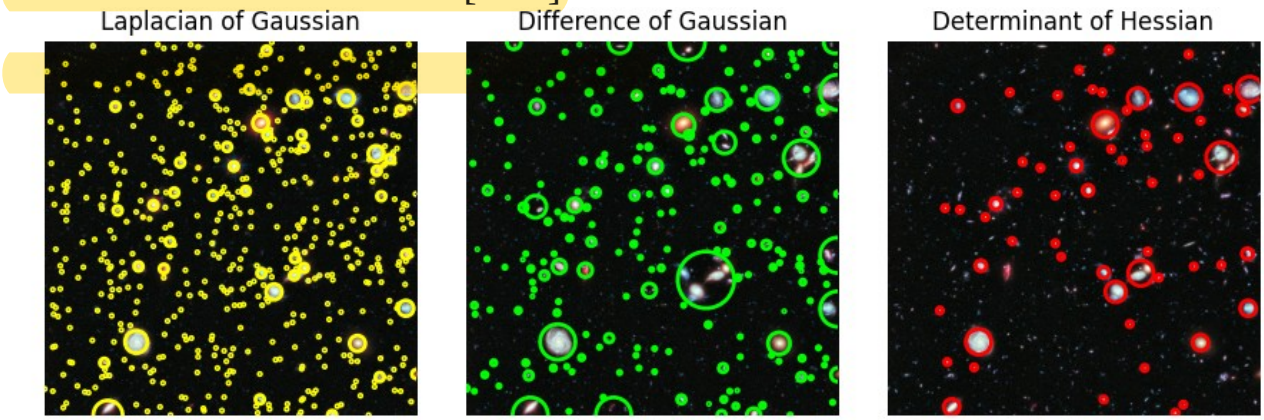
ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Μικρόπουλος Νικόλαος
Περδικάκης Σπυρίδων

Assignment - Features and Tracking

Exercise 1 - Blob Detection [3/10]



Develop a simple 'blob' detector. The detector should rely on the LoG filter.

1. Implement a function to detect blobs of a given radius. [2]

2. Add an extra parameter to the detector to select either: i) dark blobs, ii) light blobs, or iii) both. [0.5]

3. Extend the detector so instead of a single radius it accepts a range (min, max). [0.5]

Test the detector firstly in the images 'black_dots.jpg', 'white_dots.jpg' and then also in 'coins.tiff' and 'circles_{}.jpg'.

Present the results by showing the detected blobs as circles in the initial images.

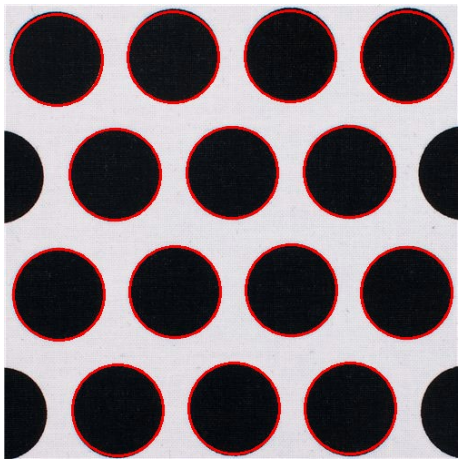
Η διαδικασία για την ανίχνευση των blobs είναι η εξής:

- Φορτώνουμε την εικόνα.
- Τη μετατρέπουμε σε grayscale.
- Για τη δοσμένη ακτίνα r, βρίσκουμε την ακτίνα με τη μέγιστη απόκριση $s=\frac{r}{\sqrt{2}}$ ώστε να την εφαρμόσουμε στο Laplacian of Gaussian φίλτρο.
- Βρίσκουμε τα blobs της εικόνας, εφαρμόζοντας το φίλτρο Laplacian of Gaussian.
- Σχεδιάζουμε στην εικόνα κύκλους με κέντρο και ακτίνα σύμφωνα με τα blobs που ανιχνεύσαμε.

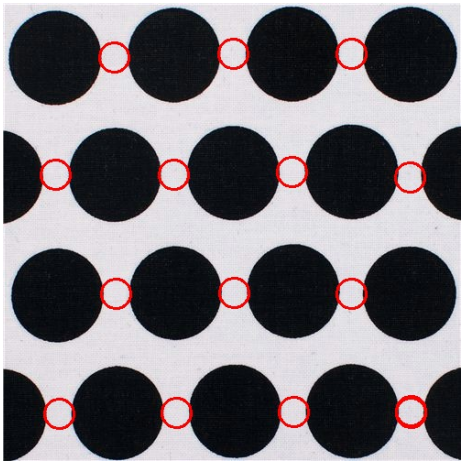
Και τα τρία υποερωτήματα απαντώνται ταυτόχρονα με τον κώδικα, ο οποίος περιέχει αναλυτικά σχόλια.

Προσπαθήσαμε να σχεδιάσουμε blobs στις κυκλικές περιοχές των εικόνων που μας ζητήθηκαν.

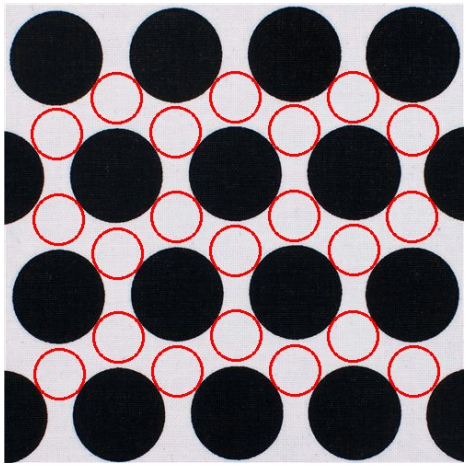
Παρακάτω παραθέτουμε τις εικόνες, με σχεδιασμένα τα ανιχνευθέντα blobs.



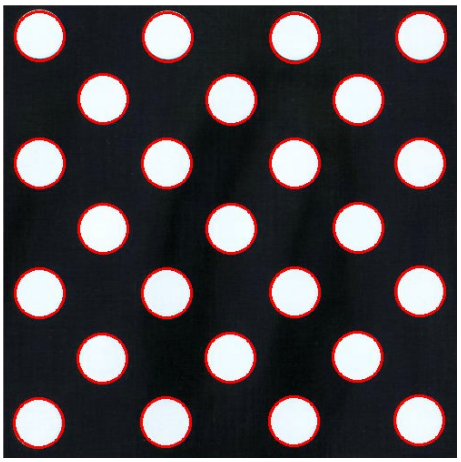
black_dots.jpg με radius=50 και threshold=0.7



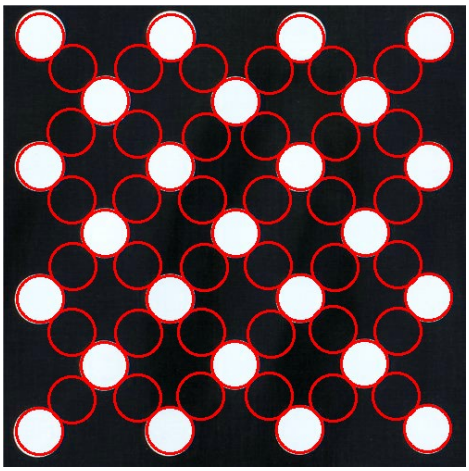
black_dots.jpg με radius=16 και threshold=0.8



