ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ
ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Ακαδημαϊκό έτος 2022-2023

ΟΡΑΣΗ ΥΠΟΛΟΓΙΣΤΩΝ

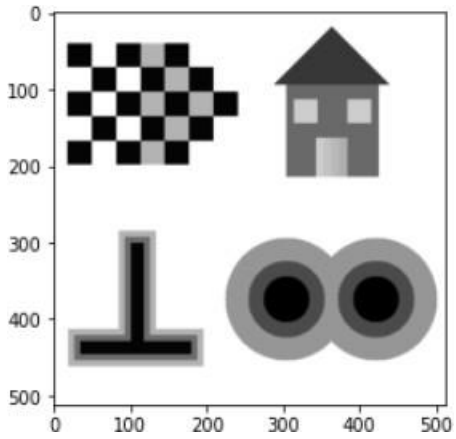
Αναφορά 1ης Εργαστηριακής Άσκησης

Δαμιανός Δημήτρης – 03119825
Καπετανάκης Αναστάσιος – 03119048

Μέρος 1ο :

1.1 Δημιουργία Εικόνων Εισόδου

Η εικόνα I της οποίας θα προσπαθήσουμε να βρούμε τις ακμές της είναι

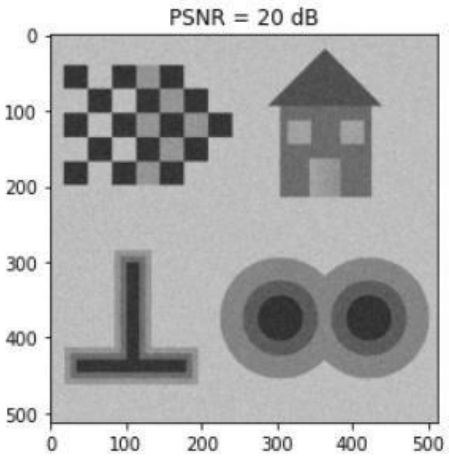


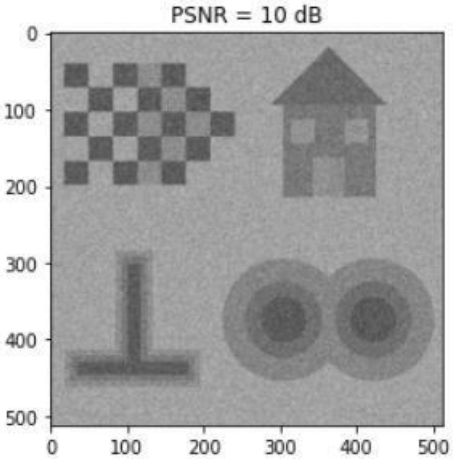
Με την βοήθεια της συνάρτησης numpy.random.normal(), προσθέτουμε θόρυβο με PSNR=20dB και 10dB, όπου η τυπική απόκλιση του λευκού θορύβου προκύπτει από τον τύπο

$$\mathrm{P}SNR = 20\log_{10}(\frac{I_{max} - I_{\min}}{\sigma_n})$$

Όπου $I_{\max}$, I_{min} η μέγιστη και ελάχιστη τιμή των pixels της εικόνας.

Μετά την προσθήκη θορύβου προκύπτουν οι εικόνες:





1.2 Υλοποίηση Αλγορίθμου Ανίχνευσης Ακμών

Με την βοήθεια του παρακάτω κώδικα δημιουργήσαμε τις προσεγγίσεις των εξής κρουστικών αποκρίσεων:
A) Δισδιάστατή Gaussian
B) Laplacian-of-Gaussian (LoG)

```
G1D = cv2.getGaussianKernel(n,sigma) #1D gaussian filter
G2D = G1D @ G1D.T #symmetric 2D gaussian filter

def LoG(n,sigma):
  x = np.linspace(-n/2,n/2,n)
  y = np.linspace(-n/2,n/2,n)
  X,Y = np.meshgrid(x,y)

  Z = (X**2 + Y**2-2*sigma**2)/(2*np.pi*sigma**6)
  Z *= np.exp(-(X**2+Y**2)/(2*sigma**2))
  return Z
```

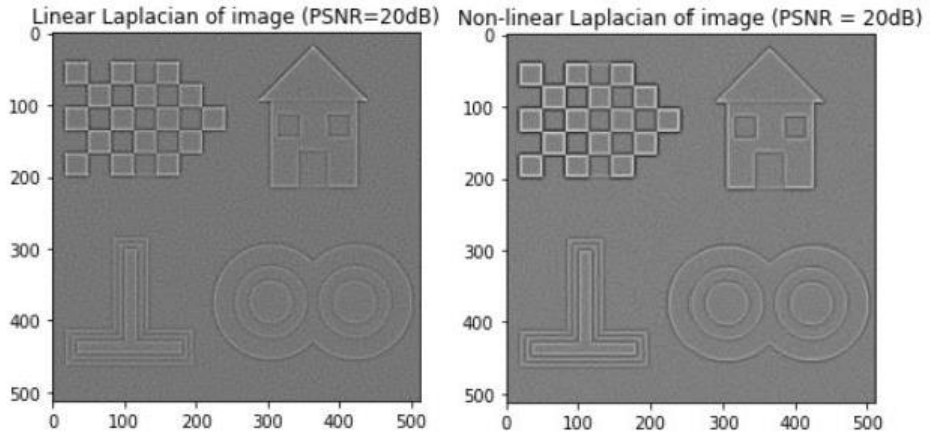
Με την βοήθεια των παραπάνω προσεγγίζουμε της Laplacian L της εξομαλυμένης εικόνας με 2 τρόπους, χρησιμοποιώντας τον παρακάτω κώδικα:

A) Γραμμική Laplacian: $L_1 = \nabla^2(G_\sigma * I) = (\nabla^2 G_\sigma) * I$

B) Μη-γραμμική Laplacian: $L_2 = (I\sigma \oplus B) + (I\sigma \ominus B) - 2I\sigma$

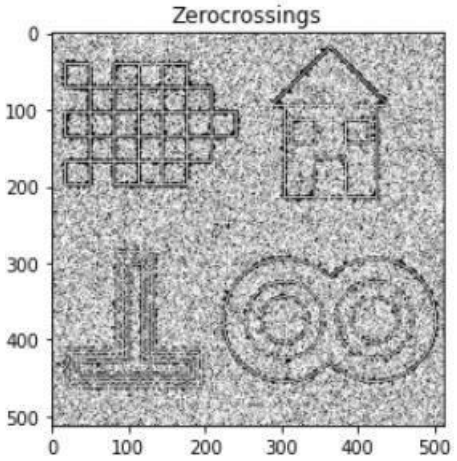
```
L1 = cv2.filter2D(image1,-1,LoG(n,sigma))

L2 =
cv2.dilate(smoothed_img,kernel)+cv2.erode(smoothed_img,kernel)-
2*smoothed_img
```

Οι εξομαλυμένες εικόνες που προκύπτουν είναι οι εξής:



Επόμενο στάδιο είναι ο υπολογισμός των σημείων μηδενισμού (zerocrossings) της Laplacian. Για να το πετύχουμε αυτό, υπολογίζουμε την Δυαδική Εικόνα Προσήμου X = (L>=0), και στην συνέχεια το περίγραμμα Y του X: Y = (X ⊕ B) − (X ⊖ B).

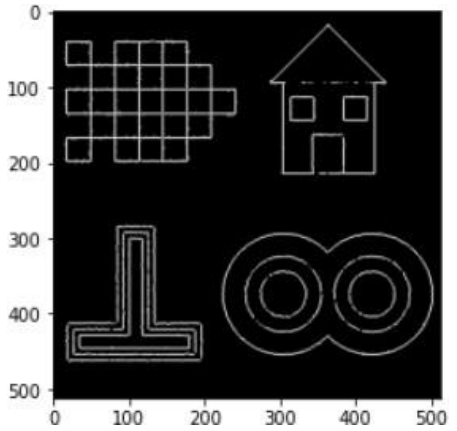
Το αποτέλεσμα του υπολογισμού zerocrossing στις παραπάνω εικόνες είναι το εξής:



Τέλος, απορρίπτουμε τα zerocrossings σε σχετικά ομαλές περιοχές, δηλαδή δεν ικανοποιούν το παρακάτω κριτήριο:

$$\mathrm{Y}\,[\mathrm{i},\mathrm{j}] = 1 \text{ και } \| \nabla \mathrm{I}\sigma[\mathrm{i},\mathrm{j}] \| > \theta\mathrm{edge} \cdot \max \mathrm{x},\mathrm{y} \| \nabla \mathrm{I}\sigma \|$$

Το αποτέλεσμα είναι οι ακμές της εικόνας (1 στα pixels ακμών,0 αλλού):



