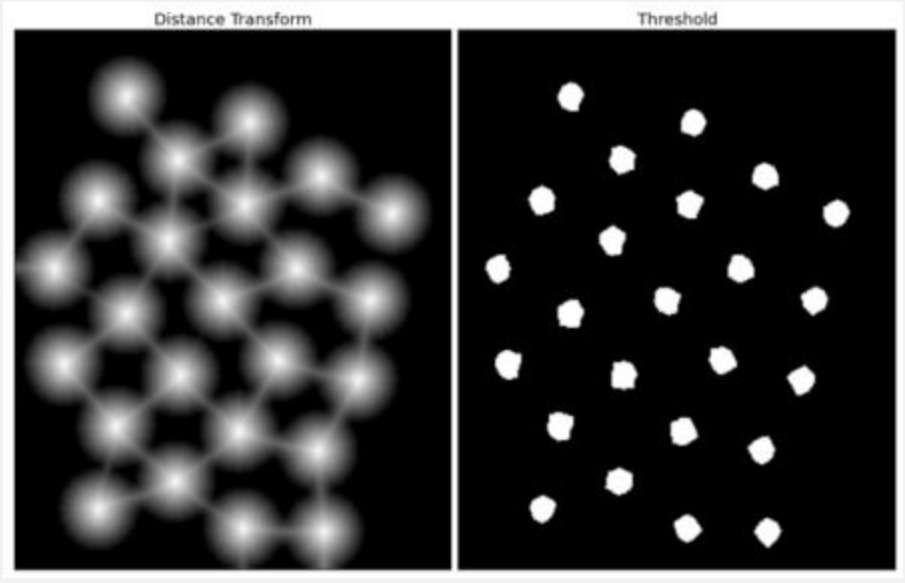
So we need to extract the area which we are sure are coins. Erosion removes the boundary pixels. So whatever remains, we can be sure it is a coin. That would work if objects were not touching each other. But since they are touching each other, another good option would be to find the distance transform and apply a proper threshold. Next we need to find the area which we are sure are not coins. For that, we dilate the result. Dilation increases object boundary to background. This way, we can make sure whatever region in the background in result is really a background, since the boundary region is removed. See the image below.



The remaining regions are those which we don't have any idea about, whether it is coins or background. The Watershed algorithm should find it. These areas are normally around the boundaries of coins where foreground and background meet (Or even two different coins meet). We call it the border. It can be obtained from subtracting sure_fg area from sure_bg area.

```
# noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv.morphologyEx(thresh,cv.MORPH_OPEN,kernel, iterations = 2)

# sure background area (RONI)
sure_bg = cv.dilate(opening,kernel,iterations=3)

# Finding sure foreground area (ROI)
dist_transform = cv.distanceTransform(opening,cv.DIST_L2,5)
ret, sure_fg = cv.threshold(dist_transform,0.7*dist_transform.max(),255,0)

# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv.subtract(sure_bg,sure_fg)
```

See the result. In the thresholded image, we get some regions of coins which we are sure of and they are detached now. (In some cases, you may be interested in only foreground segmentation, not in separating the mutually touching objects. In that case, you need not use distance transform, just erosion is sufficient. Erosion is just another method to extract sure foreground area, that's all.)



Now we know for sure which region of coins, which are background and all. So we create a marker (it is an array of the same size as that of the original image, but with int32 datatype) and label the regions inside it. The regions we know for sure (whether foreground or background) are labelled with any positive integers, but different integers, and the area we don't know for sure are just left as zero. For this we use cv.connectedComponents(). It labels the background of the image with 0, then other objects are labelled with integers starting from 1.

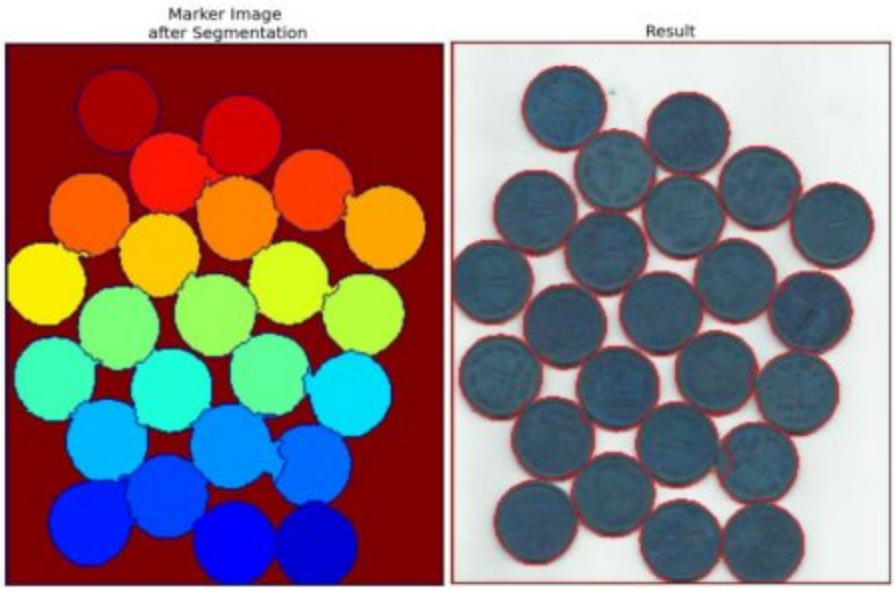
But we know that if the background is marked with 0, watersheds will consider it as an unknown area. So we want to mark it with a different integer. Instead, we will mark unknown regions, defined by unknown, with 0.

```
# Marker labelling
ret, markers = cv.connectedComponents(sure_fg)

# Add one to all labels so that sure background is not 0, but 1
markers = markers+1

# Now, mark the region of unknown with zero
markers[unknown==255] = 0
```

Now our marker is ready. It is time for the final step, apply watershed. Then the marker image will be modified. The boundary region will be marked with -1.

```
markers = cv.watershed(img,markers)
img[markers == -1] = [255,0,0]
```

See the result below. For some coins, the region where they touch are segmented properly and for some, they are not.