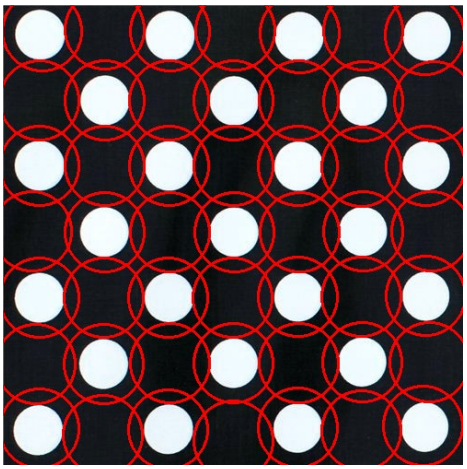
black_dots.jpg με radius=27 και threshold=0.65



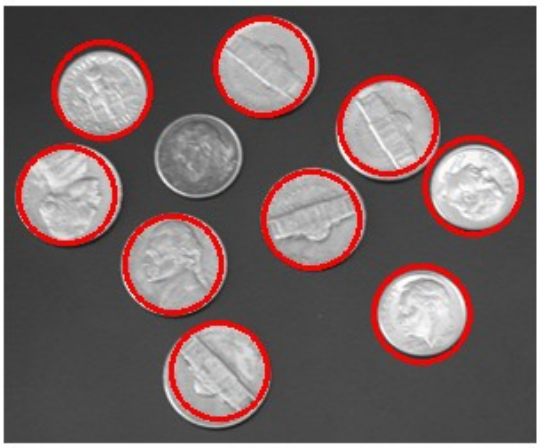
white_dots.jpg με radius=27 και threshold=0.7



white_dots.jpg με radius=25 και threshold=0.1



white_dots.jpg με radius=43 και threshold=0.4

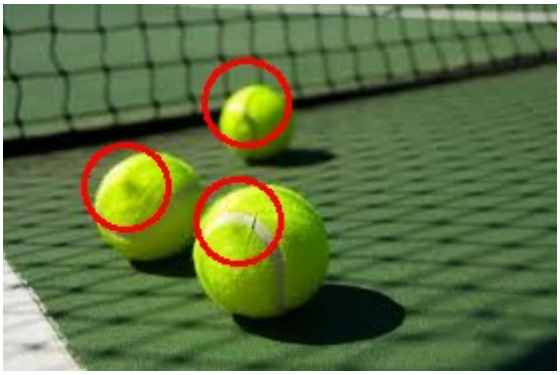


coins.tiff με radius=27 και threshold=0.6

circles_01.jpg με radius=39 και threshold=0.72

circles_02.jpg με radius=12 και threshold=0.48

circles_03.jpg με radius=75 και threshold=0.38



circles_04.jpg με radius=21 και threshold=0.65

circles_05.jpg με radius=17 και threshold=0.44

Παραθέτουμε τον κώδικα:

```
import numpy as np
import cv2 as cv
from google.colab.patches import cv_imshow
from scipy import ndimage as ndi
from google.colab import drive
drive.mount("/content/drive", force_remount=True)
root_folder = "/content/drive/My Drive/"
url_black_dots = root_folder + "/BlobDetectionImages/black_dots.jpg" #r=50 thres=0.6
url_white_dots = root_folder + "/BlobDetectionImages/white_dots.jpg" #r=27 thres=0.7
url_coins      = root_folder + "/BlobDetectionImages/coins.tiff"     #r=35 thres=0.6
url_circles_01 = root_folder + "/BlobDetectionImages/circles_01.jpg" #r=39 thres=0.79
url_circles_02 = root_folder + "/BlobDetectionImages/circles_02.jpg" #r=12 thres=0.46
url_circles_03 = root_folder + "/BlobDetectionImages/circles_03.jpg" #r=75 thres=0.38
url_circles_04 = root_folder + "/BlobDetectionImages/circles_04.jpg" #r=21 thres=0.65
url_circles_05 = root_folder + "/BlobDetectionImages/circles_05.jpg" #r=17 thres=0.44
# from PIL import Image
img = cv.imread(url_black_dots) #choose which of the above images to process
r = 1
img = cv.resize(img, dsize=(0,0), fx=r, fy=r, interpolation=cv.INTER_CUBIC)
# img = cv.cvtColor(img, cv.COLOR_BGR2RGB) #convert image to rgb
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) #convert image to grayscale

# Laplacian of Gaussian (LoG) Kernel
def logkern(sigma, n=None):
  #window size
  if n is None:
   n = np.ceil(sigma*6)
  y,x = np.ogrid[-n//2:n//2+1,-n//2:n//2+1]
  y_filter = np.exp(-(y*y/(2.*sigma*sigma)))
  x_filter = np.exp(-(x*x/(2.*sigma*sigma)))
  final_filter = (-(2*sigma**2) + (x*x + y*y) ) *  \
         (x_filter*y_filter) * (1/(2*np.pi*sigma**4))
  return final_filter

def local_maxima_3D(data_in, order=1):

  size = 1 + 2 * order
  footprint = np.ones((size, size, size))
  footprint[order, order, order] = 0
  data = data_in.copy()
  h = data.shape[0]
  w = data.shape[1]
  Z1 = np.zeros((h,w), dtype=data.dtype)
  Z2 = np.zeros((h,w,1), dtype=data.dtype)
  data = np.insert(data,0, Z1, axis=2)
  data = np.append(data, Z2, axis=2)

  filtered = ndi.maximum_filter(data, footprint=footprint, mode='nearest')
  filtered=np.delete(filtered,0,axis=2)
```

```
  filtered=np.delete(filtered,-1,axis=2)
  data=np.delete(data,0,axis=2)
  data=np.delete(data,-1,axis=2)
  mask_local_maxima = data > filtered
  coords = np.asarray(np.where(mask_local_maxima)).T
  values = data[mask_local_maxima]
  return coords, values
def detect_blobs(image,rmin,rmax,thr,choice):
 """ Detects and draws blobs in an image
  Args:
    image: the image to use
    rmin: the minimum radius
    rmax: the maximum radius
    thr: the threshold to use
    choice: enter 1 for detecting light blobs only
        enter 2 for detecting dark blobs only
        enter 3 for detecting both
  Returns:
    the image with the blobs found, drawn as a circle  """
 r=rmin
 disp_image = img.copy()

 while r <= rmax:
  s=r/np.sqrt(2.0) #find the radius with the maximum response
  log = logkern(s)  #compute Laplacian of Gaussian to find edges
  image = np.nan_to_num(image) #Replace nan values with 0
  img_log = cv.filter2D(image, cv.CV_64F, log) #apply Laplacian of Gaussian filter to find edges
  max_val = np.max(np.abs(img_log))
  thres = thr * max_val
  if choice == 1:
   img_blobs = np.abs(img_log) >= thres #detect only light blobs
  elif choice == 2:
   img_blobs = img_log < thres #detect only dark blobs
  elif choice == 3:
   #detect both light and dark blobs (instead of thres I could use a number for light blobs and a different o
ne for dark blobs)
   img_blobs =  (np.abs(img_log) >= thres) | (img_log < thres) #light blobs or dark blobs

  img_log_thres = img_log.copy()
  img_log_thres[img_blobs == False] = 0

  img_log_thres = img_log_thres[:, :, np.newaxis] #insert another column to prepare for local_maxima_3D
  coords, values = local_maxima_3D(np.abs(img_log_thres))
  for c in coords:
   disp_image=cv.circle(disp_image, (c[1],c[0]), r, (0,0,255), 2) #draw the circle
  r+=1 #do the same for the next radius
 cv_imshow(disp_image);  #display the image
 return disp_image

I2=detect_blobs(img_gray,50,50,0.7,1) #call the function to get the results
```