Συνοψίζοντας την παραπάνω προεργασία σε μία συνάρτηση εντοπισμού ακμών, φτιάξαμε την EdgeDetect():

```
def EdgeDetect(image:np.array,
               sigma:int,
               theta:int,
               type_of_filter:str):
  assert type_of_filter == 'linear' or type_of_filter == 'non-linear','Type should be "linear" or "non-linear"'

  #Filter Creation
  n = int(2*np.ceil(3*sigma)+1) #n: kernel size (n x n)

  gauccian_1D = cv2.getGaussianKernel(n,sigma)
  gauccian_2D = gauccian_1D @ gauccian_1D.T

  def LoG(n,sigma): #return Laplacian-of-Gauccian filter
    x = np.linspace(-n/2,n/2,n)
    y = np.linspace(-n/2,n/2,n)
    X,Y = np.meshgrid(x,y)

    Z = (X**2 + Y**2-2*sigma**2)/(2*np.pi*sigma**6)
    Z *= np.exp(-(X**2+Y**2)/(2*sigma**2))
    return Z

  log = LoG(n,sigma)

  #Laplacian L (smoothing the image)
  kernel = np.array([[0,1,0],
                     [1,1,1],
                     [0,1,0]], dtype=np.uint8)
  blur_img = cv2.filter2D(image,-1,gauccian_2D)

  #choose between L1 or L2 smoothing methods
  if type_of_filter == 'linear':
    L = cv2.filter2D(image,-1,log) #convolution of log and image
  elif type_of_filter == 'non-linear':
    L = cv2.dilate(blur_img,kernel)+cv2.erode(blur_img,kernel)-2*blur_img

  X = (L>=0)  #binary image
  X = X.astype(np.uint8)

  Y = cv2.dilate(X,kernel) - cv2.erode(X,kernel)

  #Finding the edges (threshold on zerocrossings)
  img_gradient = np.array(np.gradient(blur_img))

  #find the max values of both axis gradient
  max1 = np.absolute(img_gradient[0]).max()
  max2 = np.absolute(img_gradient[1]).max()

  #find edges of both axis gradients
  edges1 = (np.logical_and(Y==1,np.absolute(img_gradient[0])>theta*max1)).astype(int)
  edges2 = (np.logical_and(Y==1,np.absolute(img_gradient[1])>theta*max2)).astype(int)

  result = np.logical_or(edges1,edges2).astype(int)

  return result
```

1.3 Αξιολόγηση Αποτελεσμάτων Ανίχνευσης Ακμών

Για να αξιολογήσουμε την ποιότητα εντοπισμού ακμών στις εικόνες με θόρυβο, θα πρέπει πρώτα να εντοπίσουμε τις ακμές στην αρχική φωτογραφία, χρησιμοποιώντας τον παρακάτω τελεστή:

$$M = (I \oplus B) - (I \ominus B)$$

Το μορφολογικό φίλτρο dilate εξομαλύνει την εικόνα 'γεμίζοντας' χάσματα και κενά, με αποτέλεσμα η εικόνα να 'διογκώνεται'. Αντίθετα το erode εξομαλύνει την εικόνα αφαιρώντας pixels που έχουν μεταξύ τους κενά, με αποτέλεσμα η εικόνα να 'συρρικνώνεται'.

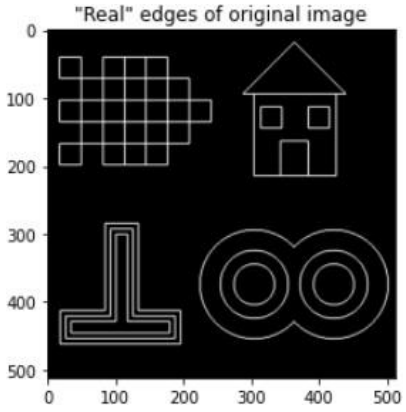
Η αφαίρεση των αποτελεσμάτων των 2 αυτών φίλτρων (της διογκωμένης και συρρικνωμένης εικόνας) μας αφήνει τα pixels της διογκωμένης εικόνας που δεν μοιράζεται με την συρρικνωμένη, το οποίο είναι καλή προσέγγιση των ακμών της πραγματικής εικόνας.

Τέλος εφαρμόζοντας ένα threshold στα εναπομένοντα pixels, καταλήγουμε σε μία δυαδική εικόνα που προσεγγίζει πολύ καλά τις πραγματικές ακμές.

Την υλοποίηση της παραπάνω διαδικασίας την πετυχαίνουμε με την συνάρτηση real_edges() :

```
def real_edges_detect(image,theta_real):
  kernel = np.array([[0,1,0],
                     [1,1,1],
                     [0,1,0]], dtype=np.uint8)

  M = cv2.dilate(image,kernel) - cv2.erode(image,kernel)
  T = M>theta_real
  T = T.astype(np.uint8)
  return T
```

Η οποία μας επιστρέφει την παρακάτω εικόνα ακμών, για threshold = 0.1:



Για τον καθορισμό της ποιότητας, υπολογίζουμε την ποσότητα

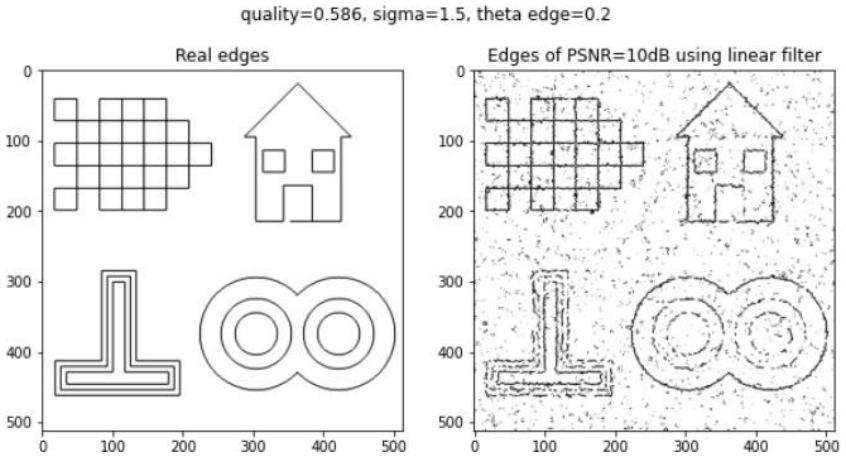
$$C = [Pr(D|T) + Pr(T|D)] /2$$

Pr(D|T):= ποσοστό ανιχνευθεισών ακμών που είναι αληθινές (precision)
Pr(T|D):= ποσοστό αληθινών ακμών που ανιχνεύθησαν (reacall)

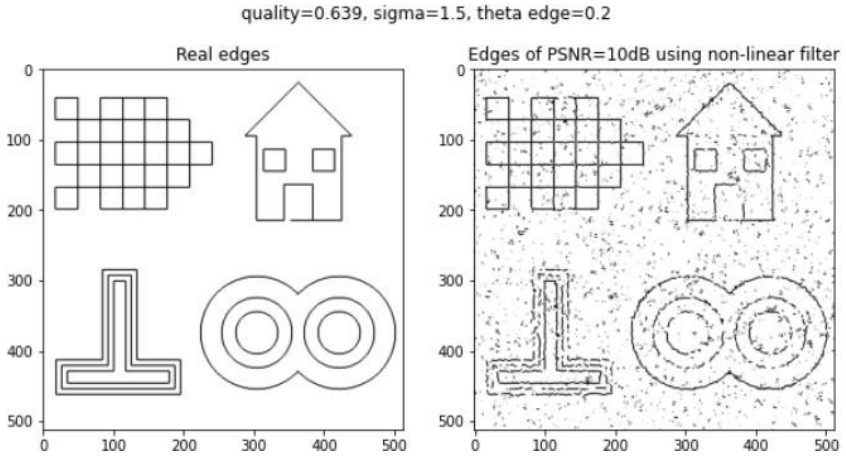
Η συνάρτηση υπολογισμού:

```
def detection_quality(real_edges,
                      my_edges):
  precision = (real_edges & my_edges).sum()/real_edges.sum()
  recall = (real_edges & my_edges).sum()/my_edges.sum()

  return (precision+recall)/2
```

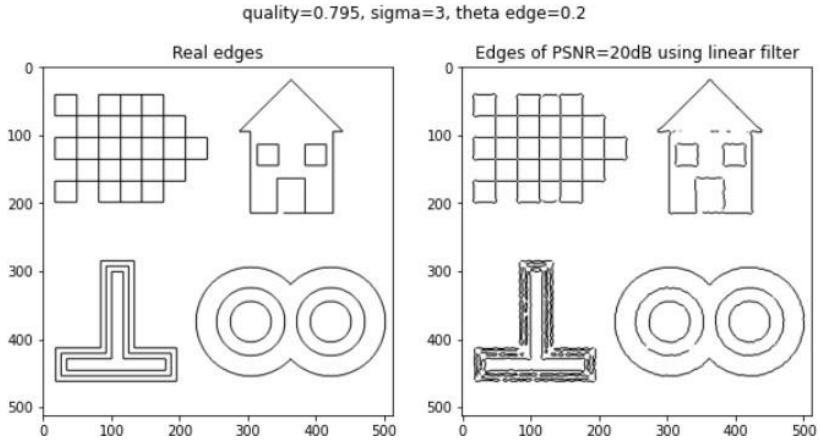
Παρακάτω μερικά παραδείγματα καθορισμού της ποιότητας
(στο Notebook περιέχονται περισσότερα)



