Τεχνική αναφορά για την Προγραμματιστική Εργασία στο μάθημα της Ψηφιακής Επεξεργασίας Εικόνας

Σταύρος Νικολαΐδης
ΑΕΜ: 3975

1. Εισαγωγή

Υπάρχουν τα εξής αρχεία:

- **demo.py,** όπου υπάρχουν όλες οι μέθοδοι/συναρτήσεις που υλοποιήθηκαν μαζί με τις βοηθητικές συναρτήσεις.
- **demo1.py,** το οποίο είναι το αρχείο που ζητήθηκε από την εκφώνηση της εργασίας.
- **demo2.py,** όπου παρουσιάζεται η λειτουργία κάθε μεθόδου που υλοποιήθηκε.
- **Φάκελος Results,** όπου είναι αποθηκευμένες οι εικόνες που προέκυψαν από διάφορες επεξεργασίες.

Αρχικά θα εξηγήσουμε τι κάνει κάθε συνάρτηση που υλοποιήθηκε μέσα από παραδείγματα του demo2.py και έπειτα θα σχολιάσουμε τα αποτελέσματα του demo1.py.

2. Δυδιάστατη Συνέλιξη - def myConv2(A, B, param)

Η συνάρτηση δέχεται 2 πίνακες (ή εικόνες), τους A και B, ως ορίσαματα. Υποθέτουμε πως ο πίνακας B είναι ο πυρήνας. Οπότε, πριν εφαρμοστεί η πράξη της συνέλιξης αντανακλούμε τον πίνακα B ως προς τους 2 άξονες. Στην συνέχεια, η συνάρτηση παίρνει τις πληροφορίες για το σχήμα των πινάκων και ανάλογα με την παράμετρο για επιστροφή πίνακα με σχήμα ίδιο με τον πίνακα A ή όχι θέτει το μέγεθος του τελικού πίνακα. Ο τελικός πίνακας προκύπτει από την πράξη της συνέλιξης η οποία υλοποιείται με 4 επαναλήψεις, «παιρνώντας» ουσιαστικά το kernel μέσα από τον πίνακα A και παίρνοντας κάθε φορά το αποτέλεσμα του αθροίσματος των πολλαπλασιασμών των επικαλυπτόμενων στοιχείων.

<u>Παράδειγμα 1</u>

Πίνακες

```
4   # Example of 2D Convolution
5   A = np.array([[1, 2, 3],
6                 [4, 5, 6],
7                 [7, 8, 9]])
8
9   B = np.array([[0, 1, 0],
10                [1, 0, 1],
11                [0, 1, 0]])
```

Αποτέλεσμα

```
[[ 0.  1.  2.  3.  0.]
 [ 1.  6.  9.  8.  3.]
 [ 4. 13. 20. 17.  6.]
 [ 7. 12. 21. 14.  9.]
 [ 0.  7.  8.  9.  0.]]
```

Παράδειγμα 2

Πίνακες

```
# Example of 2D Convolution
A = np.array([[14, 12, 1],
              [1, 3, 12],
              [2, 3, 10]])

B = np.array([[1, 0, 0],
              [1, 0, 1],
              [1, 0, 0]])
```

Αποτέλεσμα

```
[[ 0.  0. 14. 12.  1.]
 [14. 12. 16. 15. 13.]
 [ 1.  3. 29. 18. 23.]
 [ 2.  3. 13.  6. 22.]
 [ 0.  0.  2.  3. 10.]]
```

3. Ασπρόμαυρη Εικόνα - def myColorToGray(A, param)

Η συνάρτηση δέχεται μια εικόνα A. Ελέγχει αν είναι ήδη ασπρόμαυρη με βάση το σχήμα της, αν είναι τότε απλά την επιστρέφει ως έχει. Αν δεν είναι, δημιουργεί τοπικά ένα αντίγραφο της ως πίνακα numpy και έπειτα εφαρμόζει την φόρμουλα NTSC στα 3 κανάλια του αντίγραφου της εικόνας και έπειτα επιστρέφει το αποτέλεσμα αφού πρώτα μετατρέψει τον πίνακα σε εικόνα uint8.