Exercise 2 - Feature Based Tracking and RANSAC [7/10]

The aim is to estimate the pose of an object in an image sequence. The template appearance of the object is extracted from the first frame. The pose of the object in each frame is defined by the coordinates of the points on its contour (i.e. the four corners). Use the 'Melita' dataset to test your algorithm.

Steps:

1. Set the template in the first frame, extract and store features from it.
2. For each frame:

    Extract features and match them with the template features.

    Using the feature correspondences calculate the transformation between the template and the object. Implement both the affine and the homography transformation to compare their performance on the task.

    To deal with outliers use the RANSAC algorithm.

    Using the estimated transformation transform the contour points of the template and draw the contour of the object in the current frame.

Εδώ υλοποιήσαμε δύο διαφορετικούς κώδικες, έναν για τον μετασχηματισμό affine, και έναν για τον μετασχηματισμό ομογραφίας.

Και οι δύο υλοποιούνται εντός του αλγορίθμου RANSAC, ο οποίος αναλαμβάνει να διαχωρίσει τα inliers από τα outliers. Για αυτόν τον λόγο, όταν καλούμε την match_features, δεν δίνουμε το κατώφλι thres, ώστε να μας επιστρέψει όλα τα σημεία, και ο διαχωρισμός να γίνει αποκλειστικά από τον RANSAC.

Στην εργασία *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography των Martin A. Fischler and Robert C. Bolles,* και στο κεφάλαιο *B. The Maximum Number of Attempts to Find a Consensus Set* (http://www.ai.sri.com/pubs/files/836.pdf) βρίσκουμε πως οι ελάχιστες επαναλήψεις του RANSAC που πρέπει να υλοποιήσουμε, ορίζονται από τη σχέση

$k = \frac{\log(1-z)}{\log(1-b)}$, όπου:

$b = \frac{1}{\omega^{-n}}$

ω = η πιθανότητα οποιοδήποτε επιλεγμένο σημείο να είναι εντός της ανοχής λάθους του μοντέλου

n = ένα σύνολο καλών σημείων. Εδώ βάζουμε τον ελάχιστο αριθμό ζευγών που απαιτούνται. 3 για τον affine transformation και 4 για τον homography transformation.

z = η πιθανότητα τουλάχιστον ένα από τα επιλεγμένα ζεύγη σημείων να είναι σωστό

Για τη βέλτιστη απόδοση του RANSAC, χρησιμοποιήσαμε z=0.99 w=0.5.

Παρατηρήσαμε πως ο k μας έδινε κάποια αποτελέσματα, αλλά στην πράξη, όταν τον πολλαπλασιάσαμε με το 5, πήραμε πολύ ικανοποιητικότερα αποτελέσματα, χωρίς να υπάρχει σημαντική καθυστέρηση στους υπολογισμούς.

Ο affine transformation που υλοποιήθηκε είναι ο

$$\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

όπως περιγράφεται εδώ: *https://staff.fnwi.uva.nl/r.vandenboomgaard/IPCV20162017/LectureNotes/MATH/homogenous.html*

και ο homography trasformation που υλοποιήθηκε είναι αυτός που περιγράφεται από τον *David Kriegman* στο βιβλίο του *Computer Vision I*, και στο κεφάλαιο *Homography Estimation.*

Συνοπτικά, η διαδικασία που χρησιμοποιήσαμε είναι η εξής:

- Φορτώνουμε την εικόνα αναφοράς και το πλαίσιο αναφοράς.
- Από το πλαίσιο αναφοράς δημιουργούμε το template.
- Φορτώνουμε την εικόνα από την οποία θα ανιχνεύσουμε το αντικείμενο που θέσαμε στο template.

- Με την match_features βρίσκουμε τα σημεία που είναι κοινά στις δύο εικόνες, τα σχεδιάζουμε και τα ενώνουμε με μία ευθεία ανάμεσα στις δυο εικόνες.
- Εκτελούμε τον RANSAC με affine (στον δεύτερο κώδικα με homography) tranformation. Βάζοντας στον RANSAC threshold = 3 pixels, πήραμε ικανοποιητικά αποτελέσματα.
- Χρησιμοποιούμε τα inliers σημεία που ανιχνεύσαμε και σχεδιάζουμε το πλαίσιο του αντικειμένου στη δεύτερη εικόνα.

Επιλέξαμε πέντε εικόνες από το dataset μία ανά εκατό, ώστε να πάρουμε πολλές διαφορετικές γωνίες λήψης του αντικειμένου, και να δούμε την απόδοση του κάθε μετασχηματισμού.

Παρατηρήσαμε πως ο homography transformation είναι πιο αποδοτικός από τον affine όσο περισσότερο στρέφεται το αντικείμενο, πράγμα που συμφωνεί και με τη θεωρία.
Επίσης, ο RANSAC, αν και δέχεται πάρα πολλά ζεύγη σημείων που είναι εμφανώς λανθασμένα, κάνει εξαιρετική δουλειά στο να ξεχωρίσει τα inliers.

Επειδή ο RANSAC επιλέγει τυχαία σημεία, κάθε φορά που τρέχουμε τον κώδικα τα αποτελέσματα μπορεί να διαφέρουν ελαφρώς. Η διαφορά αυτή μπορεί να ελαχιστοποιηθεί κάνοντας πολύ περισσότερα iterations από 5*k, αλλά αυτό θα κάνει τη διαδικασία πιο αργή.

Οι κώδικες, περιέχουν αναλυτικά σχόλια.

Παραθέτουμε τις πέντε εικόνες, πρώτα με τον affine, και μετά για τον homography transformation ώστε να φανούν εύκολα οι διαφορές στα αποτελέσματα.