Exercise

1. Show intermediate images (image after each operation).
2. Figure out which parameter shows the number of objects.
3. Try with your image and post your results in Padlet.

Practical 10 Feature Detection and Description

Harris Corner Detector in OpenCV

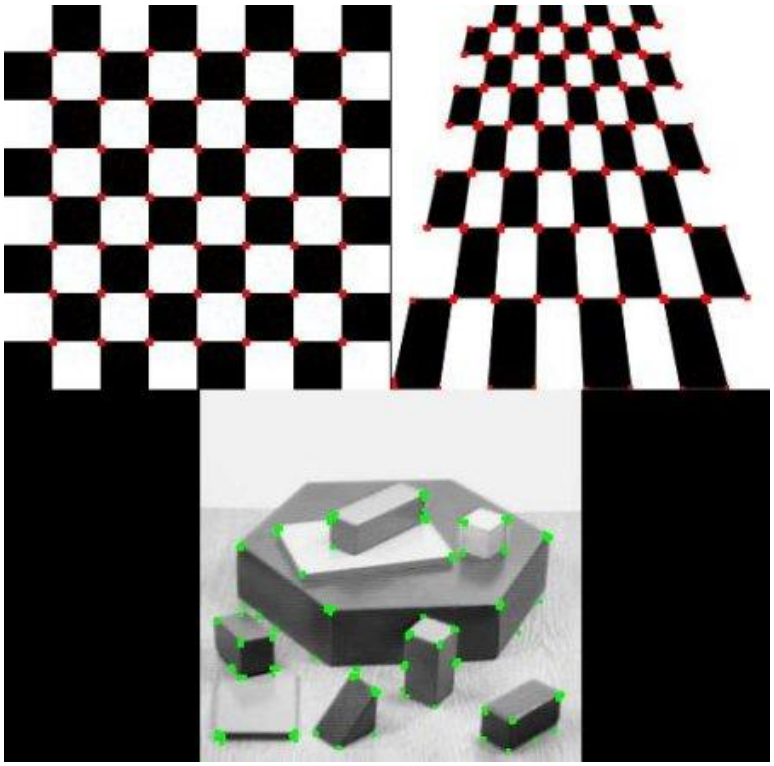
OpenCV has the function **cv.cornerHarris()** for this purpose. Its arguments are:

- **img** - Input image. It should be grayscale and float32 type.
- **blockSize** - It is the size of neighbourhood considered for corner detection
- **ksize** - Aperture parameter of the Sobel derivative used.
- **k** - Harris detector free parameter in the equation.

See the example below:

```
import numpy as np
import cv2 as cv
filename = 'chessboard.png'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)

#result is dilated for marking the corners, not important
dst = cv.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]
cv.imshow('dst',img)
if cv.waitKey(0) & 0xff == 27:
    cv.destroyAllWindows()
```

Below are the three results:



image

Shi-Tomasi Corner Detector & Good Features to Track

OpenCV has a function, **cv.goodFeaturesToTrack()**. It finds N strongest corners in the image by Shi-Tomasi method (or Harris Corner Detection, if you specify it). As usual, the image should be a grayscale image. Then you specify the number of corners you want to find. Then you specify the quality level, which is a value between 0-1, which denotes the minimum quality of the corner below which everyone is rejected. Then we provide the minimum euclidean distance between corners detected.

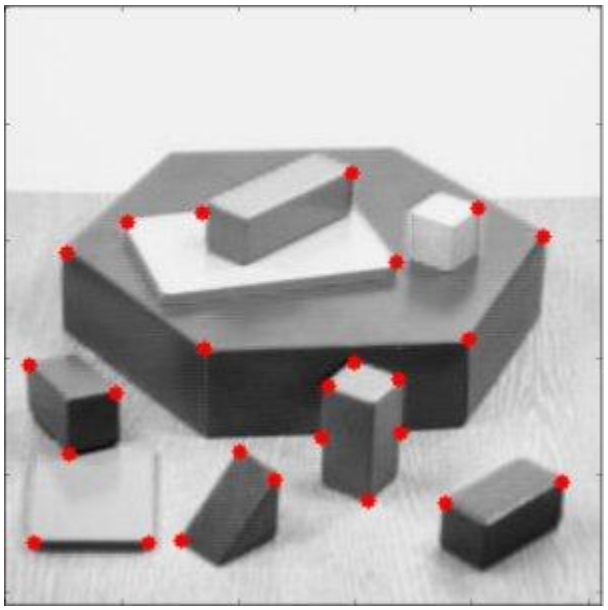
With all this information, the function finds corners in the image. All corners below quality level are rejected. Then it sorts the remaining corners based on quality in the descending order. Then

the function takes the first strongest corner, throws away all the nearby corners in the range of minimum distance and returns N strongest corners.

In below example, we will try to find 25 best corners:`import numpy as np`

```
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('blox.jpg')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
corners = cv.goodFeaturesToTrack(gray,25,0.01,10)
corners = np.int0(corners)
for i in corners:
    x,y = i.ravel()
    cv.circle(img,(x,y),3,255,-1)
plt.imshow(img),plt.show()
```

See the result below:



image

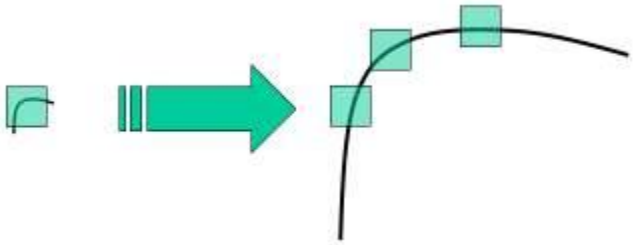
Exercise:

1. Try with your own images.
2. Modify the parameter in `cv.goodFeaturesToTrack` to get the best results.
3. Post your results in Padlet.