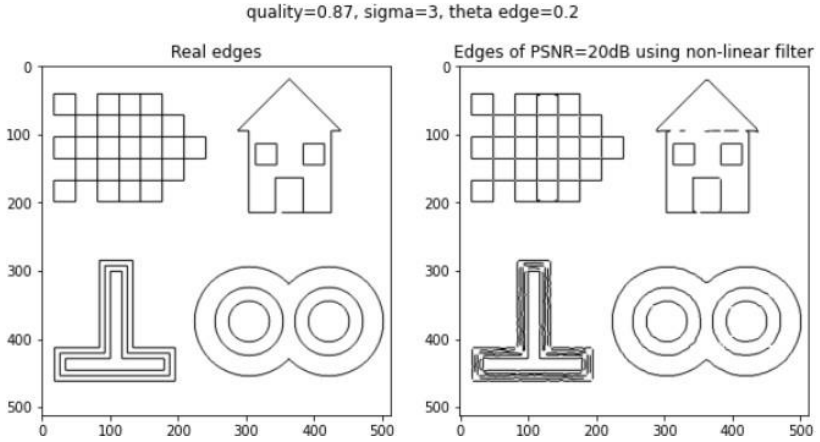
(εικόνα 1)



(εικόνα 2)



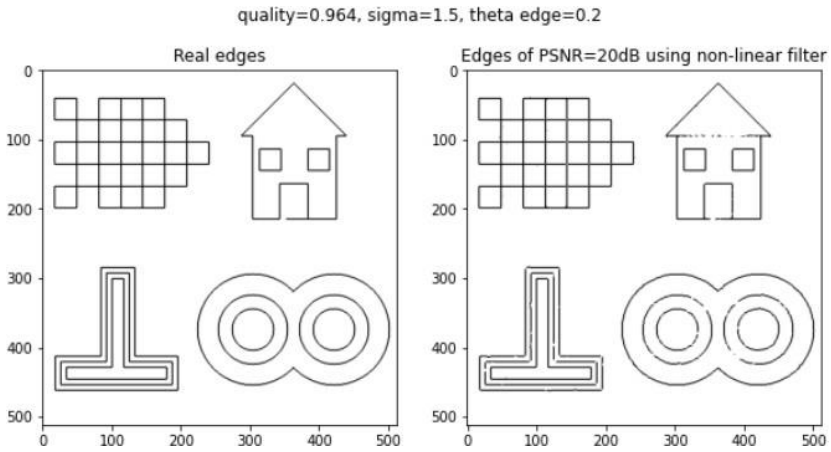
(εικόνα 3)



(εικόνα 4)

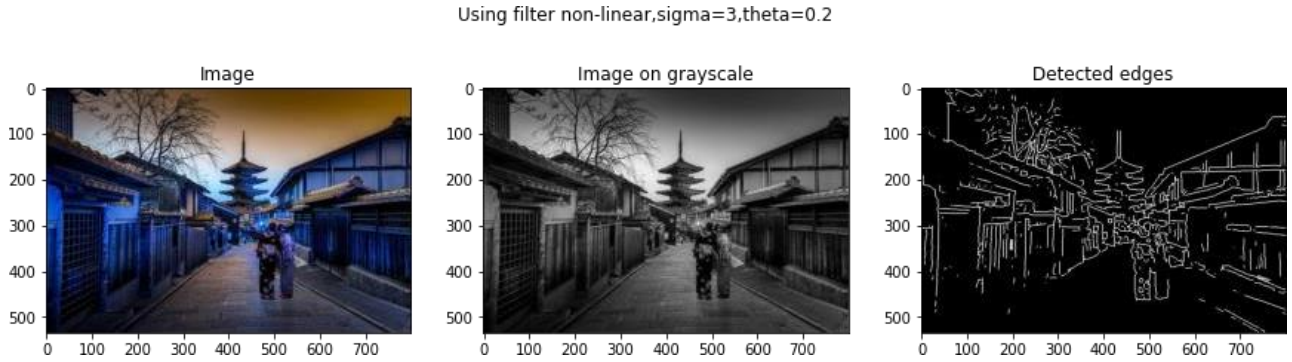
Παρατηρούμε ότι η χρήση σ=3 μειώνει σημαντικά τον θόρυβο της τελικής εικόνας, συνεπώς οδηγεί με υψηλή ποιότητας ανίχνευση ακμών, όπως βλέπουμε στις εικόνες 3,4 σε σύγκριση με τις εικόνες 1,2. Επίσης βλέπουμε ότι η χρήση της μη-γραμμικής προσέγγισης της Laplacian δίνει καλύτερα αποτελέσματα από την γραμμική, όπως φαίνεται από τις εικόνες 2,4 σε σύγκριση με τις εικόνες 1,3.

Το καλύτερο αποτέλεσμά μας είναι το παρακάτω:

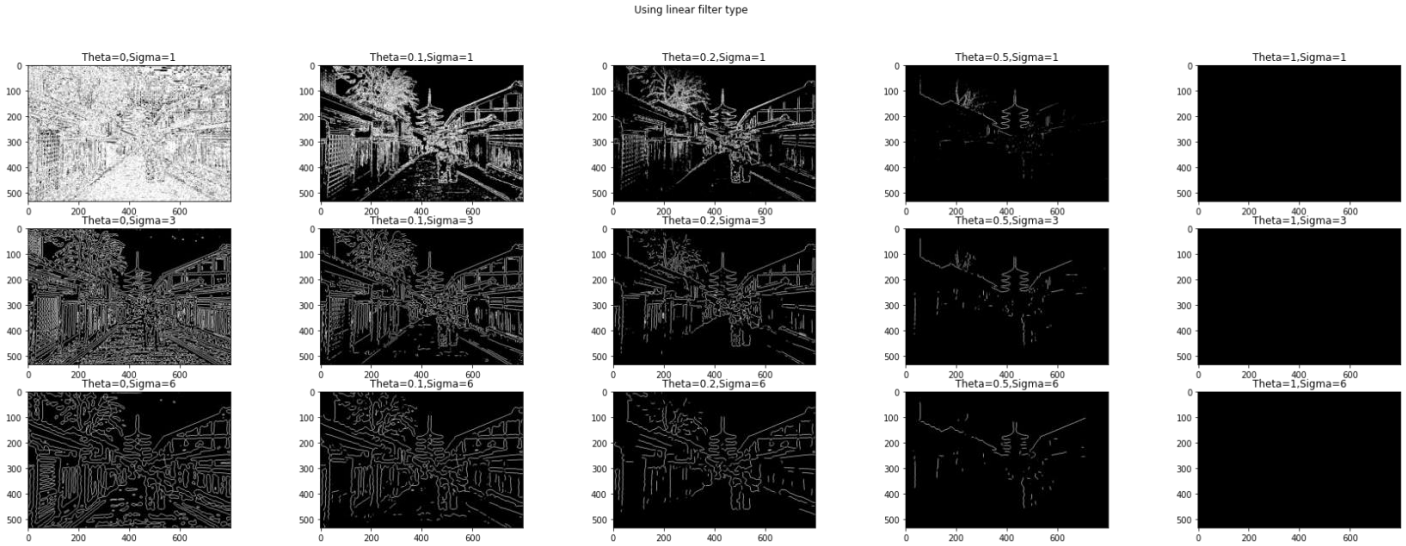


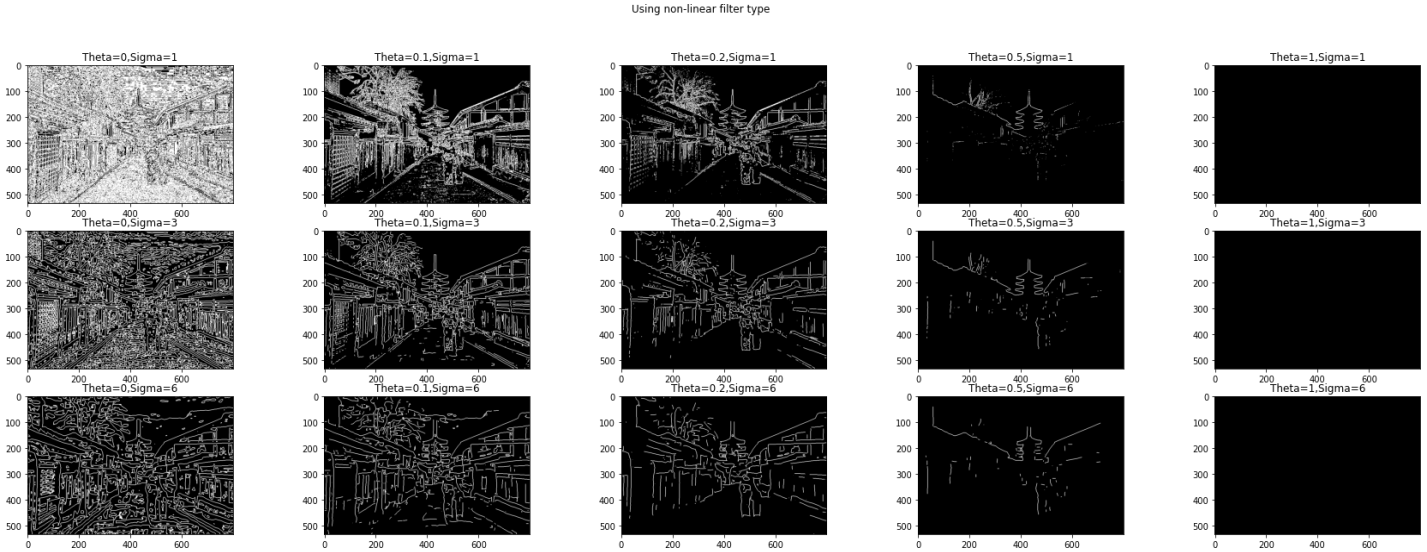
1.4 Εφαρμογή σε Πραγματικές Εικόνες

Θα εφαρμόσουμε την συνάρτηση EdgeDetect() στην ασπρόμαυρη εκδοχή της εικόνα ‘kyoto_edges.jpg’.
Τα αποτελέσματα είναι τα παρακάτω:



Τώρα θα πειραματιστούμε με για διάφορα τα σ, thresholds και φίλτρα:



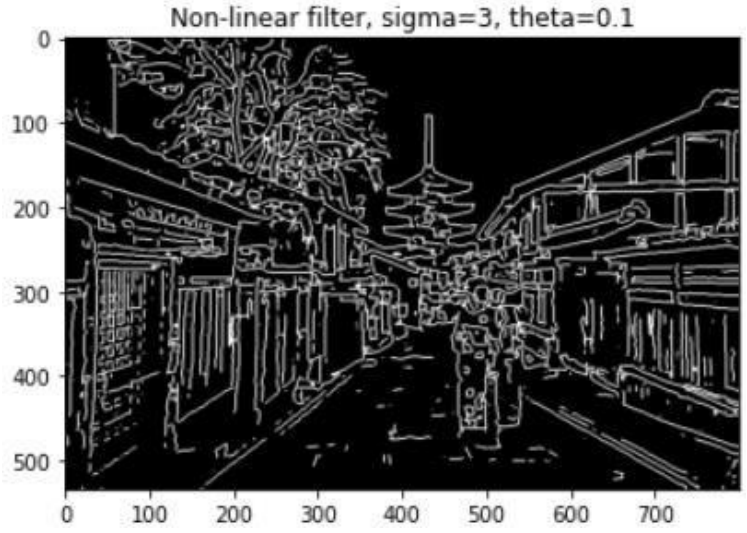


Κάνοντας μια ποιοτική ανάλυση των παραπάνω, παρατηρούμε ότι η αύξηση του threshold οδηγεί σε λιγότερη χρήσιμη πληροφορία, καθώς το πλήθος των επιλεγμένων pixels μειώνεται, ενώ για πολύ μικρά thresholds απομένει πολύ θόρυβος στη εικόνα.

Επίσης για μικρά σ (σ < 3) το παράθυρο για το εφαρμοζόμενο φίλτρο είναι αρκετά μικρό, συνεπώς στην εξομαλυμένη εικόνα απομένει ‘άχρηστη’ πληροφορία που την καθιστά δυσανάγνωστη.
Από την άλλη, τα μεγάλα σ (σ > 3) οδηγούν σε μεγάλα παράθυρα τα οποία αγνοούν χρήσιμη πληροφορία. Από τα παραπάνω καταλήγουμε ότι σ=3 δίνει το κατάλληλο παράθυρο.

Τέλος, η χρήση της μη-γραμμικής προσέγγισης της Laplacian επιτυγχάνει καλύτερα αποτελέσματα από την γραμμική, καθώς δίνει καλύτερη εξομάλυνση της εικόνας στα αρχικά στάδια της επεξεργασίας.

Το καλύτερο αποτέλεσμα μας είναι το παρακάτω:



Όπου χρησιμοποιούμε ‘μέσες’ τιμές για σ και threshold, και μπορούμε να διακρίνουμε πολλές από τις ακμές δίχως να υπάρχει πολύ θόρυβος.

Μέρος 2ο :

2.1 Ανίχνευση Γωνιών (Harris-Stephens Method)

Για να ανιχνεύσουμε τις γωνίες στις φωτογραφίες μας, θα πρέπει να υπολογίζουμε τις ιδιοτιμές λ1 και λ2 (μία για κάθε κατεύθυνση) της κάθε εικόνας και συγκρίνουμε τις τιμές μεταξύ τους.

Για τον υπολογισμό, βρίσκουμε πρώτα τα J_1, J_2, J_3 του τανυστή J:

$$J_1(x,y) = G_\rho * \left(\frac{\partial I_\sigma}{\partial x} \cdot \frac{\partial I_\sigma}{\partial x} \right)(x,y)$$
$$J_2(x,y) = G_\rho * \left(\frac{\partial I_\sigma}{\partial x} \cdot \frac{\partial I_\sigma}{\partial y} \right)(x,y)$$
$$J_3(x,y) = G_\rho * \left(\frac{\partial I_\sigma}{\partial y} \cdot \frac{\partial I_\sigma}{\partial y} \right)(x,y)$$

με χρήση του παρακάτω κώδικα:

```
Is= cv2.filter2D(image,-1,gs_2D)
gradient = np.gradient(Is)

#calculating J1,J2,J3 of J tensor
J1 = cv2.filter2D((gradient[0]*gradient[0]),-1,gp_2D)
J2 = cv2.filter2D((gradient[0]*gradient[1]),-1,gp_2D)
J3 = cv2.filter2D((gradient[1]*gradient[1]),-1,gp_2D)
```

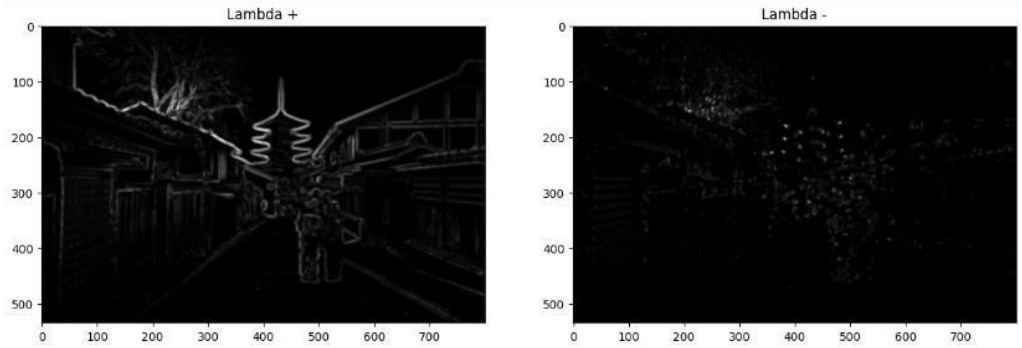
Έπειτα υπολογίζουμε τα λ1, λ2 σύμφωνα με την σχέση:

$$\lambda_\pm(x,y) = \frac{1}{2}\left(J_1 + J_3 \pm \sqrt{(J_1 - J_3)^2 + 4J_2^2} \right)$$

με την βοήθεια του κώδικα:

```
lambda_plus = (1/2)*(J1+J3+np.sqrt( (J1-J3)**2 + 4*J2**2 ))
lambda_minus = (1/2)*(J1+J3-np.sqrt( (J1-J3)**2 + 4*J2**2 ))
```

Οπτικοποιώντας τα λ1, λ2 για την kyoto_edges.png προκύπτουν τα εξής:



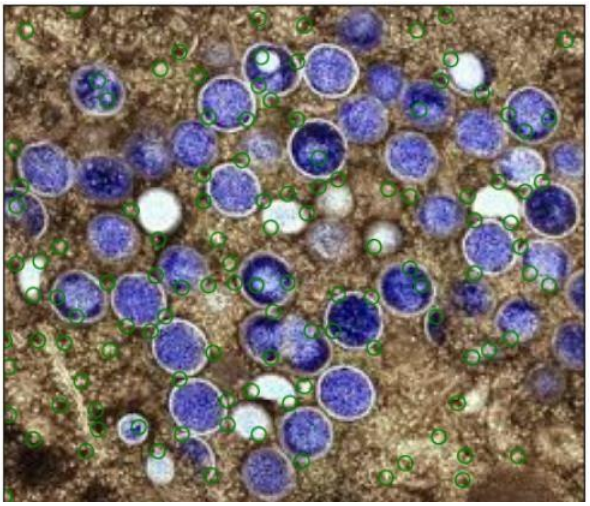
Από θεωρία γνωρίζουμε ότι τα σημεία όπου τα λ1 και λ2 έχουν υψηλές και όμοιες τιμές είναι σημείων γωνιών, ενώ οποιασδήποτε άλλος συνδυασμός των λ1 και λ2 είναι για ακμές ή αδιάφορες περιοχές (flat regions).

