Ζαρίφης Στέλιος A.M.: el20435
Ζαφείριος Αστρεινίδης A.M.: el19053

26/04/2023

ΟΡΑΣΗ ΥΠΟΛΟΓΙΣΤΩΝ

8ο Εξάμηνο 2022 – 2023

Lab Assignment 1 – Solutions

Ζαρίφης Στέλιος – el20435 Email: el20435@mail.ntua.gr

Αστρεινίδης Ζαφείριος – el19053 Email: el19053@mail.ntua.gr

Contents

Μέρος 1: Ανίχνευση Ακμών σε Γκρίζες Εικόνες 2
1.1. Δημιουργία Εικόνων Εισόδου 2
1.2. Υλοποίηση Αλγορίθμων Ανίχνευσης Ακμών 7
1.3. Αξιολόγηση των Αποτελεσμάτων Ανίχνευσης Ακμών 19
1.4. Εφαρμογή των Αλγορίθμων Ανίχνευσης Ακμών σε Πραγματικές εικόνες 23
Μέρος 2: Ανίχνευση Σημείων Ενδιαφέροντος (Interest Point Detection) 30
2.1. Ανίχνευση Γωνιών 30
2.2. Πολυκλιμακωτή Ανίχνευση Γωνιών 41
2.4. Πολυκλιμακωτή Ανίχνευση Blobs 47
2.5 Box Filters 50
Μέρος 3: Εφαρμογές σε Ταίριασμα και Κατηγοριοποίηση Εικόνων με Χρήση Τοπικών Περιγραφητών στα Σημεία Ενδιαφέροντος 57
3.1. Ταίριασμα Εικόνων υπό Περιστροφή και Αλλαγή Κλίμακας 58
3.2. Εξαγωγή Χαρακτηριστικών & Αναπαράσταση Bag of Visual Words 63

Μέρος 1: Ανίχνευση Ακμών σε Γκρίζες Εικόνες

Στο πρώτο μέρος της εργασίας, θα εξετάσουμε τις εφαρμογές γραμμικών και μη γραμμικών φίλτρων στο πρόβλημα της ανίχνευσης ακμών σε μια εικόνα. Συγκεκριμένα, θα προσθέσουμε θόρυβο σε μία "καθαρή", δοκιμαστική εικόνα με σκοπό να εφαρμόσουμε αλγορίθμους ανίχνευσης ακμών και να τους αξιολογήσουμε ώστε τελικά να τους εφαρμόσουμε σε μια πραγματική εικόνα.

```
# Imports
import cv2
import numpy as np
import matplotlib.pyplot as plt

# To save and create gif for corner detection
import os
from google.colab import files
import imageio

%matplotlib inline
```

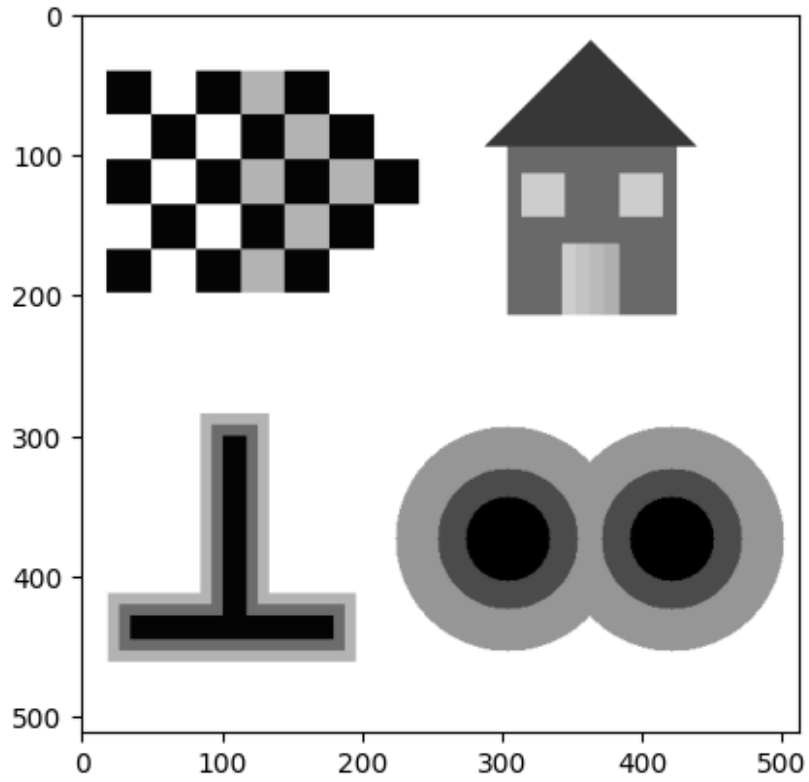
1.1. Δημιουργία Εικόνων Εισόδου

1.1.1. Εισαγωγή Εικόνας

Φορτώνουμε τη δοκιμαστική εικόνα `edgetest_23.png` για να την επεξεργαστούμε.

```
# Load an image in grayscale format using OpenCV
img = cv2.imread("edgetest_23.png", cv2.IMREAD_GRAYSCALE)

# Display the image using matplotlib's imshow function and a gray colormap
plt.imshow(img, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fab4d3e79a0>
```

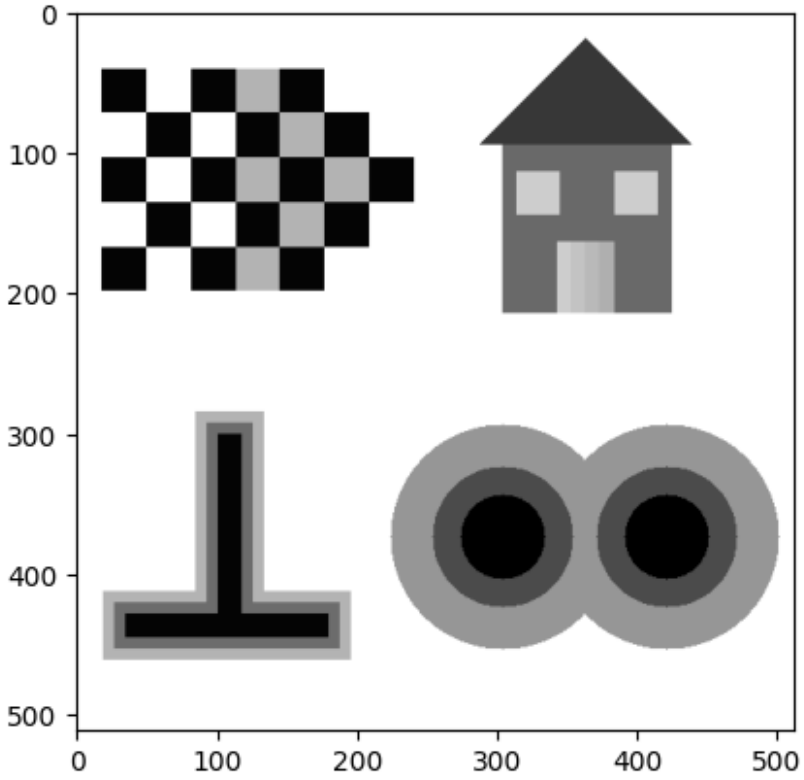


Κανονικοποιούμε την εικόνα ώστε η ένταση των pixels να βρίσκεται στο εύρος $[0,1]$. Το βήμα αυτό έχει ιδιαίτερη σημασία για τη μετέπειτα ανάλυση διότι βελτιώνει την απόδοση των επόμενων βημάτων επεξεργασίας (φιλτράρισμα και ανίχνευση ακμών). Ο θόρυβος επισκιάσει σημαντικά χαρακτηριστικά της εικόνας και δυσκολεύει τη διάκριση των πραγματικών ακμών από εκείνες που προκαλούνται από το θόρυβο.

Με την κανονικοποίηση της εικόνας, η διαφορά pixel intensity της πραγματικής εικόνας σε σχέση με τη θορυβώδη είναι πολύ μικρότερη. Έτσι, και οι διαφορές σε intensity μεταξύ γειτονικών pixels είναι μικρότερες, με αποτέλεσμα πιο smooth χαρακτηριστικά και άρα όχι τόσο μεγάλες τιμές όταν χρειαστεί να εφαρμόσουμε τα φίλτρα που κάνουν συνέλιξη με την εικόνα ή/και υπολογίζουν το gradient της. Έτσι, επιτυγχάνεται ακριβέστερη ανίχνευση ακμών οδηγώντας και άρα πιο εύρωστα φίλτρα.

```
# Normalize the image pixel values
img_norm = img.astype(np.float)/img.max()

# Display the normalized image using a grayscale colormap
plt.imshow(img_norm, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fab45fb5460>
```



1.1.2. Προσθήκη Θορύβου στην Εικόνα

Σε αυτό το βήμα, προσθέτουμε λευκό gaussian θόρυβο μέσης τιμής $\mu = 0$, τυπικής απόκλισης $\sigma = \sigma_n$ στην εικόνα. Εξετάζουμε δύο περιπτώσεις για το σ_n:

1. σ_n τέτοιο ώστε $PSNR = 20dB$
2. σ_n τέτοιο ώστε $PSNR = 10dB$

Από τον ορισμό του $PSNR$ έχουμε για το σ_n:

$$PSNR = 20\log_{10}\frac{I_{max} - I_{min}}{\sigma_n} \Leftrightarrow \sigma_n = \frac{I_{max} - I_{min}}{10^{\frac{PSNR}{20}}}$$

Άρα διακρίνουμε τις περιπτώσεις:

1. $PSNR = 20$:

$$\sigma_n = \frac{1 - 0}{10^{\frac{20}{20}}} = 1$$

2. $PSNR = 10$:

$$\sigma_n = \frac{1 - 0}{10^{\frac{10}{20}}} \approx 0.316$$

```
# Find the maximum and minimum pixel values in the normalized image
Imax_norm = np.max(img_norm)
```

```
Imin_norm = np.min(img_norm)

# Print the maximum and minimum pixel values
print("Imax =", Imax_norm)
print("Imin =", Imin_norm)
```

```
Imax = 1.0
Imin = 0.0
```

```
# calculate the deviation for PSNR values of 20 dB and 10 dB
sn_psnr20 = (Imax_norm - Imin_norm) / 10
sn_psnr10 = (Imax_norm - Imin_norm) / np.sqrt(10)

# print the deviation values
print("Deviation for PSNR = 20 dB:", sn_psnr20)
print("Deviation for PSNR = 10 dB:", sn_psnr10)
```

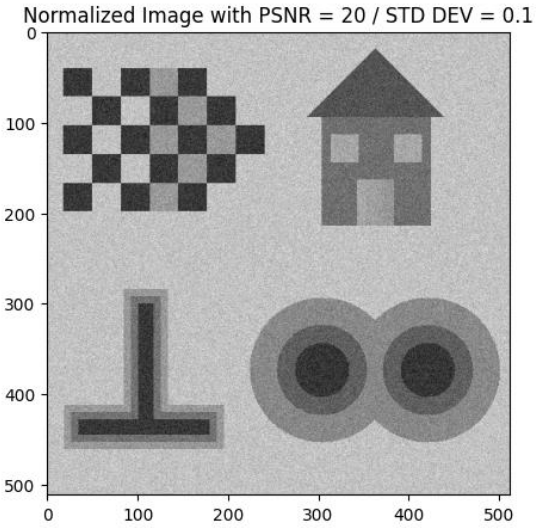
```
Deviation for PSNR = 20 dB: 0.1
Deviation for PSNR = 10 dB: 0.31622776601683794
```

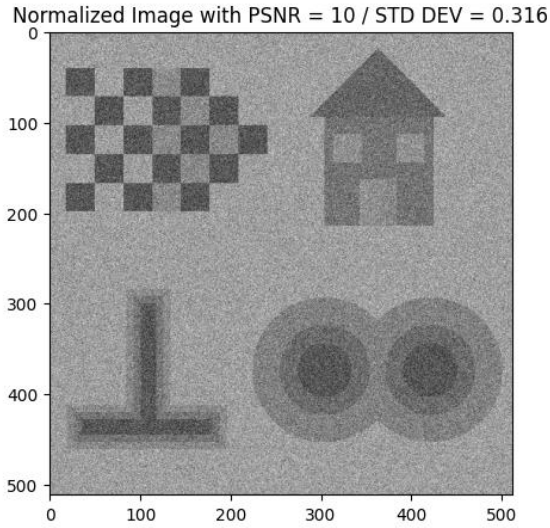
Παράγουμε τους θορύβους και τους προσθέτουμε στην εικόνα

```
# Generate random noises using normal distributions with the standard
deviations calculated
n20 = np.random.normal(0, sn_psnr20, (512, 512))
n10 = np.random.normal(0, sn_psnr10, (512, 512))

# Add the noise to the normalized images to obtain noisy images for PSNR = 20
dB and PSNR = 10 dB
I20 = img_norm + n20
I10 = img_norm + n10
```

Τυπώνουμε τις θορυβώδεις εικόνες.

```
# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(11, 11))

# Display the first image in the first column
axs[0].imshow(I20, cmap='gray')
axs[0].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1')

# Display the second image in the second column
axs[1].imshow(I10, cmap='gray')
axs[1].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316')

# Show the figure
plt.show()
```


Normalized Image with PSNR = 20 / STD DEV = 0.1
0
100
200
300
400
500
0
100
200
300
400
500


Normalized Image with PSNR = 10 / STD DEV = 0.316
0
100
200
300
400
500
0
100
200
300
400
500

1.2. Υλοποίηση Αλγορίθμων Ανίχνευσης Ακμών

1.2.1. Προσσέγγιση Συνεχών Φίλτρων με Διακριτά

Ορίζουμε τις δύο συναρτήσεις, gaussianKernel, laplacian_of_gaussian και τις χρησιμοποιούμε για να παράξουμε Laplacian of Gaussian (LoG) kernels με διάφορες τιμές του σ.

1. Η συνάρτηση gaussianKernel παράγει ένα 2D Gaussian kernel μεγέθους $n \times n$ και με τυπική απόκλιση σ.

   Το kernel size ορίζεται ως `n = int(2*np.ceil(3*sigma)+1)` για να είναι βέβαιο ότι το kernel καλύπτει τουλάχιστον το 99% της περιοχής κάτω από την Γκαουσιανή καμπύλη!

   Ύστερα, χρησιμοποιώντας τη συνάρτηση `cv2.getGaussianKernel`, παράγουμε ένα μονοδιάστατο Gaussian kernel μεγέθους n. Το εξωτερικό γινόμενο αυτού με τον εαυτό του μας δίνει το ζητούμενο διδιάστατο Gaussian kernel, το οποίο και επιστρέφει η συνάρτηση.

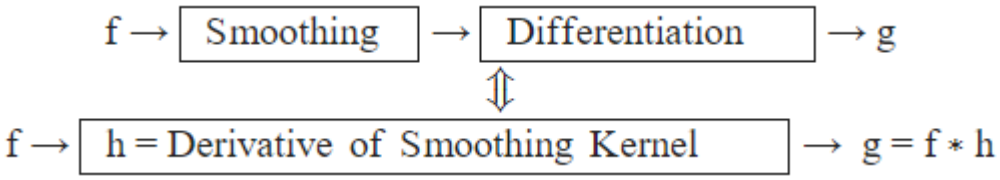
2. Η συνάρτηση `laplacian_of_gaussian` παράγει ένα 2D Laplacian of Gaussian kernel μεγέθους $n \times n$ και με τυπική απόκλιση σ.

   Για τον ίδιο λόγο με την προηγούμενη περίπτωση, το kernel size ορίζεται ως `n = int(2*np.ceil(3*sigma)+1)`.

   Χρησιμοποιώντας τη συνάρτηση `np.meshgrid` παράγουμε ένα πλέγμα από τιμές (x, y) για τις οποίες θα υπολογίσουμε τη Laplacian of Gaussian χρησιοποιώντας τη συνάρτηση:

$$LoG(x,y) = \frac{(x^2 + y^2 - 2\sigma^2)}{2\pi\sigma^6} e^{-(x^2+y^2)/(2\sigma^2)}$$

   Σημειώνουμε το λόγο για τον οποίο χρησιμοποιήσαμε την αναλυτική έκφραση της Laplacian of Gaussian:



   Όπως φαίνεται, οι δύο επιλογές είναι ισοδύναμες. Επειδή, όμως, χρειάζεται να τις υλοποιήσουμε στον υπολογιστή, δηλαδή διακριτά, έχουμε μία διαφορά:

   - Στην πρώτη περίπτωση, έχουμε συνέλιξη της εισόδου με το πρώτο φίλτρο και η έξοδός των συνελίσσεται με το δεύτερο φίλτρο και παράγεται το τελικό σήμα.
   - Στη δεύτερη περίπτωση, έχουμε υπολογίσει αναλυτικά (στο συνεχή χώρο - χωρίς σφάλμα!) τη συνέλιξη των δύο φίλτρων και εκμεταλλευόμενοι την

ιδιότητα της αντιμετάθεσης εκτελούμε συνολικά μία συνέλιξη του πλέον διακριτοποιημένου συνολικού φίλτρου με την είσοδο

Επιλέγουμε, λοιπόν, τη δεύτερη υλοποίηση στην οποία δημιουργείται μία φορά σφάλμα λόγω διακριτής συνέλιξης, σε αντίθεση με την πρώτη που έχουμε δύο συνελίξεις.

```
# Function to generate a Gaussian kernel
def gaussianKernel(sigma):
    # Calculate the kernel size based on the sigma value
    n = int(2*np.ceil(3*sigma)+1)

    # Get 1D Gaussian kernel
    gauss1D = cv2.getGaussianKernel(n, sigma)

    # Compute 2D Gaussian kernel multiplying 1D kernel with its transpose
    gauss2D = gauss1D @ gauss1D.T

    return gauss2D

# Function to generate a Laplacian of Gaussian (LoG) kernel
def laplacian_of_gaussian(sigma):
    # Calculate the kernel size based on the sigma value
    n = int(np.ceil(3 * sigma)) * 2 + 1

    # Create a meshgrid of x and y imgPoints
    x, y = np.meshgrid(np.arange(-n//2 + 1, n//2 + 1), np.arange(-n//2 + 1,
n//2 + 1))

    # Compute the Laplacian of Gaussian
    laplacian = ((x**2 + y**2 - 2 * sigma**2) / 2*np.pi*sigma**6) * np.exp(-
(x**2 + y**2) / (2 * sigma**2))

    return laplacian

# Set values of sigma for the kernels
s1 = 1.5
s2 = 2
s3 = 4

# Generate Laplacian of Gaussian kernels
LoG1_5 = laplacian_of_gaussian(s1)
LoG2 = laplacian_of_gaussian(s2)
LoG4 = laplacian_of_gaussian(s3)
```

1.2.2. Προσεγγίσεις της Λαπλασιανής της Εξομαλυμένης Εικόνας

Θα προσπαθήσουμε να προσεγγίσουμε τη Laplacian L της εξομαλυμένης εικόνας $I_\sigma(x,y) = G_\sigma * I(x,y)$ με δύο τρόπους:

1. Γραμμικά: Ως συνέλιξη της I με τη $LoG\ \ h$:
$$L_1 = \nabla^2(G_\sigma * I) = (\nabla^2 G_\sigma) * I = h * I$$
2. Μη Γραμμικά: Ως εκτίμηση της Laplacian της I_σ με μορφολογικά φίλτρα:
$$L_2 = (I_\sigma \oplus B) + (I_\sigma \ominus B) - 2I_\sigma$$

