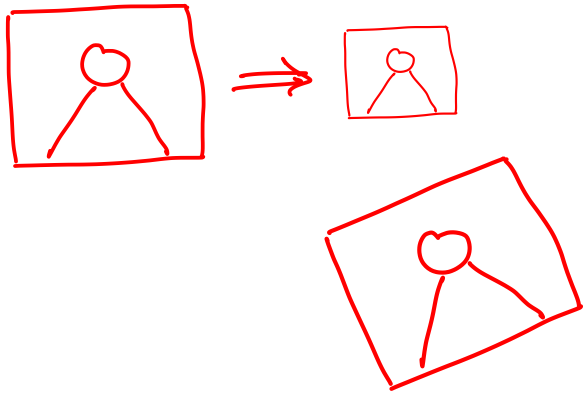
OpenCV

Transformations

Introduction to transformation

- Geometric distortions enacted upon an image
- Use transformations to correct distortions or perspective issues
- Affine → warpAffine( )
  - Transformation where points, straight lines, and planes are preserved
  - Additionally, the parallel lines will remain parallel after this transformation
  - However, an affine transformation does not preserve both the distance and angles between points.
  - E.g.
    - Scaling
    - Rotation
    - Translation

→ matrix



Perspective transformation

- In order to correct the perspective, you will need to create the transformation matrix by making use of the cv2.getPerspectiveTransform() function, where a *3 x 3* matrix is constructed
- This function needs four pairs of points (coordinates of a quadrangle in both the source and output image) and calculates a perspective transformation matrix from these points
- Then, the *M* matrix is passed to cv2.warpPerspective(), where the source image is transformed by applying the specified matrix with a specified size

```
img = cv2.imread('scan.jpg')

points_A = np.float32([[320,15], [700,215], [85,610], [530,780]])
points_B = np.float32([[0, 0], [420, 0], [0, 594], [420, 594]])

M = cv2.getPerspectiveTransform(points_A, points_B)
warped = cv2.warpPerspective(img, M, (420, 594))

cv2.imshow('perspective', warped)
cv2.imshow('original', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

perspective transformation

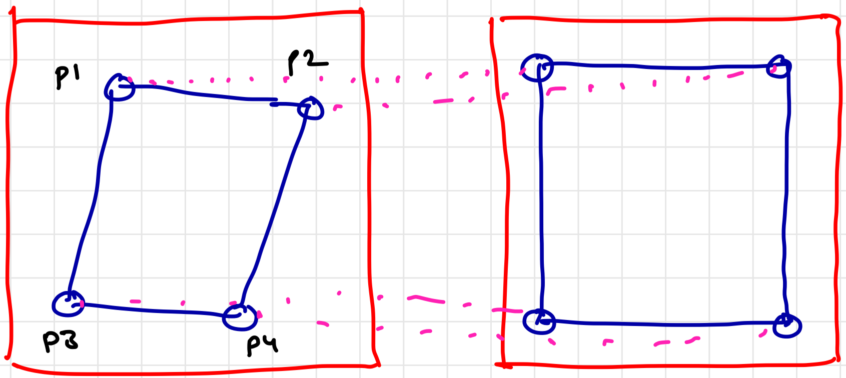


Image Filtering

Convolutions

- Convolution is a simple mathematical operation which is fundamental to many common image processing operators
- Convolution provides a way of `multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality
- This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values
- Convolution can achieve something which includes the blurring, sharpening, edge detection, noise reduction etc.

Convolutions

- In an image processing context, one of the input arrays is normally just a graylevel image. The second array is usually much smaller, and is also two-dimensional (although it may be just a single pixel thick), and is known as the kernel

original image

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}	I_{19}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}	I_{39}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}	I_{49}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}	I_{59}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}	I_{69}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}

kernel

3x3 – 25x25

Applying kernels

- OpenCV provides a function filter2D() in order to apply a kernel to an image
- To create a 5x5 kernel

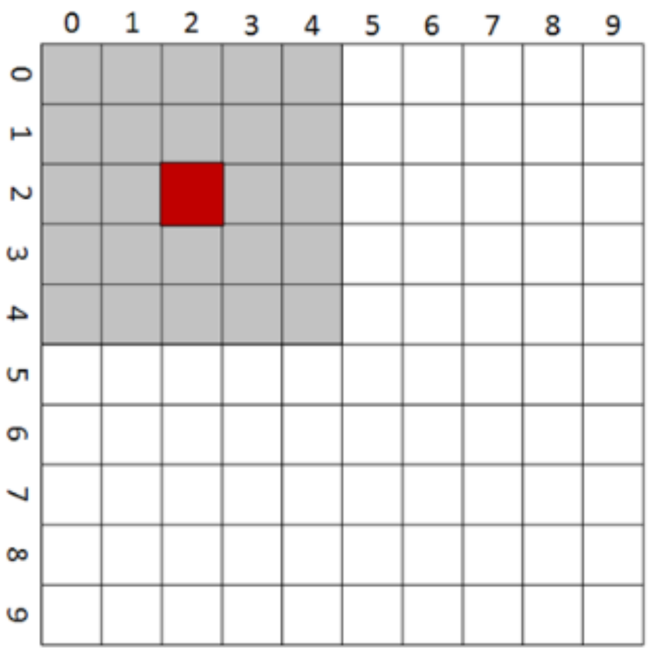
```
kernel = np.array([
        [0.04, 0.04, 0.04, 0.04, 0.04],
        [0.04, 0.04, 0.04, 0.04, 0.04],
        [0.04, 0.04, 0.04, 0.04, 0.04],
        [0.04, 0.04, 0.04, 0.04, 0.04],
        [0.04, 0.04, 0.04, 0.04, 0.04]
])
```

- OR