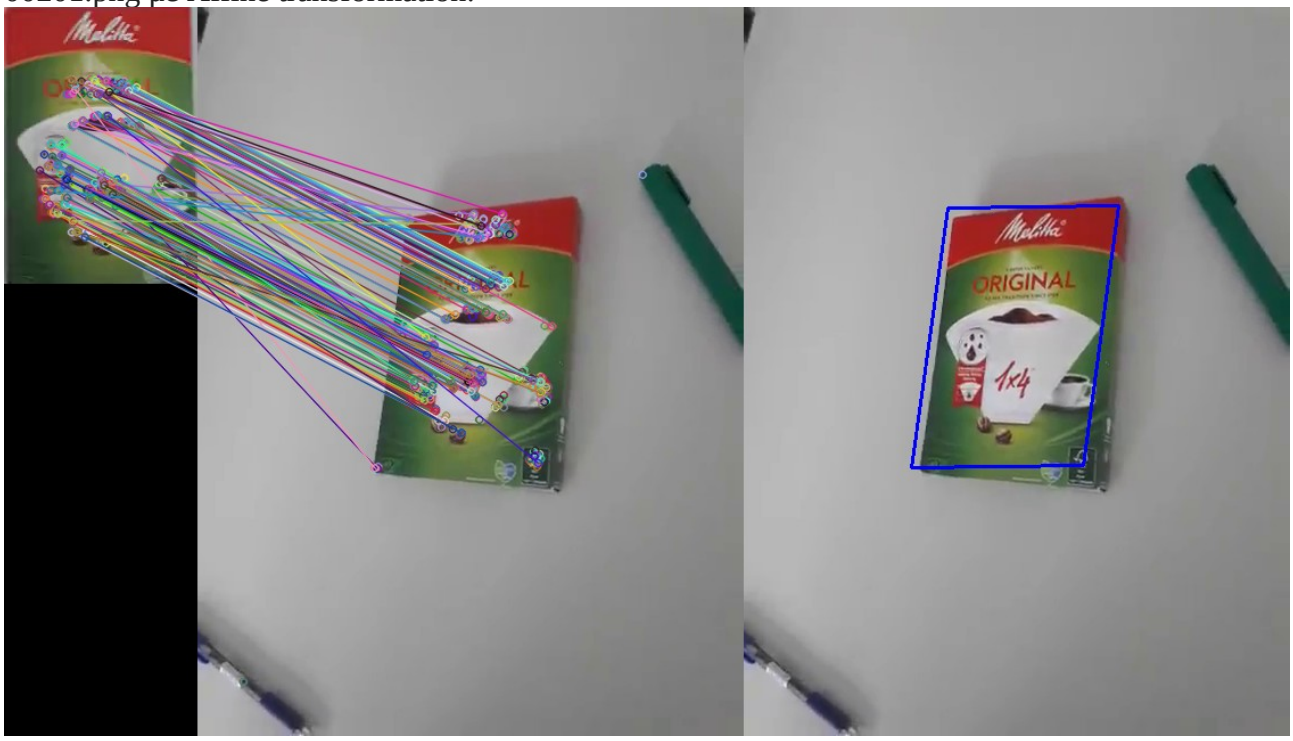
00001.png με Affine transformation:

00001.png με Homography transformation:

00101.png με Affine transformation:

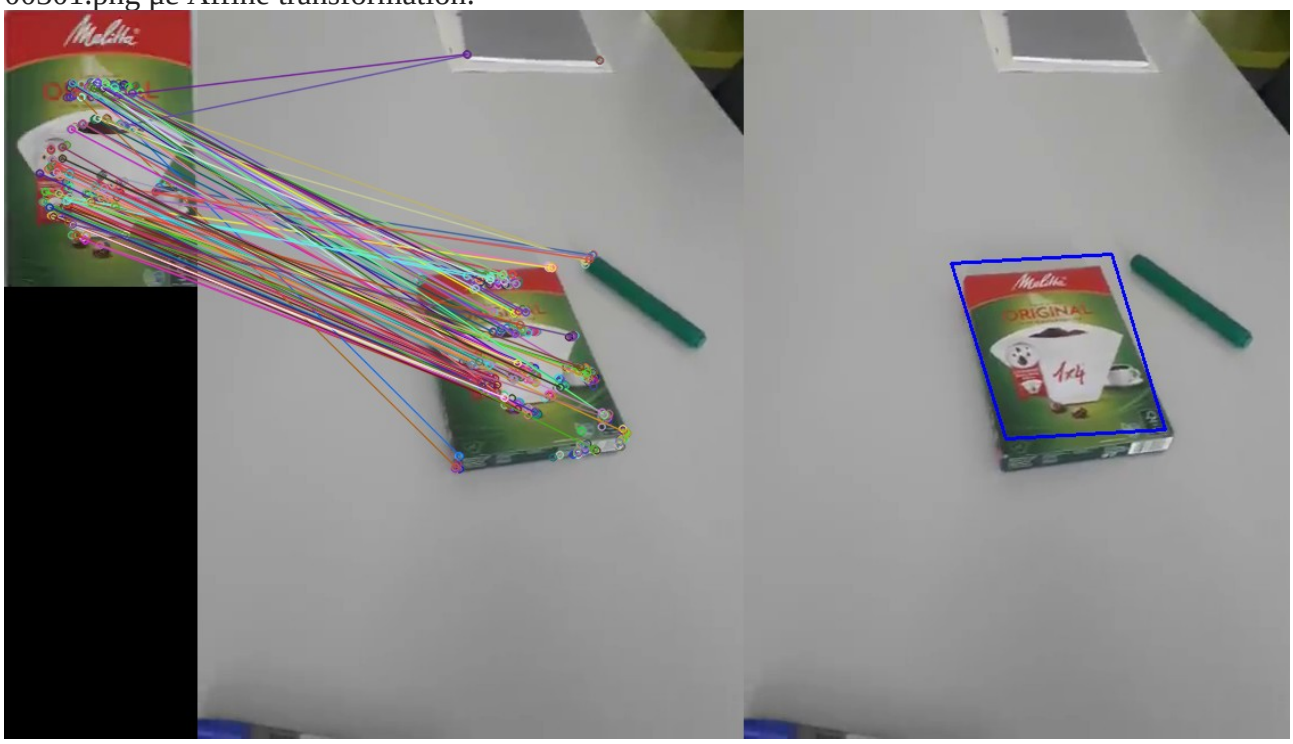
00101.png με Homography transformation:

00201.png με Affine transformation:

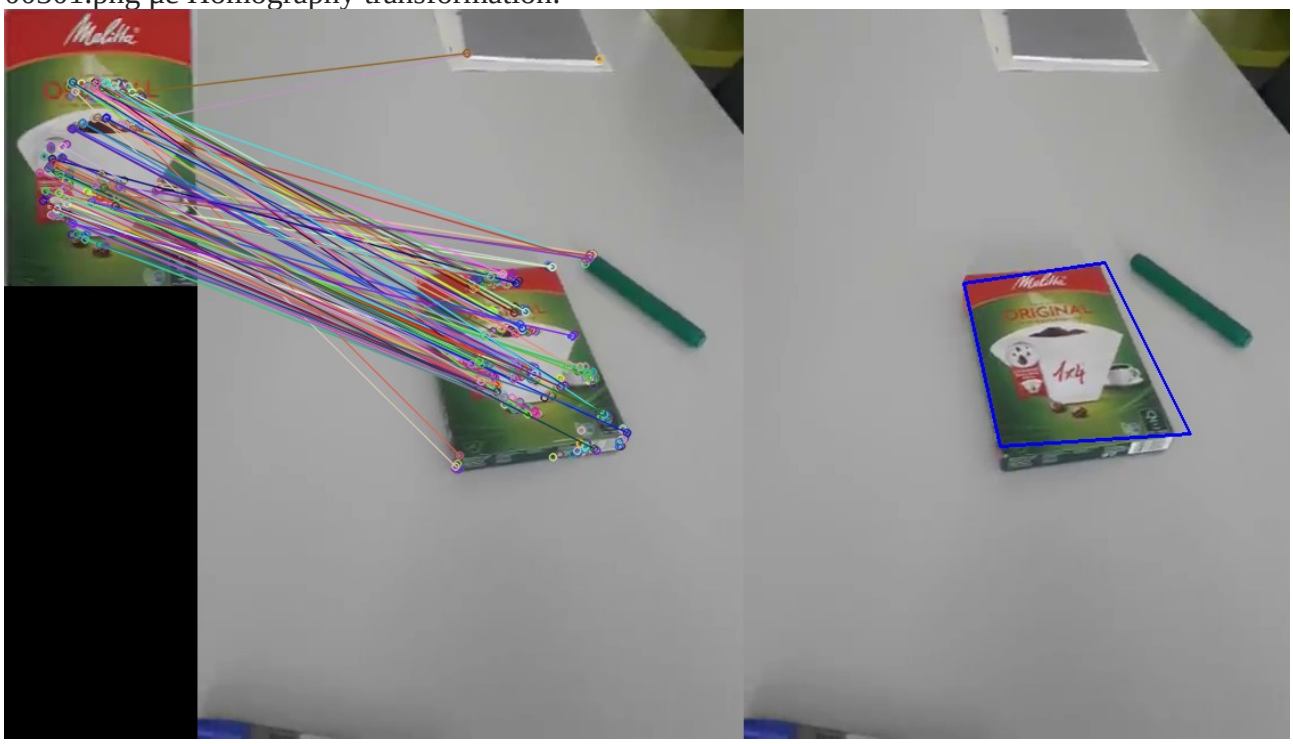


00201.png με Homography transformation:

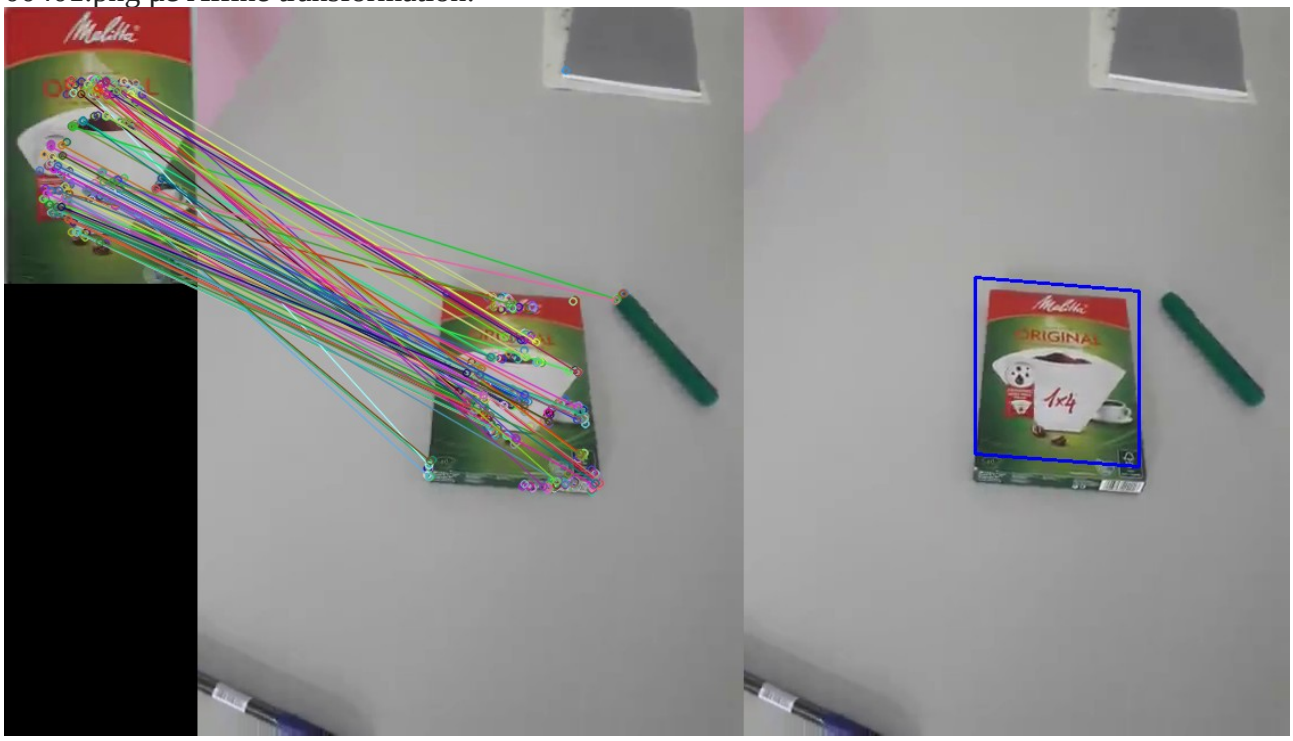
00301.png με Affine transformation:



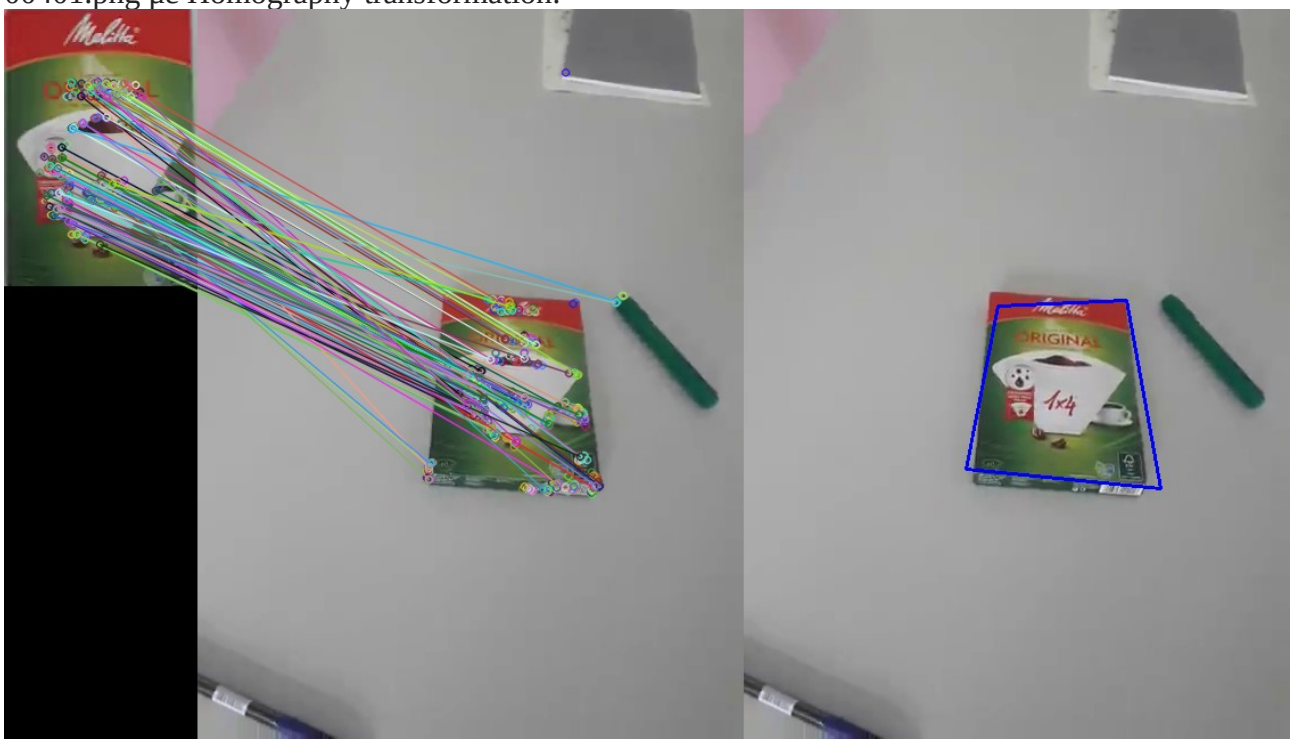
00301.png με Homography transformation:



00401.png με Affine transformation:



00401.png με Homography transformation:



Ο κώδικας που υπολογίζει RANSAC με Affine transformation:

```
from google.colab import drive
from google.colab.patches import cv2_imshow
import os
import matplotlib.pyplot as plt
import cv2
import numpy as np
import math
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/TrackingDatasets/')
root_folder = "/content/drive/My Drive/TrackingDatasets/"
from TrackingDataset import Dataset

def tohom(points):#convert points to homogenous
 n = points.shape[1]
 points_hom = np.vstack([points, np.ones((1,n), points.dtype)])
 return points_hom

def fromhom(points):#convert points to eucledian
 dims = points.shape[0] - 1
 points_eucl = points[0:dims, :]
 for d in range(dims):
  points_eucl[d,:] /= points[dims,:]
 return points_eucl

def apply_tf(points, tf):
 pointsh = tohom(points) #convert points to homogenous
 p2h = np.dot(tf, pointsh) #matrix multiplication
 p2t = fromhom(p2h) #convert points to eucledian
 return p2t

def to_pointlist(point_mat): #make the points list for the contour
 point_list = []
 for p in point_mat.T: point_list.append((p[0], p[1]))
 return point_list

def keypoints_tonumpy(keypoints):  # Get numpy array from opencv keypoints.
  p2d_np = np.zeros((2, len(keypoints)), dtype=np.float)
  for i,kp in enumerate(keypoints):
    p2d_np[0][i] = kp.pt[0]
    p2d_np[1][i] = kp.pt[1]
  return p2d_np

def match_features(template, img, thres=None): #Detects features.
 orb = cv2.ORB_create()   # Initiate ORB detector
 # find the keypoints and descriptors.
 kp0, des0 = orb.detectAndCompute(template,None)
 kp1, des1 = orb.detectAndCompute(img,None)
 # create BFMatcher object
 bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
 # Match descriptors.
 matches = bf.match(des0,des1)
 # Sort them in the order of their distance.
 matches = sorted(matches, key = lambda x:x.distance)
 distances = [m.distance for m in matches]
 print('Distance Range:', np.min(distances), np.max(distances))
 if thres is None: #if there is no threshold, return ALL matches
  matches_good = matches
 else:
  matches_good = [m for m in matches if m.distance < thres]
 print('Keypoints template/image/mathces/filtered:',len(kp0), len(kp1),len(matches), len(matches_good))
 kp0np = keypoints_tonumpy(kp0)
 kp1np = keypoints_tonumpy(kp1)
 indices0 = np.array([m.queryIdx for m in matches_good])
 indices1 = np.array([m.trainIdx for m in matches_good])
 pts0 = kp0np[:,indices0]
 pts1 = kp1np[:,indices1]
 return pts0, pts1, kp0, kp1, matches_good

def min_num_pairs(tranformation='affine'):
  return 3# The mimimum number of pairs required for affine transformation is 3.

def ransac_iters(w=0.5, n=min_num_pairs(), z=0.99):
  """ Computes the required number of iterations for RANSAC.
```

```
  Implementing chapter "B. The Maximum Number of Attempts to Find a Consensus Set "
  found in http://www.ai.sri.com/pubs/files/836.pdf
  Args:
    w: probability that any selected data point is within the
      error tolerance of the model.
    n: a subset of good data points
    z: probability that at least one of our random selections is an error-free
      set of n data points
  Returns:
    minimum number of required iterations
  """
  b = 1/(w**-n)
  k = round(math.log(1-z)/math.log(1-b))
  k = 5*k # We found that 5 times the required iterations is better in accuracy
  return k

def ransac_affine(points1, points2, ransac_thr, ransac_iter):
  """ Solves RANSAC with affine transormation.
  Implementing chapter "B. The Maximum Number of Attempts to Find a Consensus Set "
  as described in http://www.ai.sri.com/pubs/files/836.pdf
  and Affine Transform as described in https://staff.fnwi.uva.nl/r.vandenboomgaard/IPCV20162017/LectureNotes/
MATH/homogenous.html
  Args:
    ransac_thr: number of pixels to compute the distance to distinguish inliers from outliers
    ransac_iter: number of times to perform RANSAC
  Returns:
    best pair of points found
  """
  print('Calculating initial affine transform using SIFT feature matches and RANSAC')
  print('RANSAC params: thres = {}, iter = {}'.format(ransac_thr, ransac_iter))
  best_affine_transform = None
  best_num_inliers = 0
  for i in range(ransac_iter):
    random_index = np.random.choice(points1.shape[0], min_num_pairs('affine'), replace=False)# Select min_num
_pairs points randomly
    x = points1[random_index]
    x1 = points2[random_index]
    ### Implement the affine transformation
    a = np.array([
      [x[0][0], x[0][1], 1, 0, 0, 0],
      [0, 0, 0, x[0][0], x[0][1], 1],
      [x[1][0], x[1][1], 1, 0, 0, 0],
      [0, 0, 0, x[1][0], x[1][1], 1],
      [x[2][0], x[2][1], 1, 0, 0, 0],
      [0, 0, 0, x[2][0], x[2][1], 1],
    ])
    b = x1.reshape(-1)
    affine_tr, res, _, _ = np.linalg.lstsq(a, b, rcond=-1) #solve the least squares
    # Reshape affine transform matrix
     #make the calculated matrix
    affine_tr = np.reshape(affine_tr, (2,3))
    affine_tr = np.vstack([affine_tr, [0,0,1]])
    if best_affine_transform is None:
      best_affine_transform = affine_tr
    # Calculate number of inliers
    num_inliers = 0
    for j in range(points1.shape[0]):
      template_point = np.array(list(points1[j]) + [1])
      target_point = np.array(list(points2[j]) + [1])
      template_point_image = np.matmul(affine_tr, template_point)
      distance = np.sqrt(np.sum((template_point_image - target_point) ** 2))
      if distance < ransac_thr: #the inliers must have distance less than ransac_thr (in pixels)
        num_inliers += 1
    if num_inliers > best_num_inliers:
      best_affine_transform = affine_tr
      best_num_inliers = num_inliers
  return best_affine_transform

datasets = {    #the dataset frow which to load the images
  'melita': {
   'basedir': root_folder + 'Melita/',
   'img_tmpl': "img/{:05d}.png",
   'gt_filename':'groundtruth_rect.txt',},}

ds_name = 'melita'
ds = Dataset(datasets[ds_name]['basedir'],
```

```
      datasets[ds_name]['img_tmpl'],
      datasets[ds_name]['gt_filename'],
      ratio=1.)
img, bboxpoints = ds.grab(0) #Load the first image
print('bboxpoint:',bboxpoints)
cv2.rectangle(img, bboxpoints[0], bboxpoints[1], (255,0,0)) #Draw the contour
cv2_imshow(img)
### Setting the template.
template, template_contour = ds.get_template(0)
template_contour = np.array(template_contour).T
cv2_imshow(template)

framesNum=5 #number of pictures to pick from the dataset
step_frame = 100 #number of pictures to skip (in order to pick very different poses of the same object)

for i in range(framesNum):
 ### Loading another frame.
 img, _ = ds.grab(i*step_frame+1)
 ### Matching Features between template and new Frame.
 pts0, pts1, kp0, kp1, matches = match_features(template, img)
 ### Draw the first 100 matches found for distance < thres (here we haven't set any threshold)
 disp_img = np.zeros((100,100))
 disp_img1 = cv2.drawMatches(template,kp0,img,kp1,matches,disp_img)
 ### Solve Affine and display.
 at = ransac_affine(pts0.T,pts1.T,3,ransac_iters()) #Here we've set the threshold as 3, but it can be altered
 ### Tranform template contour to get the contour of the object in the current frame.
 image_contour = apply_tf(template_contour, at)
 image_contour = image_contour.astype(int)
 cp = to_pointlist(image_contour) #get the points to draw the contour
 ###Draw the contour.
 disp_img = img.copy()
 cv2.line(disp_img, cp[0], cp[1], (255,0,0), 2)
 cv2.line(disp_img, cp[1], cp[2], (255,0,0), 2)
 cv2.line(disp_img, cp[2], cp[3], (255,0,0), 2)
 cv2.line(disp_img, cp[3], cp[0], (255,0,0), 2)
 cv2_imshow(np.concatenate((disp_img1, disp_img), axis=1))
```