Εφαρμόζουμε τα LoG filters με διάφορες τιμές τυπικής απόκλισης $\sigma \in \{1.5,2,4\}$ και εξετάζουμε τις εξόδους.

```
# Apply the LoG kernels to the noisy images
I20LoG1_5 = cv2.filter2D(I20, -1, LoG1_5)
I10LoG1_5 = cv2.filter2D(I10, -1, LoG1_5)

I20LoG2 = cv2.filter2D(I20, -1, LoG2)
I10LoG2 = cv2.filter2D(I10, -1, LoG2)

I20LoG4 = cv2.filter2D(I20, -1, LoG4)
I10LoG4 = cv2.filter2D(I10, -1, LoG4)
```

Τυπώνουμε τις φιλτραρισμένες εικόνες και παρατηρούμε ότι έχουμε μια κατάσταση *trade - off*:

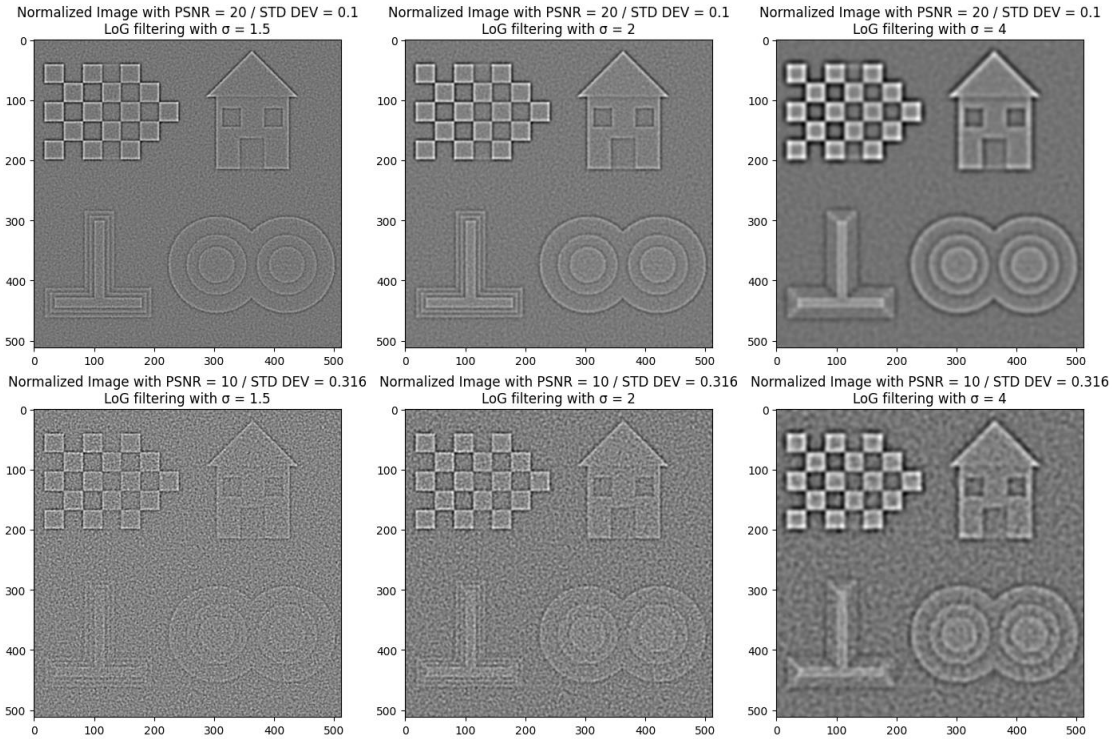
1. Για μικρές τιμές σ έχουμε στενό bandwidth του φίλτρου. Αυτό ενισχύει τις λεπτομέρειες της εικόνας (ακμές, γωνίες, ...) αλλά και το θόρυβο. Έτσι, ενώ κάποιες λεπτομέρειες φαίνονται καλύτερα, κάποιες άλλες "χάνονται" στο θόρυβο.
2. Αντίθετα, για μεγάλες τιμές σ έχουμε πλατύ bandwidth του φίλτρου. Έτσι, η εικόνα θολώνει περισσότερο αφού σε κάθε σημείο (x, y) έχουμε υπέρθεση με μεγαλύτερο πλάτος εντάσεων γειτονικών pixels.

Επομένως, είναι σημαντικό να επιλέξουμε την κατάλληλη τιμή σ που σχετίζεται τόσο με την εικόνα (και το θόρυβό της), όσο και με τη μετέπειτα επεξεργασία της εικόνας.

Η άνω σειρά περιλαμβάνει τις εξομαλυμένες εικόνες με τις διάφορες τιμές σ για θόρυβο με $PSNR = 20dB$ και η κάτω σειρά εκείνες για θόρυβο με $PSNR = 10dB$

```
# create a figure with 2 rows and 3 columns
fig, axs = plt.subplots(2, 3, figsize=(17, 11))

# define greek symbol sigma
greekSigma = "\u03C3"

# display the image filtered with LoG1_5 on the top left
axs[0][0].imshow(I20LoG1_5, cmap='gray')
# set the title for the image
axs[0][0].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1\nLoG
filtering with ' + greekSigma + ' = 1.5')
# display the image filtered with LoG1_5 on the bottom left
axs[1][0].imshow(I10LoG1_5, cmap='gray')
# set the title for the image
```

```
axs[1][0].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316\nLoG
filtering with ' + greekSigma + ' = 1.5')

# display the image filtered with LoG2 on the top middle
axs[0][1].imshow(I20LoG2, cmap='gray')
# set the title for the image
axs[0][1].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1\nLoG
filtering with ' + greekSigma + ' = 2')
# display the image filtered with LoG2 on the bottom middle
axs[1][1].imshow(I10LoG2, cmap='gray')
# set the title for the image
axs[1][1].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316\nLoG
filtering with ' + greekSigma + ' = 2')

# display the image filtered with LoG4 on the top right
axs[0][2].imshow(I20LoG4, cmap='gray')
# set the title for the image
axs[0][2].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1\nLoG
filtering with ' + greekSigma + ' = 4')
# display the image filtered with LoG4 on the bottom right
axs[1][2].imshow(I10LoG4, cmap='gray')
# set the title for the image
axs[1][2].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316\nLoG
filtering with ' + greekSigma + ' = 4')

# show the figure
plt.show()
```



Εφαρμόζουμε τα Morphological filters έχοντας πρώτα εξομαλύνει την εικόνα με διάφορες τιμές τυπικής απόκλισης $\sigma \in \{1.5,2,4\}$ και εξετάζουμε τις εξόδους.

```
# Define kernel for dilation and erosion
kern = np.array([
    [0,1,0],
    [1,1,1],
    [0,1,0]
], dtype=np.uint8)

# Create Gaussian kernels
gaussian1_5 = gaussianKernel(s1)
gaussian2 = gaussianKernel(s2)
gaussian4 = gaussianKernel(s3)

# Apply Gaussian filtering to the images
I20_gauss1_5 = cv2.filter2D(I20, -1, gaussian1_5)
I10_gauss1_5 = cv2.filter2D(I10, -1, gaussian1_5)

I20_gauss2 = cv2.filter2D(I20, -1, gaussian2)
I10_gauss2 = cv2.filter2D(I10, -1, gaussian2)

I20_gauss4 = cv2.filter2D(I20, -1, gaussian4)
I10_gauss4 = cv2.filter2D(I10, -1, gaussian4)

# Dilate and erode the filtered images
dilated_img20_s1_5 = cv2.dilate(I20_gauss1_5, kern)
dilated_img10_s1_5 = cv2.dilate(I10_gauss1_5, kern)
dilated_img20_s2   = cv2.dilate(I20_gauss2, kern)
dilated_img10_s2   = cv2.dilate(I10_gauss2, kern)
dilated_img20_s4   = cv2.dilate(I20_gauss4, kern)
dilated_img10_s4   = cv2.dilate(I10_gauss4, kern)

eroded_img20_s1_5 = cv2.erode(I20_gauss1_5, kern)
eroded_img10_s1_5 = cv2.erode(I10_gauss1_5, kern)
eroded_img20_s2   = cv2.erode(I20_gauss2, kern)
eroded_img10_s2   = cv2.erode(I10_gauss2, kern)
eroded_img20_s4   = cv2.erode(I20_gauss4, kern)
eroded_img10_s4   = cv2.erode(I10_gauss4, kern)

# Calculate the Approximation of Laplacian of the image
L2_20_s1_5 = dilated_img20_s1_5 + eroded_img20_s1_5 - 2*I20_gauss1_5
L2_10_s1_5 = dilated_img10_s1_5 + eroded_img10_s1_5 - 2*I10_gauss1_5
L2_20_s2   = dilated_img20_s2   + eroded_img20_s2   - 2*I20_gauss2
L2_10_s2   = dilated_img10_s2   + eroded_img10_s2   - 2*I10_gauss2
L2_20_s4   = dilated_img20_s4   + eroded_img20_s4   - 2*I20_gauss4
L2_10_s4   = dilated_img10_s4   + eroded_img10_s4   - 2*I10_gauss4
```

Τυπώνουμε τις φιλτραρισμένες εικόνες και παρατηρούμε ότι επίσης υπάρχει trade - off:

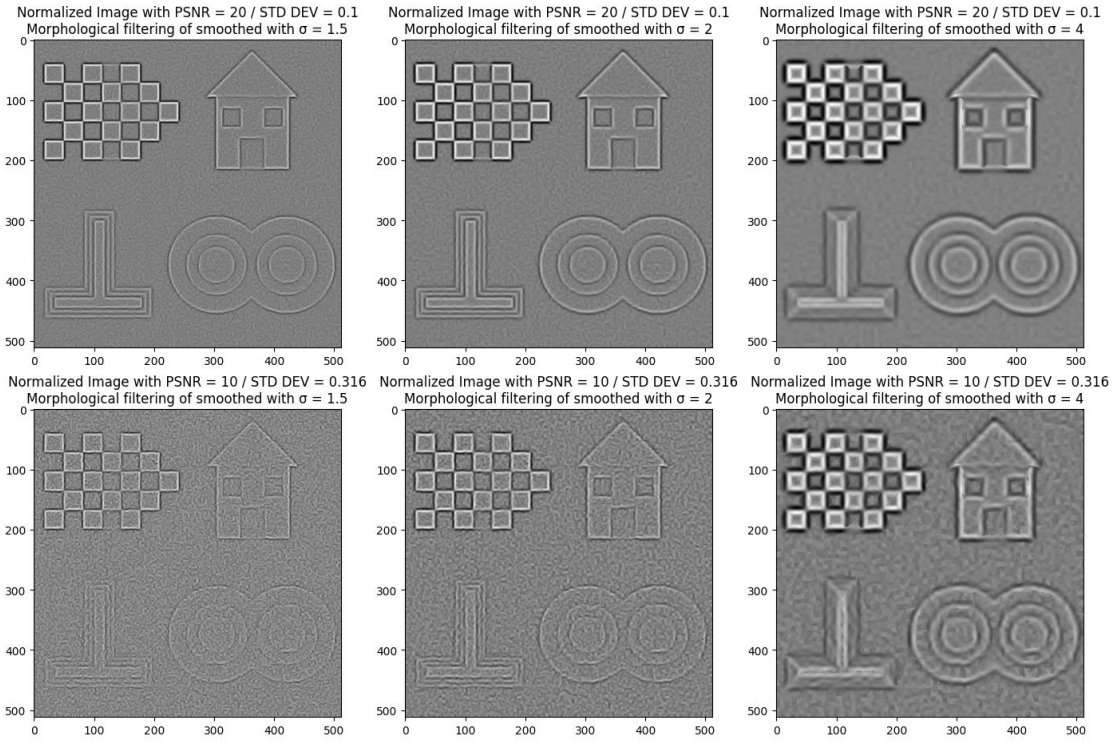
1. Για μικρές τιμές σ έχουμε στενό bandwidth του φίλτρου. Αυτό ενισχύει τις λεπτομέρειες της εικόνας (ακμές, γωνίες, ...) αλλά και το θόρυβο. Έτσι, ενώ κάποιες λεπτομέρειες φαίνονται καλύτερα, κάποιες άλλες "χάνονται" στο θόρυβο.
2. Αντίθετα, για μεγάλες τιμές σ έχουμε πλατύ bandwidth του φίλτρου. Έτσι, η εικόνα θολώνει αφού σε κάθε σημείο (x, y) έχουμε υπέρθεση με μεγαλύτερο πλάτος εντάσεων γειτονικών pixels.

Καθώς αυξάνεται το σίγμα του φίλτρου Gauss που χρησιμοποιείται για την εξομάλυνση, αυξάνεται και το μέγεθος των δομών που τα μορφολογικά φίλτρα θα είναι σε θέση να αφαιρέσουν. Επομένως, για την απομάκρυνση των ίδιων τύπων θορύβου για μεγαλύτερες τιμές sigma θα απαιτούνται μεγαλύτερα στοιχεία δόμησης (δηλαδή με περισσότερα εικονοστοιχεία ή μεγαλύτερη ακτίνα).

Επομένως, είναι και πάλι σημαντικό να επιλέξουμε την κατάλληλη τιμή σ για τη συγκεκριμένη εφαρμογή.

```
# Create a 2x3 grid of subplots with a specific figure size
fig, axs = plt.subplots(2, 3, figsize=(17, 11))

# Display the first image on the top-left subplot with a grayscale color map
and add a title
axs[0][0].imshow(L2_20_s1_5, cmap='gray')
axs[0][0].set_title('Normalized Image with PSNR = 20 / STD DEV =
0.1\nMorphological filtering of smoothed with ' + greekSigma + ' = 1.5')

# Display the second image on the bottom-left subplot with a grayscale color
map and add a title
axs[1][0].imshow(L2_10_s1_5, cmap='gray')
axs[1][0].set_title('Normalized Image with PSNR = 10 / STD DEV =
0.316\nMorphological filtering of smoothed with ' + greekSigma + ' = 1.5')

# Display the third image on the top-center subplot with a grayscale color
map and add a title
axs[0][1].imshow(L2_20_s2, cmap='gray')
axs[0][1].set_title('Normalized Image with PSNR = 20 / STD DEV =
0.1\nMorphological filtering of smoothed with ' + greekSigma + ' = 2')

# Display the fourth image on the bottom-center subplot with a grayscale
color map and add a title
axs[1][1].imshow(L2_10_s2, cmap='gray')
axs[1][1].set_title('Normalized Image with PSNR = 10 / STD DEV =
0.316\nMorphological filtering of smoothed with ' + greekSigma + ' = 2')

# Display the fifth image on the top-right subplot with a grayscale color map
and add a title
axs[0][2].imshow(L2_20_s4, cmap='gray')
axs[0][2].set_title('Normalized Image with PSNR = 20 / STD DEV =
0.1\nMorphological filtering of smoothed with ' + greekSigma + ' = 4')
```

```
# Display the sixth image on the bottom-right subplot with a grayscale color
map and add a title
axs[1][2].imshow(L2_10_s4, cmap='gray')
axs[1][2].set_title('Normalized Image with PSNR = 10 / STD DEV =
0.316\nMorphological filtering of smoothed with ' + greekSigma + ' = 4')

# Display the plot
plt.show()
```



1.2.3. Προσέγγιση των Σημείων Μηδενισμού της Λαπλασιανής της Εξομαλυμένης Εικόνας

Σε αυτό το στάδιο θα προσπαθήσουμε να προσεγγίσουμε τις διασταυρώσεις της Λαπλασιανής L της εικόνας με τη μηδενική συνάρτηση. Τα βήματα είναι τα εξής:

1. Δημιουργούμε τη Δυαδικής Εικόνας Προσήμου X της L ως την κατωφλίωση της L στο 0, δηλαδή $X = (L >= 0)$. Αυτό σημαίνει ότι τα pixels με μη αρνητικές τιμές έντασης παίρνουν την τιμή 1, ενώ τα υπόλοιπα παίρνουν την τιμή 0. Έτσι, παίρνουμε μια νέα "δυαδική" εικόνα που έχει τιμές 1 στα σημεία όπου η Λαπλασιανή της αρχικής εικόνας είναι μη αρνητική και 0 όπου είναι αρνητική.

2. Βρίσκουμε το περίγραμμα του X με χρήση μορφολογικών πράξεων. Χρησιμοποιούμε τον πυρήνα B για τη διαστολή ($\oplus$) και διάβρωση ($\ominus$) της X. Η αφαίρεση των "δυαδικών" εικόνων που προέκυψαν μας δίνει τη δυαδική εικόνα Y που προσεγγίζει το περίγραμμα της X (στο όριό της, η Y είναι ακριβώς το περίγραμμα της X). Σημείωση: Ο πυρήνας B έχει τη μορφή:

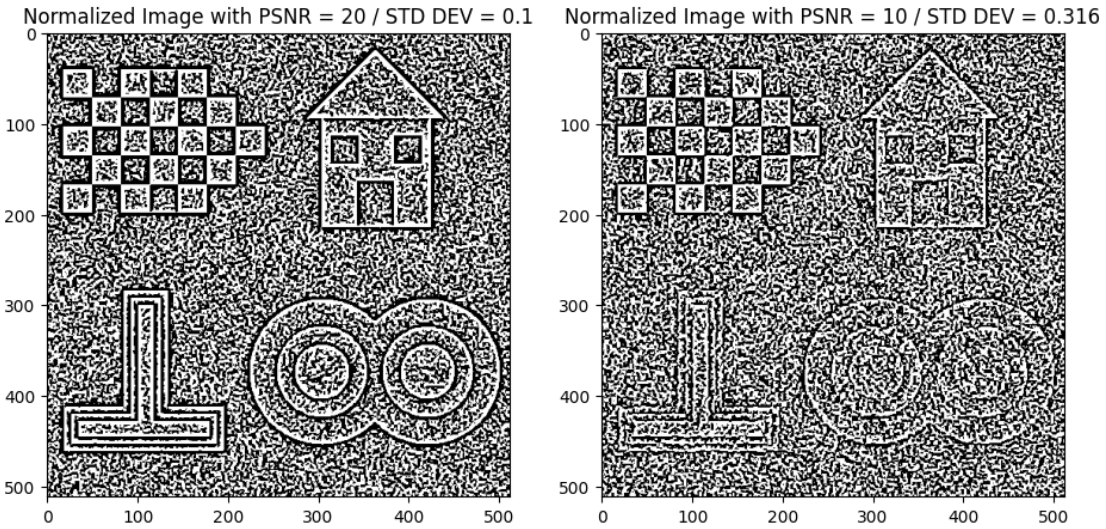
$$B = \begin{bmatrix} & o & \\ o & o & o \\ & o & \end{bmatrix}$$

Τα zero-crossings της Λαπλασιανής αντιστοιχούν στα σημεία της εικόνας όπου η L αλλάζει πρόσημο. Τα σημεία προσεγγίζονται από τα σημεία όπου η δυαδική εικόνα Y έχει την τιμή 1.

```
X20 = (L2_20_s1_5 >= 0).astype(np.uint8)
X10 = (L2_10_s1_5 >= 0).astype(np.uint8)
```

Όπως φαίνεται καθαρά στις επόμενες εικόνες, ειδικά στην όχι τόσο θορυβώδη, παρατηρούμε ότι εκεί που βρίσκονται πραγματικές ακμές, έχουμε τεράστια μεταβολή έντασης pixel εκατέρωθέν των, όπως ακριβώς ήταν αναμενόμενο από την περιγραφή του αλγορίθμου.