```
Kernel = np.ones((5, 5), np.float32) / 25
```

Blurring / Smoothing Images

- Blurring is an operation where we average the pixels within a region kernel).
- Normalize the kernel (i.e. sum to 1) otherwise it would increase intensity



$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

```
img = cv2.imreadmessi5.jpg')
kernel = np.ones((3, 3), dtype='float32') / 9
new = cv2.filter2D(img, -1, kernel)
cv2.imshow('new image', new)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Sharpening Image

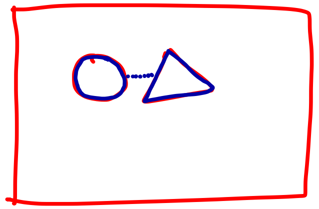
- Sharpening is the opposite of blurring
- It strengthens or emphasizing edges in an image

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & \text{intensity} & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

```
img = cv2.imreadmessi5.jpg')
kernel_sharpening = np.array([[-1, -1, -1],
                              [-1, 9, -1],
                              [-1, -1, -1]])
new = cv2.filter2D(img, -1, kernel_sharpening)
cv2.imshow('new image', new)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Edge Detection

- Edge detection is a very important area in OpenCV, especially when dealing with contours
- Edge can be defined as sudden changes (discontinuities) in an image and they can encode just as much information as pixels
- Types
  - Sobel: to emphasize vertical or horizontal edges
  - Laplacian: gets all orientations
  - Canny: optimal due to low error rate, well defined edges and accurate detection



Edge Detection - Canny Edge

- Developed by John F. Canny in 1986
- Applied Gaussian blurring
- Find intensity gradient of the image
- Applied non-maximum suppression (i.e. removes pixels that are not edges)
- Hysteresis – Applies thresholds (i.e. if pixel is within the upper or lower thresholds, it is considered on the edge)

binary image ← colors
→ black - background
→ white - edge

binarising image
→ canny ()
→ threshold ()

Arithmetic Operations

Image Addition and Subtraction

- Image addition and subtraction can be performed with:
  - cv2.add()
  - cv2.subtract()

broadcast

- These functions sum/subtraction the per-element sum/subtract of two arrays
- These function can also be used to sum/subtract an array and a scalar
- To add some value in an image we need to create a matrix with same shape as that of the image
  - M = np.ones(image.shape, dtype=np.uint8) * 30
- To apply
  - cv2.add(img, M)
  - cv2.subtract(img, M)

Image Blending

- Image blending is also image addition, but different weights are given to the images, giving an impression of transparency
- In order to do this, the cv2.addWeighted() function will be used
- This function is commonly used to get the output from the Sobel operator

Bitwise Operations

- There are some operations that can be performed at bit level using bitwise operators
- These bitwise operations are simple, and are quick to calculate
- This means that they are a useful tool when working on images
- Operations
  - **Bitwise AND**: bitwise_and = cv2.bitwise_and(img_1, img_2)
  - **Bitwise OR**: bitwise_xor = cv2.bitwise_xor(img_1, img_2)
  - **Bitwise XOR**: bitwise_xor = cv2.bitwise_xor(img_1, img_2)
  - **Bitwise NOT**: bitwise_not_1 = cv2.bitwise_not(img_1)

Morphological Operations

Introduction

- These operations are normally performed on binary images and based on the image shape
- The exact operation is determined by a kernel-structuring element, which decides the nature of the operation
- Dilation and erosion are the two basic operators in the area of morphological transformations
- Additionally, opening and closing are two important operations, which are derived from the two aforementioned operations (dilation and erosion)

Dilation

- The main effect of a dilation operation on a binary image is to gradually expand the boundary regions of the foreground object
- This means the areas of the foreground object will become larger while holes within those regions shrink
- E.g.
  - dilation = cv2.dilate(image, kernel, iterations=1)