NTSC Formula

Ουσιαστικά πρόκειται για πολλαπλασιασμό κάθε pixel σε κάθε κανάλι με μια δεδομένη τιμή για το εκάστωτε κανάλι. Επειδή φορτώσαμε την εικόνα με την βιβλιοθήκη cv2, οι πρώτες τιμές στην τρίτη διάσταση του πίνακα (δείκτης 0) αφορούν το μπλε κανάλι, οι δεύτερες (δείκτης 1) το πράσινο κανάλι και τέλος οι τρίτες (δείκτης 2) το κόκκινο κανάλι. Οι συντελεστές της φόρμουλας ορίζονται ως εξής:

- Μπλε κανάλι: 0.114
- Πράσινο κανάλι: 0.587
- Κόκκινο κανάλι: 0.299

Όπως βλέπουμε, δίνουμε περισσότερη βάση στο πράσινο κανάλι από ότι στο κόκκινο και στο μπλε. Αυτό γίνεται διότι ανθρώπινο μάτι είναι πιο ευαίσθητο στις αλλαγές του πράσινου από ότι σε αυτές του κόκκινου ή του μπλε. Οι συντελεστές, έχουν επιλεχτεί κατά προσέγγιση ώστε να μιμούνται την ανθρώπινη αντίληψη για τα χρώματα.

Παράδειγμα:

Εικόνα Εισόδου

Εικόνα Εξόδου

Παρατηρούμε ότι το αποτέλεσμα είναι αυτό ακριβώς που θέλαμε, η μετατροπή μιας έγχρωμης φωτογραφίας σε ασπρόμαυρη.

4. Εισαγωγή Θορύβου - def myImNoise(A, param)

Η συνάρτηση υλοποιεί 2 είδη προσθήκης θορύβου σε εικόνες.

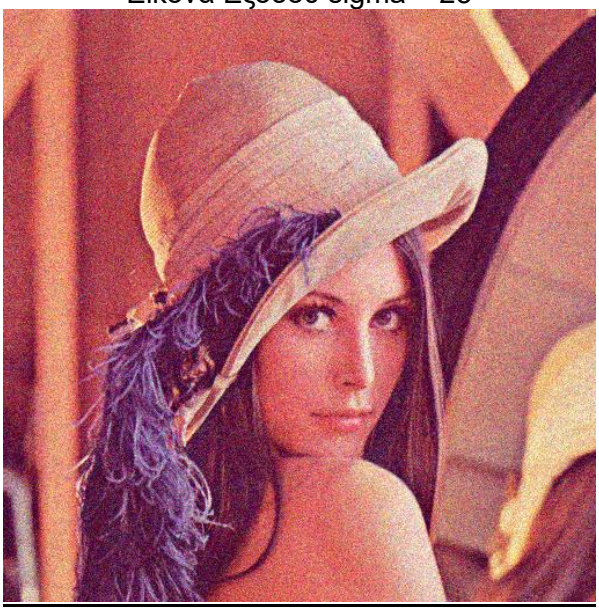
4.1 Τεχνική Gaussian Θορύβου – param = ‘gaussian’

Ο θόρουβος εδώ είναι γκαουσιανός, δηλαδή πρόκειτα για τυχαίο θόρυβο που ακολουθεί γκαουσιανή κατανομή. Ορίζουμε mean = 0 ώστε να μην υπάρχει κάποιο bias ως προς το πρόσημο των pixels. Ορίζουμε sigma, που είναι η τυπική απόκλιση της κατανομής. Μετράει τη μέση απόκλιση των τιμών από το μέσο όρο. Στο πλαίσιο του θορύβου, μια μεγαλύτερη τιμή sigma σημαίνει ότι η ένταση του θορύβου είναι υψηλότερη και οι διακυμάνσεις είναι μεγαλύτερες. Ουσιαστικά, δείχνει πόσο θα αποκλίνουν οι τιμές του θορύβου από τη μέση τιμή.

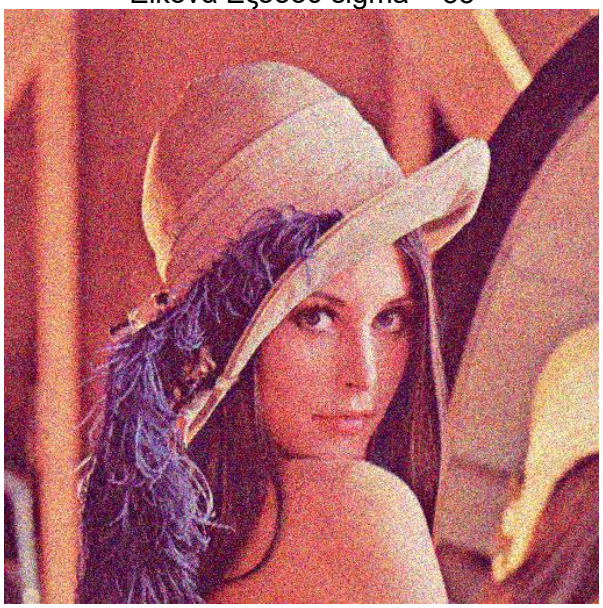
Παραδείγματα:

Εικόνα Εισόδου

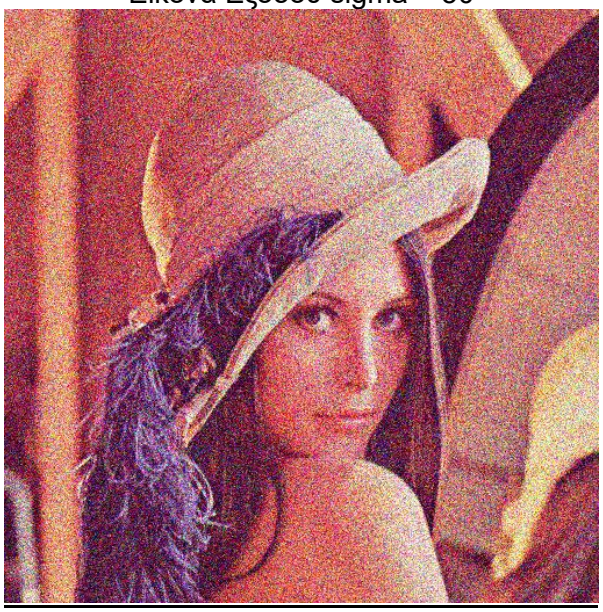
Εικόνα Εξόδου sigma = 25



Εικόνα Εξόδου sigma = 35



Εικόνα Εξόδου sigma = 50



4.2 Τεχνική Salt-and-Pepper Θορύβου – param = ‘saltandpepper’

Πρόκειται για μια πολύ απλή μεθοδολογία. Αρχικά, ορίζουμε μια μεταβλητή split η οποία είναι ουσιαστικά η ποσότητα άσπρου και μαύρου θόρυβου που θέλουμε να έχει η τελική εικόνα. Δηλαδή, με 0.5 έχουμε ίδια ποσότητα μαύρου και λευκού θορύβου. Επίσης ορίζουμε μια μεταβλητή amount για να ορίσουμε το ποσοστό της αρχικής εικόνας που θα επηρεαστεί από τον θόρυβο. Ξεκινάμε πρώτα από τον λευκό θόρυβο. Ορίζουμε τον αριθμό των pixel τα οποία θα επηρεαστούν με λευκό θόρυβο σύμφωνα με την παρακάτω φόρμουλα:

[split * A.size * amount]

όπου A.size το μέγεθος της δοθείσας εικόνας. Ακολουθεί ένας βρόχος όπου σε κάθε επανάληψη διαλέγουμε τυχαία x και y συντεταγμένες και θέτουμε την τιμή του pixel με αυτές τις συντεταγμένες ίση με 255 ώστε να γίνει λευκό αυτό το pixel.

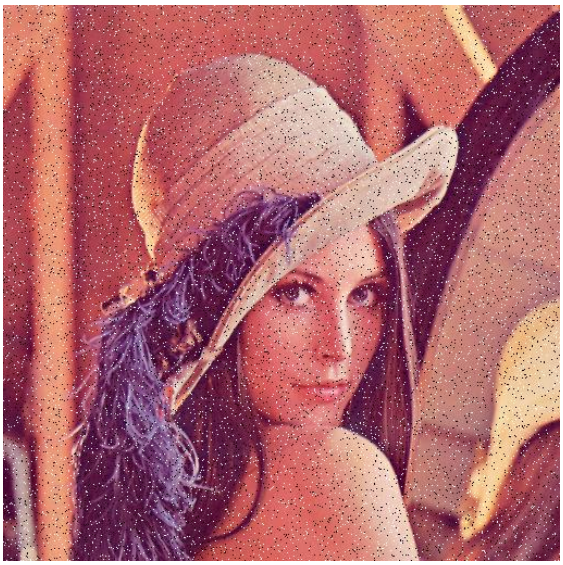
Με τον ίδιο τρόπο επιλέγουμε τυχαία pixels και θέτουμε την τιμή τους ίση με 0 ώστε να γίνουν μαύρα:

$$\lceil (1 - \text{split}) * A.\text{size} * \text{amount} \rceil$$

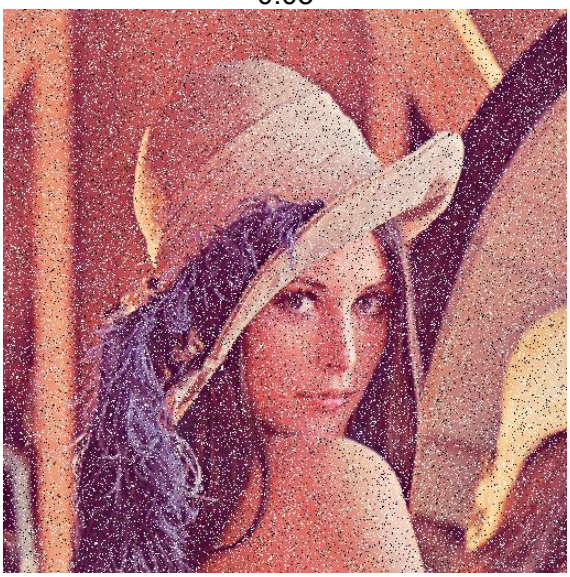
<u>Παραδείγματα:</u>

Εικόνα Εισόδου

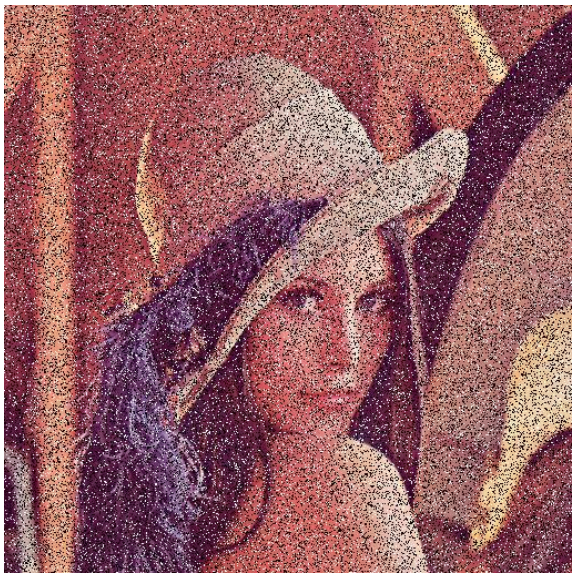
Εικόνα Εξόδου με split = 0.5 και amount = 0.02



Εικόνα Εξόδου με split = 0.7 και amount = 0.05



Εικόνα Εξόδου με split = 0.3 και amount = 0.1



5. Φιλτράρισμα - def myImFilter(A, param, kernel_size)

Η συνάρτηση υλοποιεί 2 είδη φίλτρων. Να σημειωθεί ότι επέλεξα να υλοποιήσω τα φίλτρα με τον τρόπο που μάθαμε στην θεωρία, δηλαδή να κρατήσω κενό το border της φιλτραρισμένης εικόνας. Σίγουρα θα μπορούσε να επεκταθεί αυτή η υλοποίηση και με μια παράμετρο να μπορούμε να επιλέξουμε εάν θέλουμε να γεμίσουμε το border με κάποια τεχνική.

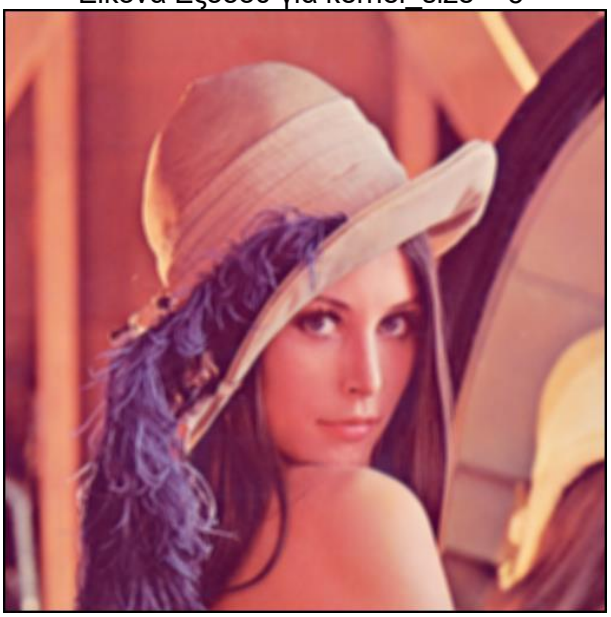
5.1 Mean

Αρχικά ορίζουμε το μέγεθος του kernel. Υπολογίζουμε το offset ώστε να πάρουμε την «γειτονιά» (η οποία είναι ανάλογη με το μέγεθος του kernel) από κάθε pixel. Υπολογίζουμε την μέση τιμή των pixel της «γειτονιάς» και την θέτουμε στο σωστό pixel (σε αυτό που βρίσκεται στο κέντρο του kernel την συγκεκριμένη στιγμή δηλαδή). Επαναλαμβάνουμε για κάθε pixel και επιστρέφουμε την φιλτραρισμένη εικόνα.