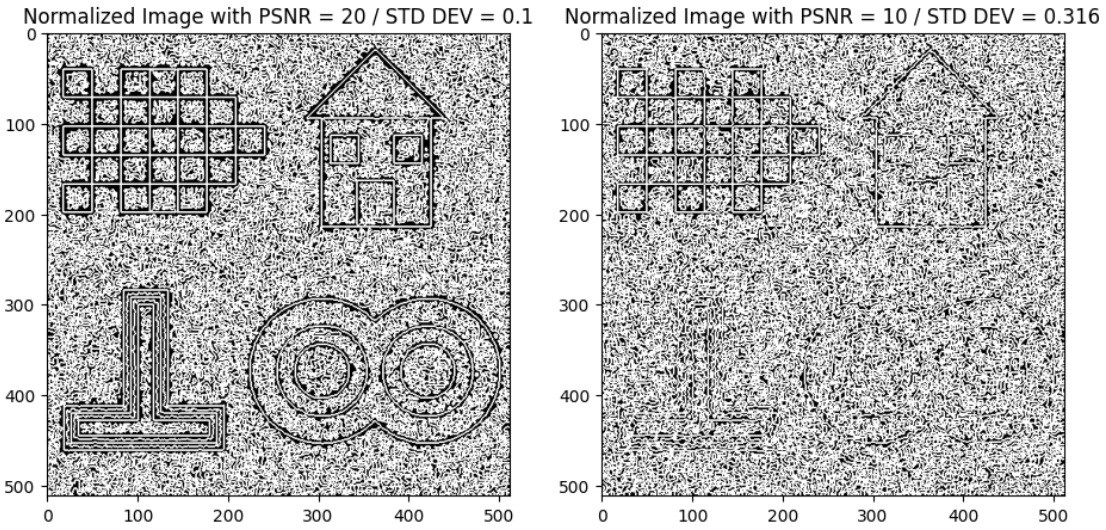
```
# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(11, 9))

# Display the first image in the first column
axs[0].imshow(X20, cmap='gray')
axs[0].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1')

# Display the second image in the second column
axs[1].imshow(X10, cmap='gray')
axs[1].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316')

# Show the figure
plt.show()
```




```
Y20 = cv2.dilate(X20, kern) - cv2.erode(X20, kern)
Y10 = cv2.dilate(X10, kern) - cv2.erode(X10, kern)
```

Βλέπουμε πως οι ακμές αρχίζουν να σχηματίζονται πιο καθαρά!

```
# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(11, 9))

# Display the first image in the first column
axs[0].imshow(Y20, cmap='gray')
axs[0].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1')

# Display the second image in the second column
axs[1].imshow(Y10, cmap='gray')
axs[1].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316')

# Show the figure
plt.show()
```



1.2.4. Απόρριψη των zero-crossings

Αφού εντοπίσαμε τα zero-crossings της Λαπλασιανής της εξομαλυμένης εικόνας πρέπει να απορρίψουμε εκείνα που αντιστοιχούν σε σχετικά ομαλές περιοχές της εξομαλυμένης εικόνας. Αυτό συμβαίνει για τον εξής λόγο:

Εκεί που η εξομαλυμένη εικόνα είναι σχετικά ομαλή, είναι πιο πιθανό να μην έχουμε πραγματική ακμή (διότι αν είχαμε, θα βλέπαμε μεγάλη μεταβολή στην ένταση των pixels προς κάποια κατεύθυνση). Ο θόρυβος, όμως, οδηγεί στη δημιουργία zero-crossings σε τέτοιες περιοχές. Αυτές τις περιπτώσεις δεν πρέπει να τις αναγνωρίσουμε ως ακμές.

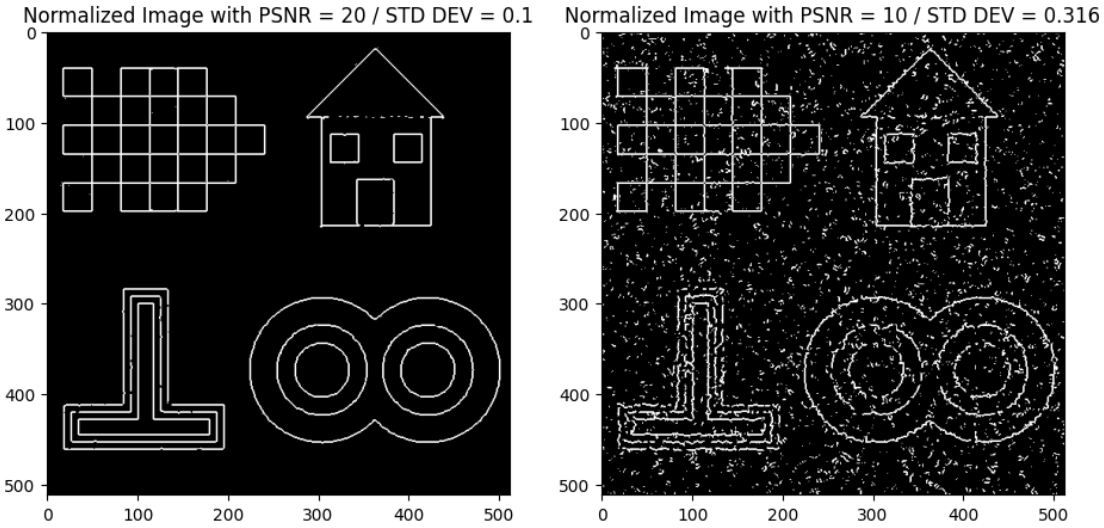
Για να το καταφέρουμε αυτό, χρησιμοποιούμε μια παράμετρο κατωφλίου θ_{edge}, την οποία συγκρίνουμε με τη μέγιστη τιμή του gradient στη γειτονιά κάθε pixel. Εάν η τιμή της κλίσης είναι μεγαλύτερη από το κατώφλι, το εξεταζόμενο pixel θεωρείται ακμή.

Συμπερασματικά, ο αλγόριθμος επιλέγει μόνο τις θέσεις (i, j) της εικόνας όπου η εξομαλυμένη εικόνα έχει αρκετά μεγάλη κλίση. Έτσι, απορρίπτονται ακμές σε ομαλές περιοχές της εικόνας όπου δεν υπάρχει μεγάλη μεταβολή της έντασης, αφού κατά πάσα πιθανότητα είναι "ψευδείς" ακμές.

Θεωρούμε, λοιπόν, σημεία πάνω σε ακμές εκείνα τα pixels (i, j) για τα οποία ισχύει η συνθήκη:

$$Y[i,j] = 1 \ \wedge \| \nabla I_\sigma[i,j] \| > \theta_{edge} \cdot \max_{x,y} \| \nabla I_\sigma \|$$

```
grad20 = np.sqrt(np.gradient(I20_gauss1_5)[0]**2 +
np.gradient(I20_gauss1_5)[1]**2)
grad10 = np.sqrt(np.gradient(I10_gauss1_5)[0]**2 +
np.gradient(I10_gauss1_5)[1]**2)
temp20 = np.zeros_like(grad20)
temp10 = np.zeros_like(grad10)
theta = 0.2
temp20 = (grad20 > theta*np.amax(grad20)).astype(np.uint8)
temp10 = (grad10 > theta*np.amax(grad10)).astype(np.uint8)
Y20 = Y20 & temp20
Y10 = Y10 & temp10

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(11, 9))

# Display the first image in the first column
axs[0].imshow(Y20, cmap='gray')
axs[0].set_title('Normalized Image with PSNR = 20 / STD DEV = 0.1')

# Display the second image in the second column
axs[1].imshow(Y10, cmap='gray')
axs[1].set_title('Normalized Image with PSNR = 10 / STD DEV = 0.316')

# Show the figure
plt.show()
```

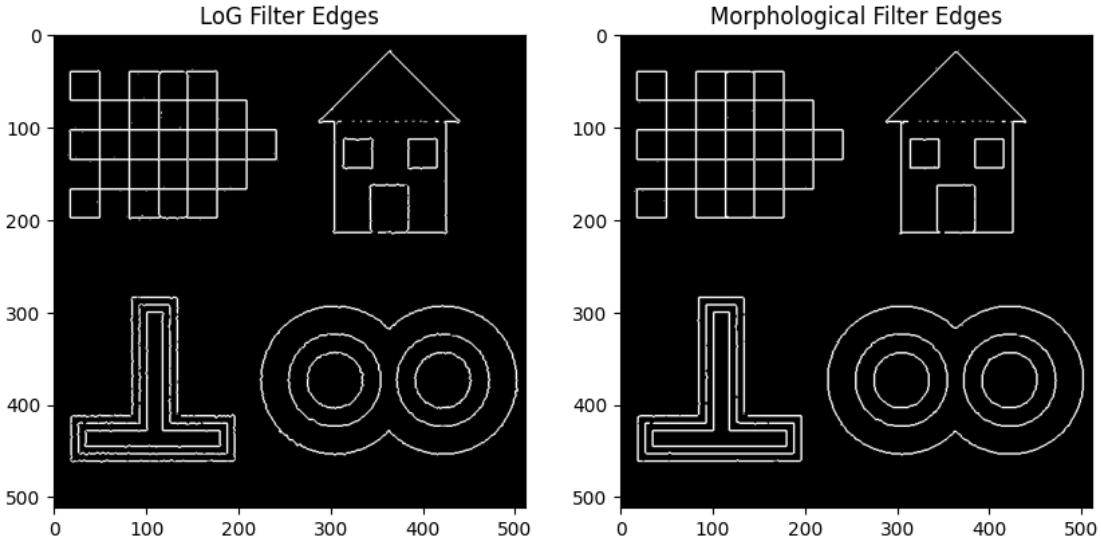


Περιλαμβάνουμε όλα τα βήματα σε μια συνάρτηση που δέχεται παραμέτρους τις τιμές $\{\sigma, \theta, method\}$, όπου $method$ αναφέρεται στη μέθοδο προσέγγισης της Λαπλασιανής της εικόνας.

```
def edgeDetect(img, sigma, theta, method = "LoG"):
    if method != "LoG" and method != "Morphological":
        print("Unknown method! Choose one of [LoG, Morphological]")

    gauss2D = gaussianKernel(sigma)
    img_Gauss = cv2.filter2D(img, -1, gauss2D)
    kernel_B = np.array([ [0,1,0],
                          [1,1,1],
                          [0,1,0] ], dtype=np.uint8)

    if method == "LoG":
        LoG = laplacian_of_gaussian(sigma)
        L = cv2.filter2D(img, -1, LoG)
    else:
        dilated_img_Gauss = cv2.dilate(img_Gauss, kernel_B)
        eroded_img_Gauss = cv2.erode(img_Gauss, kernel_B)
        L = dilated_img_Gauss + eroded_img_Gauss - 2*img_Gauss

    X = (L >= 0).astype(np.uint8)
    Y = cv2.dilate(X, kernel_B) - cv2.erode(X, kernel_B)

    grad = np.sqrt(np.gradient(img_Gauss)[0]**2 +
np.gradient(img_Gauss)[1]**2)
    temp = (grad > theta*np.amax(grad)).astype(np.uint8)
    res = Y & temp

    return res
```

Τυπώνουμε τις ακμές που βρίσκουμε με τις δύο μεθόδους. Παρατηρούμε ότι καλύτερη επίδοση έχουν τα μορφολογικά φίλτρα.

```
edges_LoG = edgeDetect(I20, sigma = 1.5, theta = 0.2, method = "LoG")
edges_Morph = edgeDetect(I20, sigma = 1.5, theta = 0.2, method =
"Morphological")

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# Display the first image in the first column
axs[0].imshow(edges_LoG, cmap='gray')
axs[0].set_title('LoG Filter Edges')

# Display the second image in the second column
axs[1].imshow(edges_Morph, cmap='gray')
axs[1].set_title('Morphological Filter Edges')

# Show the figure
plt.show()
```



1.3. Αξιολόγηση των Αποτελεσμάτων Ανίχνευσης Ακμών

1.3.1. Υπολογισμός των Αληθινών Ακμών

Θα αξιολογήσουμε τώρα την επίδοση των αλγορίθμων ανίχνευσης ακμών. Αυτό θα γίνει με χρήση της αρχικής καθαρής εικόνας. Υπολογίζουμε, λοιπόν, τη "δυαδική" εικόνα αληθινών ακμών εφαρμόζοντας τον απλό τελεστή ακμών:

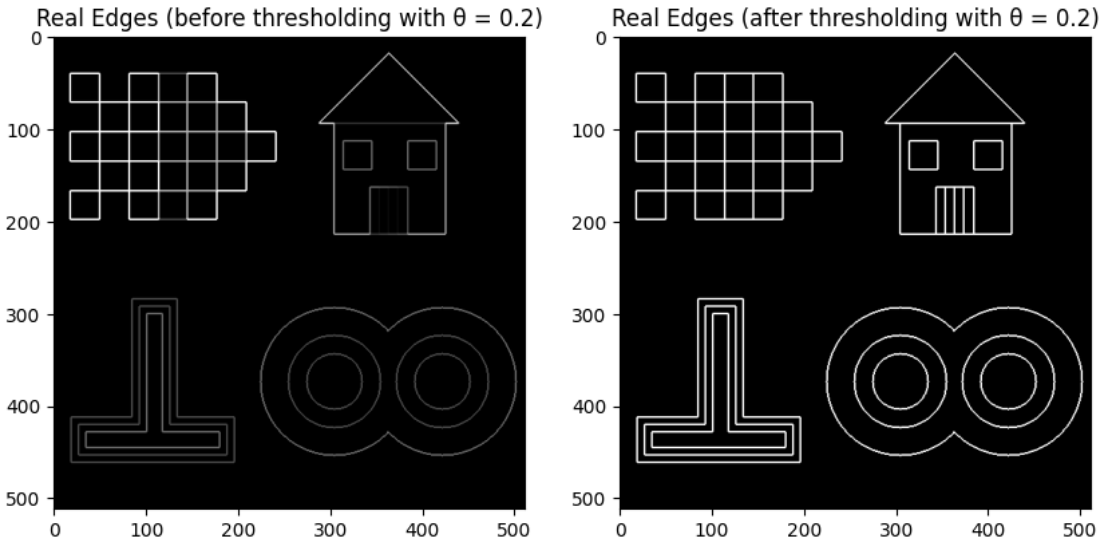
$$M = (I_0 \oplus B) - (I_0 \ominus B)$$

Εφαρμόζουμε ακόμα κατωφλίωση στην M χρησιμοποιώντας μια τιμή κατωφλίου $\theta_{realedge}$. Η δυαδική εικόνα T δίνεται στη συνέχεια από τη σχέση:

$$T = M > \theta_{realedge}$$

Αυτό σημαίνει ότι όλα τα pixels της M που έχουν τιμή μεγαλύτερη από $\theta_{realedge}$ τίθενται σε 1 για την T, ενώ σε άλλη περίπτωση τίθενται σε 0. Η δυαδική εικόνα T που προκύπτει περιέχει τις "αληθινές" ακμές της αρχικής καθαρής εικόνας.

```
M = cv2.dilate(img, kern) - cv2.erode(img, kern)
thetaReal = 0.2
T = (M > thetaReal).astype(np.uint8)

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# Display the first image in the first column
axs[0].imshow(M, cmap='gray')
axs[0].set_title('Real Edges (before thresholding with θ = ' + str(thetaReal)
+ ')')

# Display the second image in the second column
axs[1].imshow(T, cmap='gray')
axs[1].set_title('Real Edges (after thresholding with θ = ' + str(thetaReal)
+ ')')

# Show the figure
plt.show()
```



1.3.2. Ποσοτική αξιολόγηση των αποτελεσμάτων ανίχνευσης ακμών

Σε αυτό το βήμα θα συγκρίνουμε τις πραγματικές ακμές με εκείνες που ανιχνεύσαμε για να εξαγάγουμε συμπεράσματα σχετικά με την επίδοση των αλγορίθμων μας

Για τον υπολογισμό του κριτηρίου ποιότητας για το αποτέλεσμα της ανίχνευσης ακμών, υπολογίζονται το ποσοστό των ανιχνευμένων ακμών που είναι αληθείς και το ποσοστό των αληθών ακμών που ανιχνεύονται (Precision & Recall). Η Precision, $Pr(D|T)$, είναι ο λόγος του αριθμού των ανιχνευμένων ακμών που είναι αληθείς προς τον συνολικό αριθμό των ανιχνευμένων ακμών. Η Recall, $Pr(T|D)$, είναι ο λόγος του αριθμού των αληθινών ακμών που ανιχνεύονται προς τον συνολικό αριθμό των αληθινών ακμών.

Το κριτήριο ποιότητας τελικά είναι ο μέσος όρος των μεγεθών που υπολογίσαμε. Παρατήρηση: Όσο το C τείνει στο 1, τόσο καλύτερη είναι η απόδοση του αλγορίθμου μας. Ακόμα, μαθηματικά το C γράφεται ως εξής:

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

Επειδή τα σύνολα D και T είναι διακριτά, έχουμε ότι:

$$Pr(T|D) = card(D \cap T)/card(T) \wedge Pr(D|T)$$
$$= card(T \cap D)/card(D), card(\cdot): \#στοιχείων\ του\ συνόλου$$

```
def score(T,D):
    card_DandT = np.sum(T*D)

    PrTgivenD = card_DandT/np.sum(T)
    PrDgivenT = card_DandT/np.sum(D)
    C = (PrTgivenD+PrDgivenT)/2

    return C
```

```
score(T, edges_LoG)
```

```
0.9309848178036655
```

1.3.3. Προσέγγιση Βέλτιστων Παραμέτρων για την Ανίχνευση Ακμών

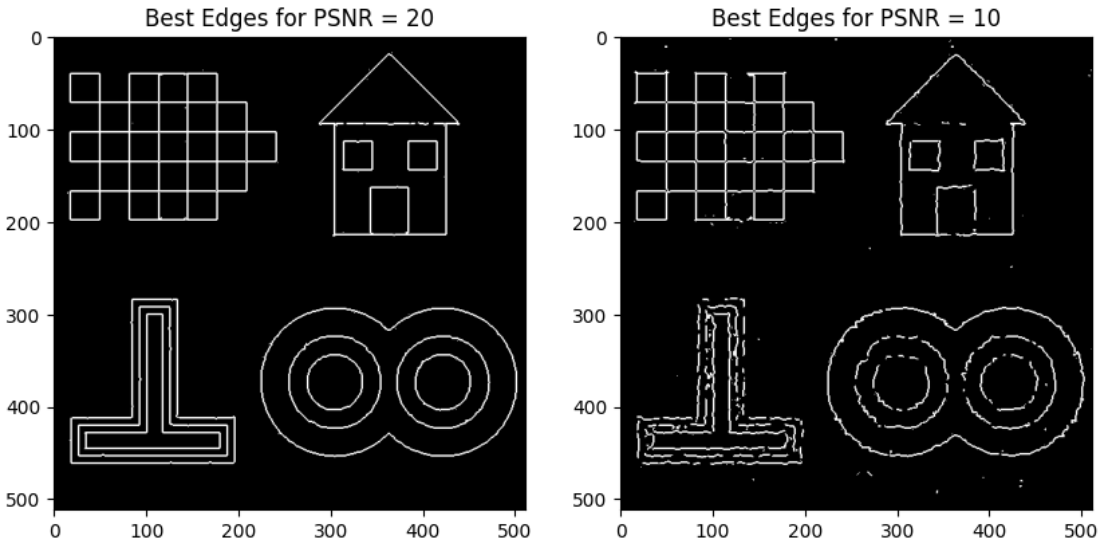
Με ένα απλό script θα προσεγγίσουμε τις βέλτιστες παραμέτρους $\{\sigma, \theta, method\}$ αξιολογώντας την απόδοση του αλγορίθμου ανίχνευσης ακμών για διάφορες τιμές τους. Αυτό που κάνουμε, είναι ότι εξετάζουμε συνδυασμούς των σ, θ για κάθε μία από τις μεθόδους $\{linear\ filtering, non\ linear\ filtering\}$ για τις δύο θορυβώδεις εικόνες και συγκρίνουμε τις επιδόσεις.

```
def gridSearch(img, noisyImg, methodList, sigmaList, thetaList):
    bestScore = 0
    bestParams = []
    M = cv2.dilate(img, kern) - cv2.erode(img, kern)

    for method in methodList:
        for sigma in sigmaList:
            for theta in thetaList:
                T = (M > theta).astype(np.uint8)
                e = edgeDetect(noisyImg, sigma = sigma, theta = theta, method
= method)
                currScore = score(T, e)
                if currScore > bestScore:
                    bestScore = currScore
                    bestParams = [method, sigma, theta, currScore]

    return bestParams

sigmaList = np.arange(1.5, 4.02, 0.02)
thetaList = np.arange(0.1, 0.32, 0.02)

bestI20 = gridSearch(img, noisyImg = I20, methodList = ["LoG",
"Morphological"], sigmaList = sigmaList, thetaList = thetaList)
print(bestI20)

bestI10 = gridSearch(img, noisyImg = I10, methodList = ["LoG",
"Morphological"], sigmaList = sigmaList, thetaList = thetaList)
print(bestI10)
```

```
['Morphological', 1.62, 0.16000000000000003, 0.9612710940073659]
['Morphological', 2.3600000000000008, 0.22000000000000003, 0.761190098653078]
```

```
bestEdgesI20 = edgeDetect(I20, sigma = bestI20[1], theta = bestI20[2], method
= bestI20[0])
bestEdgesI10 = edgeDetect(I10, sigma = bestI10[1], theta = bestI10[2], method
= bestI10[0])
```

```
# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(10, 5))

# Display the first image in the first column
axs[0].imshow(bestEdgesI20, cmap='gray')
axs[0].set_title('Best Edges for PSNR = 20')

# Display the second image in the second column
axs[1].imshow(bestEdgesI10, cmap='gray')
axs[1].set_title('Best Edges for PSNR = 10')

# Show the figure
plt.show()
```



1.4. Εφαρμογή των Αλγορίθμων Ανίχνευσης Ακμών σε Πραγματικές εικόνες

1.4.1. Εφαρμογή των Αλγορίθμων Ανίχνευσης Ακμών σε Πραγματικές εικόνες

Θα εφαρμόσουμε τώρα τους αλγορίθμους edge detection που υλοποιήσαμε στην εικόνα kyoto edges.jpg, χωρίς την προσθήκη θορύβου. Συγκεκριμένα, εφαρμόζουμε τις γραμμικές και μη γραμμικές μεθόδους για διάφορες τιμές $\sigma \in \{1.5,2,4\}$.