Παρατηρούμε ότι οι ιδιοτιμές λ1 (λ+) αντιστοιχούν τόσο σε ακμές όσο και σε γωνίες, ενώ οι λ2 (λ-) μονάχα σε γωνίες. Αυτό προκύπτει από την θεωρία, καθώς στις ακμές η μία ιδιοτιμή είναι μεγαλύτερη της άλλης (συνεπώς είναι θετική και ανήκει στα λ+) ενώ μονάχα στα γωνιακά σημεία τα λ+ και λ- έχουν την ίδια τιμή (όπως φαίνεται από τις δύο εικόνες, όπου υπάρχουν κάποια κοινά σημεία).

Υπολογίζοντας το κριτήριο γωνιότητας R:

$$R(x,y) = \lambda_-\lambda_+ - k \cdot (\lambda_- + \lambda_+)^2$$

και εφαρμόζοντας τις συνθήκες Σ1 και Σ2 για τον εντοπισμό των pixels-γωνιών, καταλήγουμε ,για τις τρεις εικόνες, στα σημεία:



Τα παραπάνω αποτελέσματα προκύπτουν για σ=1, p=1.5, k=0.1, theta_corn=0.05.

Παρατηρούμε ότι βρίσκει αρκετές γωνίες, ενώ δυσκολεύεται μονάχα στην cells.png, όπου λόγω της ιδιομορφίας των κυττάρων δεν υπάρχουν πολλές γωνίες.

Παρακάτω η συνάρτηση harris_stephens_detector() που μας δίνει τα γωνιακά σημεία σύμφωνα με την παραπάνω μέθοδο:

```
def harris_stephens_detector(s,p,image,k):
  #calculate sigma gauccians
  n_s = int(2*np.ceil(3*s)+1)
  gs_1D = cv2.getGaussianKernel(n_s,s)
  gs_2D = gs_1D @ gs_1D.T

  #calculate p gauccians
  n_p = int(2*np.ceil(3*p)+1)
  gp_1D = cv2.getGaussianKernel(n_p,p)
  gp_2D = gp_1D @ gp_1D.T

  Is= cv2.filter2D(image,-1,gs_2D)
  gradient = np.gradient(Is)

  #calculating J1,J2,J3 of J tensor
  J1 = cv2.filter2D((gradient[0]*gradient[0]),-1,gp_2D)
  J2 = cv2.filter2D((gradient[0]*gradient[1]),-1,gp_2D)
  J3 = cv2.filter2D((gradient[1]*gradient[1]),-1,gp_2D)

  #calculating the eigenvalues of J tensor
  lambda_plus = (1/2)*(J1+J3+np.sqrt( (J1-J3)**2 + 4*J2**2 ))
  lambda_minus = (1/2)*(J1+J3-np.sqrt( (J1-J3)**2 + 4*J2**2 ))

  #calculating the cornerness criterion
  R = lambda_minus*lambda_plus - k*(lambda_minus + lambda_plus)**2

  #condition 1: pixels of R that are the max in a window determned by sigma
  from cv23_lab1_part2_utils import disk_strel
  import cv23_lab1_part2_utils as utils

  ns = int(np.ceil(3*s)*2+1)
  B_sq = disk_strel(ns)
  Cond1 = ( R == cv2.dilate(R,B_sq))

  #condition 2: pixels (x,y) where R(x,y)>theta_corn*Rmax
  theta_corn = 0.05
  Rmax = R.max()
  Cond2 = ( R >theta_corn*Rmax )
  corners = np.logical_and(Cond1,Cond2)
  corners = corners.astype(int)
  rows,cols = corners.shape
  points = []
  for y in range(rows):
    for x in range(cols):
      if corners[y][x]>0:
          points.append([x,y,s])

  return corners,np.array(points)
```

2.2 Πολυκλιμακωτή Ανίχνευση Γωνιών (Harris-Laplacian)

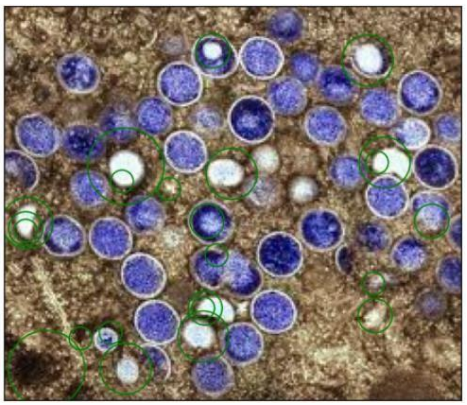
Για να πετύχουμε καλύτερα αποτελέσματα στην ανίχνευση γωνιών, μπορούμε να δοκιμάσουμε την μέθοδο που περιγράψαμε στο 2.1 (Harris-Stephens Method) για πολλές τιμές σ,p και να επιλέξουμε μεταξύ των αποτελεσμάτων τους. Για αυτό υπολογίζουμε N διαφορετικά σ,p, τα οποία προκύπτουν από τα αρχικά επί κάποιο παράγοντα κλιμάκωσης s, και εφαρμόζουμε την harris_ stephens_detector() για κάθε ζεύγος σ,p.
(Η διαδικασία αυτή υλοποιείται από την συνάρτηση find_all_corners())

Για την επιλογή των κατάλληλων σημείων ,αρχικά υπολογίζουμε την κανονικοποιημένη LoG (Laplacian-of-Gaussian) για κάθε σ όπως περιγράφεται από:

$$|LoG(\mathbf{x}, \sigma_i)| = \sigma_i^2 |L_{xx}(\mathbf{x}, \sigma_i) + L_{yy}(\mathbf{x}, \sigma_i)|, \quad i = 0, ..., N-1$$

και από τα σημεία που προκύπτουν για κάθε σ_i επιλέγουμε αυτά που η $LoG(x, y, \sigma_i)$ είναι μεγαλύτερη από την LoG(x,y) για σ_{i-1}, σ_{i+1} (είναι δηλαδή τοπικό μέγιστο).
(Η διαδικασία αυτή υλοποιείται από τις συναρτήσεις normalize_LoG(), find_all_LoG() και harris_laplace_detector()).

Εφαρμόζοντας την διαδικασία στις εικόνες, προκύπτουν:



Τα παραπάνω αποτελέσματα προέκυψαν για σ=2, p=2.5, s=1.5 και N=4.

Παρατηρούμε ότι στις φωτογραφίες όπου οι γωνίες δεν είναι τόσο ευδιάκρατικες (π.χ cells, up) η Harris-Laplacian βρίσκει περισσότερα σημεία από την Harris-Stephens.

Παρακάτω η συνάρτηση harris_laplace_detector() που μας δίνει τα παραπάνω σημεία :

```
def harris_laplace_detector(image,s0,p0,scale,N):
  all_logs,all_sigmas =
find_all_LoG(image=image,s0=s0,scale=scale,N=N)
  all_corners =
find_all_corners(image=image,s0=s0,p0=p0,scale=s,N=N,k=0.1)

  result = [] #contains the coordinates of final edges
  for index in range(0,N):
    corner_coord = all_corners[index]
    cur_log = all_logs[index]
    if(index>0 and index<N-1):
      prev_log = all_logs[index-1]
      next_log = all_logs[index+1]
    elif(index==0):
      prev_log = all_logs[index+1]
      next_log = all_logs[index+1]
    elif(index==N-1):
      prev_log = all_logs[index-1]
      next_log = all_logs[index-1]

    for p in corner_coord:
      x = int(p[0])
      y = int(p[1])
      if (cur_log[y][x]>prev_log[y][x]) and
(cur_log[y][x]>next_log[y][x]):
        result.append([x,y,all_sigmas[index]])

  return np.array(result)
```

2.3 Ανίχνευση Blobs

Ως 'blobs' ορίζονται περιοχές με ομοιογένεια που διαφέρουν σημαντικά από την γειτονιά τους. Για την ανίχνευσή τους, υπολογίζουμε τον πίνακα Hessian:

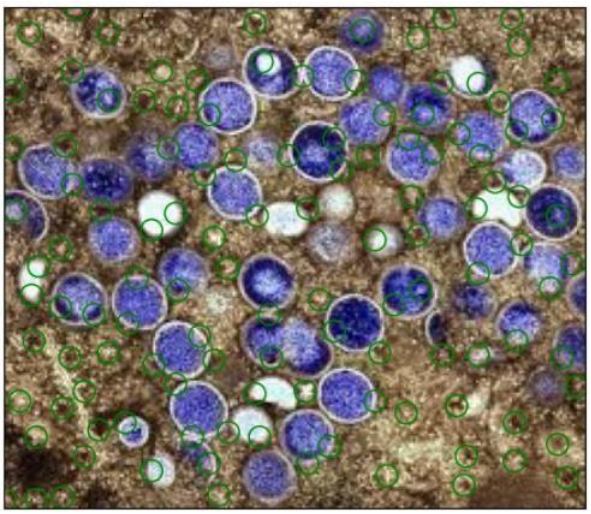
$$H(x,y) = \begin{bmatrix} L_{xx}(x,y,\sigma) & L_{xy}(x,y,\sigma) \\ L_{xy}(x,y,\sigma) & L_{yy}(x,y,\sigma) \end{bmatrix}$$

