Image processing with OpenCV

For tutorial week 7, our discussion and code samples will cover the following topics:

- Gentle introduction of image segmentation
- Edge detection
- Contour: Detection and drawing of image contours
- Contour features (useful for shape detection and image recognition)
- Blob detection using built-in OpenCV function

Image segmentation

Image segmentation is a commonly used technique in digital image processing and analysis to partition an image into multiple parts or regions, often based on the characteristics of the pixels in the image. Image segmentation could involve separating foreground from background or clustering regions of pixels based on similarities in color or shape.

Important notes

In the case of traffic sign segmentation project, the performance of traffic sign segmentation pipeline is evaluated through the computation of *Intersection over Union (IOU)*. This is possible as the traffic signs are annotated with ground truth bouding boxes.

Edge detection

Consider an image, how can we find the salient edges? Qualitatively, edges occur at boundaries between regions of different color, intensity or texture. Often, it is preferable to detect edges using only purely local information.

Edges are characterized by sudden change in pixel intensity. To detect edges, we need to look for such changes in the neighboring pixels. You can easily notice that in an edge the pixel intensity changes in a notorious way. A good way to express changes is by using derivatives. A high change in gradient indicates a major change in the image.

The method to detect edges can be performed by locating pixel locations where the gradient is higher than its neighbors (or to generalize, higher than a threshold). A Sobel operator is a discrete differentiation operator. It computes an approximation of the gradient of an image intensity function. The Sobel operator combines Gaussian smoothing and differentiation.

Image Gradients

OpenCV provides 3 types of gradient filters or HPF: Sobel, Scharr and Laplacian.

1. Sobel and Scharr Derivatives
   Sobel operators is a joint Gaussian smoothing plus differentiation operation, so it is more resistant to noise. You can specify the direction of derivatives to be taken, vertical or horizontal (by the arguments, yorder and

xorder respectively). When applying Sobel operator for edge detection, we need to convert the color image to grayscale image.

2. Laplacian Derivatives. One such example is as shown below: $kernel =$

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

In [1]:

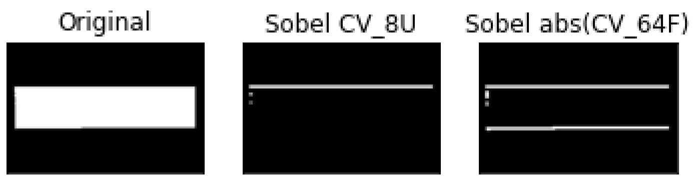
```python
import sys
assert sys.version_info >= (3, 7)

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

if not cv.useOptimized():
    cv.setUseOptimized(True)

cv.useOptimized()
```

Out[1]:

```
True
```

In [2]:

```python
img = cv.imread('images/wood_planck.jfif')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

ret, img_wood = cv.threshold(gray, 200, 255, cv.THRESH_BINARY_INV)

# Output dtype = cv.CV_8U
# detect horizontal edge
sobely8u = cv.Sobel(img_wood, cv.CV_8U, 0, 1, ksize=5)

# set output type as cv.CV_64F
sobely64f = cv.Sobel(img_wood, cv.CV_64F, 0, 1, ksize=5)
sobely_8u = cv.convertScaleAbs(sobely64f)

plt.subplot(1, 3, 1), plt.imshow(img_wood, cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(1, 3, 2), plt.imshow(sobely8u, cmap = 'gray')
plt.title('Sobel CV_8U'), plt.xticks([]), plt.yticks([])
plt.subplot(1, 3, 3), plt.imshow(sobely_8u, cmap = 'gray')
plt.title('Sobel abs(CV_64F)'), plt.xticks([]), plt.yticks([])

plt.show()
```



In [5]:

```python
img = cv.imread('images/lena.jfif', 0)

laplacian = cv.Laplacian(img, cv.CV_64F, ksize=3)
laplacian = cv.convertScaleAbs(laplacian)
sobelx = cv.Sobel(img, cv.CV_64F, 1, 0, ksize=3)
sobelx = cv.convertScaleAbs(sobelx)
sobely = cv.Sobel(img, cv.CV_64F, 0, 1, ksize=3)
sobely = cv.convertScaleAbs(sobely)
```

```
cv.imshow('Laplacian', laplacian)
cv.imshow('Sobel x', sobelx)
cv.imshow('Sobel y', sobely)

cv.waitKey(0)
cv.destroyAllWindows()
```

In [7]:
```
blur = cv.GaussianBlur(img, (5, 5), 0)

laplacian_no_noise = cv.Laplacian(blur, cv.CV_64F, ksize=3)
laplacian_no_noise = cv.convertScaleAbs(laplacian_no_noise)

cv.imshow('laplacian', laplacian)
cv.imshow('Laplacian_blur', laplacian_no_noise)
cv.waitKey(0)
cv.destroyAllWindows()
```

Exercise

1. Use convolution `cv.filter2D()` to implement Laplacian filter on 'lena.jfif'.
2. What happen if you add the results of Laplacian of Gaussian (LOG) filters into the original image.