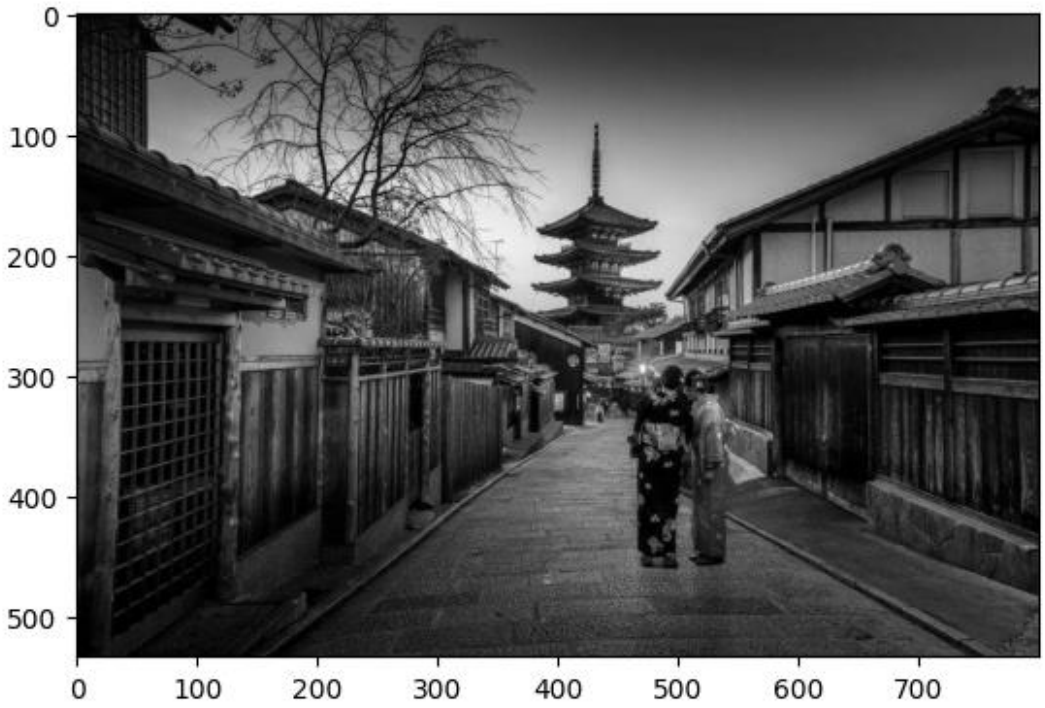
Στις εικόνες που παράγουμε παρατηρούμε, όπως και πριν, ότι για μικρές τιμές σ έχουμε μικρότερο bandwidth του πυρήνα και για μεγάλες το αντίστροφο. Αυτό σημαίνει ότι όσο μεγαλώνει το σ, τόσο σημαντικότερη είναι η συνεισφορά των εντάσεων γειτονικών pixels σε κάθε pixel, άρα τόσο χάνεται η πληροφορία. Παράλληλα, όμως, το φίλτρο γίνεται πιο εύρωστο, αφού μειώνει το θόρυβο και γίνεται πιο ευαίσθητο σε ακμές που ταιριάζουν στην κλίμακάτου

Ακόμα, η τιμή κατωφλίου θ είναι μια ακόμα παράμετρος για το πόσο ευαίσθητο είναι το φίλτρο στις ακμές. Για μικρές τιμές θ έχουμε μικρότερη ανεκτικότητα στο θόρυβο.

Για μια ακόμα φορά, είναι σημαντικό να επιλέγουμε παραμέτρους ανάλογα με τα χαρακτηριστικά της εικόνας εισόδου.

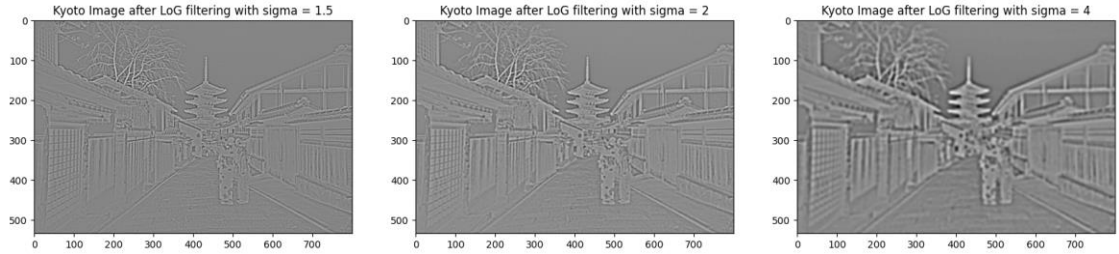
```
kyoto = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
kyoto_norm = kyoto/255
plt.imshow(kyoto, cmap = "gray")
```

```
<matplotlib.image.AxesImage at 0x7fab3f1f88e0>
```



Υπολογίζουμε τις εξόδους των γραμμικών φίλτρων για διάφορες τιμές σ.

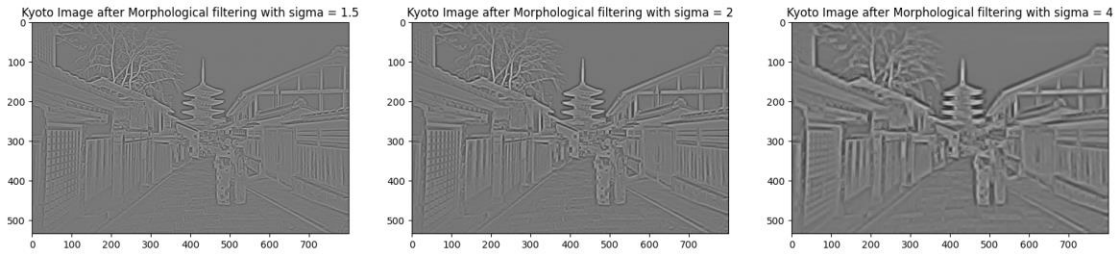
```
LoG1_5 = laplacian_of_gaussian(1.5)
LoG2 = laplacian_of_gaussian(2)
LoG4 = laplacian_of_gaussian(4)

kyotoLoG1_5 = cv2.filter2D(kyoto_norm, -1, LoG1_5)
kyotoLoG2 = cv2.filter2D(kyoto_norm, -1, LoG2)
kyotoLoG4 = cv2.filter2D(kyoto_norm, -1, LoG4)

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 3, figsize=(20, 20))

# Display the first image in the first column
axs[0].imshow(kyotoLoG1_5, cmap='gray')
axs[0].set_title('Kyoto Image after LoG filtering with sigma = 1.5')

# Display the second image in the second column
axs[1].imshow(kyotoLoG2, cmap='gray')
axs[1].set_title('Kyoto Image after LoG filtering with sigma = 2')

# Display the third image in the third column
axs[2].imshow(kyotoLoG4, cmap='gray')
axs[2].set_title('Kyoto Image after LoG filtering with sigma = 4')

# Show the figure
plt.show()
```



Υπολογίζουμε τις εξόδους των μη γραμμικών φίλτρων για διάφορες τιμές σ.

```
kern = np.array([[0,1,0],
                 [1,1,1],
                 [0,1,0]], dtype=np.uint8)

gaussian1_5 = gaussianKernel(1.5)
gaussian2 = gaussianKernel(2)
gaussian4 = gaussianKernel(4)

kyoto_gauss1_5 = cv2.filter2D(kyoto_norm, -1, gaussian1_5)

kyoto_gauss2 = cv2.filter2D(kyoto_norm, -1, gaussian2)
```

```
kyoto_gauss4 = cv2.filter2D(kyoto_norm, -1, gaussian4)

dilated_kyoto0_s1_5 = cv2.dilate(kyoto_gauss1_5, kern)
dilated_kyoto0_s2 = cv2.dilate(kyoto_gauss2, kern)
dilated_kyoto0_s4 = cv2.dilate(kyoto_gauss4, kern)

eroded_kyoto_s1_5 = cv2.erode(kyoto_gauss1_5, kern)
eroded_kyoto_s2 =   cv2.erode(kyoto_gauss2, kern)
eroded_kyoto_s4 =   cv2.erode(kyoto_gauss4, kern)

kyoto_morph_s1_5 = dilated_kyoto0_s1_5 + eroded_kyoto_s1_5 - 2*kyoto_gauss1_5
kyoto_morph_s2 = dilated_kyoto0_s2 + eroded_kyoto_s2 - 2*kyoto_gauss2
kyoto_morph_s4 = dilated_kyoto0_s4 + eroded_kyoto_s4 - 2*kyoto_gauss4

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 3, figsize=(20, 20))

# Display the first image in the first column
axs[0].imshow(kyoto_morph_s1_5, cmap='gray')
axs[0].set_title('Kyoto Image after Morphological filtering with sigma =
1.5')

# Display the second image in the second column
axs[1].imshow(kyoto_morph_s2, cmap='gray')
axs[1].set_title('Kyoto Image after Morphological filtering with sigma = 2')

# Display the third image in the third column
axs[2].imshow(kyoto_morph_s4, cmap='gray')
axs[2].set_title('Kyoto Image after Morphological filtering with sigma = 4')

# Show the figure
plt.show()
```



1.4.2 Ποσοτική Αξιολόγηση των Αποτελεσμάτων Ανίχνευσης Ακμών

Παρατηρούμε στα αποτελέσματα τα εξής:

Η επίδοση του μορφολογικού φίλτρου είναι καλύτερη από εκείνη του LoG.

Μικρό σ σημαίνει μεγαλύτερη λεπτομέρεια αλλά και περισσότερος θόρυβος (Ένα pixel αναγνωρίζεται εύκολα ως ακμή).

Μεγάλο σ σημαίνει μικρότερη λεπτομέρεια και μεγαλύτερη ανεκτικότητα στο θόρυβο (Οι πιο σημαντικές ακμές "περνούν" από το φίλτρο).

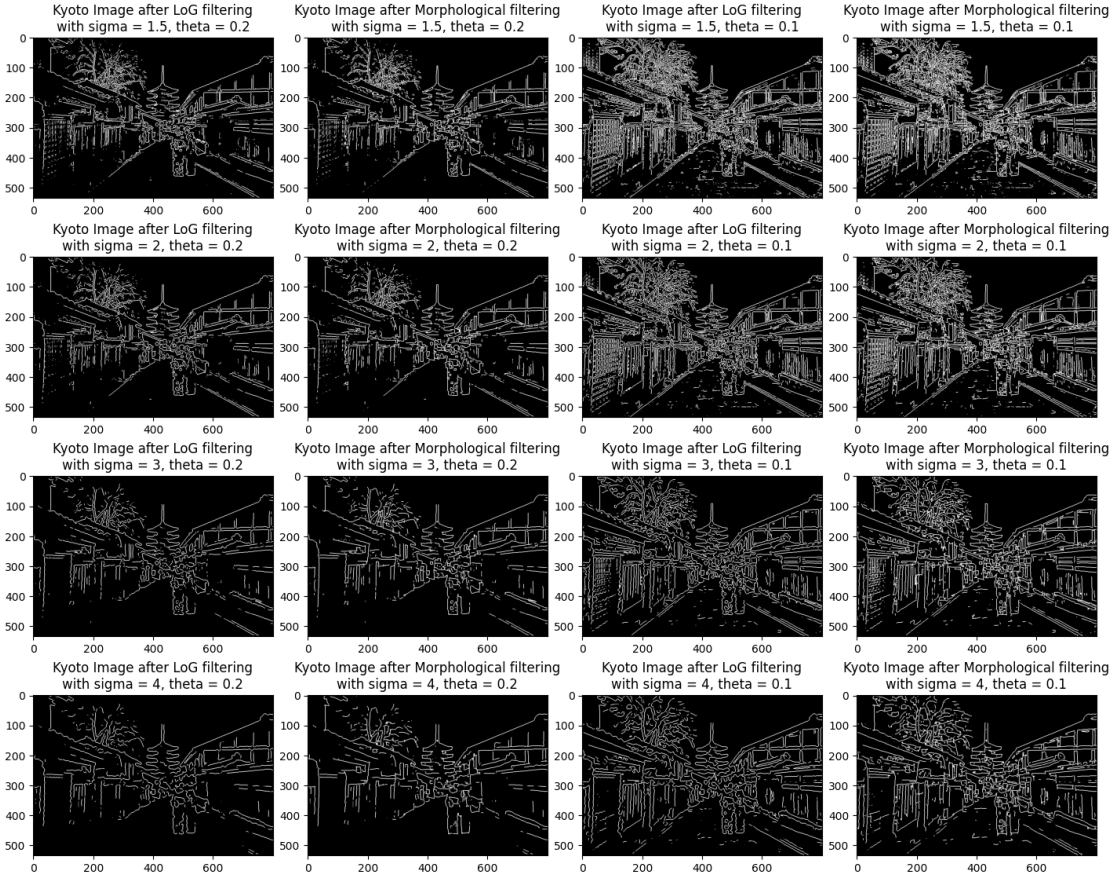
Μικρό θ σημαίνει επίσης μεγαλύτερη λεπτομέρεια, αφού με μεγαλύτερη ευκολία αναγνωρίζεται ένα pixel ως ακμή, ακόμα και αν δε θεωρείται τόσο σημαντική ακμή.

Για τον αντίθετο λόγο με το (4), μεγάλο θ σημαίνει μικρότερη λεπτομέρεια.

```
kyoto_edge_Log_sigma1_5 = edgeDetect(kyoto_norm, sigma = 1.5, theta = 0.2,
method = "LoG")
kyoto_edge_Morph_sigma1_5 = edgeDetect(kyoto_norm, sigma = 1.5, theta = 0.2,
method = "Morphological")

kyoto_edge_Log_sigma2 = edgeDetect(kyoto_norm, sigma = 2, theta = 0.2, method
= "LoG")
kyoto_edge_Morph_sigma2 = edgeDetect(kyoto_norm, sigma = 2, theta = 0.2,
method = "Morphological")

kyoto_edge_Log_sigma3 = edgeDetect(kyoto_norm, sigma = 3, theta = 0.2, method
= "LoG")
kyoto_edge_Morph_sigma3 = edgeDetect(kyoto_norm, sigma = 3, theta = 0.2,
method = "Morphological")

kyoto_edge_Log_sigma4 = edgeDetect(kyoto_norm, sigma = 4, theta = 0.2, method
= "LoG")
kyoto_edge_Morph_sigma4 = edgeDetect(kyoto_norm, sigma = 4, theta = 0.2,
method = "Morphological")

kyoto_edge_Log_sigma1_5_1 = edgeDetect(kyoto_norm, sigma = 1.5, theta = 0.1,
method = "LoG")
kyoto_edge_Morph_sigma1_5_1 = edgeDetect(kyoto_norm, sigma = 1.5, theta =
0.1, method = "Morphological")

kyoto_edge_Log_sigma2_1 = edgeDetect(kyoto_norm, sigma = 2, theta = 0.1,
method = "LoG")
kyoto_edge_Morph_sigma2_1 = edgeDetect(kyoto_norm, sigma = 2, theta = 0.1,
method = "Morphological")

kyoto_edge_Log_sigma3_1 = edgeDetect(kyoto_norm, sigma = 3, theta = 0.1,
method = "LoG")
kyoto_edge_Morph_sigma3_1 = edgeDetect(kyoto_norm, sigma = 3, theta = 0.1,
method = "Morphological")

kyoto_edge_Log_sigma4_1 = edgeDetect(kyoto_norm, sigma = 4, theta = 0.1,
method = "LoG")
```

```
kyoto_edge_Morph_sigma4_1 = edgeDetect(kyoto_norm, sigma = 4, theta = 0.1,
method = "Morphological")

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(4, 4, figsize=(14, 14), layout = "compressed")

# Display the first image in the first column
axs[0][0].imshow(kyoto_edge_Log_sigma1_5, cmap='gray')
axs[0][0].set_title('Kyoto Image after LoG filtering\nwith sigma = 1.5, theta
= 0.2')

# Display the second image in the second column
axs[1][0].imshow(kyoto_edge_Log_sigma2, cmap='gray')
axs[1][0].set_title('Kyoto Image after LoG filtering\nwith sigma = 2, theta =
0.2')

# Display the third image in the third column
axs[2][0].imshow(kyoto_edge_Log_sigma3, cmap='gray')
axs[2][0].set_title('Kyoto Image after LoG filtering\nwith sigma = 3, theta =
0.2')

# Display the third image in the fourth column
axs[3][0].imshow(kyoto_edge_Log_sigma4, cmap='gray')
axs[3][0].set_title('Kyoto Image after LoG filtering\nwith sigma = 4, theta =
0.2')
# === ===#
# Display the first image in the first column
axs[0][1].imshow(kyoto_edge_Morph_sigma1_5, cmap='gray')
axs[0][1].set_title('Kyoto Image after Morphological filtering\nwith sigma =
1.5, theta = 0.2')

# Display the second image in the second column
axs[1][1].imshow(kyoto_edge_Morph_sigma2, cmap='gray')
axs[1][1].set_title('Kyoto Image after Morphological filtering\nwith sigma =
2, theta = 0.2')

# Display the third image in the third column
axs[2][1].imshow(kyoto_edge_Morph_sigma3, cmap='gray')
axs[2][1].set_title('Kyoto Image after Morphological filtering\nwith sigma =
3, theta = 0.2')

# Display the third image in the fourth column
axs[3][1].imshow(kyoto_edge_Morph_sigma4, cmap='gray')
axs[3][1].set_title('Kyoto Image after Morphological filtering\nwith sigma =
4, theta = 0.2')
#
================================================================================
#
```

```
# Display the first image in the first column
axs[0][2].imshow(kyoto_edge_Log_sigma1_5_1, cmap='gray')
axs[0][2].set_title('Kyoto Image after LoG filtering\nwith sigma = 1.5, theta
= 0.1')

# Display the second image in the second column
axs[1][2].imshow(kyoto_edge_Log_sigma2_1, cmap='gray')
axs[1][2].set_title('Kyoto Image after LoG filtering\nwith sigma = 2, theta =
0.1')

# Display the third image in the third column
axs[2][2].imshow(kyoto_edge_Log_sigma3_1, cmap='gray')
axs[2][2].set_title('Kyoto Image after LoG filtering\nwith sigma = 3, theta =
0.1')

# Display the third image in the fourth column
axs[3][2].imshow(kyoto_edge_Log_sigma4_1, cmap='gray')
axs[3][2].set_title('Kyoto Image after LoG filtering\nwith sigma = 4, theta =
0.1')
# === ===#
# Display the first image in the first column
axs[0][3].imshow(kyoto_edge_Morph_sigma1_5_1, cmap='gray')
axs[0][3].set_title('Kyoto Image after Morphological filtering\nwith sigma =
1.5, theta = 0.1')

# Display the second image in the second column
axs[1][3].imshow(kyoto_edge_Morph_sigma2_1, cmap='gray')
axs[1][3].set_title('Kyoto Image after Morphological filtering\nwith sigma =
2, theta = 0.1')

# Display the third image in the third column
axs[2][3].imshow(kyoto_edge_Morph_sigma3_1, cmap='gray')
axs[2][3].set_title('Kyoto Image after Morphological filtering\nwith sigma =
3, theta = 0.1')

# Display the third image in the fourth column
axs[3][3].imshow(kyoto_edge_Morph_sigma4_1, cmap='gray')
axs[3][3].set_title('Kyoto Image after Morphological filtering\nwith sigma =
4, theta = 0.1')

# Show the figure
plt.show()
```