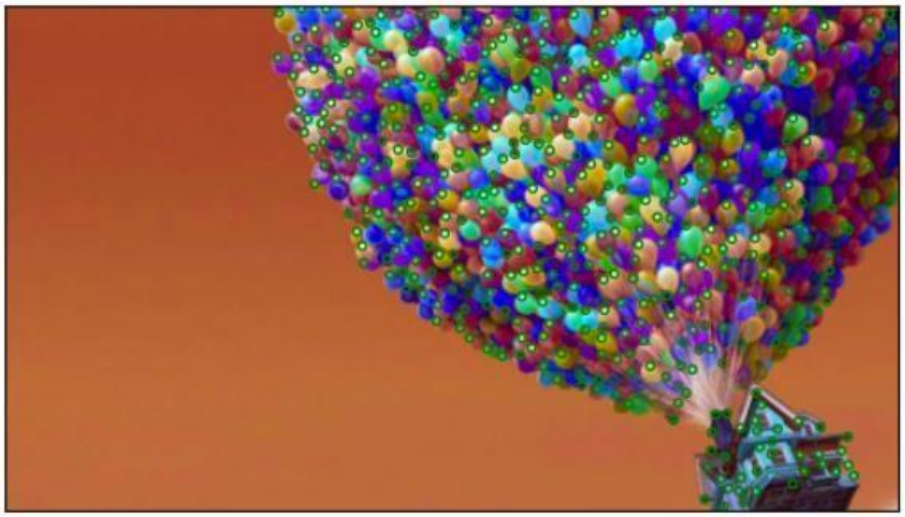
Ως κριτήριο γωνιότητας παίρνουμε το

$$R = \det(H) = Lxx * Lyy - (Lxy)^2$$

και εφαρμόζουμε τα κριτήρια Σ1 και Σ2 που είδαμε στο 2.1

Εφαρμόζοντας τα παραπάνω στις εικόνες, έχουμε τα αποτελέσματα:





Τα παραπάνω προκύπτουν για σ=1.5 και theta_corn=0.2.

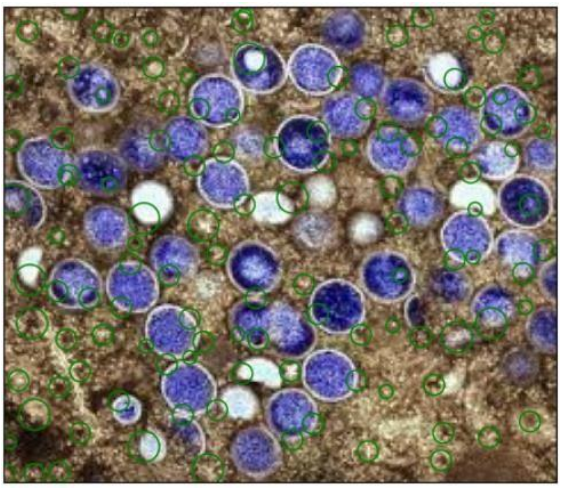
Παρατηρούμε ότι η εύρεση των γωνιών εδώ γίνεται πολύ καλύτερα.

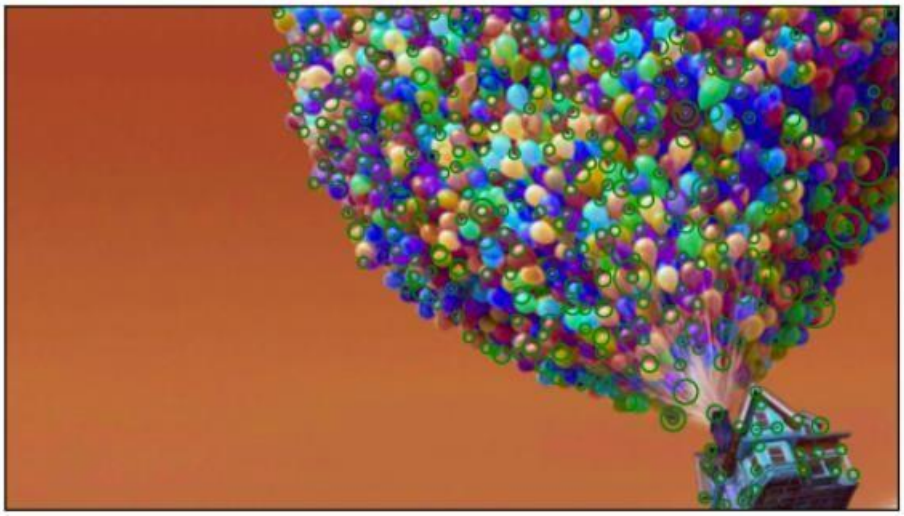
Παρακάτω η συνάρτηση hessian_detector() που μας δίνει τα παραπάνω σημεία:

```
def hessian_detector(image,sigma,theta_corn):
  n = int(2*np.ceil(3*sigma)+1)
  gauccian_1D = cv2.getGaussianKernel(n,s)
  gauccian_2D = gauccian_1D @ gauccian_1D.T
  Is = cv2.filter2D(image,-1,gauccian_2D) #blurred image

  #Calculate gradients
  Lx = np.gradient(Is)[0]
  Ly = np.gradient(Is)[1]
  Lxx = np.gradient(Lx)[0]
  Lyy = np.gradient(Ly)[1]
  Lxy = np.gradient(Lx)[1]

  #H = [[Lxx,Lxy],[Lxy,Lyy]] Hessian Matrix
  H =
np.concatenate([np.concatenate([Lxx,Lyy],axis=1),np.concatenate([Lxy,Lyy],axis=1)],axis=0)
  R = Lxx*Lyy - (Lxy)**2
  Rmax = R.max()

  #condition 1: pixels of R that are the max in a window determned by sigma
  from cv23_lab1_part2_utils import disk_strel
  import cv23_lab1_part2_utils as utils

  ns = int(np.ceil(3*sigma)*2+1)
  B_sq = disk_strel(ns)
  Cond1 = ( R == cv2.dilate(R,B_sq))

  #condition 2: pixels (x,y) where R(x,y)>theta_corn*Rmax
  theta_corn = 0.05
  Rmax = R.max()
  Cond2 = ( R >theta_corn*Rmax )

  corners = np.logical_and(Cond1,Cond2)
  h,w = corners.shape
  points = []
  for y in range(h):       #pick row
    for x in range(w):     #pick column
      if corners[y][x]>0:
        points.append([x,y,sigma])
  return corners,points
```

2.4 Πολυκλιμακωτή Ανίχνευση Blobs

Ακολουθώντας την ίδια διαδικασία με το 2.2, βρίσκουμε τα blobs για διάφορες κλίμακες και εφαρμόζοντας το κριτήριου του τοπικού μεγίστου στα διαδοχικά LoGs, προκύπτουν τα παρακάτω αποτελέσματα:





Τα παραπάνω σημεία προκύπτουν για σ=1,s=1.5,N=4 και theta_corn=0.2

Παρατηρούμε ότι με την πολυκλιμακωτή ανίχνευση εντοπίζουμε περισσότερες γωνίες από ότι πριν.

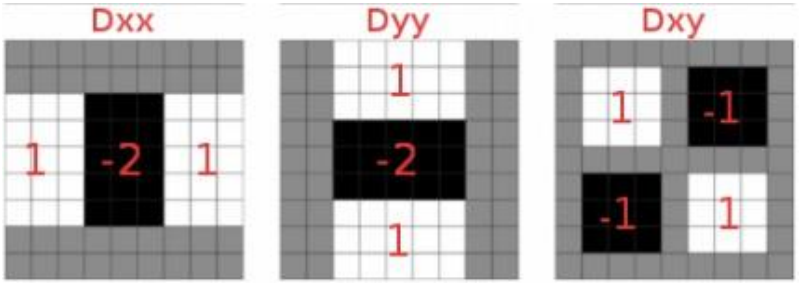
Παρακάτω η συνάρτηση hessian_laplace_detector() που μας επιστρέφει τα παραπάνω σημεία:

```
Def hessian_laplace_detector(image,s0,scale,N,theta):
  all_logs,all_sigmas = find_all_LoG(image=image,s0=s0,scale=scale,N=N)
  all_corners =
find_all_blobs(image=image,sigma=s0,scale=scale,N=N,theta=theta)
  result = [] #contains the coordinates of final edges
  for index in range(0,N):
    corner_coord = all_corners[index]
    cur_log = all_logs[index]
    if(index>0 and index<N-1):
      prev_log = all_logs[index-1]
      next_log = all_logs[index+1]
    elif(index==0):
      prev_log = all_logs[index+1]
      next_log = all_logs[index+1]
    elif(index==N-1):
      prev_log = all_logs[index-1]
      next_log = all_logs[index-1]

    for p in corner_coord:
      x = int(p[0])
      y = int(p[1])
      if (cur_log[y][x]>prev_log[y][x]) and (cur_log[y][x]>next_log[y][x]):
        result.append([x,y,all_sigmas[index]])

  return np.array(result)
```

2.5 Επιτάχυνση με την χρήση Box Filters και Ολοκληρωτικών Εικόνων

Η πολυκλιμακωτή ανίχνευση blobs απαιτεί μεγάλο υπολογιστικό κόστος, εξαιτίας των πολλαπλών συνελίξεων για διάφορα σ.