Canny Edge Detection

1. **Noise Reduction**
   Since edge detection is susceptible to noise in the image, the first step is to remove noise using 5x5 Gaussian filter.

2. **Finding Intensity Gradient of the image**. Sobel operators x and y are utilized.

3. **Non-maximum Suppression** is to filter out unwanted pixels (which do not constitute valid edges). To accomplish that, each pixel is compared to its neighboring pixels, in the positive and negative gradient direction. If the gradient magnitude of the current pixel is greater than its neighboring pixels, it is left unchanged. Otherwise, the magnitude of the current pixel is set to zero.

4. **Hysterisis thresholding**. The gradient magnitude are compared with the 2 threshold values.

Canny edge detection in OpenCV

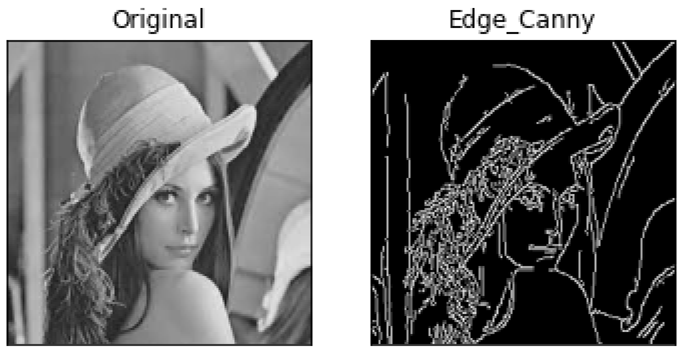
OpenCV puts all the above in single function, `cv.Canny()` .

```
# The 4th and 5th arguments are optional.
cv.Canny(img, threshold1, threshold2, apertureSize=3, L2gradient = False)
```

- First argument is input image.
- Second and third arguments are minVal and maxVal respectively.
- Fouth argument is aperture size, which is the size of Sobel operator used to find gradients. Default is 3.
- Last argument is L2 gradient which specifies the equation to find the gradient magnitude. It can be `True` or `False` .

In [8]:
```
img = cv.imread('images/lena.jfif', 0)
edges = cv.Canny(img, 100, 200)
```

```
plt.subplot(121), plt.imshow(img, cmap='gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.imshow(edges, cmap='gray')
plt.title('Edge_Canny'), plt.xticks([]), plt.yticks([])

plt.show()
```



In [2]:
```
img = cv.imread('images/lena.jfif')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

edges = cv.Canny(gray, 100, 300)

cv.imshow('Canny', edges)
cv.waitKey(0)
cv.destroyAllWindows()
```

Generally speaking, Canny edge detection produces the best results in most applications because it provides more flexibility in how edges are identified and connected.

Zero-parameter / automatic Canny edge detector

From the examples above, we know that `threshold1` and `threshold2` are two of the most crucial parameters to control the quality of edge detection results. This begs a question: Can we automate Canny edge detection by selecting the 2 parameters algorithmically? The answer is yes.

The example code on the implementation will be shown in the class.

Further reading

For more information on the determination of threshold range, please refer to this pyimagesearch and stackabuse posts.

Exercise

Create a trackbar that control the hysterisis thresholds and display the resulting images from the changes in the thresholds.

What are contours?

While isolated edges can be used for some applications, like line detection, they become even more useful when linked into continuous contours.

Contours can be explained simply as curve joining all the continuous points (along the boundary), having the same color and intensity. The contours are a useful tool for shape analysis as well as object detection and recognition. As such, it is often one of the most important steps for numerous applications, such as *image segmentation*.

When we join all the points on the boundary of an object, we get a contour.

Some suggestions in the OpenCV implementation:

- 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, object to be found should be white and background should be black.

Find contours

This is done through function `cv.findContours()` .

- First argument is source image, which should be 8-bit single channel image. The image is treated as binary image.
- Second argument is contour retrieval mode. Refer to online documentation for more info.
  - cv.RETR_EXTERNAL
  - cv.RETR_LIST
  - cv.RETR_CCOMP
  - cv.RETR_TREE
- Third argument is contour approximation method. Refer to online documentation for more info.
  - cv.CHAIN_APPROX_NONE
  - cv.CHAIN_APPROX_SIMPLE
  - cv.APPROX_TC89_L1
  - cv.APPROX_TC89_KCOS
- Outputs: 1. contours and 2. hierarchy

Draw contours

To draw contours, `cv.drawContours()` function is used.