Kyoto Image after LoG filtering with sigma = 1.5, theta = 0.2
Kyoto Image after Morphological filtering with sigma = 1.5, theta = 0.2
Kyoto Image after LoG filtering with sigma = 1.5, theta = 0.1
Kyoto Image after Morphological filtering with sigma = 1.5, theta = 0.1
Kyoto Image after LoG filtering with sigma = 2, theta = 0.2
Kyoto Image after Morphological filtering with sigma = 2, theta = 0.2
Kyoto Image after LoG filtering with sigma = 2, theta = 0.1
Kyoto Image after Morphological filtering with sigma = 2, theta = 0.1
Kyoto Image after LoG filtering with sigma = 3, theta = 0.2
Kyoto Image after Morphological filtering with sigma = 3, theta = 0.2
Kyoto Image after LoG filtering with sigma = 3, theta = 0.1
Kyoto Image after Morphological filtering with sigma = 3, theta = 0.1
Kyoto Image after LoG filtering with sigma = 4, theta = 0.2
Kyoto Image after Morphological filtering with sigma = 4, theta = 0.2
Kyoto Image after LoG filtering with sigma = 4, theta = 0.1
Kyoto Image after Morphological filtering with sigma = 4, theta = 0.1

Μέρος 2: Ανίχνευση Σημείων Ενδιαφέροντος (Interest Point Detection)

Στο δεύτερο μέρος θα επικεντρωθούμε στον εντοπισμό σημείων ενδιαφέροντος, όπως είναι οι γωνίες. Θα υλοποιήσουμε τη μέθοδο Harris-Stephens για τον εντοπισμό γωνιών και ύστερα τη Harris-Laplacian για multiscale ανίχνευση γωνιών.

Ακόμα, στο μέρος 2 γίνεται χρήση των συναρτήσεων του αρχείου utils.py για την οπτικοποίηση των σημείων ενδιαφέροντος που θα βρούμε

2.1. Ανίχνευση Γωνιών

2.1.1. Υπολογισμός του τανυστή της εικόνας

Κανονικοποιούμε την εικόνα και βρίσκουμε την εξομαλυμένη εικόνα με τη βοήθεια ενός Gaussian πυρήνα. Ύστερα υπολογίζουμε τον τανυστή J ως εξής:

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

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

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

Όπου $I_s = G_\sigma * I$ και είναι συνολικά:

$$\mathrm{J} = \begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix}$$

```
kyoto_norm = kyoto.astype(np.float)/255
sigma = 2
gaussian_r2_5 = gaussianKernel(2.5)
gaussian_s2 = gaussianKernel(2)

Is2 = cv2.filter2D(kyoto_norm, -1, gaussian_s2)

[Isx, Isy] = np.gradient(Is2)

J1 = cv2.filter2D(Isx*Isx, -1, gaussian_r2_5)
J2 = cv2.filter2D(Isx*Isy, -1, gaussian_r2_5)
J3 = cv2.filter2D(Isy*Isy, -1, gaussian_r2_5)
```

2.1.2. Υπολογισμός των ιδιοτιμών του τανυστή σε κάθε pixel

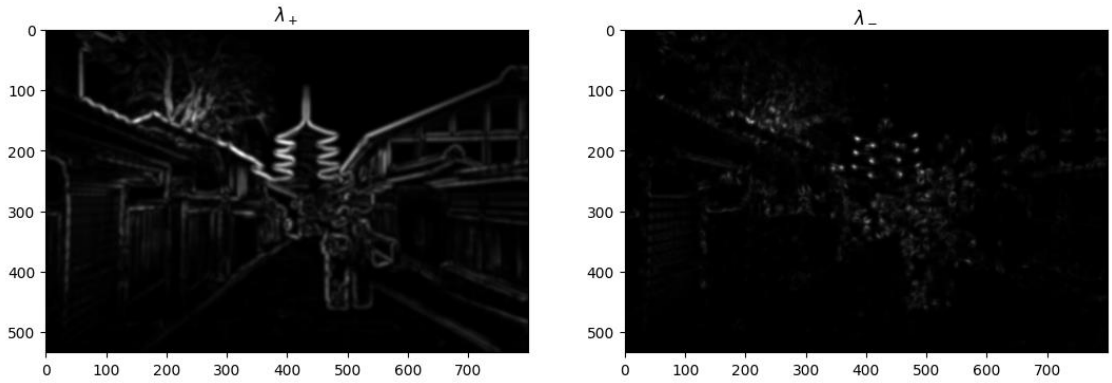
Υπολογίζουμε τις ιδιοτιμές του τανυστή χρησιμοποιώντας τον τύπο:

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

Ο τανυστής περιγράφει τη μορφολογία της εικόνας σε μια γειτονιά γύρω από κάθε pixel. Οι ιδιοτιμές αντιστοιχούν στις principal curvatures [2] της εικόνας, δηλαδή στο μέγιστο και ελάχιστο εκείνης της γειτονιάς. Αν κανείς φανταστεί την εικόνα ως αντιστοίχιση από το επίπεδο των pixels (x, y) στην έντασή τους $I(x, y)$ καταλήγει σε μια 3D επιφάνεια. Τότε, οι principal curvatures μετρούν πόσο στρεβλώνεται η επιφάνεια γύρω από το σημείο (x, y)

[1]: https://en.wikipedia.org/wiki/Principal_curvature

```
lambdaPlus = (J1 + J3 + np.sqrt((J1 - J3)**2 + 4*J2**2))/2
lambdaMinus = (J1 + J3 - np.sqrt((J1 - J3)**2 + 4*J2**2))/2

# Create a figure with 1 row and 2 columns
fig, axs = plt.subplots(1, 2, figsize=(13, 13))

# Display the first image in the first column
axs[0].imshow(lambdaPlus, cmap='gray')
axs[0].set_title('$\lambda_+$')

# Display the second image in the second column
axs[1].imshow(lambdaMinus, cmap='gray')
axs[1].set_title('$\lambda_-$')

# Show the figure
plt.show()
```



Ορίζουμε τις συναρτήσεις του αρχείου utils.py

```
import sys
import numpy as np
```

```
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

def interest_points_visualization(I_, kp_data_, ax=None):
    '''
    Plot keypoints chosen by detectos on image.
    Args:
        I_: Image (if colored, make sure it is in RGB and not BGR).
        kp_data_: Nx3 array, as described in assignment.
        ax: Matplotlib axis to plot on (if None, a new Axes object is
created).
    Returns:
        ax: Matplotlib axis where the image was plotted.
    '''
    try:
        I = np.array(I_)
        kp_data = np.array(kp_data_)
    except:
        print('Conversion to numpy arrays failed, check if the inputs (image
and keypoints) are in the required format.')
        exit(2)

    try:
        assert(len(I.shape) == 2 or (len(I.shape) == 3 and I.shape[2] == 3))
    except AssertionError as e:
        print('interest_points_visualization: Image must be either a 2D
matrix or a 3D matrix with the last dimension having size equal to 3.',
file=sys.stderr)
        exit(2)

    try:
        assert(len(kp_data.shape) == 2 and kp_data.shape[1] == 3)
    except AssertionError as e:
        print('interest_points_visualization: kp_data must be a 2D matrix
with 3 columns.', file=sys.stderr)
        exit(2)

    if ax is None:
        _, ax = plt.subplots()

    ax.set_aspect('equal')
    ax.imshow(I)
    ax.tick_params(bottom=False, left=False, labelbottom=False,
labelleft=False)

    for i in range(len(kp_data)):
        x, y, sigma = kp_data[i]
        circ = Circle((x, y), 3*sigma, edgecolor='y', fill=False,
linewidth=1)
```

```
            ax.add_patch(circ)

    return ax

def disk_strel(n):
    '''
        Return a structural element, which is a disk of radius n.
    '''
    r = int(np.round(n))
    d = 2*r+1
    x = np.arange(d) - r
    y = np.arange(d) - r
    x, y = np.meshgrid(x,y)
    strel = x**2 + y**2 <= r**2
    return strel.astype(np.uint8)
```

Ορίζουμε:

1. Την ευαισθησία στη γωνιότητα: k
2. Το κατώφλι για την απόφαση γωνίας: $\theta_c orn$
3. Την τυπική απόκλιση σ

Ώστε να υπολογίσουμε:

1. Το κριτήριο γωνιότητας: $R(x,y) = \lambda_- \lambda_+ - k \cdot (\lambda_- + \lambda_+)^2$
2. Τις συνθήκες:

$$(x,y) \in \left(R \oplus B_{square}\right) \wedge \left(R(x,y) > \theta_{corn} * R_{max}\right)$$

Και τελικά τις συναληθεύουμε. Κρατάμε έναν πίνακα για τις γωνίες που βρήκαμε και τις οπτικοποιούμε.

```
k = 0.05
thetaCorn = 0.005
sigma = 2

# Read the input image in BGR and grayscale
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)

# Calculate the eigenvalues
R = lambdaMinus*lambdaPlus - k*(lambdaMinus + lambdaPlus)**2

# Size of the disk
ns = np.ceil(3*sigma)*2 + 1

# Create the disk
B_square = disk_strel(ns)
```

```
# Both conditions must hold
condition1 = (R == cv2.dilate(R, B_square))
condition2 = (R > thetaCorn*np.amax(R))
cornersDetected = (condition1*condition2).astype(np.uint8)

# Obtain the coordinates of corner points
imgPoints = np.nonzero(cornersDetected == 1)

# Convert the coordinates to the format expected by
interest_points_visualization function and store them in kp_data list
kp_data = []
for i in range(np.sum(cornersDetected)):
    kp_data.append([imgPoints[1][i],imgPoints[0][i],2])

# Convert the BGR image to RGB format and visualize the detected corner
points on it using interest_points_visualization function
img = cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB)
temp = interest_points_visualization(img, kp_data)
```

```
def corner_detection(Img_BGR, Img_Grey, sigma, p, theta_corn, k, show =
False):
    kyoto_norm = kyoto.astype(np.float)/255
    sigma = 2
    gaussian_r2_5 = gaussianKernel(2.5)
    gaussian_s2 = gaussianKernel(2)

    Is2 = cv2.filter2D(kyoto_norm, -1, gaussian_s2)

    [Isx, Isy] = np.gradient(Is2)
```

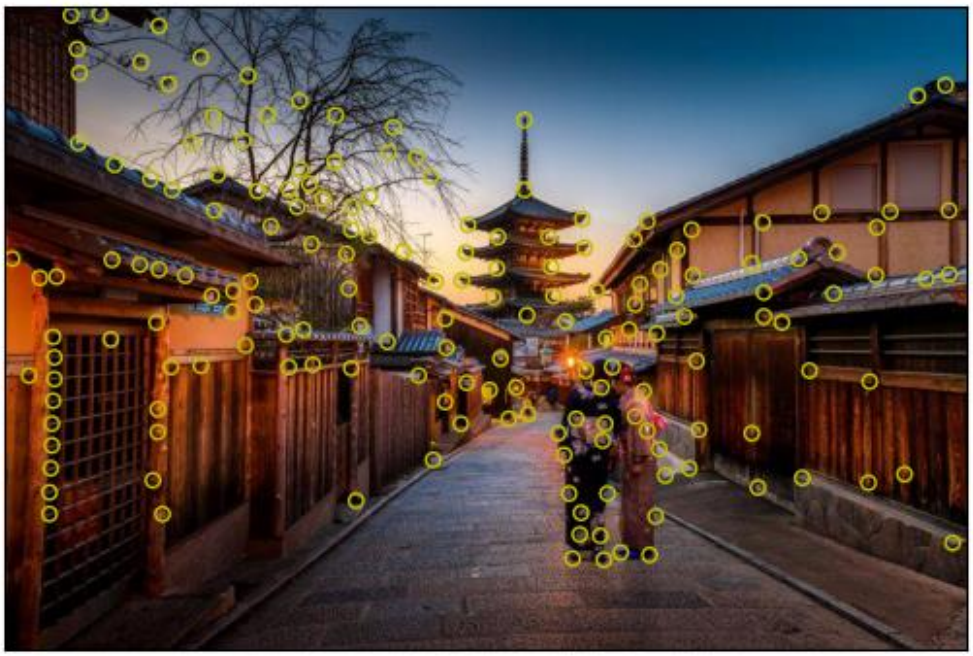
```
    J1 = cv2.filter2D(Isx*Isx, -1, gaussian_r2_5)
    J2 = cv2.filter2D(Isx*Isy, -1, gaussian_r2_5)
    J3 = cv2.filter2D(Isy*Isy, -1, gaussian_r2_5)

    lambdaPlus = (J1 + J3 + np.sqrt((J1 - J3)**2 + 4*J2**2))/2
    lambdaMinus = (J1 + J3 - np.sqrt((J1 - J3)**2 + 4*J2**2))/2

    R = lambdaMinus*lambdaPlus - k*(lambdaMinus + lambdaPlus)**2
    ns = np.ceil(3*sigma)*2 + 1
    B_square = disk_strel(ns)
    condition1 = (R == cv2.dilate(R,B_square))
    condition2 = (R > thetaCorn*np.amax(R))
    cornersDetected = (condition1*condition2).astype(np.uint8) # or &

    imgPoints = np.nonzero(cornersDetected == 1)                    #find the
imgPoints of all elements==1
    kp_data = []
    for i in range(np.sum(cornersDetected)):
        kp_data.append([imgPoints[1][i],imgPoints[0][i],2])  #store them the
way interest_points_visualization wants them

    if show:
        img = cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB)
        interest_points_visualization(img, kp_data)  #call only if you want
to show points on image
    return cornersDetected

def edgeDetection(ImgBGR, ImgGrey, sigma, r, thetaCorn, k):
    gaussian_r = gaussianKernel(r)
    gaussian_s = gaussianKernel(sigma)

    Is2 = cv2.filter2D(kyoto_norm, -1, gaussian_s)

    [Isx, Isy] = np.gradient(Is2)

    J1 = cv2.filter2D(Isx*Isx, -1, gaussian_r2_5)
    J2 = cv2.filter2D(Isx*Isy, -1, gaussian_r2_5)
    J3 = cv2.filter2D(Isy*Isy, -1, gaussian_r2_5)
    lambdaPlus = (J1 + J3 + np.sqrt((J1 - J3)**2 + 4*J2**2))/2
    lambdaMinus = (J1 + J3 - np.sqrt((J1 - J3)**2 + 4*J2**2))/2

    R = lambdaMinus*lambdaPlus - k*(lambdaMinus + lambdaPlus)**2
    ns = np.ceil(3*sigma)*2 + 1
    B_square = disk_strel(ns)
    condition1 = (R == cv2.dilate(R,B_square))
    condition2 = (R > thetaCorn*np.amax(R))
    cornersDetected = (condition1*condition2).astype(np.uint8)
```

```
    imgPoints = np.nonzero(cornersDetected == 1)
    kp_data = []
    for i in range(np.sum(cornersDetected)):
        kp_data.append([imgPoints[1][i],imgPoints[0][i],2])

    img = cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB)
    temp = interest_points_visualization(img, kp_data)

    return temp
```

Στην πιο κάτω εικόνα παρατηρούμε ότι ο αλγόριθμος έχει καταφέρει να εντοπίσει αρκετές γωνίες, με μερικα λάθη όμως, όπως για παράδειγμα πάνω στους ανθρώπους της εικόνας.

```
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
edgeDetection(ImgBGR, ImgGrey, sigma = 2, r = 2.5, thetaCorn = 0.005, k =
0.05)
```

```
<Axes: >
```



Παρατήρηση:

Στις επόμενες 3 εικόνες μειώσαμε ξεχωριστά τις τιμές των σ, θ_{corn}, k. Βλέπουμε ότι:

1. Για μικρότερο σ ανιχνεύσαμε περισσότερες γωνίες. Αυτό το αποδίδουμε στο γεγονός ότι το μικρότερο bandwidth του Gaussian φίλτρου δεν εξομαλύνει αρκετά το θόρυβο και επίσης επιτρέπει να "περάσουν" περισσότερες λεπτομέρειες της εικόνας με αποτέλεσμα μεγαλύτερη ευαισθησία στις απότομες μεταβολές (gradient) της έντασης των pixels.