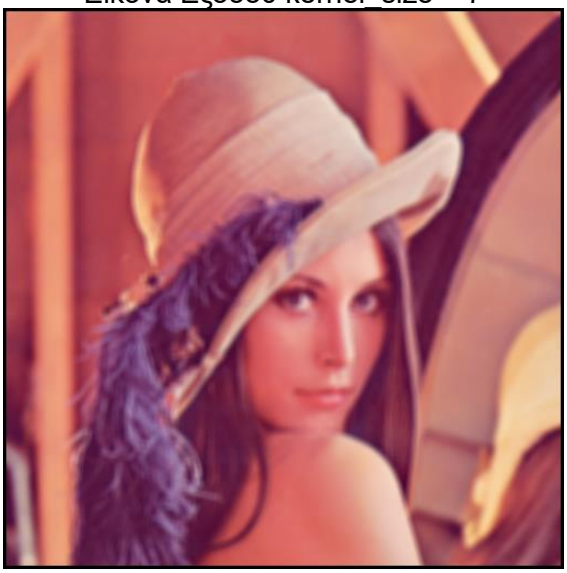
Παραδείγματα:

Εικόνα Εισόδου

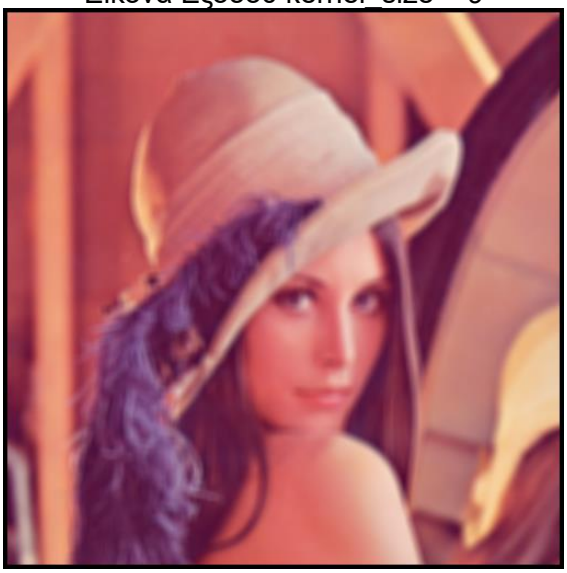
Εικόνα Εξόδου για kernel_size = 5



Εικόνα Εξόδου kernel_size = 7



Εικόνα Εξόδου kernel_size = 9



Όπως παρατηρούμε, όσο αυξάνουμε το μέγεθος του kernel τόσο πιο πολύ «χάνουμε» πληροφορία από τις ακμές του προσώπου και των αντικειμένων και οδηγούμαστε σε μια πιο εξομαλυμένη εικόνα.

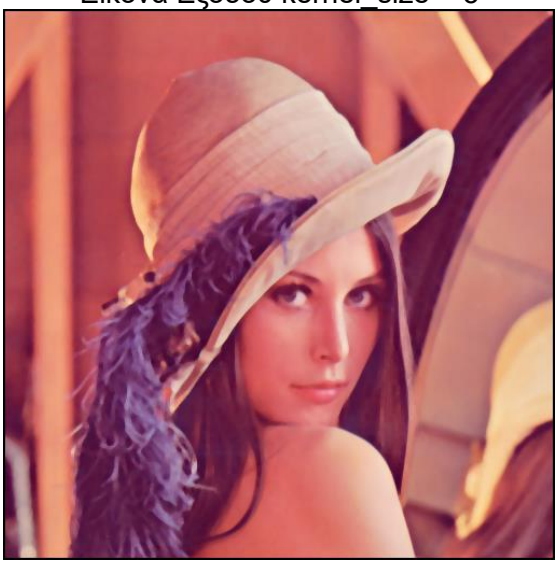
5.2 Median

Και εδώ πρώτα ορίζουμε το μέγεθος του kernel. Υπολογίζουμε το offset ώστε να πάρουμε την «γειτονιά» (η οποία είναι ανάλογη με το μέγεθος του kernel) από κάθε pixel. Αυτή τη φορά όμως διατάσουμε τις τιμές των pixels της γειτονιάς σε αύξουσα σειρά με quicksort και παίρνουμε το μεσαίο στοιχείο αυτής της διάταξης (τον διάμεσο δηλαδή) και θέτουμε αυτή την τιμή στο σωστό pixel (σε αυτό που βρίσκεται στο κέντρο του kernel την συγκεκριμένη στιγμή δηλαδή). Επαναλαμβάνουμε για κάθε pixel και επιστρέφουμε την φιλτραρισμένη εικόνα.

