Learning outcomes

Today, we will explore different types of image segmentation techniques:

- Clustering based
  - k-means clustering
- Anisotropic image segmentation
- Region based
  - Watershed transformation

Setup

In [1]:

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

import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
from util_func import *

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

cv.useOptimized()
```

Out[1]:

```
True
```

K-means clustering

- Learn to use `cv.kmeans()` function in OpenCV for data clustering.

Understanding Parameters

Input parameters

1. samples: It should be of **np.float32** data type, and each feature should be put in a single column.
2. nclusters(K): Number of clusters required for clustering.
3. criteria: It is the iteration termination criteria. When this criteria is satisfied, algorithm iteration stops. It should be a tuple of 3 parameters. They are `(type, max_iter, epsilon)` :
   A. type of termination criteria. It has 3 flags as below:
      - `cv.TERM_CRITERIA_EPS` - stop the algorithm iteration if specified accuracy, *epsilon*, is reached.
      - `cv.TERM_CRITERIA_MAX_ITER` - stop the algorithm after the specified number of iterations, *max_iter*.
      - `cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER` - stop the iteration when any of the above condition is met.

   B. max_iter - An integer specifying maximum number of iterations.

   C. epsilon - Required accuracy.
4. attempts: Flag to specify the number of times the algorithm is executed using different initial labellings. The algorithm returns the labels that yield the best compactness. This compactness is returned as output.

5. flags: This flag is used to specify how initial centers are taken. Normally 2 flags are used for this: `cv.KMEANS_PP_CENTERS` and `cv.KMEANS_RANDOM_CENTERS` .

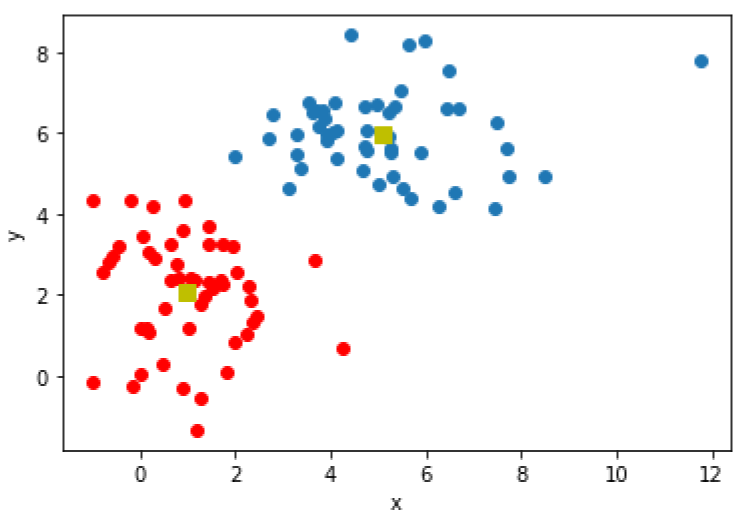
Ouput parameters

1. compactness: It is the sum of squared distance from each point to their corresponding centers.
2. labels: This is the label array.
3. centers: This is array of centers of clusters.

Demo on simple 2D data

In [2]:

```
np.random.seed(55)

mean1 = (1, 2)
cov1 = [[1, 0], [0, 2]]
X1, Y1 = np.random.multivariate_normal(mean1, cov1, 50).T
dat1 = np.concatenate((X1[:, None], Y1[:, None]), axis = 1)

mean2 = (5, 6)
cov2 = [[2, 0], [0, 1]]
X2, Y2 = np.random.multivariate_normal(mean2, cov2, 50).T
dat2 = np.concatenate((X2[:, None], Y2[:, None]), axis = 1)

# vertical stack and transpose
Z = np.vstack((dat1, dat2))
Z = np.float32(Z)

# define criteria and apply kmeans()
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret, label, center = cv.kmeans(Z, 2, None, criteria, 10, cv.KMEANS_RANDOM_CENTERS)