2. Για μικρότερο θ_{corn} ανιχνεύσαμε περισσότερες γωνίες. Αυτό συμβαίνει απλά επειδή απαιτείται μικρότερη τιμή του κριτηρίου γωνιότητας $R(x, y)$ για να θεωρηθεί ένα pixel τμήμα γωνίας.
3. Για μικρότερο k ανιχνεύσαμε και πάλι περισσότερες γωνίες. Όπως και στο (2), μιρκότερο k σημαίνει μεγαλύτερο $R(x, y)$, οπότε για τα ίδια pixels έχουμε μεγαλύτερη "τιμή γωνιότητας".

```
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
edgeDetection(ImgBGR, ImgGrey, sigma = 1, r = 2.5, thetaCorn = 0.005, k =
0.05)
```

```
<Axes: >
```

```
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
edgeDetection(ImgBGR, ImgGrey, sigma = 2, r = 2.5, thetaCorn = 0.001, k =
0.05)
```

```
<Axes: >
```

```
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
edgeDetection(ImgBGR, ImgGrey, sigma = 2, r = 2.5, thetaCorn = 0.005, k =
0.01)
```

```
<Axes: >
```

```
ImgBGR = cv2.imread('kyoto_edges.jpg')
ImgGrey = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
```

```
# define parameter value lists
sigma_values = [1.5, 2, 2.5]
r_values = [1.5, 2, 2.5]
thetaCorn_values = [0.005, 0.01, 0.02, 0.05, 0.1]
k_values = [0.05, 0.1, 0.15]

# create directory for saving output images
output_dir = "./cvLab1/kyotoEdges/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# loop over all combinations of parameter values and plot the results
for sigma in sigma_values:
    for r in r_values:
        for thetaCorn in thetaCorn_values:
            for k in k_values:
                edgeDetection(ImgBGR, ImgGrey, sigma, r, thetaCorn, k)
                # save the plot with the specified filename
                filename = output_dir + "kyoto_s" +
"{:03d}".format(int(sigma*10)) + "_r" + "{:03d}".format(int(r*10)) + "_theta"
+ "{:03d}".format(int(thetaCorn*1000)) + "_k" + "{:03d}".format(int(k*100)) +
".jpg"
                plt.savefig(filename)
                # close the plot to save memory
                plt.close()

download = False
if download:
    # create a ZIP archive of the kyotoEdges folder
    !zip -r kyotoEdges.zip ./cvLab1/kyotoEdges/

    # download the ZIP archive
    files.download("kyotoEdges.zip")
```

Φτιάχνουμε ένα GIF file με στιγμιότυπα τις διάφορες ακμές καθώς μεταβάλλονται τα $\sigma, \rho, \theta_{corn}, k$

```
# define input and output filenames
input_dir = "./cvLab1/kyotoEdges/"
output_filename = "kyotoEdges.gif"

# get list of all image filenames in input directory
filenames = sorted(os.listdir(input_dir))

# create list of image arrays from the input files
images = []
for filename in filenames:
    if filename.endswith(".jpg"):
        filepath = os.path.join(input_dir, filename)
```

```
        image = imageio.imread(filepath)
        images.append(image)

# create the GIF animation with a frame duration of 0.2 seconds
duration = 0.2
imageio.mimsave(output_filename, images, duration=duration)

from IPython.display import Image
Image(filename='kyotoEdges.gif')
```

2.2. Πολυκλιμακωτή Ανίχνευση Γωνιών

Σε αυτό το βήμα, θα υλοποιήσουμε τον Harris-Laplacian αλγόριθμο για την πολυκλιμακωτή ανίχνευση γωνιών

2.2.1. Πολυκλιμακωτή Ανίχνευση Γωνιών

Εφαρμόζουμε τον προηγούμενο αλγόριθμο για τις διάφορες κλίμακες:

$$(\sigma_0, \sigma_1, \ldots, \sigma_{N-1}) = (s^0\sigma_0, s^1\sigma_0, \ldots, s^{N-1}\sigma_0)$$
$$(\rho_0, \rho_1, \ldots, \rho_{N-1}) = (s^0\rho_0, s^1\rho_0, \ldots, s^{N-1}\rho_0)$$

```
s = 1.5
s0 = 2
r0 = 2.5
N = 4
thetaCorn = 0.1

sigmas = []
rs = []
cornersDetected = []
L = []
ImgGreyNorm = ImgGrey.astype(np.float)/255

for i in range(N):
    sigmas.append(s**i*s0)
    rs.append(s**i*r0)
    cornersDetected.append(corner_detection(ImgBGR, ImgGrey, sigmas[i],
rs[i], thetaCorn, k))

    gaussian = gaussianKernel(sigmas[i])
    LoG = laplacian_of_gaussian(sigmas[i])

    L.append(sigmas[i]**2*np.abs(cv2.filter2D(ImgGreyNorm, -1, LoG)))
```

2.2.2. Αυτόματη Επιλογή Χαρακτηριστικής Κλίμακας

Τώρα, διατρέχουμε τα pixels και απορρίπτουμε αυτά που δε μεγιστοποιούν τη $LoG\ metric$.

Σημείωση: Χρησιμοποιούμε μια κοινή τακτική στην επεξεργασία εικόνας κατά την οποία θεωρούμε την εικόνα συνεχή, δηλαδή ότι οι απέναντι πλευρές της είναι ενωμένες. Έτσι λύνουμε το πρόβλημα της ασυνέχειας (θα μπορούσαμε να χρησιμοποιήσουμε και άλλες τεχνικές, όπως padding)

```
imgPoints = []
kp_data2 = []

for i in range(N):
    imgPoints.append(np.nonzero(cornersDetected[i] == 1))
```

```
    for j in range(np.sum(cornersDetected[i])):
        curr = L[i][imgPoints[i][0][j]][imgPoints[i][1][j]]

        prev = L[(i - 1) % N][imgPoints[i][0][j]][imgPoints[i][1][j]]
        next =  L[(i + 1) % N][imgPoints[i][0][j]][imgPoints[i][1][j]]

        if (curr > prev) & (curr > next):

kp_data2.append([imgPoints[i][1][j],imgPoints[i][0][j],sigmas[i]])

img = cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB)
interest_points_visualization(img, kp_data2)

def harrisLaplacian(ImgBGR, ImgGrey, N=4, s0=2, r0=2.5, s=1.5, thetaCorn=0.2,
k=0.1):
    sigmas = [s ** i * s0 for i in range(N)]
    rs = [s ** i * r0 for i in range(N)]
    cornersDetected = [corner_detection(ImgBGR, ImgGrey, sigmas[i], rs[i],
thetaCorn, k) for i in range(N)]
    L = [sigmas[i] ** 2 * np.abs(cv2.filter2D(ImgGrey.astype(np.float)/255, -
1, laplacian_of_gaussian(sigmas[i]))) for i in range(N)]

    kp_data2 = []
    for i in range(N):
        imgPoints = np.nonzero(cornersDetected[i] == 1)
        for j in range(len(imgPoints[0])):
            curr = L[i][imgPoints[0][j], imgPoints[1][j]]

            prev = L[(i - 1) % N][imgPoints[0][j], imgPoints[1][j]]
            next = L[(i + 1) % N][imgPoints[0][j], imgPoints[1][j]]

            if (curr > prev) and (curr > next):
                kp_data2.append([imgPoints[1][j], imgPoints[0][j],
sigmas[i]])

    img = cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB)
    interest_points_visualization(img, kp_data2)

harrisLaplacian(ImgBGR, ImgGrey, N = 4, s0 = 2, r0 = 2.5, s = 1.5, thetaCorn =
0.005, k = 0.05)
```

2.3. Ανίχνευση Blobs

2.3.1. Υπολογισμός της Ορίζουσας του Hessian Matrix

Στο βήμα αυτό θα ανιχνεύσουμε blobs. Τα blobs είναι περιοχές στην εικόνα που διαφέρουν σημαντικά (σε ένταση pixel) από τα γειτονικά τους σημεία. Χρησιμοποιούμε τον Hessian πίνακα:

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

Ο πίνακας περιλαμβάνει πληροφορία σχετική με την τοπική καμπυλότητα της έντασης και η ορίζουσά του είναι ένα μέτρο αυτής

```
Img = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
Img = Img.astype(np.float)/255
sigma = 2
Theta_corn = 0.005
k = 0.05

gauss2D = gaussianKernel(sigma)
Is= cv2.filter2D(Img, -1, gauss2D)

[Lx,Ly] = np.gradient(Is)
[Lxx,Lxy] = np.gradient(Lx)
[Lyx,Lyy] = np.gradient(Ly)
R = Lxx*Lyy - Lxy*Lxy
```

2.3.2. Απόρριψη Σημείων Ενδιαφέροντος που δεν Πληρούν Συνθήκες

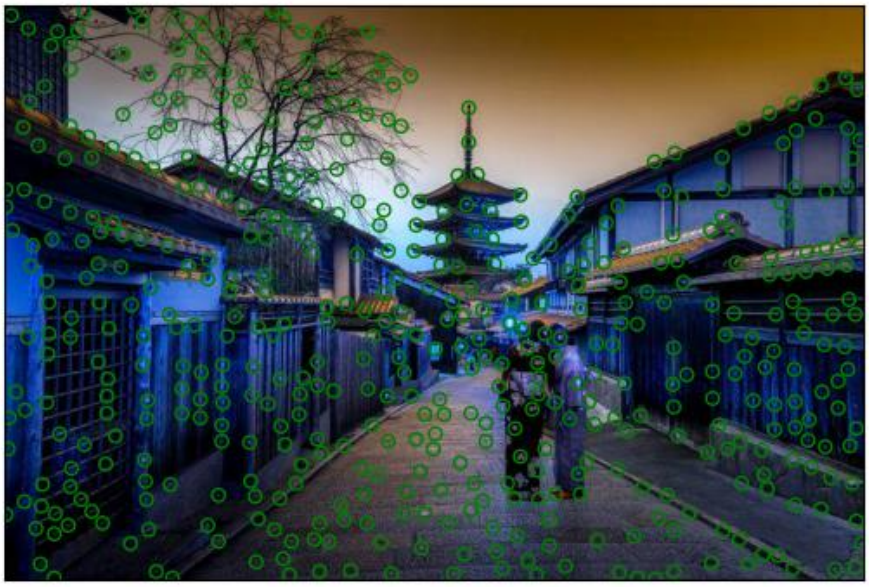
Θεωρούμε blobs τα σημεία ενδιαφέροντος που αντιστοιχούν σε τοπικά μέγιστα της ορίζουσας του πίνακα $H(x,y)$ και η τιμή τους ξεπερνά ένα κατώφλι (που βοηθά στην απόρριψη σημείων που πιθανά ανιχνεύτηκαν λόγω θορύβου - βέβαια το trade - off είναι ότι χάνονται και καλώς ανιχνευθέντα σημεία). Παρατηρούμε ότι η διαδικασία είναι πολύ όμοια με εκείνη του βήματος (2.2)

```
B_sq = disk_strel(ns)
Cond1 = ( R==cv2.dilate(R,B_sq) )
Cond2 = (R > Theta_corn*np.amax(R))
corners = (Cond1*Cond2).astype(np.uint8)

def blob_detect(Img, sigma = 2, Theta_corn = 0.005, k = 0.05):
    Img = Img.astype(np.float)/Img.max()
    ns = int(np.ceil(3*sigma)*2+1)
    gauss2D = gaussianKernel(sigma)
    Is= cv2.filter2D(Img, -1, gauss2D)

    [Lx,Ly] = np.gradient(Is)
    [Lxx,Lxy] = np.gradient(Lx)
    [Lyx,Lyy] = np.gradient(Ly)
```

```
    R = Lxx*Lyy - Lxy*Lxy

    Bsquare = disk_strel(ns)
    Cond1 = (R==cv2.dilate(R,Bsquare))
    Cond2 = (R > Theta_corn*np.amax(R))
    corners = (Cond1*Cond2).astype(np.uint8)

    coords = np.nonzero(corners == 1)

    kp_data = []
    for i in range(np.sum(corners)):
        kp_data.append([coords[1][i], coords[0][i], sigma])
    blobs = []
    for i in range(len(corners)):
        for j in range(len(corners[i])):
            if int(corners[i][j]) == 1:
                blobs.append([j, i, sigma])

    return np.array(blobs)
```

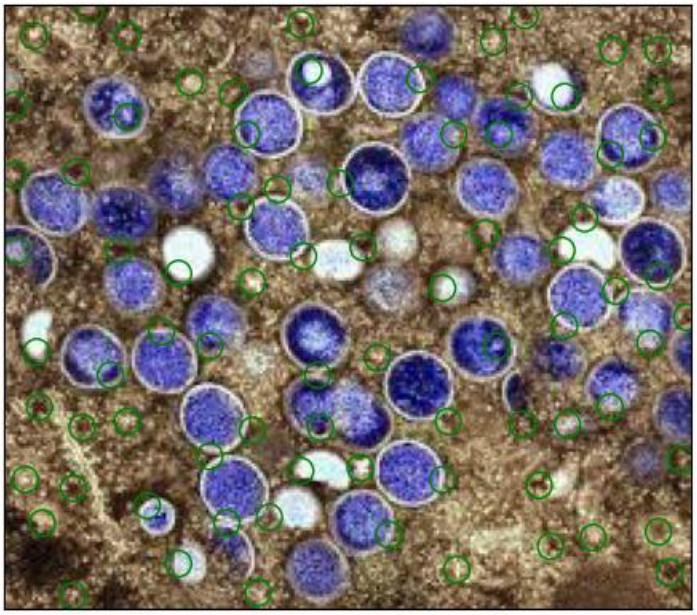
Μπορούμε να παρατηρήσουμε στις εικόνες ότι πράγματι, σημειωμένες είναι περιοχές που έχουν μεγάλη διαφορά σε ένταση pixel με τα γειτονικά τους σημεία. Για παράδειγμα, μπορεί κανείς να δει πως υπάρχουν πολλοί κύκλοι στις περιφέρειες των κυττάρων, όπου έχουμε απότομη μεταβολή έντασης. Μάλιστα, σε ένα κύτταρο (για $\sigma = 2$ και σε περισσότερα για $\sigma = 3$ έχουμε κύκλους να περιβάλλουν τελείως μικρότερες λευκές περιοχές, αφού το bandwidth (που σχετίζεται με το σ) επιτρέπει την ανίχνευσή των.

```
Img = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
ImgBGR = cv2.imread('kyoto_edges.jpg')
blobs = blob_detect(Img, 2, 0.005, 0.05)
interest_points_visualization(cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB), blobs)
```



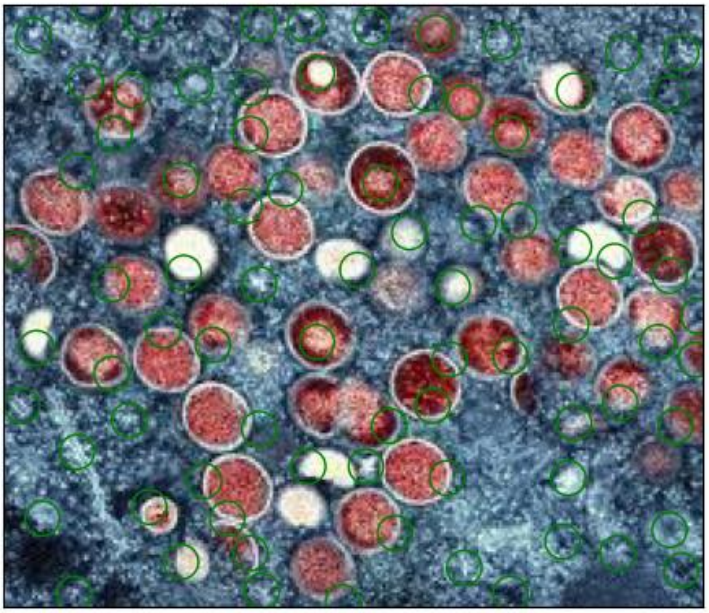
```
Img = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
ImgBGR = cv2.imread('cells.jpg')
x = blob_detect(Img, 1.5, 0.005, 0.05)
interest_points_visualization(cv2.cvtColor(ImgBGR, cv2.COLOR_BGR2RGB), x)
```

```
<AxesSubplot:>
```



```
Img = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('kyoto_edges.jpg')

ImgCells = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGRCells = cv2.imread('cells.jpg')

ImgUp = cv2.imread('up.png', cv2.IMREAD_GRAYSCALE)
Img_BGRUp = cv2.imread('up.png')

kp = blob_detect(Img, 3, 0.005, 0.05)
kpCells = blob_detect(ImgCells, 2, 0.005, 0.05)
kpUp = blob_detect(ImgUp, 3, 0.005, 0.05)

interest_points_visualization(cv2.cvtColor(Img_BGR, cv2.COLOR_BGR2RGB), kp)
interest_points_visualization(cv2.cvtColor(Img_BGRCells, cv2.COLOR_BGR2RGB),
kpCells)
interest_points_visualization(cv2.cvtColor(Img_BGRUp, cv2.COLOR_BGR2RGB),
kpUp)
```

```
<AxesSubplot:>
```



2.4. Πολυκλιμακωτή Ανίχνευση Blobs

Πρέπει να απορριφθούν ορισμένα blobs. Αυτό το κομμάτι δεν είναι έτοιμο ακόμα :(

```
sigma = 2
r = 2.5
k = 0.05
thetCorn = 0.005
s = 1.5
N = 4
multiscale_blobs = dict()
scales = [(s**i) * sigma for i in range(N)]
for i in range(N):
    multiscale_blobs[i] = blob_detect(Img, sigma = scales[i], Theta_corn =
0.005, k = 0.05)

if False:
    multiscale_blobs = []
    sigma = 2
    theta_blob = 0.005
    s = 1.5
    N = 4
    k = 0.05
    scales = [(s**i) * sigma for i in range(N)]
    for i in range(N):
        multiscale_blobs.append(blob_detect(Image, Img_BGR, sigma =
scales[i], Theta_corn = theta_blob, k = k))
    # np.shape(multiscale_blobs)

def multiscale_BlobDetect (Img, sigma, theta_blob, s, N, k):
    multiscale_blobs = []
    smoothedImgs = []
    LoGimg = []

    scales = [(s**i) * sigma for i in range(int(N))]
    # ns = [int(np.ceil(3*((s**i)*sigma))*2+1) for i in range(N)]
    for i in range(int(N)):
        gauss2D = gaussianKernel(sigma = scales[i])
        Is = cv2.filter2D(Img, -1, gauss2D)
        smoothedImgs.append(Is)
        IsGrad = np.gradient(Is)
        [Ixx, Ixy] = np.gradient(IsGrad[0])
        [Iyx, Iyy] = np.gradient(IsGrad[1])
        LoGimg.append(((s**i)*scales[i])**2*abs(Ixx + Iyy))

        multiscale_blobs.append(blob_detect(Img, sigma = scales[i],
Theta_corn = theta_blob, k = k))

    i = -1
    edges =[]
```

```
    for arr in multiscale_blobs:
        i+=1
        for lis in arr:
            curr = LoGimg[i][int(lis[1])][int(lis[0])]

            if i == 0:
                # print("In i == 0")
                nxt  = LoGimg[i+1][int(lis[1])][int(lis[0])]
                prev = nxt-0.1
            elif i == N-1:
                prev = LoGimg[i-1][int(lis[1])][int(lis[0])]
                nxt =  prev-0.1
            else:
                nxt =  LoGimg[i+1][int(lis[1])][int(lis[0])]
                prev = LoGimg[i-1][int(lis[1])][int(lis[0])]
            if (curr >= prev) and (curr >= nxt):
                edges.append([lis[0],lis[1],lis[2]])

    return np.array(edges)

Image = cv2.imread('kyoto_edges.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('kyoto_edges.jpg')
edges = multiscale_BlobDetect(Image, sigma = 2,theta_blob = 0.005, s = 1.5,N
= 4,k = 0.05)

Image = cv2.imread('kyoto_edges.jpg')
ImageRGB = cv2.cvtColor(Image, cv2.COLOR_BGR2RGB)
interest_points_visualization(ImageRGB, edges, None)
```

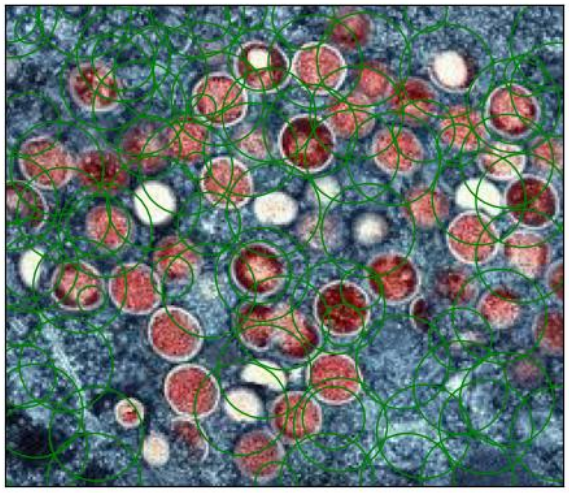
```
<AxesSubplot:>
```

```
Image = cv2.imread('up.png', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('up.png')
blobs = multiscale_BlobDetect(Image,sigma = 2,theta_blob = 0.005, s = 1.5,N =
4,k = 0.05)

Image = cv2.imread('up.png')
ImageRGB = cv2.cvtColor(Image, cv2.COLOR_BGR2RGB)
interest_points_visualization(ImageRGB, blobs, None)
```

```
<AxesSubplot:>
```

```
Image = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('cells.jpg')
blobs = multiscale_BlobDetect(Image,sigma = 2,theta_blob = 0.005, s = 1.5,N =
4,k = 0.05)

Image = cv2.imread('cells.jpg')
ImageRGB = cv2.cvtColor(Image, cv2.COLOR_BGR2RGB)
interest_points_visualization(ImageRGB, blobs, None)
```

```
<AxesSubplot:>
```



2.5 Box Filters

Τα box filters χρησιμοποιούνται για εξομαλύνουν και θολώνουν τις εικόνες. Αυτό συμβαίνει αν σε κάθε pixel (i, j) της αρχικής εικόνας θέσουμε την τιμή του ως το μέσο όρο των τιμών των γειτόνων του. Χρησιμοποιούνται για μείωση θορύβου και απόρριψη λεπτομερειών κάποιας κλίμακας και μικρότερων.