SIFT (Scale-Invariant Feature Transform)

Theory

In the last section, we saw some corner detectors like Harris etc. They are rotation-invariant, which means, even if the image is rotated, we can find the same corners. It is obvious because corners remain corners in the rotated image also. But what about scaling? A corner may not be a corner if the image is scaled. For example, check a simple image below. A corner in a small image within a small window is flat when it is zoomed in the same window. So Harris corner is not scale invariant.



image

SIFT in OpenCV

Now let's see SIFT functionalities available in OpenCV. Note that these were previously only available in the opencv contrib repo, but the patent expired in the year 2020. So they are now included in the main repo. Let's start with keypoint detection and draw them. First we have to construct a SIFT object. We can pass different parameters to it which are optional and they are well explained in docs.

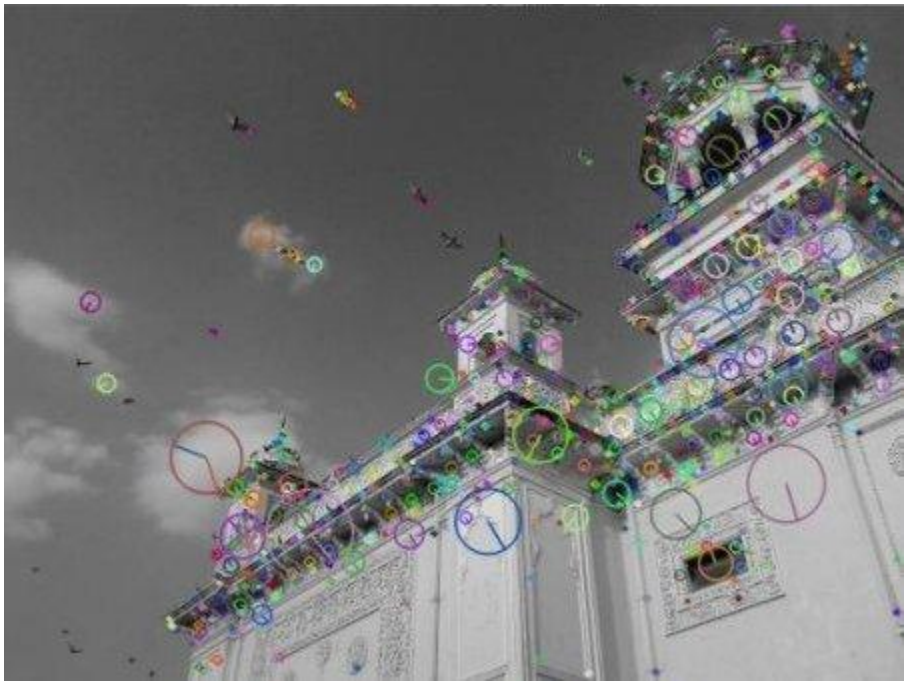
```
import numpy as np
import cv2 as cv
img = cv.imread('home.jpg')
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
sift = cv.SIFT_create()
kp = sift.detect(gray,None)
img=cv.drawKeypoints(gray,kp,img)
cv.imwrite('sift_keypoints.jpg',img)
```

sift.detect() function finds the keypoint in the images. You can pass a mask if you want to search only a part of the image. Each keypoint is a special structure which has many attributes like its (x,y) coordinates, size of the meaningful neighbourhood, angle which specifies its orientation, response that specifies strength of keypoints etc.

OpenCV also provides **cv.drawKeyPoints()** function which draws the small circles on the locations of keypoints. If you pass a flag, **cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS** to it, it will draw a circle with the size of the keypoint and it will even show its orientation. See below example.

```
img=cv.drawKeypoints(gray,kp,img,flags=cv.DRAW_MATCHES_FLAGS_DRAW_RI
      CH_KEYPOINTS)
cv.imwrite('sift_keypoints.jpg',img)
```

See the two results below:



image

Now to calculate the descriptor, OpenCV provides two methods.

1. Since you already found keypoints, you can call **sift.compute()** which computes the descriptors from the keypoints we have found. Eg: kp,des = sift.compute(gray,kp)

2. If you didn't find keypoints, directly find keypoints and descriptors in a single step with the function, **sift.detectAndCompute()**.

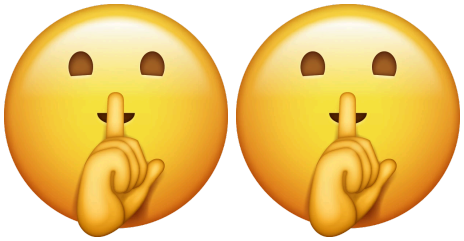
We will see the second method:

```
sift = cv.SIFT_create()
kp, des = sift.detectAndCompute(gray,None)
```

Here kp will be a list of keypoints and des is a numpy array of shape ***Number_of_Keypoints×128***

So we got keypoints, descriptors etc. Now we want to see how to match keypoints in different images. That we will learn in the coming sections.