```
img = cv2.imread('opencv.png')

kernel = np.ones((5, 5), np.uint8)
cv2.imshow('original', img)

erosion = cv2.dilate(img, kernel, iterations=1)
cv2.imshow('Erosion', erosion)
```

Erosion

- The main effect of an erosion operation on a binary image is to gradually erode away the boundary regions of the foreground object

reduce

- This means that the areas of the foreground object will become smaller, and the holes within those areas will get bigger
- E.g.
  - erosion = cv2.erode(image, kernel, iterations=1)

opposite of dilation process

```
img = cv2.imread('opencv.png')

kernel = np.ones((5, 5), np.uint8)
cv2.imshow('original', img)

erosion = cv2.erode(img, kernel, iterations=1)
cv2.imshow('Erosion', erosion)
```

Thresholding Techniques

Image Segmentation

- Image segmentation is a key process in many computer vision applications
- It is commonly used to partition an image into different regions that, ideally, correspond to real-world objects extracted from the background

object detection object recognition

- Therefore, image segmentation is an important step in image recognition and content analysis
- Image thresholding is a simple, yet effective, image segmentation method, where the pixels are partitioned depending on their intensity value

edges

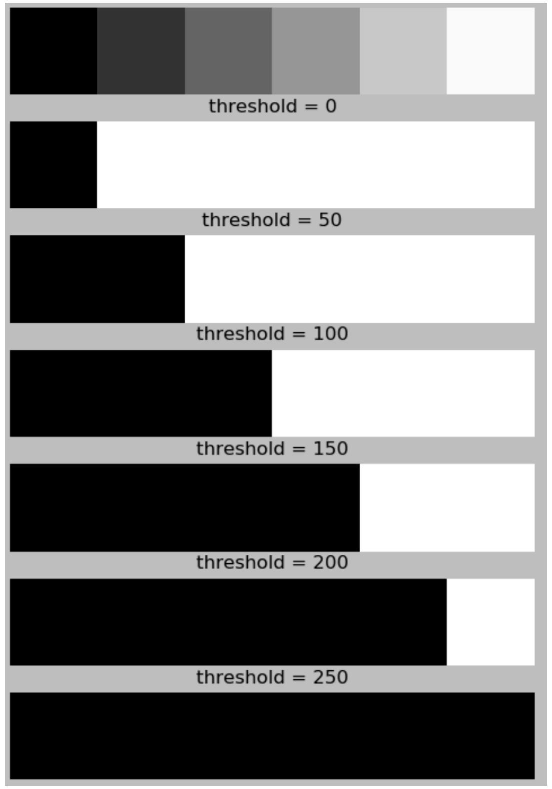
- It can be used to partition an image into a foreground and background
- The objective of image segmentation is to modify the representation of an image into another representation that is easier to process

↳ binary image - black & white

Simple Thresholding

- It is used for image segmentation