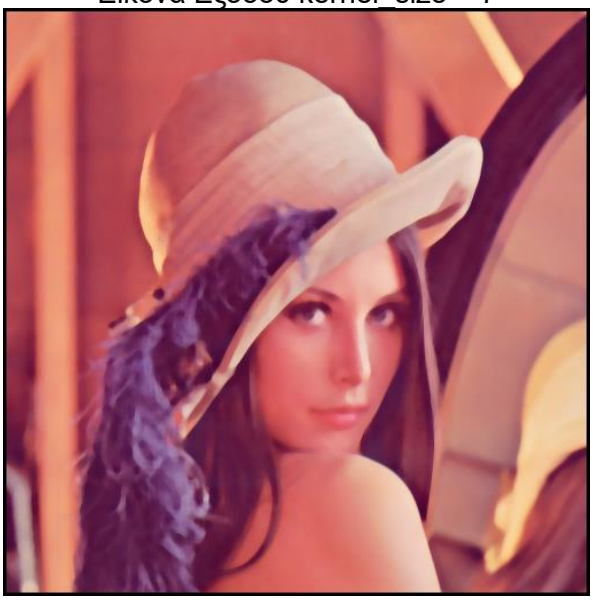
<u>Παραδείγματα:</u>

Εικόνα Εισόδου

Εικόνα Εξόδου kernel_size = 5



Εικόνα Εξόδου kernel_size = 7



Εικόνα Εξόδου kernel_size = 9

Εδώ, παρόλο που οδηγούμαστε ξανά προς μια πιο οξομαλυμένη εικόνα, παρατηρούμε ότι οι ακμές διατηρούνται.

6. Ανίχνευση Ακμών - def myEdgeDetection(A, param):

Η συνάρτηση υλοποιεί 3 διαφορετικές τεχνικές Edge Detection.

Αρχικά, όμως, πρέπει να μετατρέψουμε την εικόνα εισόδου σε ασπρόμαυρη, σε περίπτωση που ήταν έγχρωμη. Αυτό το κάνουμε ελέγχοντας το σχήμα της εικόνας και έαν διαθέτει κανάλια πέρα από ύψος και πλάτος, τότε καλούμε την myColorToGray και την χρησιμοποιούμε για να μετατρέψουμε την εικόνα σε ασπρόμαυρη. Αν ήταν ήδη ασπρόμαυρη συνεχίζουμε κανονικά.

6.1 Τεχνική Sobel

Για αυτή την τεχνική θα χρησιμοποιήσουμε τις μάσκες Sobel:

-1	0	1
-2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1

Μάσκες Sobel

Ουσιαστικά πρόκειται για τον kernel με τον οποίο θα συνελίξουμε την εικόνα ώστε να πάρουμε την κλίση της εικόνας. Ο αριστερά πίνακας αφορά την οριζόντια κλίση της εικόνας (άξονας x) ενώ ο δεξιά την κάθετη κλίση της εικόνας (άξονας y).

Για να πάρουμε την κάθε κλίση συνελίσουμε με χρήση της myConv2, ξεχωριστά, την ασπρόμαυρη εικόνα μια φορά με τον kernel για την οριζόντια κλίση και μια φορά με τον kernel για την κάθετη.

Έχοντας πλέον τις κλίσεις της εικόνας, παίρνουμε το μέτρο (magnitude) τους και το αποτέλεσμα είναι το edge detection της εικόνας με την τεχνική Sobel.

Παράδειγμα:

Εικόνα Εισόδου

Εικόνα Εξόδου

Όπως παρατηρούμε, η εικόνα εξόδου έχει έντονη απεικόνιση των ακμών τόσο του προσώπου όσο και των αντικειμένων με λευκό χρώμα, ενώ κάθε άλλο αντικείμενο ή επιφάνεια έχει καλυφθεί με μαύρο.

6.2 Τεχνική Prewitt

Με τον ίδιο τρόπο θα υλοποιήσουμε και την τεχνική prewitt. Το μόνο που αλλάζει είναι οι πίνακες των μασκών Prewitt. Η διαδικασία παραμένει ίδια.