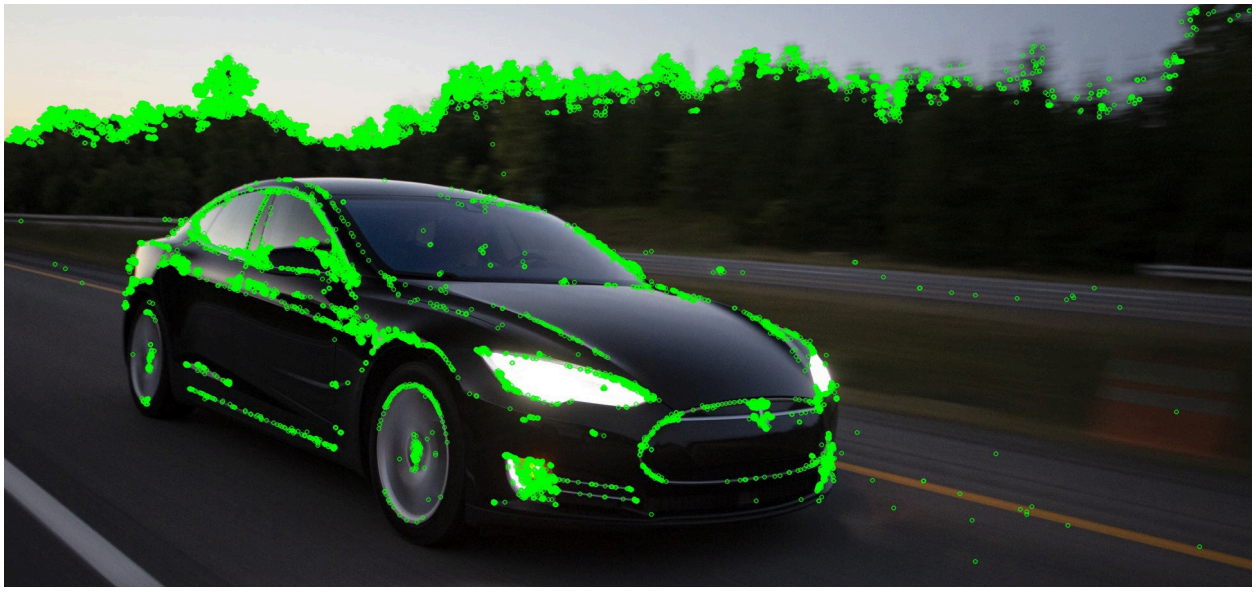
FAST Algorithm for Corner Detection



SURF is fast when compared to SIFT but not as fast to use it with real time devices like mobile phones. So the FAST algorithm was introduced with reduced processing time. However FAST gives us only the key points and we may need to compute descriptors with other algorithms like SIFT and SURF.

FAST = really FASTTTT!

```
import cv2

img = cv2.imread('your image here.jpg')
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

fast = cv2.FastFeatureDetector_create()
fast.setNonmaxSuppression(False)

kp = fast.detect(gray_img, None)
kp_img = cv2.drawKeypoints(img, kp, None, color=(0, 255, 0))

cv2.imshow('FAST', kp_img)
cv2.waitKey()
```



ORB (Oriented FAST and Rotated BRIEF)

SIFT and SURF are good in what they do, ~~but what if you have to pay a few dollars every year to use them in your applications? Yeah, they are patented!!!~~ (The patent expires in 2020, Yeah!) To solve that problem, OpenCV devs came up with a new "FREE" alternative to SIFT & SURF, and that is ORB.

Even though it computes less key points when compared to SIFT and SURF yet they are effective. It uses FAST and BRIEF techniques to detect the key points and compute the image descriptors respectively.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
```

```
# Initiate ORB detector
orb = cv.ORB_create()

# find the keypoints with ORB
kp = orb.detect(img,None)
```

```
# compute the descriptors with ORB
kp, des = orb.compute(img, kp)

# draw only keypoints location,not size and orientation
img2 = cv.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
plt.imshow(img2), plt.show()
```

Practical 11 Feature Matching

We know a great deal about feature detectors and descriptors. It is time to learn how to match different descriptors. OpenCV provides two techniques, Brute-Force matcher and FLANN based matcher.

Brute-Force (BF) Matcher

BF Matcher matches the descriptor of a feature from one image with all other features of another image and returns the match based on the distance. It is slow since it checks matches with all the features.

book_cover.jpeg book_cover_rotated.jpeg

```
import cv2

img1 = cv2.imread('book_cover.jpeg', 0)
img2 = cv2.imread('book_cover_rotated.jpeg', 0)

orb = cv2.ORB_create(nfeatures=500)
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

# matcher takes normType, which is set to cv2.NORM_L2 for SIFT and SURF,
cv2.NORM_HAMMING for ORB, FAST and BRIEF
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des1, des2)
matches = sorted(matches, key=lambda x: x.distance)
# draw first 50 matches
match_img = cv2.drawMatches(img1, kp1, img2, kp2, matches[:50], None)
cv2.imshow('Matches', match_img)
cv2.waitKey()
```

Matches with ORB and Brute-Force

FLANN based matcher

Fast Library for Approximate Nearest Neighbors (FLANN) is optimized to find the matches with search even with large datasets hence its fast when compared to Brute-Force matcher.

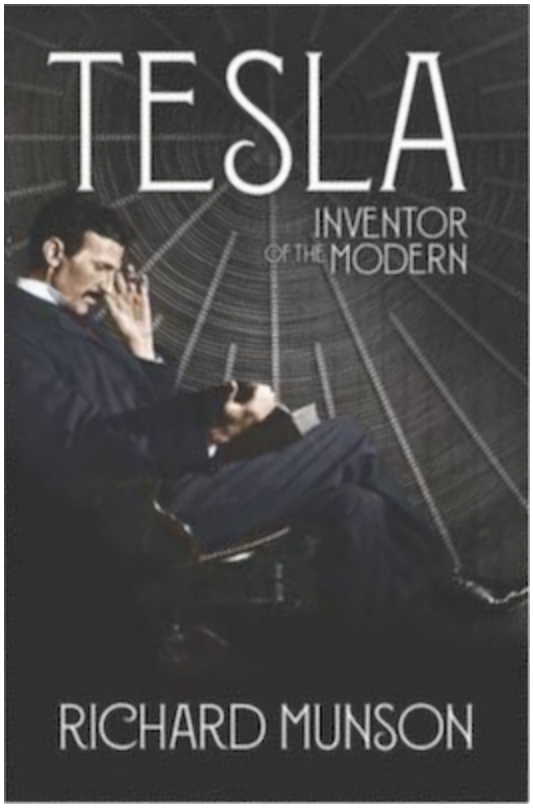
With an ORB and FLANN matcher let us extract the tesla book cover from the second image and correct the rotation with respect to the first image.