Η διαδικασία για τον υπολογισμό του μέσου όρου είναι κάνοντας συνέλιξη της εικόνας με τον παρακάτω kernel.

$$box_{mn} = \frac{1}{mn} \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

Source: https://docs.nvidia.com/vpi/algo_box_filter.html

Παρατήρηση: Μεγάλος kernel σημαίνει μεγαλύτερη εξομάλυνση και διατήρηση λιγότερων λεπτομερειών και αντίστροφα.

2.5.1. Υπολογίζουμε την ολοκληρωτική εικόνα

```
Image = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('cells.jpg')
Image = Image.astype(np.float)/Image.max()

imageInt = np.cumsum(np.cumsum(Image, axis=0), axis=1)
```

2.5.1. Ορίζουμε το μέγεθος της εικόνας καθώς και τα παράθυρα που χρειάζεται να αθροίσουμε

```
sigma = 1.5
n = int(np.ceil(3*sigma)*2+1)

DxxHeight = int(4*np.floor(n/6)+1)
DxxWidth = int(2*np.floor(n/6)+1)

DyyHeight = int(2*np.floor(n/6)+1)
DyyWidth = int(4*np.floor(n/6)+1)

DxyHeight = int(2*np.floor(n/6)+1)
DxyWidth = int(2*np.floor(n/6)+1)

padUp = (DxxHeight - 1)//2
padDown = padUp
padLeft = (3*DxxWidth - 1)//2
padRight = padLeft

imageIntPadxx = np.pad(imageInt, ((padUp, padDown), (padLeft, padRight)))
imageIntPadyy = np.pad(imageInt, ((padLeft, padRight), (padUp, padDown)))
imageIntPadxy = np.pad(imageInt, ((DxxWidth, DxxWidth), (DxxWidth,
DxxWidth)))
```

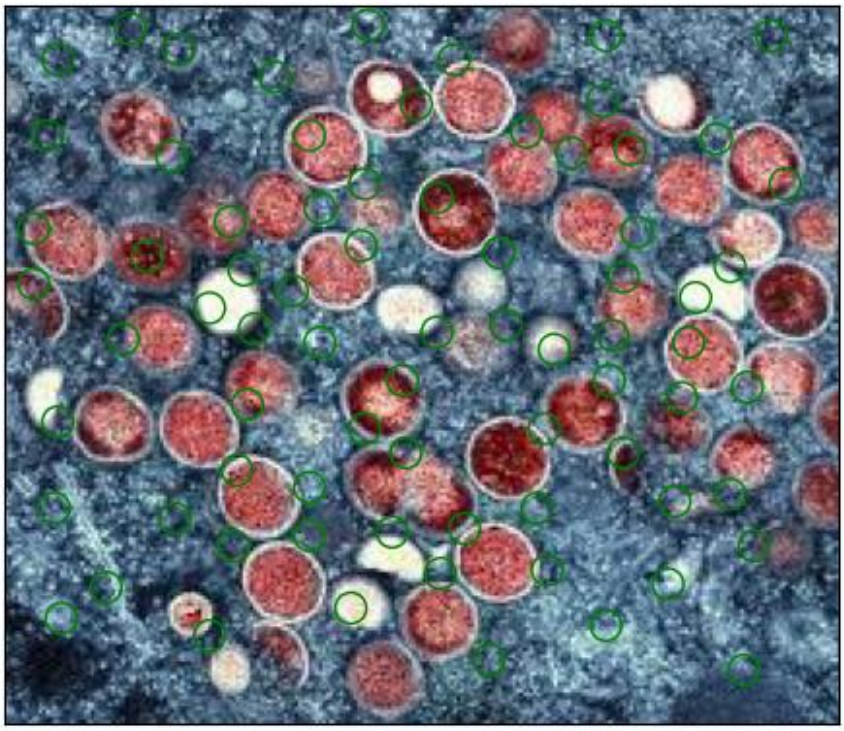
```
padDim = (3*DxxHeight//2, 3*DxxHeight//2)
imageIntPadxx = np.pad(imageInt, (padDim, padDim))
imageIntPadyy = np.pad(imageInt, (padDim, padDim))
imageIntPadxy = np.pad(imageInt, (padDim, padDim))
```

2.5.2. Υπολογισμός των L_{xx}, L_{xy}, L_{yy} της Ολοκληρωτικής Εικόνας

```
Exx = imageIntPadxx - np.roll(imageIntPadxx,-DxxHeight+1,axis=0) -
np.roll(imageIntPadxx,-DxxWidth+1,axis=1) + np.roll(imageIntPadxx,(-
DxxHeight+1,-DxxWidth+1),axis=(0,1))
Lxx = Exx + np.roll(Exx,-2*DxxWidth,axis=0) - 2*np.roll(Exx,-DxxWidth,axis=0)

Eyy = imageIntPadyy - np.roll(imageIntPadyy,-DyyWidth+1,axis=1) -
np.roll(imageIntPadyy,-DyyHeight+1,axis=0) + np.roll(imageIntPadyy,(-
DyyWidth+1,-DyyHeight+1),axis=(1,0))
Lyy = Eyy + np.roll(Eyy,-2*DyyHeight,axis=1) - 2*np.roll(Eyy,-
DyyHeight,axis=1)

Exy = imageIntPadxy - np.roll(imageIntPadxy,-DxyHeight+1,axis=1) -
np.roll(imageIntPadxy,-DxyWidth+1,axis=0) + np.roll(imageIntPadxy,(-
DxyHeight+1,-DxyWidth+1),axis=(0,1))
Lxy = Exy - np.roll(Exy,-DxyWidth-1,axis=1) - np.roll(Exy,-DxyWidth-1,axis=0)
+ np.roll(Exy,(-DxyHeight-1,-DxyWidth-1),axis=(0,1))
```

2.5.3. Εύρεση των Σημείων Ενδιαφέροντος

```
R = Lxx*Lyy - (0.9*Lxy)**2
R = (R - R.min())/R.max()

threshold = (R.min() + R.max()) / 2
R_bin = (R > threshold).astype(int)
n = np.ceil(3*sigma)*2+1
B_square = disk_strel(n)
theta_corn = 0.05

corners = ((R == cv2.dilate(R,B_sq)) & (R >
theta_corn*np.amax(R))).astype(np.uint8)

cornerPositions = np.nonzero(corners == 1)
kp_data = []

for i in range(np.sum(corners)):
    kp_data.append([cornerPositions[1][i],cornerPositions[0][i],sigma])
img = cv2.cvtColor(Img_BGR, cv2.COLOR_BGR2RGB)
temp = interest_points_visualization(img, kp_data)
```



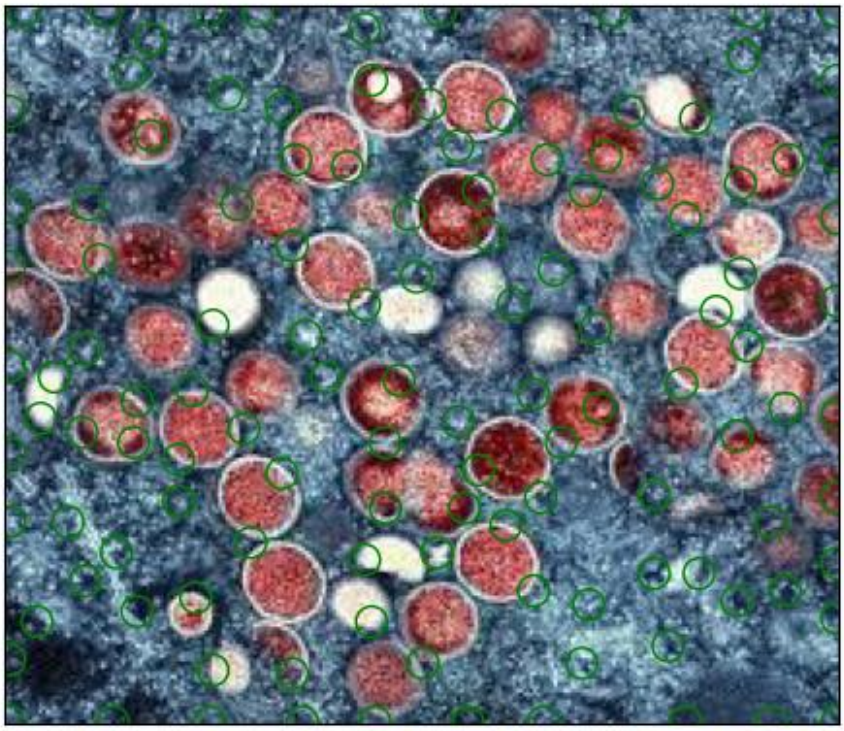
Ορίζουμε μια συνάρτηση για τα box filters

```
def BoxFilters(img,sd,theta_corn):

    theta_corn = 0.005

    img = img.astype(np.float)/img.max()
    intImg = np.cumsum(np.cumsum(img, axis=0), axis=1)
    ns = int(np.ceil(3*sd)*2+1)
# ======================================== #
    DxxHeight = int(4*np.floor(ns/6)+1)
    DxxWidth = int(2*np.floor(ns/6)+1)

    DyyHeight = int(2*np.floor(ns/6)+1)
    DyyWidth = int(4*np.floor(ns/6)+1)

    DxyHeight = int(2*np.floor(ns/6)+1)
    DxyWidth = int(2*np.floor(ns/6)+1)

    padUp = (DxxHeight - 1)//2
    padDown = padUp
    padLeft = (3*DxxWidth - 1)//2
    padRight = padLeft
# ======================================== #

    paddedIntImg = intImg
```

```
    paddedIntImgxx = np.pad(paddedIntImg, ((padUp, padDown), (padLeft,
padRight)))
    paddedIntImgyy = np.pad(paddedIntImg, ((padLeft, padRight), (padUp,
padDown)))
    paddedIntImgxy = np.pad(paddedIntImg, ((DxyHeight, DxyWidth), (DxyHeight,
DxyWidth)))

    Lyy = np.zeros((img.shape[0], img.shape[1]))
    Lxx = np.zeros((img.shape[0], img.shape[1]))
    Lxy = np.zeros((img.shape[0], img.shape[1]))

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            Lxx1 = (paddedIntImgxx[i][j] - paddedIntImgxx[i][j + DxxWidth -
1] +
                    paddedIntImgxx[i + DxxHeight - 1][j + DxxWidth - 1] -
paddedIntImgxx[i + DxxHeight - 1][j])

            Lxx2 = -2*(paddedIntImgxx[i][j - paddedIntImgxx[i][j + 2*DxxWidth
- 1] + DxxWidth] +
                       paddedIntImgxx[i + DxxHeight - 1][j + 2*DxxWidth - 1]
- paddedIntImgxx[i + DxxHeight - 1][j + DxxWidth])

            Lxx3 = (paddedIntImgxx[i][j + 2*DxxWidth] - paddedIntImgxx[i][j +
3*DxxWidth - 1] +
                    paddedIntImgxx[i + DxxHeight - 1][j + 3*DxxWidth - 1] -
paddedIntImgxx[i + DxxHeight - 1][j + 2*DxxWidth])

            Lyy1 = (paddedIntImgyy[i][j] - paddedIntImgyy[i][j + DxxHeight -
1] +
                    paddedIntImgyy[i + DxxWidth - 1][j + DxxHeight - 1] -
paddedIntImgyy[i + DxxWidth - 1][j])

            Lyy2 = -2*(paddedIntImgyy[i + DxxWidth][j] - paddedIntImgyy[i +
DxxWidth][j + DxxHeight - 1] +
                       paddedIntImgyy[i + 2*DxxWidth - 1][j + DxxHeight - 1]
- paddedIntImgyy[i + 2*DxxWidth - 1][j])

            Lyy3 = (paddedIntImgyy[i + 2*DxxWidth][j] - paddedIntImgyy[i +
2*DxxWidth][j + DxxHeight - 1] +
                    paddedIntImgyy[i + 3*DxxWidth - 1][j + DxxHeight - 1] -
paddedIntImgyy[i + 3*DxxWidth - 1][j])

            Lxy1 = (paddedIntImgxy[i][j] - paddedIntImgxy[i][j+DxxWidth-1] +
                    paddedIntImgxy[i + DxxWidth-1][j + DxxWidth-1] -
paddedIntImgxy[i + DxxWidth-1][j])

            Lxy2 = -(paddedIntImgxy[i + DxxWidth+1][j] - paddedIntImgxy[i +
DxxWidth+1][j + DxxWidth-1] +
```

```
                    paddedIntImgxy[i + 2*DxxWidth][j + DxxWidth-1] -
paddedIntImgxy[i + 2*DxxWidth][j])

            Lxy3 = -(paddedIntImgxy[i][j + DxxWidth+1] - paddedIntImgxy[i][j
+ 2*DxxWidth] +
                    paddedIntImgxy[i + DxxWidth - 1][j + 2*DxxWidth] -
paddedIntImgxy[i + DxxWidth - 1][j + DxxWidth+1])

            Lxy4 = (paddedIntImgxy[i + DxxWidth+1][j + DxxWidth+1] -
paddedIntImgxy[i + DxxWidth+1][j + 2*DxxWidth] +
                    paddedIntImgxy[i + 2*DxxWidth][j + 2*DxxWidth] -
paddedIntImgxy[i + 2*DxxWidth][j + DxxWidth+1])

            Lxx[i][j] = Lxx1 + Lxx2 + Lxx3
            Lyy[i][j] = Lyy1 + Lyy2 + Lyy3
            Lxy[i][j] = Lxy1 + Lxy2 + Lxy3 + Lxy4

    R = Lxx*Lyy - (0.9*Lxy)**2
    R = (R - R.min())/R.max()

    Rmax = max(R)
    Bsquare = disk_strel(ns)

    points = []
    dilation = cv2.dilate(R, Bsquare)
    for i in range(len(R)):
        for j in range(len(R[i])):
            if  R[i][j]==dilation[i][j] and R[i][j] > theta_corn*Rmax:
                points.append([j,i,sd])

    return np.array(points)

import cv2
import numpy as np

img = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
points = BoxFilters(img,1.5,0.005)

I = cv2.imread('cells.jpg')
I = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)
interest_points_visualization(I, points, None)
```

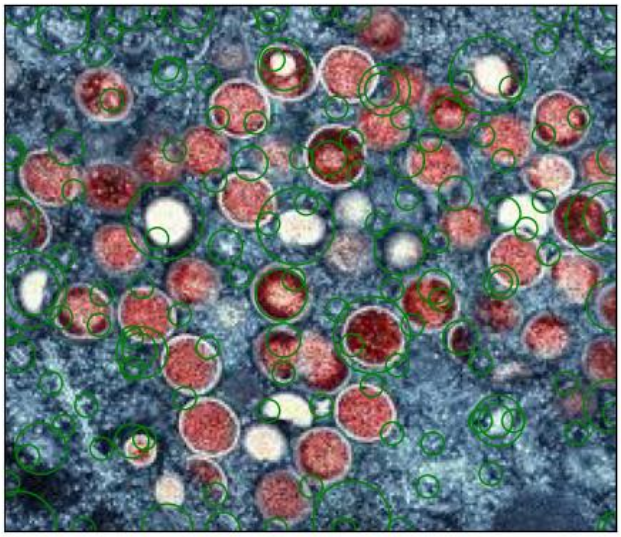
```
<AxesSubplot:>
```



```
def MultiscaleBox(I,sd,N,s,theta_corn):

    Is_total = []
    Isxx = []
    Isyy = []
    total_points =[]
    LoG = []

    #sat = IntegralImage(I)
    for i in range(N):

        ns = int(np.ceil(3*((s**i)*sd))*2+1)
        gauss1D = cv2.getGaussianKernel(ns, (s**i)*sd) # Column vector
        Gs = gauss1D @ gauss1D.T # Symmetric gaussian kernel

        Is = cv2.filter2D(I,-1,Gs)
        Is_total.append(Is)
        gradIs = np.gradient(Is)
        temp1 = np.gradient(gradIs[0])
        temp2 = np.gradient(gradIs[1])

        Isxx = temp1[0]
        Isyy = temp2[1]

        LoG.append(((s**i)*sd)**2*abs(Isxx + Isyy))
```

```
            points = BoxFilters(I, (s**i)*sd, theta_corn)
            total_points.append(points)

        i = -1
        edges =[]
        for arr in total_points:
            i+=1
            for currEdge in arr: #lis: [269. 308.   2.]
                curr = LoG[i][int(currEdge[1])][int(currEdge[0])]

                if i == 0:
                    nxt = LoG[i+1][int(currEdge[1])][int(currEdge[0])]
                    prev = next-1 #pseudovalue

                elif i == N-1:
                    prev = LoG[i-1][int(currEdge[1])][int(currEdge[0])]
                    nxt =  prev-1 #pseudovalue
                else:
                    nxt =  LoG[i+1][int(currEdge[1])][int(currEdge[0])]
                    prev = LoG[i-1][int(currEdge[1])][int(currEdge[0])]

                if (curr > prev) and (curr > nxt):
                    edges.append([currEdge[0], currEdge[1], currEdge[2]])
        return np.array(edges)

img = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
edges = MultiscaleBox(img,1.5,4,1.5,0.005)

I = cv2.imread('cells.jpg')
I = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)
interest_points_visualization(I, edges, None)
```

```
<AxesSubplot:>
```



Μέρος 3: Εφαρμογές σε Ταίριασμα και Κατηγοριοποίηση Εικόνων με Χρήση Τοπικών Περιγραφητών στα Σημεία Ενδιαφέροντος

Χρησιμοποιούμε τους local feature descriptors SURF (Speed Up Robust Features) και HOG (Histogram of Oriented Gradients) στις εικόνες που υποβάλλονται σε περιστροφές και μεγεθύνσεις.

Ο SURF descriptor χρησιμοποιείται σε εφαρμογές object recognition, image registration, classification, ή 3D reconstruction. Αν και οι δημιουργοί του εμπνεύστηκαν από τον SIFT (Scale-Invariant Feature Transform) descriptor, ο SURF είναι γρηγορότερος και πιο εύρωστος.

Source: https://en.wikipedia.org/wiki/Speeded_up_robust_features

Ο HOG descriptor χρησιμοποιείται σε εφαρμογές object detection και εξαγάγει συμπεράσματα μετρώντας εμφανίσεις του προσανατολισμού του gradient σε εντοπισμένες περιοχές της εικόνας.