Ο κώδικας που υπολογίζει RANSAC με Homography transformation:

```
from google.colab import drive
import os
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/TrackingDatasets/')
root_folder = "/content/drive/My Drive/TrackingDatasets/"
print(os.path.isdir(root_folder))
import numpy as np
np.set_printoptions(precision=2, suppress=True)
import cv2
from google.colab.patches import cv2_imshow
import numpy as np
import random
import math
from TrackingDataset import Dataset

def tohom(points):
 n = points.shape[1]
 points_hom = np.vstack([points, np.ones((1,n), points.dtype)])
 return points_hom

def fromhom(points):
 dims = points.shape[0] - 1
 points_eucl = points[0:dims, :]
 for d in range(dims):
  points_eucl[d,:] /= points[dims,:]
 return points_eucl

def apply_tf(points, tf):
 pointsh = tohom(points) #convert to homogenous
 p2h = np.dot(tf, pointsh) #matrix multiplication
 p2t = fromhom(p2h) #convert to eucledian
 return p2t

def to_pointlist(point_mat): #make a list of points
 point_list = []
 for p in point_mat.T: point_list.append((p[0], p[1]))
 return point_list

def to_pointmat(point_list): #make an array of points
 point_mat = np.array(point_list).T
 return point_mat

def keypoints_tonumpy(keypoints):
  # Get numpy array from opencv keypoints.
  p2d_np = np.zeros((2, len(keypoints)), dtype=np.float)
  for i,kp in enumerate(keypoints):
    p2d_np[0][i] = kp.pt[0]
    p2d_np[1][i] = kp.pt[1]
  return p2d_np

def match_features(template, img, thres=None):
 # Initiate ORB detector
 orb = cv2.ORB_create()
 # find the keypoints and descriptors.
 kp0, des0 = orb.detectAndCompute(template,None)
 kp1, des1 = orb.detectAndCompute(img,None)
 # create BFMatcher object
 bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
 # Match descriptors.
 matches = bf.match(des0,des1)
 #print(len(matches))
 # Sort them in the order of their distance.
 matches = sorted(matches, key = lambda x:x.distance)
 distances = [m.distance for m in matches]
 print('Distance Range:', np.min(distances), np.max(distances))
 if thres is None:
  matches_good = matches
 else:
  matches_good = [m for m in matches if m.distance < thres]
 print('Keypoints template/image/mathces/filtered:',len(kp0), len(kp1),len(matches), len(matches_good))
 kp0np = keypoints_tonumpy(kp0)
 kp1np = keypoints_tonumpy(kp1)
 indices0 = np.array([m.queryIdx for m in matches_good])
```

```
 indices1 = np.array([m.trainIdx for m in matches_good])
 pts0 = kp0np[:,indices0]
 pts1 = kp1np[:,indices1]
 return pts0, pts1, kp0, kp1, matches_good

def min_num_pairs():
  # For Homography transformation, the minimum number of pairs is 4.
  return 4

def ransac_iters(w=0.5, n=min_num_pairs(), z=0.99):
  """ Computes the required number of iterations for RANSAC.
  Implementing chapter "B. The Maximum Number of Attempts to Find a Consensus Set "
  found in http://www.ai.sri.com/pubs/files/836.pdf
  Args:
    w: probability that any selected data point is within the
      error tolerance of the model.
    n: a subset of good data points
    z: probability that at least one of our random selections is an error-free
      set of n data points
  Returns:
    minimum number of required iterations
  """
  b = 1/(w**-n)
  k = round(math.log(1-z)/math.log(1-b))
  k = 5*k # I found that 5 times the required iterations is better in accuracy
  return k

def pickup_samples(pts1, pts2):
  """ Randomly select k corresponding point pairs.
  Note that here we assume that pts1 and pts2 have
  been already aligned: pts1[k] corresponds to pts2[k].
  This function makes use of min_num_pairs()
  Args:
    pts1 and pts2: point coordinates from Image 1 and Image 2
  Returns:
    pts1_rnd and pts2_rnd: min_num_pairs randomly selected points
              from pts1 and pts2  """
  # Get the random min_num_pairs (here it is 4) indexes in range
  random_num = random.sample(range(0, len(pts2)), min_num_pairs())
  pts1_rnd = []
  pts2_rnd = []
  # For each random numbers, take the pts points into the arrays
  for i in random_num:
    pts1_rnd.append(pts1[i])
    pts2_rnd.append(pts2[i])
  return pts1_rnd, pts2_rnd

def compute_homography(pts1, pts2):
  """ Construct homography matrix and solve it by SVD, as described in
  "Homography Estimation" by David Kriegman
  Args:
    pts1: the coordinates of interest points in img1, array (N, 2)
    pts2: the coordinates of interest points in img2, array (M, 2)
  Returns:
    H: homography matrix as array (3, 3)  """
  # Making A matrix using 4 interest points, for every interest points,
  # we need 2 rows with 9 columns. So A becomes 8 by 9 matrix.
  A = np.empty((8,9))
  pts1_rnd, pts2_rnd = pickup_samples(pts1, pts2)
  # For every 4 points
  for i in range(4):
    # Extracting each points from pts1 and pts2
    x = pts1_rnd[i][0]
    y = pts1_rnd[i][1]
    x1 = pts2_rnd[i][0]
    y1 = pts2_rnd[i][1]
    # Putting the values to A matrix to get the H
    A[i*2][0] = -x
    A[i*2][1] = -y