```
import cv2
import numpy as np


def get_corrected_img(img1, img2):
    MIN_MATCHES = 50

    orb = cv2.ORB_create(nfeatures=500)
    kp1, des1 = orb.detectAndCompute(img1, None)
    kp2, des2 = orb.detectAndCompute(img2, None)

    index_params = dict(algorithm=6,
```

```
                        table_number=6,
                        key_size=12,
                        multi_probe_level=2)
    search_params = {}
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)

    # As per Lowe's ratio test to filter good matches
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)

    if len(good_matches) > MIN_MATCHES:
                    src_points = np.float32([kp1[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
                    dst_points = np.float32([kp2[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
        m, mask = cv2.findHomography(src_points, dst_points, cv2.RANSAC, 5.0)
                corrected_img = cv2.warpPerspective(img1, m, (img2.shape[1],
img2.shape[0]))

        return corrected_img
    return img2
```

To extract the book cover from image 2 and correct its orientation with respect to image 1

```
im1 = cv2.imread('book_cover.jpeg')
im2 = cv2.imread('book_cover_rotated.jpeg')

img = get_corrected_img(im2, im1)
cv2.imshow('Corrected image', img)
cv2.waitKey()
```



corrected image

When to Use Which?

Criteria	Brute-Force (BF)	FLANN
Dataset size	Small	Medium to large
Descriptor type	Binary (e.g., ORB, BRIEF)	Float-based (e.g., SIFT, SURF), ORB*
Speed	Slower	Faster
Accuracy	High	Approximate, can include false matches

Feature Matching + Homography to find Objects

Goal

In this chapter,

- We will mix up the feature matching and findHomography from the calib3d module to find known objects in a complex image.

Basics

So what we did in the last session? We used a queryImage, found some feature points in it, we took another trainImage, found the features in that image too and we found the best matches among them. In short, we found locations of some parts of an object in another cluttered image. This information is sufficient to find the object exactly on the trainImage.

For that, we can use a function from calib3d module, ie **cv.findHomography()**. If we pass the set of points from both the images, it will find the perspective transformation of that object. Then we can use **cv.perspectiveTransform()** to find the object. It needs at least four correct points to find the transformation.

We have seen that there can be some possible errors while matching which may affect the result. To solve this problem, the algorithm uses RANSAC or LEAST_MEDIAN (which can be decided by the flags). So good matches which provide correct estimation are called inliers and remaining are called outliers. **cv.findHomography()** returns a mask which specifies the inlier and outlier points.

So let's do it !!!

1. First, as usual, let's find SIFT features in images and apply the ratio test to find the best matches.

   book.jpg book_in_scene.jpg

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
MIN_MATCH_COUNT = 10
img1 = cv.imread('book_in_scene.jpg', 0)  # queryImage
img2 = cv.imread('book.jpg',0) # trainImage

# Initiate SIFT detector
sift = cv.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)
flann = cv.FlannBasedMatcher(index_params, search_params)
```

```
matches = flann.knnMatch(des1,des2,k=2)

# store all the good matches as per Lowe's ratio test.
good = []
for m,n in matches:
   if m.distance < 0.7*n.distance:
       good.append(m)
```

2. Now we set a condition that at least 10 matches (defined by MIN_MATCH_COUNT) are to be there to find the object. Otherwise simply show a message saying not enough matches are present.

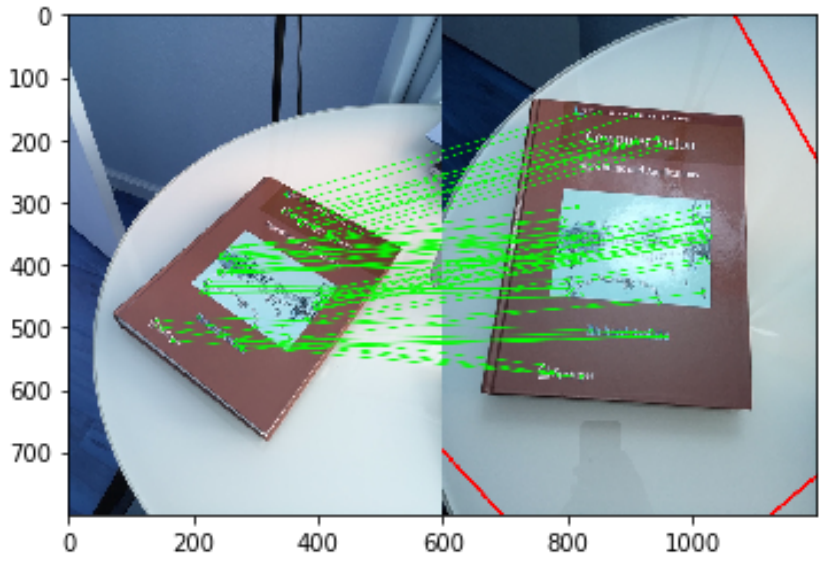
   If enough matches are found, we extract the locations of matched keypoints in both the images. They are passed to find the perspective transformation. Once we get this 3x3 transformation matrix, we use it to transform the corners of queryImage to corresponding points in trainImage. Then we draw it.

```
if len(good)>MIN_MATCH_COUNT:
   src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
   dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)
   M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC,5.0)
   matchesMask = mask.ravel().tolist()
   h,w,d = img1.shape
   pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
   dst = cv.perspectiveTransform(pts,M)
   img2 = cv.polylines(img2,[np.int32(dst)],True,255,3, cv.LINE_AA)
else:
       print( "Not enough matches are found - {}/{}".format(len(good),
       MIN_MATCH_COUNT) )
   matchesMask = None
```