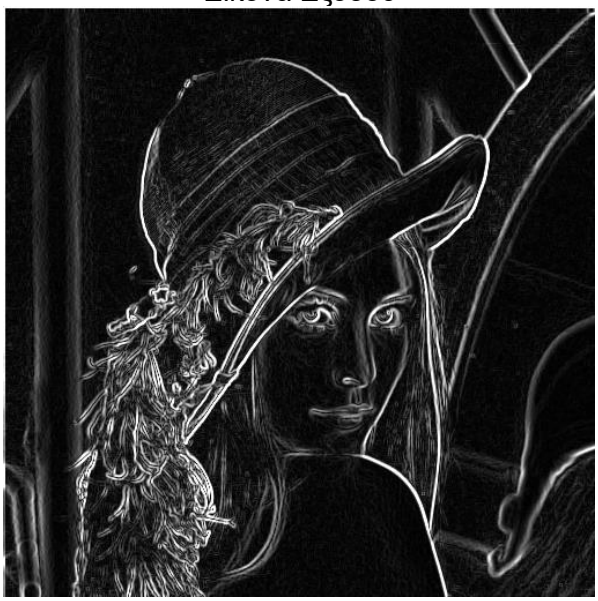
-1	0	1
-1	0	1
-1	0	1

1	1	1
0	0	0
-1	-1	-1

Μάσκες Prewitt

Εικόνα Εισόδου

Εικόνα Εξόδου



Η μόνη διαφορά από την Sobel όπως παρατηρούμε είναι ότι η εικόνα είναι ελαφρώς πιο σκοτεινή με λιγότερα γκρι σημεία και με λιγότερες ακμές σε επιφάνειες (π.χ. πρόσωπο).

6.3 Τεχνική Laplacian

Εδώ η τεχνική διαφέρει από τις 2 προηγούμενες αρκετά. Αρχικά, δεν έχουμε 2 kernels αλλά 1 και επίσης δεν χρειαζόμαστε το μέτρο της κλίσης, πράγμα που σημαίνει λιγότερες πράξεις. Ο λόγος είναι γιατί μπορούμε να προσεγγίσουμε τον τελεστή Lagrance από τον παρακάτω τύπο.

Προσέγγιση του τελεστή Laplace :

$$\nabla^2 f(x,y) \approx f(x,y) - \frac{1}{4}[f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y)]$$

Αυτό στον κώδικα μοντελοποιείται με την χρήση του παρακάτω πίνακα πυρήνα:

```
laplacian_kernel = np.array([[0, 1, 0],
                             [1, -4, 1],
                             [0, 1, 0]])
```

Επομένως, το μόνο που χρειάζεται είναι να εφαρμόσουμε 2D Συνέλιξη ανάμεσα στην ασπρόμαυρη εικόνα και τον laplacian_kernel. Το αποτέλεσμα αυτής της συνέλιξης είναι η εικόνα με το Edge Detection.

Εικόνα Εισόδου

Εικόνα Εξόδου

Όπως βλέπουμε, δεν έχουμε τόσο έντονες ακμές και ειδικά για τα αντικείμενα. Όμως είναι ξεκάθαρο ότι πρόκειται για έναν άνθρωπο στην φωτογραφία.

7. Σχολιασμός Αποτελεσμάτων - demo1.py

Αρχικά διαβάζουμε την έγχρωμη εικόνα.

7.1 Grayscale

Μετατρέπουμε την εικόνα σε ασπρόμαυρη.

Εικόνα A

7.2 Εισαγωγή θορύβου

Εισάγουμε στην εικόνα A γκαουσιανό θόρυβο με χρήση της myImNoise.

Εικόνα B

7.3 Φιλτράρισμα

Αποθορυβοποιούμε την εικόνα με χρήση της myImFilter.

Εικόνα C

Όπως παρατηρούμε, το φίλτρο είναι αποτελεσματικό στην αφαίρεση του τυχαίου θορύβου, όμως θολώνει την εικόνα αρκετά.

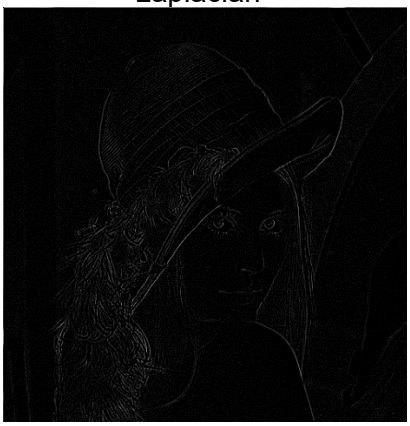
7.3 Ανίχνευση Ακμών

Τα αποτελέσματα και για τις τρεις τεχνικές πάνω στην εικόνα Α παρουσιάστηκαν και σχολιάστηκαν πλήρως στην Ενότητα 6.