- The simplest thresholding methods replace each pixel in the source image with a black pixel if the pixel intensity is less than some predefined constant (the threshold value), or a white pixel, if the pixel intensity is greater than the threshold value
- OpenCV provides the cv2.threshold() function to threshold images
- E.g.
  - ret1, thresh1 = cv2.threshold(gray_image, 50, 255, cv2.THRESH_BINARY)

gray image



Adaptive Thresholding

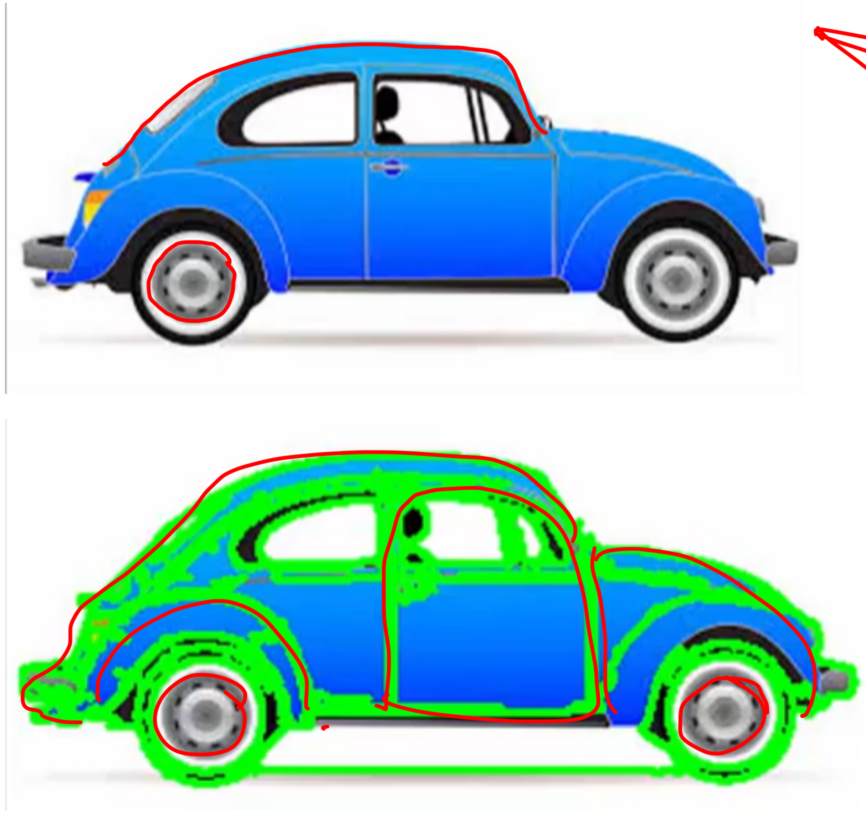
- Sometimes the simple thresholding’s result is not very good due to the different illumination conditions in the different areas of the image
- In these cases, you can try adaptive thresholding
- In OpenCV, the adaptive thresholding is performed by the cv2.adapativeThreshold() function

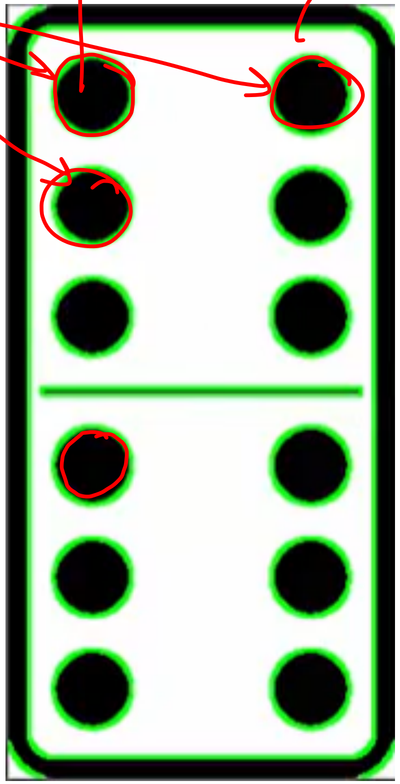
Contours

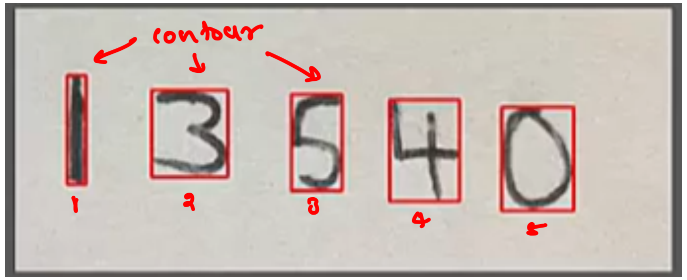
Image Segmentation

- Segmentation is partitioning images into different regions

contour circle rectangle







Introduction to Contours

- Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity
- The contours are a useful tool for shape analysis and object detection and recognition.
- For better accuracy, use binary images. So before finding contours, apply threshold or canny edge detection.
- In OpenCV, finding contours is like finding white object from black background. So remember, object to be found should be white and background should be black.

Finding contours

- See, there are three arguments in cv.findContours() function, first one is source image, second is contour retrieval mode, third is contour approximation method
- And it outputs a modified image, the contours and hierarchy
- contours is a Python list of all the contours in the image
- Each individual contour is a Numpy array of (x,y) coordinates of boundary points of the object