3. Finally we draw our inliers (if successfully found the object) or matching keypoints (if failed).

```
draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)
img3 = cv.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
plt.imshow(img3, 'gray'),plt.show()
```

4. See the result.



Full Code

Python

```python
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
MIN_MATCH_COUNT = 10
img1 = cv.imread('ttss1.jpg', 0)  # queryImage
img2 = cv.imread('ttss2.jpg',0) # trainImage
from skimage import data, img_as_float
from skimage.metrics import structural_similarity as ssim

# Initiate SIFT detector
sift = cv.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 50)
flann = cv.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1,des2,k=2)

# store all the good matches as per Lowe's ratio test.
good = []
for m,n in matches:
```

```
    if m.distance < 0.7*n.distance:
        good.append(m)
if len(good)>MIN_MATCH_COUNT:
    src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
    dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)
    M, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC,5.0)
    matchesMask = mask.ravel().tolist()
    h,w = img1.shape
    # Warp img2 to align with img1
    aligned_img2 = cv.warpPerspective(img2, M, (w, h))

    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv.perspectiveTransform(pts,M)
    img2 = cv.polylines(img2,[np.int32(dst)],True,255,3, cv.LINE_AA)
else:
    print( "Not enough matches are found - {}/{}".format(len(good),
MIN_MATCH_COUNT) )
    matchesMask = None

draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)

def ssim_compare(img1_path, img2_path) :
    img1 = cv.imread('ttss1.jpg', 0)
    img2 = cv.imread('ttss2.jpg', 0)
    dim = (6022,5513)
    # print("Img1 Resolution:", img1.shape)
    # print("Img2 Resolution:", img2.shape)
    img1 = cv.resize(img1, dim)
    img2 = cv.resize(img2, dim)
    # print("Img1 Res :", img1.shape)
    # print("Img2 Res :", img2.shape)
    ssim_score, dif = ssim(img1, img2, full=True)
    return ssim_score
ssim_val = ssim_compare('ttss1.jpg', 'ttss2.jpg')
print(ssim_val)

img3 = cv.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
plt.imshow(img3, 'gray'),plt.show()
```

When to Use What?

Method	Best For	Fast?	Robust?
SSIM	Visual/perceptual similarity	✅	✅
Histogram	Colour/light similarity	✅	❌
Feature match ratio	Geometric/object match	❌	✅
MSE	Pixel-by-pixel difference	✅	❌
SSIM after Homography	**Align + compare**	❌	✅✅✅

Let's us incorporate SSIM into the homography features matching

>> Inside the if-else loop after getting the size of img1

Python

```
# Warp img2 to align with img1
aligned_img2 = cv.warpPerspective(img2, M, (w, h))

# Compute SSIM between img1 and aligned img2
ssim_score, _ = ssim(img1, aligned_img2, full=True)
print(f"SSIM after homography alignment: {ssim_score:.4f}")
```

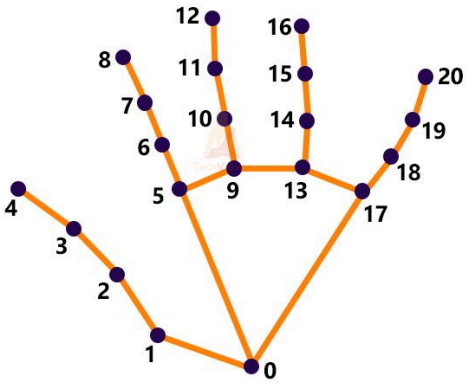
Practical 12 Hand Gesture Recognition

Gesture recognition is an active research field in Human-Computer Interaction technology. It has many applications in virtual environment control and sign language translation, robot control, or music creation. In this machine learning project on Hand Gesture Recognition, we are going to make a real-time Hand Gesture Recognizer using the MediaPipe framework and Tensorflow in OpenCV and Python.

What is MediaPipe?