Source: https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients

3.1. Ταίριασμα Εικόνων υπό Περιστροφή και Αλλαγή Κλίμακας

```
from cv23_lab1_part3_utils import *
```

3.1.1. Εκτίμηση της Περιστροφής και της Κλίμακας των Εικόνων και Αποτίμησή των

```
Image = cv2.imread('cells.jpg', cv2.IMREAD_GRAYSCALE)
Img_BGR = cv2.imread('cells.jpg')
Image = Image.astype(np.float)/Image.max()

SURFScaleError, SURFThetaError = matching_evaluation(
    lambda I: corner_detection(ImgGrey = I, sigma = 2,
                               r = 2.5, thetaCorn = 0.05, k = 0.005),
    lambda I, kp: featuresSURF(I, kp.astype(float)))


SURF features: Avgerage Scale Error (Image 1): 0.036
SURF features: Avgerage Theta Error (Image 1): 2.832
SURF features: Avgerage Scale Error (Image 2): 0.002
SURF features: Avgerage Theta Error (Image 2): 0.233
SURF features: Avgerage Scale Error (Image 3): 0.009
SURF features: Avgerage Theta Error (Image 3): 2.186

Avg. Scale Error for Image 1 with SURF features: 0.036
Avg. Theta Error for Image 1 with SURF features: 2.832
Avg. Scale Error for Image 2 with SURF features: 0.002
Avg. Theta Error for Image 2 with SURF features: 0.233
Avg. Scale Error for Image 3 with SURF features: 0.009
Avg. Theta Error for Image 3 with SURF features: 2.186

HOGScaleError, HOGThetaError = matching_evaluation(
    lambda I: corner_detection(ImgGrey = I, sigma = 2,
                               r = 2.5, thetaCorn = 0.05, k = 0.005),
    lambda I, kp: featuresHOG(I,kp))


HOG features: Avgerage Scale Error (Image 1): 0.232
HOG features: Avgerage Theta Error (Image 1): 22.234
HOG features: Avgerage Scale Error (Image 2): 0.506
HOG features: Avgerage Theta Error (Image 2): 22.049
HOG features: Avgerage Scale Error (Image 3): 0.284
HOG features: Avgerage Theta Error (Image 3): 16.883

Avg. Theta Error for Image 1 with HOG features: 22.234
Avg. Scale Error for Image 2 with HOG features: 0.506
Avg. Theta Error for Image 2 with HOG features: 22.049
Avg. Scale Error for Image 3 with HOG features: 0.284
Avg. Theta Error for Image 3 with HOG features: 16.883
```

3.1.2. Ικανότητα Εκτίμησης της Περιστροφής και κλίμακας των Εικόνων για SURF Περιγραφητές

```
# 3.1.2. SURF
avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: corner_detection(I, 2, 2.5, 0.05, 0.005),
    lambda I, kp: featuresSURF(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: harrisLaplacian(I,2,4,1.5,2.5,0.005,0.05),
    lambda I, kp: featuresSURF(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: blob_detect(I, 2,0.005,0.05),
    lambda I, kp: featuresSURF(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: multiscale_BlobDetect(I,1.5,0.005,1.5,4,0.05),
    lambda I, kp: featuresSURF(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: MultiscaleBox(I,1.5,4,1.5,0.005),
    lambda I, kp: featuresSURF(I,kp))
```

```
SURF features: Average Scale Error (Image 1) with CornerDetection: 0.036
SURF features: Average Theta Error (Image 1) with CornerDetection: 2.832
SURF features: Average Scale Error (Image 2) with CornerDetection: 0.002
SURF features: Average Theta Error (Image 2) with CornerDetection: 0.233
SURF features: Average Scale Error (Image 3) with CornerDetection: 0.009
SURF features: Average Theta Error (Image 3) with CornerDetection: 2.186

SURF features: Average Scale Error (Image 1) with HarrisLaplacian: 0.136
SURF features: Average Theta Error (Image 1) with HarrisLaplacian: 11.621
SURF features: Average Scale Error (Image 2) with HarrisLaplacian: 1.042
SURF features: Average Theta Error (Image 2) with HarrisLaplacian: 4.99
SURF features: Average Scale Error (Image 3) with HarrisLaplacian: 0.103
SURF features: Average Theta Error (Image 3) with HarrisLaplacian: 16.647

SURF features: Average Scale Error (Image 1) with Blobs: 0.027
SURF features: Average Theta Error (Image 1) with Blobs: 7.759
SURF features: Average Scale Error (Image 2) with Blobs: 0.01
SURF features: Average Theta Error (Image 2) with Blobs: 0.229
SURF features: Average Scale Error (Image 3) with Blobs: 0.001
SURF features: Average Theta Error (Image 3) with Blobs: 0.054

SURF features: Average Scale Error (Image 1) with MultiscaleBlobs: 0.069
SURF features: Average Theta Error (Image 1) with MultiscaleBlobs: 10.511
SURF features: Average Scale Error (Image 2) with MultiscaleBlobs: 0.146
SURF features: Average Theta Error (Image 2) with MultiscaleBlobs: 14.692
SURF features: Average Scale Error (Image 3) with MultiscaleBlobs: 0.055
```

```
SURF features: Average Theta Error (Image 3) with MultiscaleBlobs: 8.261

SURF features: Average Scale Error (Image 1) with MultiscaleBox: 0.001
SURF features: Average Theta Error (Image 1) with MultiscaleBox: 0.096
SURF features: Average Scale Error (Image 2) with MultiscaleBox: 0.002
SURF features: Average Theta Error (Image 2) with MultiscaleBox: 0.078
SURF features: Average Scale Error (Image 3) with MultiscaleBox: 0.001
SURF features: Average Theta Error (Image 3) with MultiscaleBox: 0.045

SURF features: Average Scale Error (Image 1) with CornerDetection: 0.036
SURF features: Average Theta Error (Image 1) with CornerDetection: 2.832
SURF features: Average Scale Error (Image 2) with CornerDetection: 0.002
SURF features: Average Theta Error (Image 2) with CornerDetection: 0.233
SURF features: Average Scale Error (Image 3) with CornerDetection: 0.009
SURF features: Average Theta Error (Image 3) with CornerDetection: 2.186

SURF features: Average Scale Error (Image 1) with HarrisLaplacian: 0.136
SURF features: Average Theta Error (Image 1) with HarrisLaplacian: 11.621
SURF features: Average Scale Error (Image 2) with HarrisLaplacian: 1.042
SURF features: Average Theta Error (Image 2) with HarrisLaplacian: 4.99
SURF features: Average Scale Error (Image 3) with HarrisLaplacian: 0.103
SURF features: Average Theta Error (Image 3) with HarrisLaplacian: 16.647

SURF features: Average Scale Error (Image 1) with Blobs: 0.027
SURF features: Average Theta Error (Image 1) with Blobs: 7.759
SURF features: Average Scale Error (Image 2) with Blobs: 0.01
SURF features: Average Theta Error (Image 2) with Blobs: 0.229
SURF features: Average Scale Error (Image 3) with Blobs: 0.001
SURF features: Average Theta Error (Image 3) with Blobs: 0.054

SURF features: Average Scale Error (Image 1) with MultiscaleBlobs: 0.069
SURF features: Average Theta Error (Image 1) with MultiscaleBlobs: 10.511
SURF features: Average Scale Error (Image 2) with MultiscaleBlobs: 0.146
SURF features: Average Theta Error (Image 2) with MultiscaleBlobs: 14.692
SURF features: Average Scale Error (Image 3) with MultiscaleBlobs: 0.055
SURF features: Average Theta Error (Image 3) with MultiscaleBlobs: 8.261

SURF features: Average Scale Error (Image 1) with MultiscaleBox: 0.001
SURF features: Average Theta Error (Image 1) with MultiscaleBox: 0.096
SURF features: Average Scale Error (Image 2) with MultiscaleBox: 0.002
SURF features: Average Theta Error (Image 2) with MultiscaleBox: 0.078
SURF features: Average Scale Error (Image 3) with MultiscaleBox: 0.001
SURF features: Average Theta Error (Image 3) with MultiscaleBox: 0.045
```

```
avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: corner_detection(I, 2, 2.5, 0.05, 0.005),
    lambda I, kp: featuresHOG(I,kp))
```

```
avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: harrisLaplacian(I,2,4,1.5,2.5,0.005,0.05),
    lambda I, kp: featuresHOG(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: blob_detect(I, 2,0.005,0.05),
    lambda I, kp: featuresHOG(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: multiscale_BlobDetect(I,1.5,0.005,1.5,4,0.05),
    lambda I, kp: featuresHOG(I,kp))

avg_scale_errors, avg_theta_errors = matching_evaluation(
    lambda I: MultiscaleBox(I,1.5,4,1.5,0.005),
    lambda I, kp: featuresHOG(I,kp))
```

```
HOG features: Average Scale Error (Image 1) with CornerDetection: 0.232
HOG features: Average Theta Error (Image 1) with CornerDetection: 22.234
HOG features: Average Scale Error (Image 2) with CornerDetection: 0.506
HOG features: Average Theta Error (Image 2) with CornerDetection: 22.049
HOG features: Average Scale Error (Image 3) with CornerDetection: 0.284
HOG features: Average Theta Error (Image 3) with CornerDetection: 16.883

HOG features: Average Scale Error (Image 1) with HarrisLaplacian: 0.63
HOG features: Average Theta Error (Image 1) with HarrisLaplacian: 14.363
HOG features: Average Scale Error (Image 2) with HarrisLaplacian: 0.481
HOG features: Average Theta Error (Image 2) with HarrisLaplacian: 32.468
HOG features: Average Scale Error (Image 3) with HarrisLaplacian: 0.144
HOG features: Average Theta Error (Image 3) with HarrisLaplacian: 25.419

HOG features: Average Scale Error (Image 1) with Blobs: 0.186
HOG features: Average Theta Error (Image 1) with Blobs: 7.231
HOG features: Average Scale Error (Image 2) with Blobs: 0.1
HOG features: Average Theta Error (Image 2) with Blobs: 13.674
HOG features: Average Scale Error (Image 3) with Blobs: 0.154
HOG features: Average Theta Error (Image 3) with Blobs: 27.219

HOG features: Average Scale Error (Image 1) with MultiscaleBlobs: 0.165
HOG features: Average Theta Error (Image 1) with MultiscaleBlobs: 24.57
HOG features: Average Scale Error (Image 2) with MultiscaleBlobs: 0.462
HOG features: Average Theta Error (Image 2) with MultiscaleBlobs: 20.107
HOG features: Average Scale Error (Image 3) with MultiscaleBlobs: 0.25
HOG features: Average Theta Error (Image 3) with MultiscaleBlobs: 29.258

HOG features: Average Scale Error (Image 1) with MultiscaleBox: 0.138
HOG features: Average Theta Error (Image 1) with MultiscaleBox: 14.919
HOG features: Average Scale Error (Image 2) with MultiscaleBox: 0.199
HOG features: Average Theta Error (Image 2) with MultiscaleBox: 20.013
```

```
HOG features: Average Scale Error (Image 3) with MultiscaleBox: 0.175
HOG features: Average Theta Error (Image 3) with MultiscaleBox: 15.399

HOG features: Average Scale Error (Image 1) with CornerDetection: 0.232
HOG features: Average Theta Error (Image 1) with CornerDetection: 22.234
HOG features: Average Scale Error (Image 2) with CornerDetection: 0.506
HOG features: Average Theta Error (Image 2) with CornerDetection: 22.049
HOG features: Average Scale Error (Image 3) with CornerDetection: 0.284
HOG features: Average Theta Error (Image 3) with CornerDetection: 16.883

HOG features: Average Scale Error (Image 1) with HarrisLaplacian: 0.63
HOG features: Average Theta Error (Image 1) with HarrisLaplacian: 14.363
HOG features: Average Scale Error (Image 2) with HarrisLaplacian: 0.481
HOG features: Average Theta Error (Image 2) with HarrisLaplacian: 32.468
HOG features: Average Scale Error (Image 3) with HarrisLaplacian: 0.144
HOG features: Average Theta Error (Image 3) with HarrisLaplacian: 25.419

HOG features: Average Scale Error (Image 1) with Blobs: 0.186
HOG features: Average Theta Error (Image 1) with Blobs: 7.231
HOG features: Average Scale Error (Image 2) with Blobs: 0.1
HOG features: Average Theta Error (Image 2) with Blobs: 13.674
HOG features: Average Scale Error (Image 3) with Blobs: 0.154
HOG features: Average Theta Error (Image 3) with Blobs: 27.219

HOG features: Average Scale Error (Image 1) with MultiscaleBox: 0.138
HOG features: Average Theta Error (Image 1) with MultiscaleBox: 14.919
HOG features: Average Scale Error (Image 2) with MultiscaleBox: 0.199
HOG features: Average Theta Error (Image 2) with MultiscaleBox: 20.013
HOG features: Average Scale Error (Image 3) with MultiscaleBox: 0.175
HOG features: Average Theta Error (Image 3) with MultiscaleBox: 15.399
```

3.2. Εξαγωγή Χαρακτηριστικών & Αναπαράσταση Bag of Visual Words

- Εξάγουμε τα χαρακτηριστικά των εικόνων.
- Χρησιμοποιώντας kMeans βρίσκουμε τα κεντροειδή των clusters των local descriptors (words) του train set
- Εκπαιδεύουμε έναν SVM Classifier για να κάνει προβλέψεις στο test set

Σημείωση: Όπως στην Επεξεργασία Φυσικής γλώσσας έχουμε αναπαράσταση των λέξεων με vectors που προκύπτουν από το πλήθος εμφανίσεων των λέξεων, στην Όραση Υπολογιστών έχουμε vectors που προκύπτουν από το πλήθος εμφανίσεων ορισμένων τοπικών χαρακτηριστικών εικόνας.

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: corner_detection(I, 2, 2.5, 0.05, 0.005),
    lambda I, kp: featuresSURF(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'SURF descriptors accuracy for CornerDetection:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_332.png
bike_348.png
Time for feature extraction: 13.449
SURF descriptors accuracy for CornerDetection: 53.931%
```

```
# HarrisLaplacian(I,sd,N,s,p,theta_corn,k)
# Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: harrisLaplacian(I,2,4,1.5,2.5,0.005,0.05),
    lambda I, kp: featuresSURF(I,kp))
```

```
accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'SURF descriptors accuracy for Harris - Laplacian:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 391.073
SURF descriptors accuracy for Harris - Laplacian: 57.241%
```

```
detect_fun = lambda I: blob_detect(I, 2,0.005,0.05)
#HarrisLaplacian(I,2,4,1.5,2.5,0.005,0.05)
desc_fun = lambda I, kp: featuresSURF(I,kp)
#Extract features from the provided dataset.
feats = FeatureExtraction(detect_fun, desc_fun)

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)
```

```
print(f'SURF descriptors accuracy for Blobs: {round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 34.130
SURF descriptors accuracy for Blobs: 55.034%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: multiscale_BlobDetect(I,1.5,0.005,1.5,4,0.05),
    lambda I, kp: featuresSURF(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'SURF descriptors accuracy for MultiscaleBlobs:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 99.154
SURF descriptors accuracy for MultiscaleBlobs: 60.690%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: MultiscaleBox(I,1.5,4,1.5,0.005),
    lambda I, kp: featuresSURF(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'SURF descriptors accuracy for MultiscaleBox:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 3245.022
SURF descriptors accuracy for MultiscaleBox: 57.931%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: corner_detection(I, 2, 2.5, 0.05, 0.005),
    lambda I, kp: featuresHOG(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
```

```
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print('HOG descriptors accuracy for CornerDetection:
{:.3f}%'.format(100.0*np.mean(accs)))
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_332.png
bike_348.png
Time for feature extraction: 18.972
HOG descriptors accuracy for CornerDetection: 64.138%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: harrisLaplacian(I,2,4,1.5,2.5,0.005,0.05),
    lambda I, kp: featuresHOG(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'HOG descriptors accuracy for Harris - Laplacian:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 261.252
HOG descriptors accuracy for Harris - Laplacian: 63.724%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: blob_detect(I, 2,0.005,0.05),
    lambda I, kp: featuresHOG(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'HOG descriptors accuracy for Blobs: {round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 41.199
HOG descriptors accuracy for Blobs: 64.966%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: multiscale_BlobDetect(I,1.5,0.005,1.5,4,0.05),
    lambda I, kp: featuresHOG(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
```

```
    accs.append(acc)

print(f'HOG descriptors accuracy for multiscale_BlobDetect:
{round(100*np.mean(accs), 3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_330.png
bike_331.png
bike_332.png
bike_348.png

Time for feature extraction: 195.411
HOG descriptors accuracy for multiscale_BlobDetect: 66.897%
```

```
#Extract features from the provided dataset.
feats = FeatureExtraction(
    lambda I: MultiscaleBox(I,1.5,4,1.5,0.005),
    lambda I, kp: featuresHOG(I,kp))

accs = []
for k in range(5):
    # Split into a training set and a test set.
    data_train, label_train, data_test, label_test = createTrainTest(feats,
k)

    # Perform Kmeans to find centroids for clusters.
    BOF_tr, BOF_ts = BagOfWords(data_train, data_test)

    # Train an svm on the training set and make predictions on the test set
    acc, preds, probas = svm(BOF_tr, label_train, BOF_ts, label_test)
    accs.append(acc)

print(f'HOG descriptors accuracy for MultiscaleBox: {round(100*np.mean(accs),
3)}%')
```

```
person_001.png
person_002.png
person_003.png
person_004.png
...
bike_043.png
bike_045.png
bike_049.png
bike_050.png
```

```
Time for feature extraction: 3436.469
HOG descriptors accuracy for MultiscaleBox: 67.448%
```

Παραθέτουμε συνολικά τα αποτελέσματα για επισκόπηση

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from prettytable import PrettyTable

data = {'Features': ["SURF", "HOG"],
        'Corner Detection': ["Time: 13.449 | Acc: 53.931%", "Time: 18.972 |
Acc: 64.138%"],
        'Harris - Laplacian': ["Time: 391.073 | Acc: 57.241%", "Time: 261.252
| Acc: 63.724%"],
        'Blobs': ["Time: 34.130 | Acc: 55.034%", "Time: 41.199 | Acc:
64.690%"],
        'Multiscale Blobs': ["Time: 99.154  | Acc: 60.090%", "Time: 195.441 |
Acc: 66.897%"],
        'Multiscale Box' : ["Time: 3245.022 | Acc: 56.964%", "Time: 3436.469
| Acc: 67.488%"]
        }

# create a pretty table object
table = PrettyTable()

# add columns
table.add_column('Features', data['Features'])
table.add_column('Corner Detection', data['Corner Detection'])
table.add_column('Harris - Laplacian', data['Harris - Laplacian'])
table.add_column('Blobs', data['Blobs'])
table.add_column('Multiscale Blobs', data['Multiscale Blobs'])
table.add_column('Multiscale Box', data['Multiscale Box'])

# print the table
print(table)
```

```
+----------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
| Features |    Corner Detection   |   Harris - Laplacian  |         Blobs         |    Multiscale Blobs   |     Multiscale Box    |
+----------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
|   SURF   | Time: 13.449 | Acc: 53.931% | Time: 391.073 | Acc: 57.241% | Time: 34.130 | Acc: 55.034% | Time: 99.154  | Acc: 60.090% | Time: 3245.022 | Acc: 56.964% |
|   HOG    | Time: 18.972 | Acc: 64.138% | Time: 261.252 | Acc: 63.724% | Time: 41.199 | Acc: 64.690% | Time: 195.441 | Acc: 66.897% | Time: 3436.469 | Acc: 67.488% |
+----------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
```