    A[i*2][2] = -1
    A[i*2][3] = 0
    A[i*2][4] = 0
    A[i*2][5] = 0
    A[i*2][6] = x1*x
    A[i*2][7] = x1*y
    A[i*2][8] = x1
```

```
    A[i*2+1][0] = 0
    A[i*2+1][1] = 0
    A[i*2+1][2] = 0
    A[i*2+1][3] = -x
    A[i*2+1][4] = -y
    A[i*2+1][5] = -1
    A[i*2+1][6] = y1*x
    A[i*2+1][7] = y1*y
    A[i*2+1][8] = y1

  # Computing the SVD decomposition
  u,s,v = np.linalg.svd(A)
  # Picking right most vector (which is the eigenvector that has the smallest eigenvalue)
  H = v[8]
  # reshape it to 3*3 size
  H = np.reshape(H, (3,3))
  return H

def transform_pts(pts, H):
  """ Transform pst1 through the homography matrix to compare pts2 to find inliners
  Args:
    pts: interest points in img1, array (N, 2)
    H: homography matrix as array (3, 3)
  Returns:
    transformed points, array (N, 2)  """
  # Making homogeneous array of pts by adding 1 to last colums
  ones = np.ones((len(pts), 1), dtype=np.int8)
  homo_pts = np.append(pts, ones, axis=1)
  # Transform pts by multiplying H and homo_pts
  trans_pts = np.dot(homo_pts, np.transpose(H))
  # Make matrix from N * 3 to N * 2 by dividing x and y by trans_pts[i][2]
  trans_pts_ = np.empty((len(pts), 2))
  for i in range(len(pts)):
    trans_pts_[i][0] = trans_pts[i][0]/trans_pts[i][2]
    trans_pts_[i][1] = trans_pts[i][1]/trans_pts[i][2]
  return trans_pts_

def count_inliers(H, pts1, pts2, threshold=3):#Here we've set the threshold as 3, but it can be altered
  """ Count inliers
    Tips: We provide the default threshold value, but you're free to test other values
  Args:
    H: homography matrix as array (3, 3)
    pts1: interest points in img1, array (N, 2)
    pts2: interest points in img2, array (N, 2)
    threshold: scale down threshold
  Returns:
    number of inliers   """
  # inliers count
  inliers = 0
  # First, transform pts1 using H matrix
  pts1transf = transform_pts(pts1, H)
  # Calculate distances between transformed pts1 and pts2
  for i in range(len(pts2)):
    distance = math.sqrt((pts1transf[i][0]-pts2[i][0])**2 + (pts1transf[i][1]-pts2[i][1])**2)
    if distance<threshold: #if distance is smaller than threshold, add 1
      inliers += 1
  return inliers

def ransac(pts1, pts2):
  """ RANSAC algorithm
  Args:
    pts1: matched points in img1, array (N, 2)
    pts2: matched points in img2, array (N, 2)
  Returns:
    best homography observed during RANSAC, array (3, 3)
  """
  # To keep track of inlier counts, use maximum_inlier variable
  maximum_inlier = None
  # To keep track of homography, use best_Homography variable
  best_Homography = None
  # For ransac_iters times, doing ransac algorithm
  for i in range(ransac_iters()):
    H = compute_homography(pts1, pts2) # Compute homography
    cnt = count_inliers(H, pts1, pts2)
    # If it's first loop, put value directly to maximum_inlier
```

```
    if maximum_inlier == None:
      maximum_inlier = cnt
      best_Homography = H
    # If it's maximum inlier, then save homography
    if cnt > maximum_inlier:
      maximum_inlier = cnt
      best_Homography = H
  # After the loop, best_Homography has the best homography
  return best_Homography

datasets = {
  'melita': {
   'basedir': root_folder + 'Melita/',
   'img_tmpl': "img/{:05d}.png",
   'gt_filename':'groundtruth_rect.txt',
  },
}
ds_name = 'melita'
ds = Dataset(datasets[ds_name]['basedir'],
       datasets[ds_name]['img_tmpl'],
       datasets[ds_name]['gt_filename'],
       ratio=1.)

img, bboxpoints = ds.grab(0)
print('bboxpoint:',bboxpoints)
cv2.rectangle(img, bboxpoints[0], bboxpoints[1], (255,0,0))
cv2_imshow(img)
### Setting the template.
template, template_contour = ds.get_template(0)
template_contour = np.array(template_contour).T
print(template_contour)
cv2_imshow(template)

framesNum=5 #number of pictures to pick
step_frame = 100 #number of pictures to skip (in order to pick very different poses of the same object)

for i in range (framesNum):
 ### Loading another frame.
 img, _ = ds.grab(i*step_frame+1)
 # cv2_imshow(img)
 ### Matching Features between template and new Frame.
 pts0, pts1, kp0, kp1, matches = match_features(template, img)
 # Draw the first 100 matches found for distance < thres
 disp_img = np.zeros((100,100))
 disp_img1 = cv2.drawMatches(template,kp0,img,kp1,matches,disp_img)
 ### Implent Ransac using homography and display.
 at = ransac(pts0.T,pts1.T)
 # Tranform template contour to get the contour of the object in
 # the current frame.
 image_contour = apply_tf(template_contour, at)
 image_contour = image_contour.astype(int)
 print('Image Contour:\n', image_contour)
 cp = to_pointlist(image_contour)
 #Draw the contour.
 disp_img = img.copy()
 cv2.line(disp_img, cp[0], cp[1], (255,0,0), 2)
 cv2.line(disp_img, cp[1], cp[2], (255,0,0), 2)
 cv2.line(disp_img, cp[2], cp[3], (255,0,0), 2)
 cv2.line(disp_img, cp[3], cp[0], (255,0,0), 2)
 # cv2_imshow(disp_img)
 cv2_imshow(np.concatenate((disp_img1, disp_img), axis=1))
```