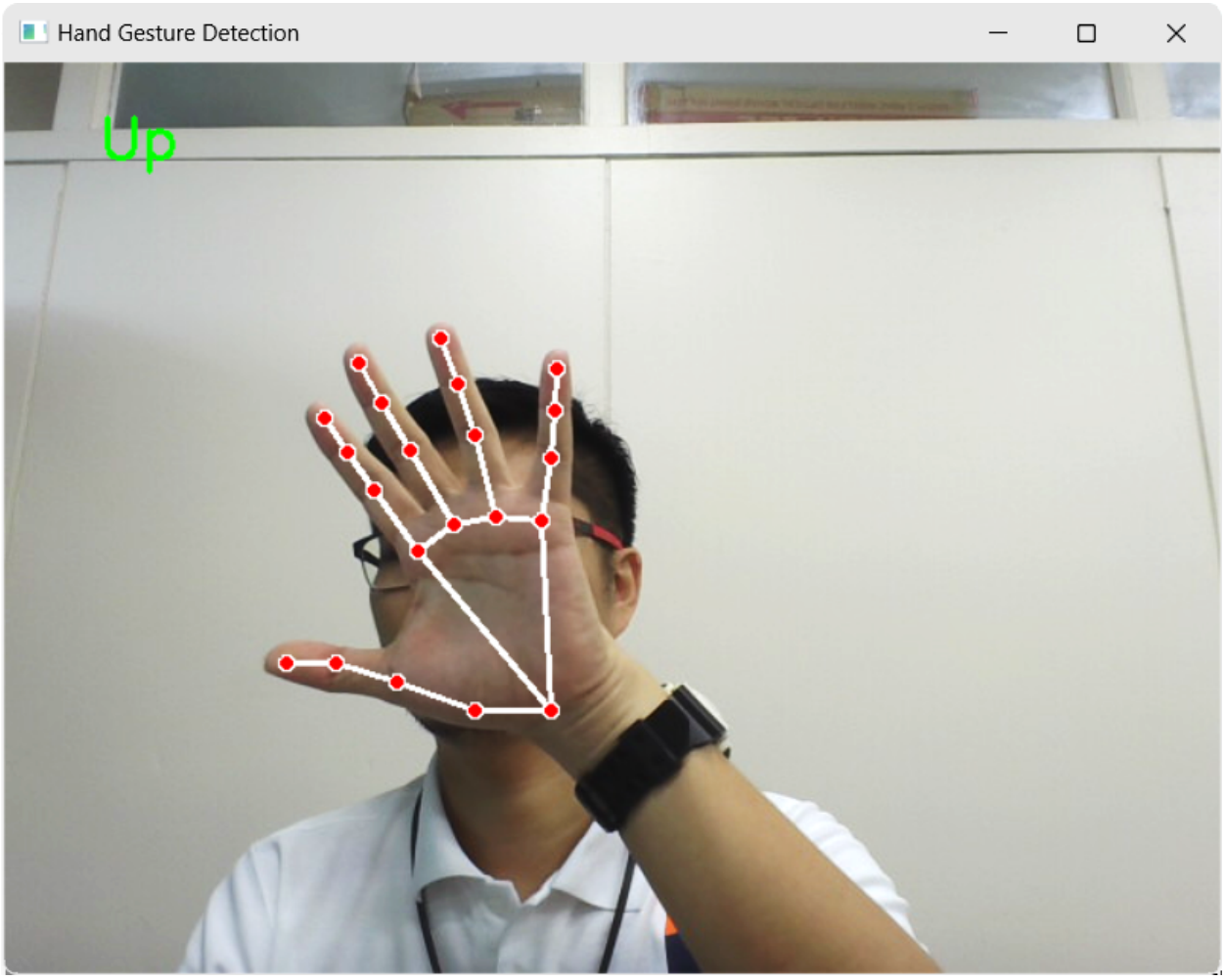
MediaPipe is a customizable machine learning solutions framework developed by Google. It is an open-source and cross-platform framework, and it is very lightweight. MediaPipe comes with some pre-trained ML solutions such as face detection, pose estimation, hand recognition, object detection, etc.

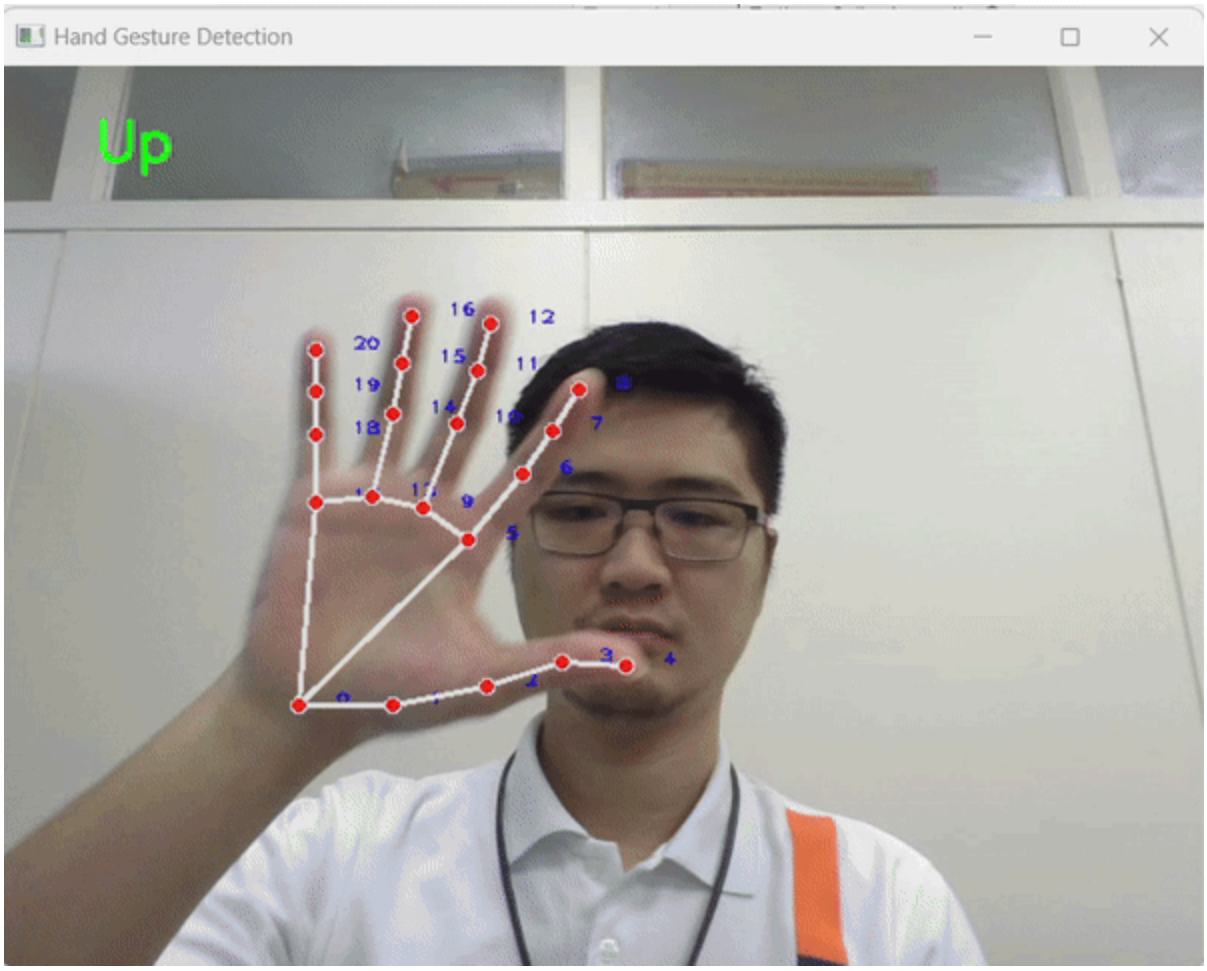
We'll first use MediaPipe to recognize the hand and the hand key points. MediaPipe returns a total of 21 key points for each detected hand.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