# Now label the data in cluster
A = Z[label.ravel() == 0]
B = Z[label.ravel() == 1]

# Plot the data
plt.scatter(A[:, 0], A[:, 1])
plt.scatter(B[:, 0], B[:, 1], c = 'r')
plt.scatter(center[:, 0], center[:, 1], s = 80, c = 'y', marker = 's')
plt.xlabel('x'), plt.ylabel('y')
plt.show()
```



Demo on images

In [4]:
```
img = cv.imread('images/flower.jfif')
img_rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)

plt.figure(), plt_img(img_rgb)
plt.show()
```

In [5]:
```
# 2 helper functions to visualize the distribution of clusters
def centroid_histogram(clust_labels):
    # Create histogram based on the number of pixels assigned to each cluster
    numLabels = len(np.unique(clust_labels))
    hist, _ = np.histogram(clust_labels, bins = numLabels)

    # Normalize the histogram, such that it sums to one
    hist = hist.astype("float32")
    hist /= hist.sum()

    return hist

def plot_colors(hist, centroids):
    # Initialize bar chart representing relative frequency
    # of each of the colors
    bar = np.zeros((50, 300, 3), dtype = np.uint8)
    startX = 0

    # loop over the percentage of each cluster and the color of each cluster
    for (percent, color) in zip(hist, centroids):
        # plot the relative percentage of each cluster
        endX = startX + (percent*300)
        cv.rectangle(bar, (int(startX), 0), (int(endX), 50),
                     color.astype("uint8").tolist(), -1)
        startX = endX

    # return bar chart
    return bar
```

In [6]:
```
# Reshape the image to 2D matrix
img_reshape = img_rgb.reshape((-1, 3))

# Convert uint8 to float
img_reshape = np.float32(img_reshape)

# define criteria, number of clusters and apply k-means clustering
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
attempts = 10
ret, label, center = cv.kmeans(img_reshape, K, None, criteria, attempts, cv.KMEANS_PP_CENT

# convert the center back to uint8
```

```
center = np.uint8(center)

#
res = center[label.flatten()]
result_image = res.reshape((img_rgb.shape))
```

In [9]:

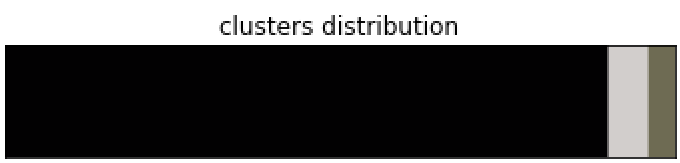
```
result_image_gray = cv.cvtColor(result_image, cv.COLOR_RGB2GRAY)
```

In [10]:

```
plt.figure(figsize = (12, 6))
plt.subplot(1, 2, 1), plt_img(result_image, "results")
plt.subplot(1, 2, 2), plt_img(img, "mask overlay")
plt.imshow(result_image_gray, cmap=plt.cm.nipy_spectral, alpha=.5)
# plt.colorbar()
plt.show()
```

In [11]:

```
hist = centroid_histogram(label)
bar = plot_colors(hist, center)