```
im = cv.imread('test.jpg')
imgray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(imgray, 127, 255, 0)
im2, contours, hierarchy = cv.findContours(thresh, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
```

Draw the contours

- To draw the contours, **cv.drawContours** function is used
- It can also be used to draw any shape provided you have its boundary points
- Its first argument is source image, second argument is the contours which should be passed as a Python list, third argument is index of contours (useful when drawing individual contour)
- To draw all contours, pass -1 and remaining arguments are color, thickness etc.

thickness

- To draw all the contours in an image:
  - cv.drawContours(img, contours, -1, (0,255,0), 3)
- To draw an individual contour, say 4th contour:
  - cv.drawContours(img, contours, 3, (0,255,0), 3)
- But most of the time, below method will be
  - cv.drawContours(img, [cnt], 0, (0,255,0), 3)

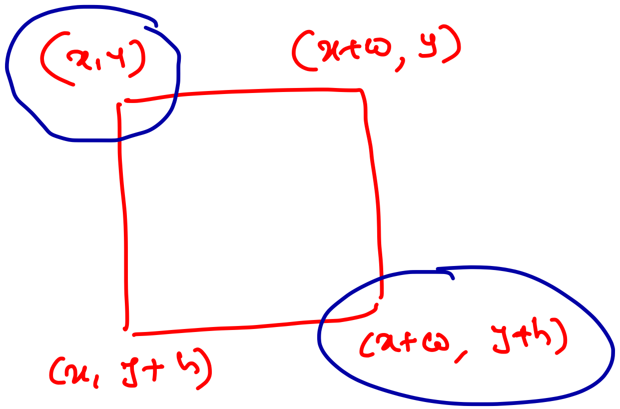
Shape Detection

- Use approxPolyDP() to detect the shape

```
approx = cv2.approxPolyDP(c, 0.01 * cv2.arcLength(c, True), True)
```

- Use boundingRect(c) to detect the bounding rectangle of the contour

```
(x, y, w, h) = cv2.boundingRect(c)
```



Feature Detection

- face
- eye
- object

Cascading classifiers

- Cascading is a particular case of ensemble learning based on the concatenation of several classifiers, using all information collected from the output from a given classifier as additional information for the next classifier in the cascade
- Unlike voting or stacking ensembles, which are multiexpert systems, cascading is a multistage one
- Cascading classifiers are trained with several hundred "positive" sample views of a particular object and arbitrary "negative" images of the same size
- After the classifier is trained it can be applied to a region of an image and detect the object in question
- To search for the object in the entire frame, the search window can be moved across the image and check every location for the classifier
- This process is most commonly used in image processing for object detection and tracking, primarily facial detection and recognition

Cascading classifiers in OpenCV

- Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001
- It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images

```
eyeCascade = cv2.CascadeClassifier’(/haarcascade_eye.xml’)

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
eyes = eyeCascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

for (x, y, w, h) in eyes :
    cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
```