- First argument is source image
- Second argument is the contours (Python list)
- Third argument is the index of contours (useful when drawing individual contour). To draw all contours, pass -1.
- Remaining arguments are color, thickness and etc.

```
{python}
# 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)
# or
cnt = contours[4]
cv.drawContours(img, [cnt], 0, (0, 255, 0), 3)
```

Steps for detecting and drawing contours in OpenCV

1. Read the image in grayscale format.
2. Apply binary thresholding or Canny edge detection on the image.
3. Find the contours from the resulting image in (2).
4. Draw contours on original image

In [3]:
```
# Create a new image
img = np.zeros((256, 256), dtype=np.uint8)

cv.rectangle(img, (25, 25), (231, 231), 255, -1)

contours, hierarchy = cv.findContours(img, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
```

In [5]:
```
len(contours)
```

Out[5]:
```
1
```

In [6]:
```
contours[0]
```

Out[6]:
```
array([[[ 25,  25]],

       [[ 25, 231]],

       [[231, 231]],

       [[231,  25]]], dtype=int32)
```

In [8]:
```
contours[0].shape
```

Out[8]:
```
(4, 1, 2)
```

In [9]:
```
cnt = contours[0]
img_bgr = cv.cvtColor(img, cv.COLOR_GRAY2BGR)
cv.drawContours(img_bgr, [cnt], 0, (0, 255, 0), 3)

cv.imshow('contours', img_bgr)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [3]:
```
#1: Grayscale image
img = cv.imread('images/monitor.jfif')
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

#2: apply binary thresholding
ret, thresh = cv.threshold(img_gray, 200, 255, cv.THRESH_BINARY_INV)

cv.imshow('img', img)
cv.imshow('Binary image', thresh)
cv.waitKey(0)
cv.destroyAllWindows()

# find and draw contours
contours, hierarchy = cv.findContours(thresh, cv.RETR_TREE,
                                      cv.CHAIN_APPROX_NONE)
img_copy = img.copy()
cv.drawContours(img_copy, contours, -1, (0, 255, 0), 1,
                    cv.LINE_AA)

cv.imshow('Contour', img_copy)
cv.waitKey(0)
cv.destroyAllWindows()
```

Try `cv.RETR_EXTERNAL` to retrieve just the external boundaries (contours).

In [4]:
```
contours, hierarchy = cv.findContours(thresh, cv.RETR_EXTERNAL,
                                      cv.CHAIN_APPROX_SIMPLE)
img_copy = img.copy()
cv.drawContours(img_copy, contours, -1, (0, 255, 0), 1, cv.LINE_AA)

cv.imshow('Contour (simple)', img_copy)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [5]:
```
img = cv.imread('images/wood_planck.jfif')
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

_, thresh = cv.threshold(img_gray, 180, 255, cv.THRESH_BINARY_INV)

cv.imshow('original', img)
cv.imshow('thresholded image', thresh)
cv.waitKey(0)
cv.destroyAllWindows()

contours, _ = cv.findContours(thresh, cv.RETR_EXTERNAL,
                              cv.CHAIN_APPROX_SIMPLE)
img_copy = img.copy()
cv.drawContours(img_copy, contours, -1, (0, 255, 0), 1, cv.LINE_AA)

cv.imshow('external contour', img_copy)
cv.waitKey(0)
cv.destroyAllWindows()
```

How to extract the contour that encloses the object of interest from a list of contours?

The following example demonstrates one very simple way to choose from a set of contours: length (how many contour points). A more formalized and effective way would be to leverage the information from the contour features, which will discuss later in this tutorial.

In [6]:
```
img_bgr = cv.imread('images/monitor.jfif')
img = cv.imread('images/monitor.jfif', 0)

ret, thresh = cv.threshold(img, 180, 255, cv.THRESH_BINARY_INV)

cv.imshow('image', img)
cv.imshow('threshold', thresh)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [7]:
```
contours, _ = cv.findContours(thresh, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)
length = [len(i) for i in contours]
```

In [10]:

```
contours[0].shape
```

Out[10]:
```
(6, 1, 2)
```

In [11]:
```
Idx_max = np.argmax(length)
cnt = contours[Idx_max]

cv.drawContours(img_bgr, [cnt], 0, (255, 0, 0))
cv.imshow('contour', img_bgr)
cv.waitKey(0)
cv.destroyAllWindows()
```

Contour features

1. Moments

Image moments help you to calculate some features like center of mass of the object, area of object etc.

In [12]:
```
M = cv.moments(cnt)
print(M)
```

```
{'m00': 36579.0, 'm10': 3863880.6666666665, 'm01': 3751910.6666666665, 'm20': 523504181.16
66666, 'm11': 393058326.5, 'm02': 499571920.1666666, 'm30': 79890535773.40001, 'm21': 5242
3706511.73333, 'm12': 51853809510.96667, 'm03': 76030428569.2, 'mu20': 115358146.38604271,
'mu11': -3260191.9940478206, 'mu02': 114738172.64357662, 'mu30': 221383865.03416443, 'mu2
1': -583399634.7706699, 'mu12': -247734700.35206413, 'mu03': 1251895430.7594147, 'nu20':
0.08621535325487124, 'nu11': -0.0024365735169228054, 'nu02': 0.08575200275133085, 'nu30':
0.0008651001633300612, 'nu21': -0.002279746625838872, 'nu12': -0.0009680711360966657, 'nu0
3': 0.004892022918901492}
```

In [13]:
```
cx = int(M['m10'] / M['m00'])
cy = int(M['m01'] / M['m00'])

print(f'centroid: {(cx, cy)}')
```

```
centroid: (105, 102)
```

In [14]:
```
print(f'area: {cv.contourArea(cnt):.3f}')
```

```
area: 36579.000
```

In [18]:
```
print(f'perimeter: {cv.arcLength(cnt, True):.3f}')
```

```
perimeter: 1066.049
```

2. Contour approximation

It approximates a contour shape to another shape with less number of vertices depending upon the precision we specify.

```
{python}
eps = 0.1*cv.arcLength(cnt, True)
approx = cv.approxPolyDP(cnt, eps, True)
```

A wise selection of eps is needed to get the correct output.

3. Convex hull

4. Bounding rectangle

Straight bounding rectangle

```
{python}
x, y, w, h = cv.boundingRect(cnt)
cv.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

Rotated rectangle

Here the bounding rectangle is drawn with a minimum area. The function is `cv.minAreaRect()` . It returns a Box 2D structure which contains centers, (x, y), dimension, $(width, height)$ and angle.

```
{python}
rect = cv.minAreaRect(cnt)
box = cv.boxPoints(rect)
box = np.int0(box)
cv.drawContours(img, [box], 0, (0, 0, 255), 2)
```

5. Minimum enclosing circle

```
{python}
(x, y), radius = cv.minEnclosingCircle(cnt)
center = (int(x), int(y))
radius = int(radius)
cv.circle(img, center, radius, (0, 0, 255), 2)
```

6. Fitting an ellipse

```
{python}
ellipse = cv.fitEllipse(cnt)
cv.ellipse(img, ellipse, (0, 255, 0), 2)
```

In [2]:

```
img = cv.imread('images/car.jfif', 0)

ret, thresh = cv.threshold(img, 200, 250, cv.THRESH_BINARY_INV)
contours, _ = cv.findContours(thresh, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)

print(len(contours))
```

Out[2]:

```
1
```

In [5]:

```
# draw contours
cnt = contours[0]
img_bgr = cv.imread('images/car.jfif')
img_bgr_copy = img_bgr.copy()
cv.drawContours(img_bgr_copy, [cnt], 0, (0, 0, 255), 2, cv.LINE_AA)

cv.imshow('contour', img_bgr_copy)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [6]:

```
x, y, w, h = cv.boundingRect(cnt)
cv.rectangle(img_bgr, (x, y), (x+w, y+h), (0, 255, 0), 2)
cv.imshow('car with bounding rectangle', img_bgr)
cv.waitKey(0)
```

Out[6]:
```
-1
```

Extract region of interest

In computer vision, blob detection methods aim at detecting regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions. The OpenCV `cv.SimpleBlobDetector()` class comes in handy in this case. The online documentation can be found here. The algorithm can be described as the following:

1. Convert the source image to binary image by applying thresholding with several thresholds from minThreshold (inclusive) to maxThreshold (exclusive) with distance thresholdStep between neighboring thresholds.
   - params.thresholdStep = 10 (default)
   - params.minThreshold = 50 (default)
   - params.maxThreshold = 220 (default)
2. Extract connected components from every binary image by `cv.findContours()` and calculate their centers.
3. Group centers from several binary images by their coordinates. Close centers form that corresponds to one blob, which is controlled by minDistBetweenBlobs parameter.
   - params.minDistBetweenBlobs = 10 (default)
4. From the groups, estimate final centers of blobs and their radiuses and return as locations and size of keypoints.

This link provides an overview of parameter setting. Filtrations of returned blobs based on several characteristics:
1) color
2) area
3) circularity. Formula: $circularity = \frac{4 \times \pi \times area}{perimeter \times perimeter}$
4) convexity. Convexity indicates how far the contour of an object deviates from a convex outline.
5) inertia_ratio. The inertia of a blob is "the inertial resistance of the blob to rotation about its principal axes". It depends on how the mass of the blob (I guess in this case the area) is distributed throughout the blob's shape.

> The ratio $\frac{I_{min}}{I_{max}}$ gives us some idea of how rounded the object is. The ratio will be zero for line and 1 for circle.

Default values of parameters are tuned to extract dark circular blobs.

In [19]:
```
img = cv.imread('images/airplanes.jfif')
blur = cv.GaussianBlur(img, (3, 3), 0)
gray = cv.cvtColor(blur, cv.COLOR_BGR2GRAY)

# parameters
params = cv.SimpleBlobDetector_Params()
params.filterByArea = False
params.filterByCircularity = False
params.filterByConvexity = False
params.filterByInertia = False

detector = cv.SimpleBlobDetector_create(params)
keypoints = detector.detect(gray)
```

```
blank = np.zeros((1, 1))
img_keypoints = cv.drawKeypoints(img, keypoints, blank, (0, 0, 255),
                                 cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv.imshow("keypoints", img_keypoints)
cv.waitKey(0)
```

```
Out[19]: -1
```

```
In [8]: keypoints
```

```
Out[8]: (<KeyPoint 000001B22251B4E0>,
         <KeyPoint 000001B22251B300>,
         <KeyPoint 000001B22251BB70>,
         <KeyPoint 000001B2224DC4E0>,
         <KeyPoint 000001B2224DCAB0>,
         <KeyPoint 000001B2224DCB10>)
```

The keypoint attributes could be found from this link and this Stackoverflow post. It can also be accessed as below.

```
In [9]: cv.KeyPoint_convert(keypoints)
```

```
Out[9]: array([[104.92087 , 157.41397 ],
               [201.69695 , 112.26543 ],
               [ 34.014908, 106.528755],
               [116.88195 ,  50.501286],
               [116.16433 ,  54.39373 ],
               [115.      , 142.      ]], dtype=float32)
```

Issues with cv.SimpleBlobDetector() function

```
In [20]:
circle_white = np.zeros((150, 150), dtype=np.uint8)
cv.circle(circle_white, (75, 75), 40, 200, -1)

circle_black = np.ones((150, 150))*255
circle_black = np.uint8(circle_black)
cv.circle(circle_black, (75, 75), 40, 50, -1)

row_img = np.concatenate((circle_black, circle_white, circle_black), axis=1)
img = np.concatenate((row_img, row_img, row_img), axis=0)

cv.imshow('circles', img)
cv.waitKey(0)
```

```
Out[20]: -1
```

```
In [21]:
# setup the parameters for SimpleBlobDetector
params = cv.SimpleBlobDetector_Params()
# Change thresholds
params.minThreshold = 0
params.maxThreshold = 255

# Filter by Area.
params.filterByArea = False

# Filter by Circularity
params.filterByCircularity = True
```

```python
# Filter by Convexity
params.filterByConvexity = False

# Filter by Inertia
params.filterByInertia = False

# filter by color
# params.filterByColor = True
# params.blobColor = 255

# create a detector
detector = cv.SimpleBlobDetector_create(params)

# detect blobs
keypoints = detector.detect(img)

# draw detected blobs as red circles
im_with_keypoints = cv.drawKeypoints(img, keypoints, np.array([]), (0, 0, 255),
                                    cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv.imshow("Keypoints", im_with_keypoints)
cv.waitKey()
```

Out[21]:
```
-1
```

Exercise

- Load the 'dice.jfif' and perform blob detection to detect as many pips on the dice as possible. You may use trackbar to find the best parameter settings.

In [ ]:
```python

```

Activity

1. Experiment with different edge detectors: Sobel, Laplacian, Prewitt, Scharr derivatives and Canny operators (all with aperture size of 3) on image named 'pineapple.jfif'. Comment on the results.
2. Write a program to identify the white object (probably laptop) present in the image 'electronic.jfif'. Draw bounding boxes on the objects.
3. Isolate the clock with the aid of edge detection and contours' properties. The example result should be as follows: segmented_clock

In [ ]:
```python

```