These key points will be fed into a pre-trained gesture recognizer network to recognize the hand pose.


Hand Gesture Detection
Up



🔹 Step 1: Install Required Libraries

```
Python
pip install opencv-python mediapipe numpy pycaw comtypes pyautogui
```

🔹 Step 2: Download MediaPipe Gesture Model

- Download from: [Gesture recognition task guide | Google AI Edge]
- Place the file `gesture_recognizer.task` in the same folder as this notebook.

🔹 Step 3: Import Required Modules

Python

```
import cv2
import mediapipe as mp
import numpy as np
import webbrowser
from pycaw.pycaw import AudioUtilities, IAudioEndpointVolume
from ctypes import cast, POINTER
from comtypes import CLSCTX_ALL
import pyautogui
```

🔹 Step 4: Configure Gesture Recognizer

Python

```
BaseOptions = mp.tasks.BaseOptions
GestureRecognizer = mp.tasks.vision.GestureRecognizer
GestureRecognizerOptions = mp.tasks.vision.GestureRecognizerOptions
GestureRecognizerResult = mp.tasks.vision.GestureRecognizerResult
VisionRunningMode = mp.tasks.vision.RunningMode

last_result = None

def print_result(result: GestureRecognizerResult, output_image: mp.Image,
timestamp_ms: int):
    global last_result
    last_result = result
```

🔹 Step 5: Set Up Recognizer Options and Volume Control

```python
options = GestureRecognizerOptions(
    base_options=BaseOptions(model_asset_path='gesture_recognizer.task'),
    running_mode=VisionRunningMode.LIVE_STREAM,
    result_callback=print_result
)

# Set up Windows volume control
devices = AudioUtilities.GetSpeakers()
interface = devices.Activate(IAudioEndpointVolume._iid_, CLSCTX_ALL, None)
volume = cast(interface, POINTER(IAudioEndpointVolume))
vol_min, vol_max = volume.GetVolumeRange()[:2]
```

🔹 Step 6: Launch Webcam and Recognize Gestures

```python
opened_youtube = False
cap = cv2.VideoCapture(0)

with GestureRecognizer.create_from_options(options) as recognizer:
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            continue

        frame = cv2.flip(frame, 1)
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=frame)
        timestamp = int(cv2.getTickCount() / cv2.getTickFrequency() * 1000)
        recognizer.recognize_async(mp_image, timestamp)

        if last_result and last_result.hand_landmarks:
            for hand_landmarks in last_result.hand_landmarks:
                landmarks = []
                for idx, landmark in enumerate(hand_landmarks):
```

```
                    x = int(landmark.x * frame.shape[1])
                    y = int(landmark.y * frame.shape[0])
                    landmarks.append((x, y))
                    cv2.circle(frame, (x, y), 5, (0, 255, 0), -1)
                    cv2.putText(frame, str(idx), (x + 5, y - 5),
cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 0), 1)

                connection_pairs = [(0,1), (1,2), (2,3), (3,4), (5,6), (6,7),
(7,8),
                                    (9,10), (10,11), (11,12), (13,14), (14,15),
(15,16),
                                    (17,18), (18,19), (19,20)]
                for start, end in connection_pairs:
                    if start < len(landmarks) and end < len(landmarks):
                        cv2.line(frame, landmarks[start], landmarks[end], (0,
255, 255), 2)

        if last_result and last_result.gestures:
            gesture = last_result.gestures[0][0].category_name
            score = last_result.gestures[0][0].score
            cv2.putText(frame, f'Gesture: {gesture} ({score:.2f})', (10, 30),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

            current_vol = volume.GetMasterVolumeLevel()
            if gesture == "Thumb_Up":
                volume.SetMasterVolumeLevel(min(current_vol + 1.0, vol_max),
None)
            elif gesture == "Thumb_Down":
                volume.SetMasterVolumeLevel(max(current_vol - 1.0, vol_min),
None)
            elif gesture == "Victory" and not opened_youtube:
                webbrowser.open("https://www.youtube.com")
                opened_youtube = True

        cv2.imshow('Gesture Control', frame)
        if cv2.waitKey(5) & 0xFF == 27:
            break

cap.release()
cv2.destroyAllWindows()
```

😍😍 **Post your masterpiece in the Padlet.**