Για αυτό το λόγο προσεγγίζουμε τους συντελεστές Lxx,Lyy,Lxy της Hessian μήτρας μέσω των box filters Dxx, Dyy, Dxy και των ολοκληρωτικών εικόνων.



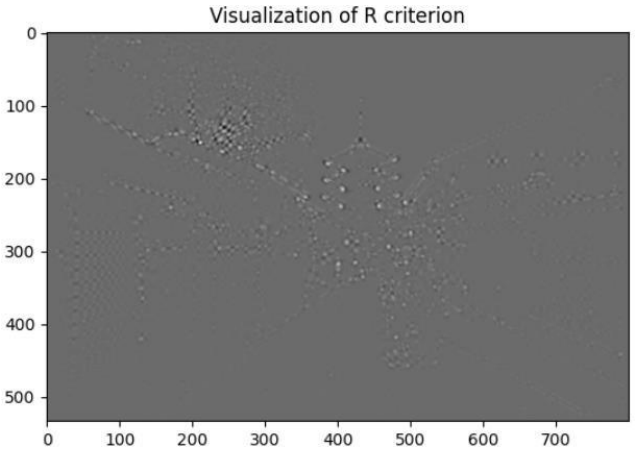
Με τον παρακάτω κώδικα υλοποιούμε τα box filters:

```
    cumsum = np.cumsum(image, 0, dtype = "float64")
    cumsum = np.cumsum(cumsum, 1, dtype = "float64")
    n = 2*np.ceil(3*sigma)+1
    dim1 = int(4*np.floor(n/6)+1)
    dim2 = int(2*np.floor(n/6)+1)
    rect_box = np.asarray([[1]*dim2]*dim1)
    trans_rect_box = np.transpose(rect_box)
    squares = np.asarray([[1]*dim2]*dim2)

    filter_n = 3*dim2
    image_y_size = int(image.shape[0])
    image_x_size = int(image.shape[1])

    Lxx = np.asarray([[0]*image_x_size]*image_y_size);
    Lyy = np.asarray([[0]*image_x_size]*image_y_size);
    Lxy = np.asarray([[0]*image_x_size]*image_y_size);

    def calc_box_sum(cumsum, y, x, dimy, dimx):
        return cumsum[y - dimy//2][x - dimx//2] + cumsum[y + dimy//2][x + dimx//2] -
cumsum[y - dimy//2][x + dimx//2] - cumsum[y + dimy//2][x - dimx//2]

    for i in range(filter_n//2, image_y_size - filter_n//2):
        for j in range(filter_n//2, image_x_size - filter_n//2):
            Lxx[i, j] = calc_box_sum(cumsum, i, j - dim2, dim1, dim2) -
2*calc_box_sum(cumsum, i, j, dim1, dim2) + calc_box_sum(cumsum, i, j + dim2, dim1,
dim2)
            Lyy[i][j] = calc_box_sum(cumsum, i - dim2, j, dim2, dim1) -
2*calc_box_sum(cumsum, i, j, dim2, dim1) + calc_box_sum(cumsum, i + dim2, j, dim2,
dim1)
            Lxy[i][j] = calc_box_sum(cumsum, i - dim2//2 - 1, j - dim2//2 - 1, dim2,
dim2) - calc_box_sum(cumsum, i - dim2//2 - 1, j + dim2//2 + 1, dim2, dim2) \
                        - calc_box_sum(cumsum, i + dim2//2 + 1, j - dim2//2 - 1,
dim2, dim2) + calc_box_sum(cumsum, i + dim2//2 + 1, j + dim2//2 + 1, dim2, dim2)
```

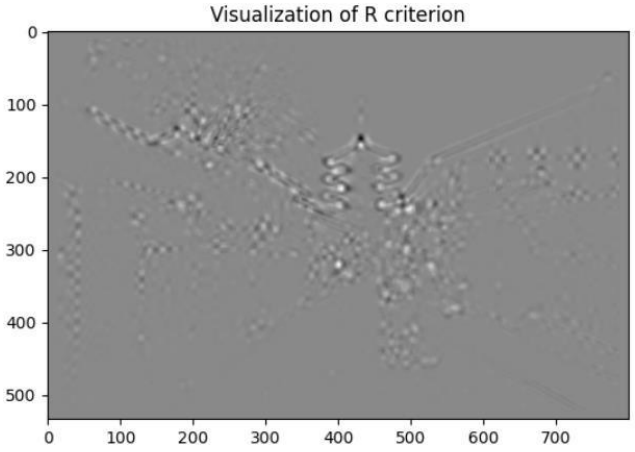
Υπολογίζουμε το κριτήριο γωνιότητας:

$$R(x,y) = L_{xx}(x,y)L_{yy}(x,y) - (0.9L_{xy}(x,y))^2$$

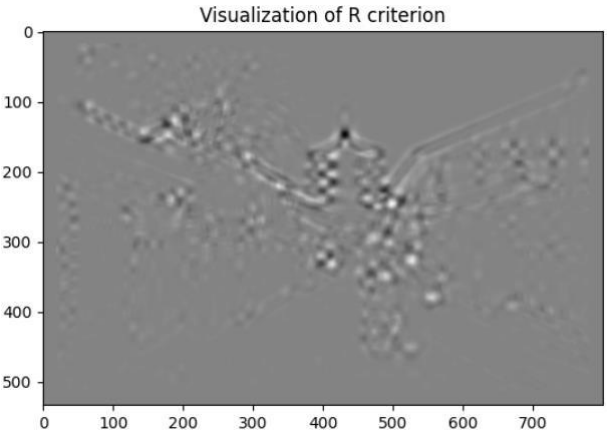
Παρακάτω οι οπτικοποιήσεις του R για σίγμα 2,4,6 και 8:



sigma=2



sigma=4

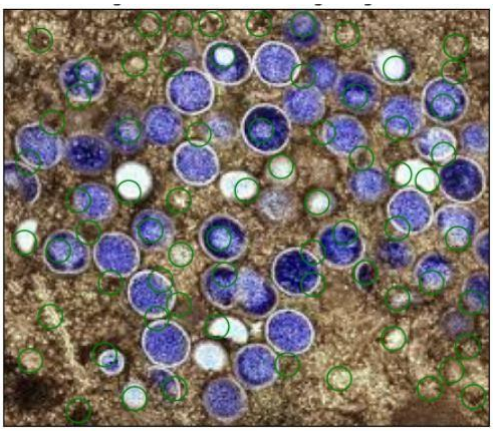


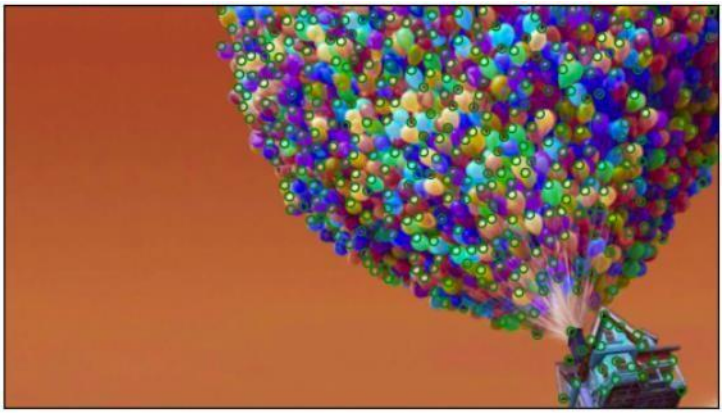
sigma=6

sigma=8

Παρατηρούμε ότι με την αύξηση του σ, η εικόνα «θολώνει» περισσότερο, συνεπώς ο εντοπισμός γωνιακών σημείων δεν είναι τόσο εύκολος.

Παρακάτω οι γωνίες που προέκυψαν για σ=2 στις φωτογραφίες:





Παρατηρούμε ότι η παραπάνω βρίσκει πολλές γωνίες, παρά το γεγονός ότι είναι μια προσέγγιση της hessian.