Sobel

Prewitt

Laplacian



7.4 Συνδιασμός Φιλτραρίσματος και Ανίχνευσης Ακμών

Πριν την ανίχνευση ακμών κάνουμε χρήση της ImFilter με φίλτρο μέσου (mean) και κατασκευάζουμε τις εικόνες Ε (με μέγεθος πυρήνα 9) και F (με μέγεθος πυρήνα 3).

Εικόνα Ε

Εικόνα F

7.4.1 Τεχνική Sobel

Ανίχνευση Ακμών για E

Ανίχνευση Ακμών για F

Όπως παρατηρούμε, για μεγαλύτερο kernel_size δεν εντοπίζονται τόσες ακμές από λεπτομέρειες (π.χ. επιφάνεια προσώπου) μιας και μια ιδιότητα του φίλτρου μέσου είναι ότι εξομαλύνει τις ακμές.

7.4.2 Τεχνική Prewitt

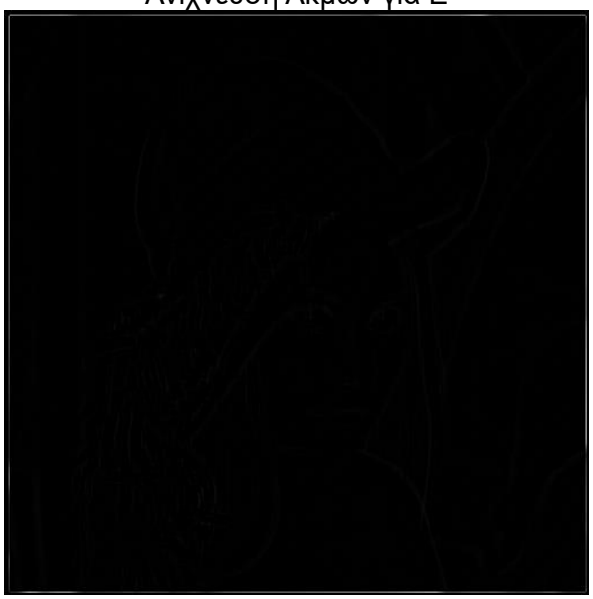
Ανίχνευση Ακμών για E

Ανίχνευση Ακμών για F

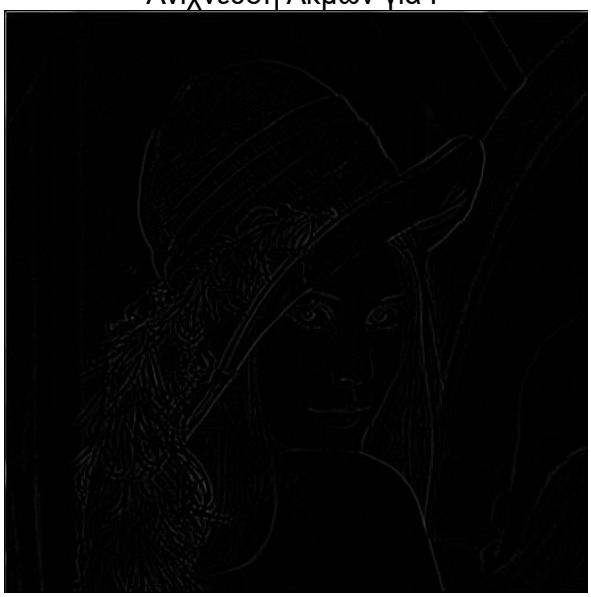
Με παρόμοιο τρόπο βλέπουμε να λειτουργεί και η τεχνική Prewitt μιας και το μόνο πράγμα στο οποίο διαφέρει είναι μια μικρή διαφορά στην φωτεινότητα της εικόνας.

7.4.3 Τεχνική Laplacian

Ανίχνευση Ακμών για E



Ανίχνευση Ακμών για F



Όπως βλέπουμε, η μέθοδος δυσκολεύεται πολύ (και κυρίως στην εικόνα E) για να βρει τις ακμές καθώς το φίλτρο που εφαρμόσαμε προηγουμένως επιδρά αρνητικά σε αυτές. Επίσης γνωρίζαμε από πριν πως αυτή η τεχνική είναι πιο «αδύναμη» ως προς την πληροφορία που παρέχει από τις 2 προηγούμενες.