plt.figure(), plt_img(bar, title="clusters distribution")
plt.show()
```



Exercise

Try k-means clustering on image named 'electronic.jfif' in the following color spaces:

- RGB
- HSV
- CIELAB

Comments on the results.

Structure tensor

Structure tensors are a matrix representation of partial derivative information. In the field of image processing and computer vision, it is typically used to represent the gradient or "edge" information. It also has a more powerful description of local patterns as opposed to the directional derivative through its coherence measure.

Background

Gradient info serves several purposes. It can relate the structure of objects in an image, identify features of interest for recognition/classification directly or provide the basis of further processing for various computer vision tasks.

In math, the structure tensor, also referred to as the 2nd moment matrix, is a matrix derived from the gradient of a function.

Interpretation

The importance of the 2D structure tensor stems from the fact eigenvalues λ_1, λ_2 and the corresponding e_1, e_2 summarizes the gradient within the window defined by W.

By expanding the effective radius of window function W (that is increasing its variance), one can make the structure tensor more robust in the face of noise, at the cost of diminished spatial resolution.

Important formula

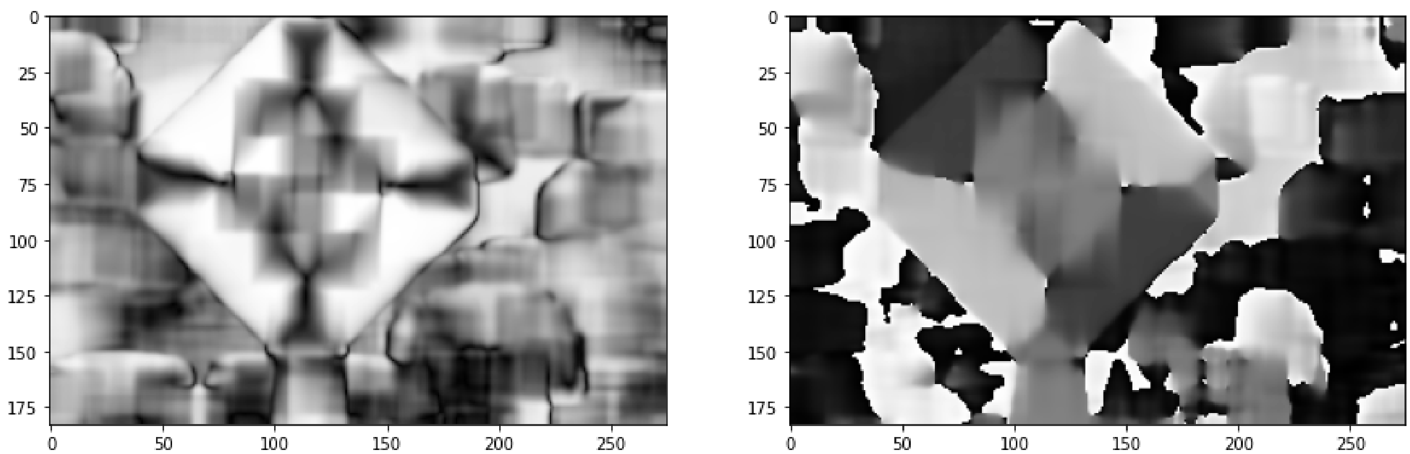
Copy the function named `calcGST()` and paste it on the notebook. Make some modification to incorporate more filtering and gradient computation options.

In [12]:

```python
def calcGST(inputIMG, w):
    img = np.float32(inputIMG)

    # Gradient structure tensor components
    imgDiffX = cv.Scharr(img, cv.CV_32F, 1, 0)
    imgDiffY = cv.Scharr(img, cv.CV_32F, 0, 1)
    imgDiffXY = cv.multiply(imgDiffX, imgDiffY)
    imgDiffXX = cv.multiply(imgDiffX, imgDiffX)
    imgDiffYY = cv.multiply(imgDiffY, imgDiffY)

    J11 = cv.boxFilter(imgDiffXX, cv.CV_32F, (w, w))
    J22 = cv.boxFilter(imgDiffYY, cv.CV_32F, (w, w))
    J12 = cv.boxFilter(imgDiffXY, cv.CV_32F, (w, w))

    # eigenvalue
    tmp1 = J11+J22
    tmp2 = J11-J22
    tmp2 = cv.multiply(tmp2, tmp2)
    tmp3 = cv.multiply(J12, J12)
    tmp4 = np.sqrt(tmp2 + 4.0*tmp3)

    lambda1 = 0.5*(tmp1+tmp4)
    lambda2 = 0.5*(tmp1 - tmp4)

    # coherency
    imgCoherencyOut = cv.divide(lambda1-lambda2, lambda1+lambda2)

    # orientation calculation
    imgOrientationOut = cv.phase(J22-J11, 2.0*J12, angleInDegrees = True)
    imgOrientationOut = 0.5*imgOrientationOut