Παρακάτω η συνάρτηση Hessian_aprox() που βρίσκει τα παραπάνω σημεία:

```
def Hessian_aprox(image, sigma, visualize_R = False):
    # Calculate cummulative sum of image
    cumsum = np.cumsum(image, 0, dtype = "float64")
    cumsum = np.cumsum(cumsum, 1, dtype = "float64")
    n = 2*np.ceil(3*sigma)+1
    dim1 = int(4*np.floor(n/6)+1)
    dim2 = int(2*np.floor(n/6)+1)
    rect_box = np.asarray([[1]*dim2]*dim1)
    trans_rect_box = np.transpose(rect_box)
    squares = np.asarray([[1]*dim2]*dim2)
    filter_n = 3*dim2
    image_y_size = int(image.shape[0])
    image_x_size = int(image.shape[1])
    # Calculate Lxx
    Lxx = np.asarray([[0]*image_x_size]*image_y_size);
    Lyy = np.asarray([[0]*image_x_size]*image_y_size);
    Lxy = np.asarray([[0]*image_x_size]*image_y_size);

    def calc_box_sum(cumsum, y, x, dimy, dimx):
        return cumsum[y - dimy//2][x - dimx//2] + cumsum[y + dimy//2][x + dimx//2] -
cumsum[y - dimy//2][x + dimx//2] - cumsum[y + dimy//2][x - dimx//2]

    for i in range(filter_n//2, image_y_size - filter_n//2):
        for j in range(filter_n//2, image_x_size - filter_n//2):
            Lxx[i, j] = calc_box_sum(cumsum, i, j - dim2, dim1, dim2) -
2*calc_box_sum(cumsum, i, j, dim1, dim2) + calc_box_sum(cumsum, i, j + dim2, dim1, dim2)
            Lyy[i][j] = calc_box_sum(cumsum, i - dim2, j, dim2, dim1) -
2*calc_box_sum(cumsum, i, j, dim2, dim1) + calc_box_sum(cumsum, i + dim2, j, dim2, dim1)
            Lxy[i][j] = calc_box_sum(cumsum, i - dim2//2 - 1, j - dim2//2 - 1, dim2,
dim2) - calc_box_sum(cumsum, i - dim2//2 - 1, j + dim2//2 + 1, dim2, dim2) \
                        - calc_box_sum(cumsum, i + dim2//2 + 1, j - dim2//2 - 1, dim2,
dim2) + calc_box_sum(cumsum, i + dim2//2 + 1, j + dim2//2 + 1, dim2, dim2)
    # Find interest points
    R = Lxx*Lyy - (0.9*Lxy)**2
    # Visualize the criterion
    if visualize_R:
        plt.imshow(R, cmap='gray', vmin=np.amin(R), vmax=np.amax(R))
        plt.title("Visualization of R criterion")
        plt.show()
    # Find local maxima
    B_sq = disk_strel(n)
    Cond1 = ( R==cv2.dilate(R,B_sq) )
    # Find corner condition
    Rmax = np.amax(R)
    Cond2 = R > theta_corn*Rmax
    # Find corners
    intersection = Cond1 & Cond2
    corners = []
    for i in range(0, intersection.shape[0]):
        for j in range(0, intersection.shape[1]):
            if intersection[i][j] == True:
                corners.append([j, i, sigma])

    return corners
```

Επεκτείνοντας την ιδέα των box filters σε πολυκλιμακωτή ανίχνευση, καταφέρνουμε να πετύχουμε καλύτερα αποτελέσματα:

Η συνάρτηση που μας δίνει τα παραπάνω αποτελέσματα:

```
def box_multiscale(image,sigma,scale,N):
    all_logs,all_sigmas =
find_all_LoG(image=image,s0=sigma,scale=scale,N=N)
    all_corners = find_all_box(image=image,sigma=sigma,scale=scale,N=N)
    result = []
    for index in range(0,N):
        corner_coord = all_corners[index]
        cur_log = all_logs[index]
        if(index>0 and index<N-1):
              prev_log = all_logs[index-1]
              next_log = all_logs[index+1]
        elif(index==0):
              prev_log = all_logs[index+1]
              next_log = all_logs[index+1]
        elif(index==N-1):
              prev_log = all_logs[index-1]
              next_log = all_logs[index-1]

        for p in corner_coord:
              x = int(p[0])
              y = int(p[1])
              if (cur_log[y][x]>prev_log[y][x]) and
(cur_log[y][x]>next_log[y][x]):
                result.append([x,y,all_sigmas[index]])

    return np.array(result)
```

Μέρος 3ο:

3.1 Ταίριασμα Εικόνων υπό Περιστροφή και Αλλαγής Κλίμακας

Θα εξετάσουμε την ικανότητα εύρεσης της περιστροφής και της κλίμακας με την χρήση των σημείων ενδιαφέροντος (που βρίσκουμε με τις συναρτήσεις του Μέρους 2) και τοπικών περιγραφητών (που μας δίνονται).

Χρησιμοποιώντας το παράδειγμα από το test_matching_evluation.py, υπολογίζουμε για κάθε περιγραφητή και συνάρτησή μας το μέσο σφάλμα.

Έχοντας δημιουργήσει ανώνυμες συναρτήσεις και για τις 5 συναρτήσεις μας, προκύπτουν τα παρακάτω αποτελέσματα για την χρήση των συναρτήσεων με SURF:

```
Results for Harris Detector and SURF:
Avg. Scale Error for Image 1 : 0.003316394599950531
Avg. Theta Error for Image 1 : 0.17663001260578187

Results for Hessian Detector and SURF:
Avg. Scale Error for Image 1 : 0.004502154343958246
Avg. Theta Error for Image 1 : 0.5103729873356031

Results for Multiscale Hessian Detector and SURF:
Avg. Scale Error for Image 1 : 0.1020810701620162
Avg. Theta Error for Image 1 : 7.292960134156691

Results for Multiscale Box and SURF:
Avg. Scale Error for Image 1 : 0.002110593067357041
Avg. Theta Error for Image 1 : 0.135049968328658
```

```
Detector harrisDetector multiscale and feature extraction method SURF
Avg. Scale Error for Image 1: 0.002
Avg. Theta Error for Image 1: 0.118
```

Αντίστοιχα, για τη χρήση των συναρτήσεων με HOG προκύπτουν τα παρακάτω αποτελέσματα:

```
Results for Harris Detector and HOG:
Avg. Scale Error for Image 1 : 0.13440981000010664
Avg. Theta Error for Image 1 : 14.114276263215611

Results for Harris Detector and HOG:
Avg. Scale Error for Image 1 : 0.21604820472385858
Avg. Theta Error for Image 1 : 25.659940371484133

Results for Multiscale Hessian Detector and HOG:
Avg. Scale Error for Image 1 : 0.475621136975784 95
Avg. Theta Error for Image 1 : 32.0057095693603

Results for Multiscale Box and HOG:
Avg. Scale Error for Image 1 : 0.13152315709186113
Avg. Theta Error for Image 1 : 7.266968451457085
```

```
Detector harrisDetector multiscale and feature extraction method HOG
Avg. Scale Error for Image 1: 0.122
Avg. Theta Error for Image 1: 13.672
```

Παρατηρούμε ότι ,συγκριτικά, ο τοπικός περιγραφητής SURF δίνει γενικότερα μικρότερο σφάλμα από τον HOG στις προβλέψεις του, ενώ από τις συναρτήσεις εύρεσης σημείων ενδιαφέροντος αυτή με τα καλύτερα αποτελέσματα είναι και στις δύο περιπτώσεις ο multiscale box (η πολυκλιμακωτή ανίχνευση με box filters).

Σημείωση: ο έλεγχος του harris multiscale έγινε σε διαφορετικό notebook από τα υπόλοιπα, για αυτό και διαφέρει.

3.2 Κατηγοριοποίηση Εικόνων

Για να αξιολογήσουμε την επίδοση και την καταλληλόλητα των παραπάνω ανιχνευτών και περιγραφητών, θα τους χρησιμοποιήσουμε για να κατηγοριοποιήσουμε εικόνες σε 3 κλάσεις: αυτοκίνητο, ποδήλατο, άνθρωπος.

Συγκεκριμένα, θα χρησιμοποιήσουμε μονάχα τις πολυκλιμακωτές εκδοχές των ανιχνευτών, και η διαδικασία που θα ακολουθηθεί είναι η εξής:
Αρχικά, θα χρησιμοποιήσουμε την συνάρτηση FeatureExtraction που μας δίνεται για να εξαγάγουμε χαρακτηριστικά για κάθε ζεύγος ανιχνευτή-περιγραφητή πάνω σε όλη τη βάση.
Στην συνέχεια διαχωρίζουμε τις εικόνες σε train και test μέσω της createTrainTest.
Και τέλος, με την βοήθεια της BoVW και SVM ταξινομητή κάνουμε την κατηγοριοποίηση.

Χρησιμοποιώντας τον κώδικα που μας δόθηκε στο example_classification.py για όλα τα ζεύγη, καταλήξαμε στα παρακάτω αποτελέσματα.
Για την χρήση SURF:

```
Mean accuracy for Hessian-Laplace with SURF descriptors: 47.448%

Mean accuracy for Multiscale Box with SURF descriptors: 57.655%
```

Για την χρήση HOG:

```
Mean accuracy for Hessian-Laplace with HOG descriptors: 59.862%

Mean accuracy for Multiscale Box with HOG descriptors: 64.000%
```

Συμπεραίνουμε ότι η χρήση HOG δίνει αρκετά καλύτερα αποτελέσματα στην κατηγοριοποίηση, ενώ και πάλι η ανίχνευση με χρήση multiscale box filters είναι καλύτερη, ανεξάρτητα του περιγραφητή.