    return imgCoherencyOut, imgOrientationOut
```

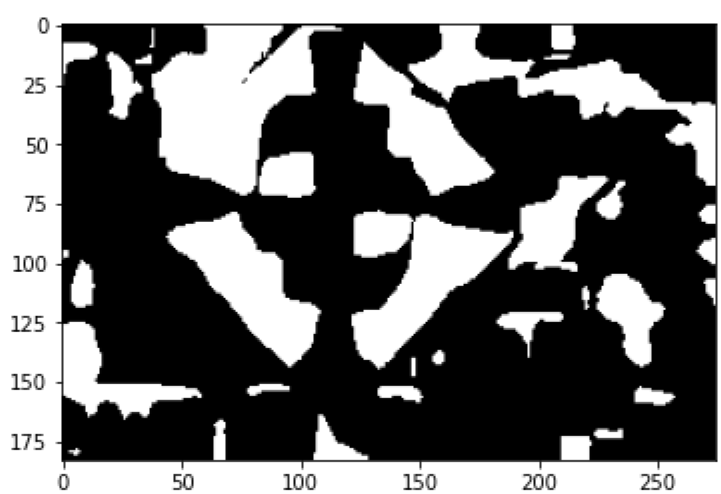
In [14]:

```python
# Display coherency and orientation maps
W = 25

im = cv.imread(cv.samples.findFile('images/traffic_sign.jfif'))
# blur = cv.GaussianBlur(im, (7, 7), 0)
gray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
gray = cv.GaussianBlur(gray, (7, 7), 0)

imgCoherency, imgOrientation = calcGST(gray, W)

# Visualize
imgCoherency_norm = cv.normalize(imgCoherency, None, alpha = 0, beta = 1, norm_type = cv.N
                                dtype = cv.CV_32F)
imgOrientation_norm = cv.normalize(imgOrientation, None, alpha = 0, beta = 1, norm_type =
                                dtype = cv.CV_32F)

plt.figure(figsize = (15, 15))
plt.subplot(1, 2, 1)
plt.imshow(imgCoherency_norm, 'gray')
plt.subplot(1, 2, 2)
plt.imshow(imgOrientation_norm, 'gray')
plt.show()
```



In [5]:

```python
# Coherency threshold
_, imgCoherencyBin = cv.threshold(imgCoherency, 0.75, 255, cv.THRESH_BINARY)

plt.imshow(np.uint8(imgCoherencyBin), 'gray')
plt.show()
```



The code implementation will be shown in the class on how to extract foreground from *Orientation* and

Coherency map.

Watershed segmentation

> Any grayscale image can be viewed as a topographic surface where high intensity denotes peaks and hills while low intensity denotes valleys. You starts filling every isolated valleys (local minima) with different colored water (labels). As the water rises, depending on the peaks (gradients) nearby, water from different valleys, obviously with different colors will start to merge. To avoid that, you build barriers in the locations where water merges... The barriers you created gives you the segmentation results. (Adapted from this link)

However, this would produce over-segmentation results due to noise in images. Segmentation using watershed transform works better if we can identify foreground and background locations with markers. This begs a question: how can we determine the background / foreground markers? The answer is there are plenty of possible solutions based on the problems. Here, we will look at 2 ways to determine the markers:

Markers determination

Morphological gradient

As shown in example 3.

Thresholding + distance transform

As shown in example 4.

Example 3

In [2]:

```
from scipy import ndimage as ndi
from skimage.segmentation import watershed
```

In [3]:

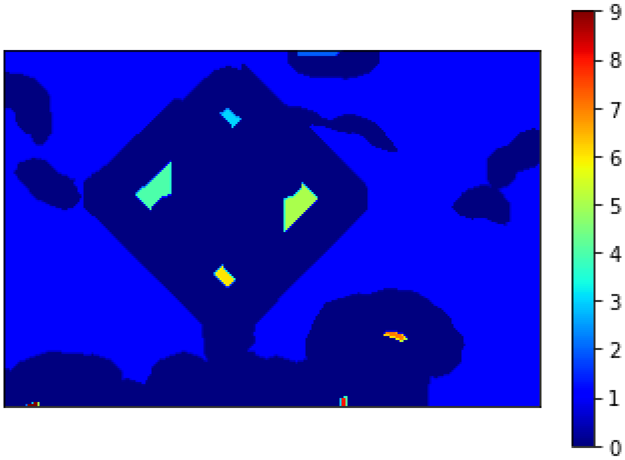
```
img = cv.imread("images/traffic_sign.jfif")
show_img("original", img)
blur = cv.GaussianBlur(img, (5, 5), 0)
blur = cv.pyrMeanShiftFiltering(blur, 15, 20, maxLevel=2)

gray = cv.cvtColor(blur, cv.COLOR_BGR2GRAY)
show_img("grayscale", gray)
```

In [4]:

```
# morphological gradient kernel
kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, (11, 11))
loc_grad = cv.morphologyEx(gray, cv.MORPH_GRADIENT, kernel, iterations=2)

show_img("local gradient", loc_grad)
```

In [7]:

```
markers = loc_grad < 80
s = np.ones((3, 3), dtype=int)
markers = ndi.label(markers, structure=s)[0]

plt.imshow(markers, cmap=plt.cm.jet)
plt.colorbar()
plt.xticks([]), plt.yticks([])
plt.show()
```



In [8]:

```
edge = cv.Canny(gray, 250, 500)
labels = watershed(edge, markers)
np.unique(labels, return_counts=True)
```

Out[8]:

```
(array([1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array([40736,   279,  1234,  2325,  2295,  1270,  1066,   736,   384],
       dtype=int64))
```

In [9]:

```
gray_ori = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

plt.figure()
plt.imshow(gray_ori, cmap=plt.cm.gray)
plt.imshow(labels, cmap=plt.cm.nipy_spectral, alpha=0.5)
plt.xticks([]), plt.yticks([])
plt.show()
```

As of how to extract the traffic sign region, I would leave it as exercise for the readers.

Example 4

In [10]:

```
img = cv.imread("images/traffic_sign1.jpg")
(h, w) = img.shape[:2]
blur = cv.GaussianBlur(img, (5, 5), 0)
```

In [13]:

```
# Convert to Lab
img_lab = cv.cvtColor(blur, cv.COLOR_BGR2Lab)

img_lab = img_lab.reshape((h*w, 3))
img_lab = np.float32(img_lab)
# criteria
```

```
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
flags = cv.KMEANS_PP_CENTERS

compactness, labels, centers = cv.kmeans(img_lab, 10, None, criteria, 10, flags)

quant = centers.astype(np.uint8)[labels]
quant = quant.reshape((h, w, 3))

bgr = cv.cvtColor(quant, cv.COLOR_Lab2BGR)

show_img("kmeans quantization", bgr)
```

In [20]:

```
img_lab[..., 0].min()
```

Out[20]:

```
6.0
```

In [14]:

```
gray = cv.cvtColor(bgr, cv.COLOR_BGR2GRAY)

th = cv.threshold(gray, 0, 255, cv.THRESH_BINARY | cv.THRESH_OTSU)[1]

show_img("threshold", th)
```

In [15]:

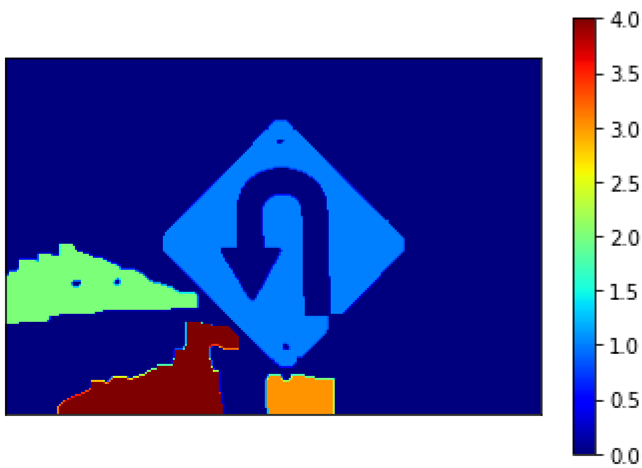
```
from skimage.feature import peak_local_max
```

In [16]:

```
kernel = np.ones((3, 3), dtype=np.uint8)
th = cv.morphologyEx(th, cv.MORPH_OPEN, kernel, iterations=3)

dist_transform = cv.distanceTransform(th, cv.DIST_L2, 3)

coords = peak_local_max(dist_transform, footprint=np.ones((100, 100)), labels=th)

mask = np.zeros(dist_transform.shape, dtype=bool)
mask[tuple(coords.T)] = True
markers, _ = ndi.label(mask)
labels = watershed(-dist_transform, markers, mask=th)
```

In [17]:

```
plt.imshow(labels, cmap=plt.cm.jet)
plt.colorbar(), plt.xticks([]), plt.yticks([])
plt.show()
```



Example 5

In [35]:
```python
import random as rng
rng.seed(3107)
```

In [36]:
```python
img = cv.imread("images/cups_coffee.webp")
cv.imshow("source", img)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [37]:
```python
# change the background to black
# img[np.all(img == 255, axis=2)] = 0
blur = cv.GaussianBlur(img, (7, 7), 0)

# do the Laplacian filter
kernel = np.ones((3, 3), np.float32)
kernel[1, 1] = -8

imgLaplacian = cv.filter2D(blur, cv.CV_32F, kernel)
sharp = np.float32(img)
# The purpose of subtraction is basically to extract the high frequency components
# of an image
imgResult = sharp - imgLaplacian

# convert back to 8 bits
imgResult = np.clip(imgResult, 0, 255)
imgResult = imgResult.astype('uint8')
imgLaplacian = np.clip(imgLaplacian, 0, 255)
imgLaplacian = imgLaplacian.astype('uint8')

cv.imshow("black blackground", img)
cv.imshow('New sharped Image' , imgResult)
cv.imshow('Laplacian', imgLaplacian)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [38]:
```python
bw = cv.cvtColor(imgResult, cv.COLOR_BGR2GRAY)
_, bw = cv.threshold(bw, 0, 255, cv.THRESH_BINARY | cv.THRESH_OTSU)
cv.imshow("binary", bw)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [39]:
```python
dist = cv.distanceTransform(bw, cv.DIST_L2, 5)

cv.normalize(dist, dist, 0, 1.0, cv.NORM_MINMAX)
cv.imshow("distance transform", dist)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [40]:
```python
dist = cv.threshold(dist, 0.5, 1.0, cv.THRESH_BINARY)[1]

kernel1 = np.ones((3, 3), dtype=np.uint8)
dist = cv.dilate(dist, kernel1)
cv.imshow("peaks", dist)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [41]:
```python
dist_8u = dist.astype('uint8')
```

```
# find total markers
contours, _ = cv.findContours(dist_8u, cv.RETR_EXTERNAL,
                              cv.CHAIN_APPROX_SIMPLE)
# Create the marker image for the watershed algorithm
markers = np.zeros(dist.shape, dtype=np.int32)
# Draw the foreground markers
for i in range(len(contours)):
    cv.drawContours(markers, contours, i, (i+1), -1)
# Draw the background marker
# cv.circle(markers, (5,5), 3, (255,255,255), -1)
markers_8u = (markers * 10).astype('uint8')
cv.imshow('Markers', markers_8u)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [12]:

```
markers_8u.shape
```

Out[12]:

```
(640, 480)
```

In [42]:

```
cv.watershed(imgResult, markers)
#mark = np.zeros(markers.shape, dtype=np.uint8)
mark = markers.astype('uint8')
mark = cv.bitwise_not(mark)
cv.imshow('Markers', mark)
cv.waitKey(0)
cv.destroyAllWindows()
```

In [43]:

```
# Generate random colors
colors = []
for contour in contours:
    colors.append((rng.randint(0,256), rng.randint(0,256), rng.randint(0,256)))
# Create the result image
dst = np.zeros((markers.shape[0], markers.shape[1], 3), dtype=np.uint8)
# Fill labeled objects with random colors
for i in range(markers.shape[0]):
    for j in range(markers.shape[1]):
        index = markers[i,j]
        if index > 0 and index <= len(contours):
            dst[i,j,:] = colors[index-1]
# Visualize the final image
cv.imshow('Final Result', dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

Comment on the outcome of example 5 (distance transform): By inspecting the outcome of `dst`, it seems like the marker-controlled watershed had missed the 5th paper cup entirely (including the lid) and the 4th cup paper cup. Oversegmentation occurs on the first cup as well.

Weekly activity

1. Apply k-means clustering on 'zebra.jfif' to segment out the zebra.
    - You are required to determine the optimal k by plotting the within cluster sum of squares vs number of clusters (2-10).
    - Apply the clustering method on 3 color spaces: BGR, HSV and LAB. Compare the results obtained.

In [ ]: