Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

Εργασία 2: Hough, Harris, Rot και αποκοπή εικόνων

(Σημείωση: Επειδή χρησιμοποιώ την OpenCV για να ανοίγω τις εικόνες , πρέπει να βρίσκονται στο ίδιο path για να μπορεί να τρέξει ο κώδικας)

Στο πρώτο ερώτημα της συγκεκριμένης εργασίας μας ζητείται να υλοποιήσουμε τον μετασχηματισμό του Hough κατασκευάζοντας μια συνάρτηση που δέχεται ως είσοδο μια binary εικόνα η οποία έχει προέλθει από έναν edge detector πάνω σε μια grayscale εικόνα. Για edge detector χρησιμοποιήθηκε η βιβλιοθήκη της opencv. Επιλέχθηκε ο Canny edge detector διότι υπερτερεί γενικά έναντι της απόδοσης των υπόλοιπων αλγορίθμων ανίχνευσης ακμών. Για την εφαρμογή του Canny edge detector, εξομαλύνουμε αρχικά την εικόνα με ένα Gaussian φίλτρο, όπως προτείνεται από την θεωρία, επιλέγοντας μια συγκεκριμένη τιμή για την τυπική του απόκλιση και με αυτή την τιμή βρίσκουμε τις διαστάσεις του Gaussian φίλτρου οι οποίες σύμφωνα με την θεωρία πρέπει να είναι ο μικρότερος περιττός ακέραιος με τιμή μεγαλύτερη του 6σ. Για την εφαρμογή του Hough transform ακολουθήθηκαν 2 διαφορετικές υλοποιήσεις στον τρόπο με τον οποίο βρίσκω τις ευθείες που προκύπτουν. Ο πρώτος τρόπος ήταν με αναφορά τις διαφάνειες του μαθήματος όπου το $\rho \in [0, \rho_{max}]$ $και$ $\theta \in [0, 2\pi]$, ενώ ο δεύτερος τρόπος ήταν με αναφορά το βιβλίο του Gonzales(το οποίο συμβουλεύτηκα επίσης και για την επιλογή των thresholds στον Canny edge detector καθώς και για την επιλογή των διαστάσεων της Gaussian μάσκας), στο οποίο θεωρεί το $\rho \in [-\rho_{max}, \rho_{max}]$ $και$ $το$ $\theta \in [-90°, 90°]$. Μεταξύ αυτών των δύο υλοποιήσεων δεν παρατηρήθηκαν σημαντικές διαφορές. Ωστόσο, με την υλοποίηση με βάση το βιβλίο του Gonzales παρατήρησα ότι προκύπτουν περισσότερες γραμμές πάνω στην εικόνα από ότι με την υλοποίηση που παρουσιάζεται στις διαφάνειες. Παρακάτω, φαίνονται τα αποτελέσματα των 2 υλοποιήσεων, όπου πρώτα παρουσιάζω τα αποτελέσματα της δεύτερης υλοποίησης. Με την εξής επιλογή των φίλτρων **(n=15, d_rho = 1, d_theta = π/180◦)**:

```
img = cv2.GaussianBlur(img, ksize: (37,37), sigmaX = 6) # applying Gaussian mask to smoothe the image
edges = cv2.Canny(img, 25, 75) # applying Canny edge detector with these thresholds
```

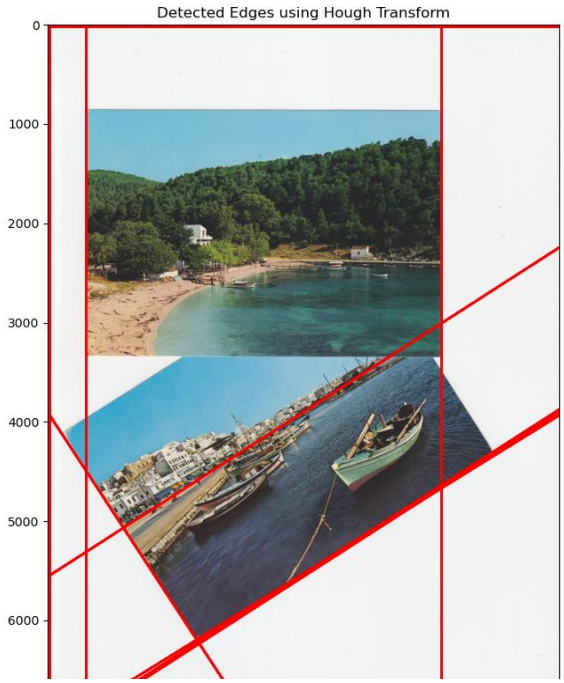


Figure 1 Δεύτερη Υλοποίηση Έγχρωμη Εικόνα

Figure 2 Δεύτερη Υλοποίηση Grayscale Εικόνα

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

Στην συνέχεια παρουσιάζω τα αποτελέσματα της πρώτης υλοποίησης δηλαδή με το $\rho \in [0, \rho_{max}]$ $και$ $\theta \in [0, 2\pi]$.

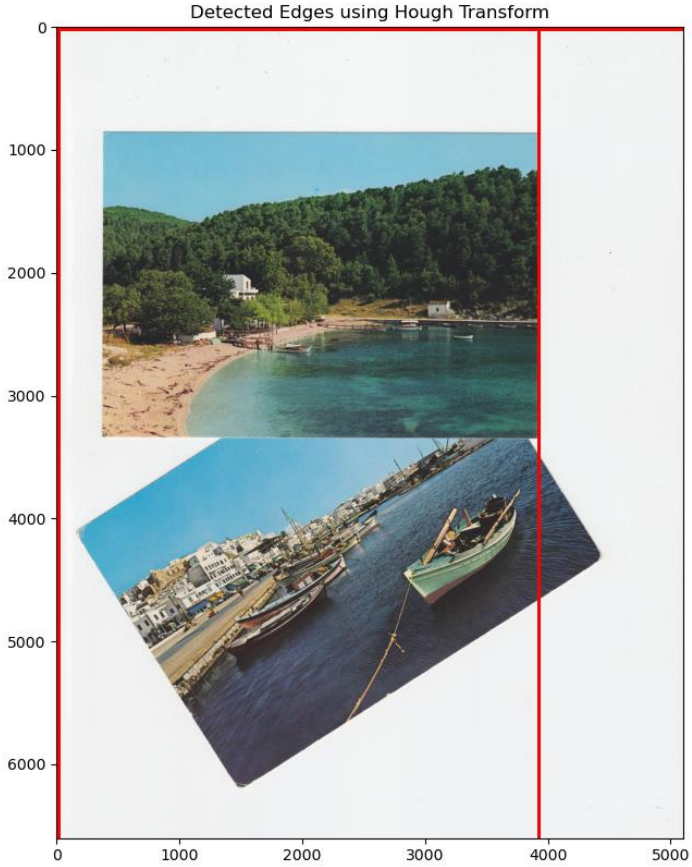


Figure 3 Πρώτη Υλοποίηση Έγχρωμη Εικόνα

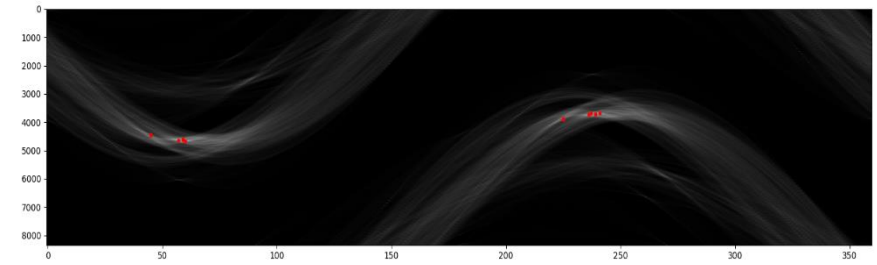
Figure 4 Πρώτη Υλοποίηση Grayscale Εικόνα

Όπου για λόγους σύγκρισης χρησιμοποιήθηκαν προφανώς οι ίδιες παράμετροι εισόδου. Τέλος παρουσιάζονται παρακάτω οι πίνακες μετασχηματισμού H για τις δύο υλοποιήσεις που αναφέρθηκαν προηγουμένως. Τα παρακάτω γραφήματα έγιναν για διαφορετικά thresholds του Canny edge detector και διαφορετική Gaussian μάσκα από ότι ο σχεδιασμός. Ο λόγος που έγινε αυτό είναι για να είναι διακριτά με το μάτι τα αποτελέσματα ειδάλλως οι εικόνες των πινάκων μετασχηματισμού ήταν σχεδόν μαύρες.

```
img = cv2.GaussianBlur(img, ksize: (7, 7), sigmaX=1)# applying Gaussian mask to smooth the image
edges = cv2.Canny(img, 100, 300)# applying Canny edge detector with these thresholds
```

Figure 5 Επιλογή Φίλτρων για τον Σχεδιασμό των Hough πινάκων

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335



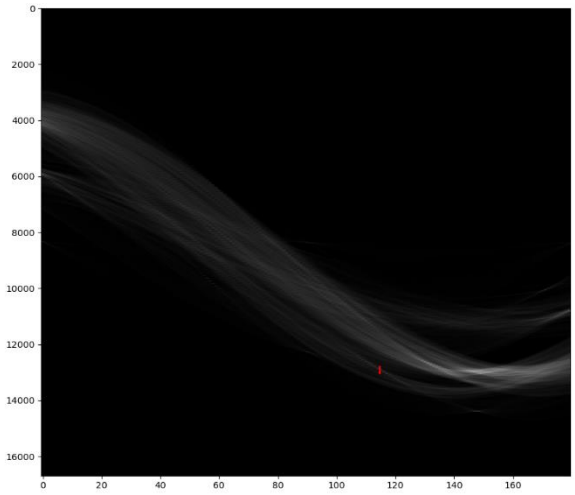


Figure 6 Πίνακας H για πρώτη υλοποίηση

Στην συνέχεια εξηγώ συνοπτικά λίγο τον κώδικα της πρώτης υλοποίησης

```
def my_hough_transform(img_binary, d_rho, d_theta, n):

    H, W = img_binary.shape

    rho_max = np.ceil(np.sqrt(H**2 + W**2)).astype(int) # get maximum rho value to be equal to the diagonal of the input image

    rho_values = np.arange(-rho_max, rho_max, d_rho) # create a rho vector with  d_rho step
    theta_values = np.arange(-np.pi/2, np.pi/2, d_theta) # create a theta vector with d_theta step

    Hough = np.zeros((2*rho_max, len(theta_values))) # initialize the Hough matrix

    edge_pixels = np.argwhere(img_binary > 0) # find all the edge pixels

    cos_theta = np.cos(theta_values) # vector of cosine values
    sin_theta = np.sin(theta_values) # vector of sine values


    x, y = edge_pixels[:,1], edge_pixels[:,0] # get the corresponding coordinates of the pixels

    rho = x[:,np.newaxis] * cos_theta + y[:,np.newaxis] * sin_theta # create a vector of all the rho values

    rho_index = np.floor(rho/d_rho).astype(int) # round down all the rho values as integers
    np.add.at(Hough, (rho_index + rho_max, np.arange(len(theta_values))), 1) # add one vote to the element with the
    # rho and theta index


    loc_max = find_local_maxima(Hough, n) # function to find the n greatest local maxima of the Hough matrix

    L = np.zeros((n,2)) # initializing the line matrix
    rho_index = loc_max[:, 0] # retreiving the rho indices
    theta_index = loc_max[:, 1] # retreiving the theta_indices

    L[:, 0] = rho_index - rho_max
    L[:, 1] = theta_values[theta_index]


   # Determine pixels not contributing to the local maxima
    maxima_votes = np.zeros((H, W), dtype=bool)

    for i, j in loc_max:
        rho = rho_values[i]
        theta = theta_values[j]
        for y, x in edge_pixels:
            rho_computed = x * np.cos(theta) + y * np.sin(theta)
            closest_rho_index = int(np.floor(rho_computed / d_rho))
            if closest_rho_index == i:
                maxima_votes[y, x] = True

    res = np.sum((img_binary > 0) & (~maxima_votes))

    return [Hough, L, res]
```

```
def find_local_maxima(array, n):

    masked_array = array

    array = np.pad(array, (1,1), mode='constant', constant_values = (0,0)) # padding the array with zeros

    mask = (array[1:-1, 1:-1] > array[:-2, 1:-1]) & \
           (array[1:-1, 1:-1] > array[2:, 1:-1]) & \
           (array[1:-1, 1:-1] > array[1:-1, :-2]) & \
           (array[1:-1, 1:-1] > array[1:-1, 2:]) & \
           (array[1:-1, 1:-1] > array[:-2, :-2]) & \
           (array[1:-1, 1:-1] > array[:-2, 2:]) & \
           (array[1:-1, 1:-1] > array[2:, :-2]) & \
           (array[1:-1, 1:-1] > array[2:, 2:]) # comparing each pixel with all its 8 neighbours to decide if it is a local maximum

    local_maxima = np.argwhere(mask) # getting the indices of all the local maxima

    # sorting the local maxima in declining order
    sorted_indices = np.argsort(masked_array[mask])[::-1]
    local_maxima = local_maxima[sorted_indices]

    if n >= len(local_maxima):
        return local_maxima
    else:
        return local_maxima[:n] # returning the first n local maxima
```

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

Αρχικά ορίζω την μέγιστη ακτίνα να είναι ίση με την διαγώνιο. Στην συνέχεια φτιάχνω δυο διανύσματα τιμών, ένα για τις γωνίες και ένα για τις ακτίνες με βήμα όσο τα ορίσματα d_rho και d_theta. Έπειτα, αρχικοποιώ τον πίνακα Hough να έχει πλήθος γραμμών όσο το διπλάσιο της μέγιστης ακτίνας και πλήθος στηλών όσο το πλήθος των γωνιών. Κατόπιν, βρίσκω τις ακμές και φτιάχνω το διάνυσμα rho κάνοντας τα εσωτερικά γινόμενα. Αφού υπολογίσω όλες αυτές τις ακτίνες για όλες τις ακμές που εντόπισα στην εικόνα, προσθέτω στον πίνακα Hough τις ψήφους με την εντολή $np.add.at$ η οποία προσθέτει μια ψήφο στο κάθε στοιχείο με το αντίστοιχο rho και theta index(ο τρόπος με τον υπολογίζω το index του rho είναι ότι σε κάθε rho προσθέτω την τιμή rho_max διότι με τον τρόπο που το υπολογίζω θα προκύψουν σίγουρα αρνητικά rho και επειδή αντιπροσωπεύουν δείκτες σε πίνακα θα ήθελα πάντα να είναι θετικά). Στην συνέχεια, αφού βρω τον Hough matrix με όλες τις ψήφους, εντοπίζω τα n τοπικά μέγιστα μέσω της δικιάς μου συνάρτησης $find\_local\_maxima(array, n)$ η οποία δέχεται σαν όρισμα έναν πίνακα και αφού τον κάνει κατάλληλα padding με μηδενικά συγκρίνει το κάθε pixel της εικόνας με τα 8 γειτονικά του και σε περίπτωση που είναι μεγαλύτερο από τους γείτονες του το εντάσσει στα τοπικά μέγιστα. Στην συνέχεια αφού βρω όλα τα τοπικά μέγιστα του Hough matrix τα ταξινομώ με φθίνουσα σειρά προκειμένου να μπορέσω να λάβω τα n μεγαλύτερα που με ενδιαφέρουν για τον σχεδιασμό των ευθειών πάνω στις εικόνες. Τέλος, αφού λάβω τα τοπικά μέγιστα από την συνάρτηση μου, παίρνω όλους τους δείκτες για τα rho και για τα theta και στην συνέχεια τους σώζω στον πίνακα των γραμμών L, αφού πρώτα κάνω την αντίστροφη διαδικασία από πριν για να ανακτήσω τις σωστές ακτίνες. Επίσης, για να βρω τα pixels που δεν συμμετέχουν στα n τοπικά μέγιστα δημιουργώ έναν πίνακα διαστάσεων όσο και η αρχική εικόνα στο οποίον κάθε τιμή είναι Boolean και μέσω του οποίου βάζω True στις τιμές που ανήκουν στις επικρατέστερες ευθείες και False στις υπόλοιπες, οπότε με αυτό τον τρόπο βρίσκω όλα τα pixels που ενώ είναι edges δεν ανήκουν στις επικρατέστερες ευθείες.

Η διαφορά της πρώτης υλοποίησης είναι ότι ο πίνακας Hough που φτιάχνω έχει διαστάσεις όσες το πλήθος των στοιχείων rho_values και ότι δεν χρειάζεται να προσθέτω και να αφαιρώ την μέγιστη τιμή του rho για να πάρω τους σωστούς δείκτες, διότι εδώ πέρα το ρ θα παίρνει μόνο θετικές τιμές και άρα μπορεί να αντιστοιχηθεί η τιμή του με την αντίστοιχη γραμμή του πίνακα Hough

(Σημείωση: Ο κώδικας που υπέβαλλα περιέχει την δεύτερη υλοποίηση με βάση το βιβλίο του Gonzales δηλαδή με $\boldsymbol{\rho} \in [-\boldsymbol{\rho_{max}}, \boldsymbol{\rho_{max}}]$ $\boldsymbol{και}$ $\boldsymbol{\theta} \in [-\frac{\boldsymbol{\pi}}{\boldsymbol{2}}, \frac{\boldsymbol{\pi}}{\boldsymbol{2}}]$)

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

HARRIS CORNER

Στο δεύτερο ερώτημα αυτής της εργασίας μας ζητείται να υλοποιήσουμε τον αλγόριθμο Harris Corner Detector για τον εντοπισμό σημείων ενδιαφέροντος πάνω στην εικόνα, όπως είναι δηλαδή οι ακμές, οι γωνίες καθώς επίσης και οι ομαλές περιοχές. Ο τρόπος με τον οποίο εντοπίζουμε αυτά τα σημεία ενδιαφέροντος είναι μέσω της εξής σχέσης :

$$R(p_1, p_2) = det(\mathbf{M}(p_1, p_2)) - kTrace(\mathbf{M}(p_1, p_2))^2$$

Με αυτό τον τρόπο κάθε pixel της εικόνας με συντεταγμένες (p_1, p_2) λαμβάνει μια τιμή η οποία αν είναι κάποια πολύ μεγάλη θετική τιμή θα καθιστά το σημείο ως γωνία, αν είναι μια πολύ μικρή τιμή τότε το pixel θα βρίσκεται σε ακμή, ενώ αν είναι αρνητική τότε το pixel θα θεωρείται ότι βρίσκεται σε ομοιόμορφη περιοχή. Στα πλαίσια της εργασίας, ενδιαφερόμαστε μόνοι για τον εντοπισμό γωνιών με την εφαρμογή του αλγορίθμου του Harris. Επομένως, για να μπορέσουμε να αποφασίσουμε αν ένα pixel ανήκει σε γωνία θα πρέπει η τιμή του $R(p_1, p_2)$ να ξεπερνάει κάποιο threshold. Για threshold επιλέχθηκε η τιμή 0.05, ενώ για την τιμή του k επιλέχθηκε πάλι το 0.05 και για τυπική απόκλιση για το Gaussian παράθυρο με το οποίο συνελίσεται ο πίνακας A, των οποίων τα στοιχεία περιέχουν τις παραγώγους της εικόνας , επιλέχθηκε η τιμή $\sigma = 1$. Τέλος, ο τρόπος με τον οποίο γίνεται ο υπολογισμός των παραγώγων είναι με την χρήση 3x3 μάσκας Sobel όπου με την μάσκα $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ υπολογίζω την μερική παράγωγο στην κατεύθυνση των x και με την μάσκα $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$ υπολογίζω την μερική παράγωγο στην κατεύθυνση των y. Τα αποτελέσματα του Harris corner για αυτές τις παραμέτρους που ανέφερα φαίνονται παρακάτω:

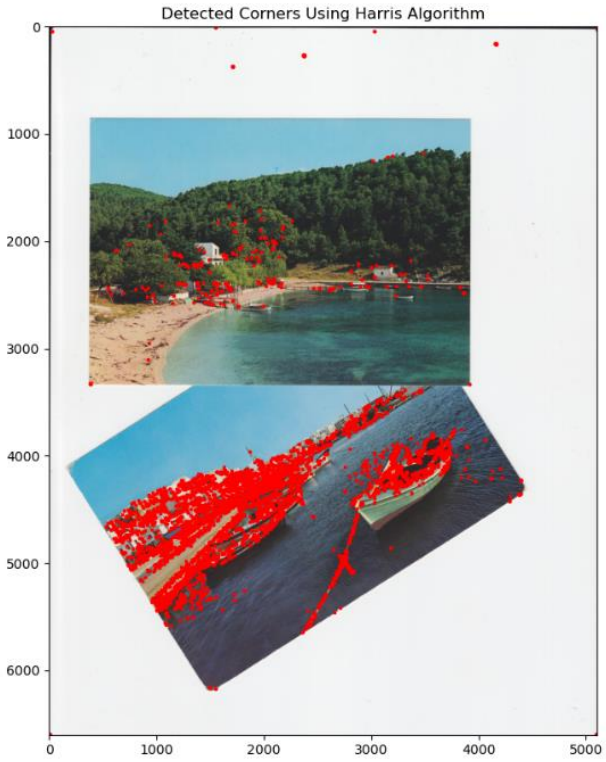


Figure 9 Harris Corner Έγχρωμη Εικόνα

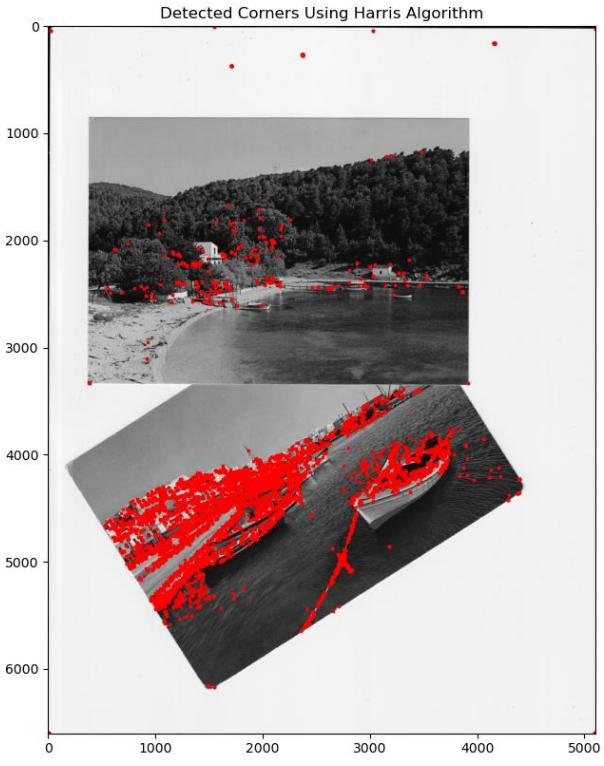


Figure 10 Harris Corner Grayscale

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

Τέλος, ακολουθεί μια σύντομη εξήγηση του κώδικα σε Python μέσω της οποίας πραγματοποιείται η υλοποίηση του αλγορίθμου Harris Corner Detector καθώς επίσης και η εύρεση αυτών των γωνιών.

```python
def my_corner_harris(img, k, sigma):

    Sobel_y_kernel = 0.25 * np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])# Sobel kernel to find the partial dervivative in the y axis
    Sobel_x_kernel = np.transpose(Sobel_y_kernel) # Sobel kernel to find the partial derivative in the x axis

    # Applying the masks to create elements of the A matrix
    I1 = signal.convolve2d(img, Sobel_x_kernel, mode='same')
    I2 = signal.convolve2d(img, Sobel_y_kernel, mode='same')

    I12 = I1 * I2
    I1  = I1 * I1
    I2  = I2 * I2

    # creating the Gaussian window to be odd in order to have a central pixel
    if round(4*sigma) % 2 == 0:
        N = round(4*sigma) - 1
    else:
        N = round(4*sigma)

    # creating the Gaussian window
    u1 = np.arange(-N, N + 1, 1)
    u2 = np.arange(-N, N + 1, 1)
    w = np.exp(-(u1[np.newaxis,:] ** 2 + u2[:, np.newaxis] ** 2) / 2*sigma**2)


    # applying the Gaussian mask to the elements of the A matrix therefore creating the M matrix
    I1  = signal.convolve2d(I1, w, mode='same')
    I2  = signal.convolve2d(I2, w, mode='same')
    I12 = signal.convolve2d(I12, w, mode='same')

    H, W = img.shape
    harris_response = np.zeros((H,W))


    # for efficiency (because the Image is EXTREMELY LARGE) I break it into chunks and
    chunk_size = 100

    # I calculate parts of the M matrix and finding the R value for a chunk of pixels of the original image

    for i in range(0, H, chunk_size):
        for j in range(0, W, chunk_size):

            i_end = min(i + chunk_size, H)
            j_end = min(j + chunk_size, W)


            I1_chunk = I1[i:i_end, j:j_end]
            I2_chunk = I2[i:i_end, j:j_end]
            I12_chunk = I12[i:i_end, j:j_end]


            M = np.empty((i_end - i, j_end - j, 2, 2))
            M[:, :, 0, 0] = I1_chunk
            M[:, :, 0, 1] = I12_chunk
            M[:, :, 1, 0] = I12_chunk
            M[:, :, 1, 1] = I2_chunk


            det_M = np.linalg.det(M)
            trace_M = np.trace(M, axis1=2, axis2=3)


            R_chunk = det_M - k * np.power(trace_M, 2)

            harris_response[i:i_end, j:j_end] = R_chunk


    return harris_response
```

```python
def my_corner_peaks(harris_response, rel_threshold):

    # I calculate the threshold for which I want to find the corners
    rel_threshold = harris_response.max() * rel_threshold
    # I am finding the indices for which harris_response values are greater than this threshold
    indices = np.argwhere(harris_response > rel_threshold)

    # creating an array with the pixel values where each value contains the coordinates of the pixels
    corner_locations = np.array(indices)

    return corner_locations
```

Αρχικά φτιάχνω τις μάσκες που προανέφερα με τις οποίες θα πραγματοποιήσω την συνέλιξη. Στην συνέχεια υπολογίζω τις μερικές παραγώγους της εικόνας και έπειτα φτιάχνω τον πίνακα Α όπως αναφέρεται στην εκφώνηση της εργασίας. Έπειτα ορίζω το Gaussian παράθυρο να μου δίνει πάντα τον μικρότερο περιττό έτσι ώστε να έχει πάντα ένα κεντρικό pixel και να είναι

συμμετρική η συνέλιξη της Gaussian μάσκας με τον πίνακα Α. Κατόπιν, για λόγους απόδοσης για να τρέχει πιο γρήγορα ο κώδικας σπάω την εικόνα σε μικρότερα chunks στα οποία βρίσκω για κάθε pixel το harris_response του. Αφού ολοκληρωθεί αυτή η διαδικασία, επιστρέφω το harris_response. Αφότου βρω το harris_response για κάθε pixel καλώ την συνάρτηση corner_peaks με την οποία θα αναγνωρίσω ποια pixels θα θεωρούνται γωνίες και ποια όχι. Αρχικά βρίσκω τους δείκτες του πίνακα harris_response(δηλαδή τις συντεταγμένες των pixels) για τους οποίους ισχύει ότι η τιμή τους είναι μεγαλύτερη από ένα threshold που έχουμε ορίσει.**(Σημείωση : Για την πραγματοποίηση της συνέλιξης με τις μάσκες χρησιμοποιήθηκε η $signal.convolve2d$ από την βιβλιοθήκη της $scipy$.)**

ROTATION

Στο τρίτο ερώτημα αυτής της εργασίας μας ζητείται να υλοποιήσουμε την περιστροφή μιας εικόνας κατά μια γωνία αντίστροφα από την φορά του ρολογιού ενώ η συνάρτηση μας θα πρέπει να λειτουργεί ανεξάρτητα από το αν η εικόνα είναι grayscale ή RGB. Για την πραγματοποίηση αυτής της λειτουργίας θα πρέπει επίσης να ρυθμίσουμε σωστά τις διαστάσεις της εικόνας και πραγματοποιήσουμε $bilinear\ interpolation$ στα pixel της εικόνας έτσι ώστε να βρούμε την νέα τιμή τους στην περιεστρεμμένη εικόνα. Παρακάτω παρατίθενται τα αποτελέσματα από την περιστροφή της εικόνας πρώτα κατά $54°$ μοίρες και έπειτα κατά $213°$ όπως ζητείται στην εκφώνηση, ενώ επίσης παρουσιάζονται τα αποτελέσματα και για grayscale και για RGB εικόνα.

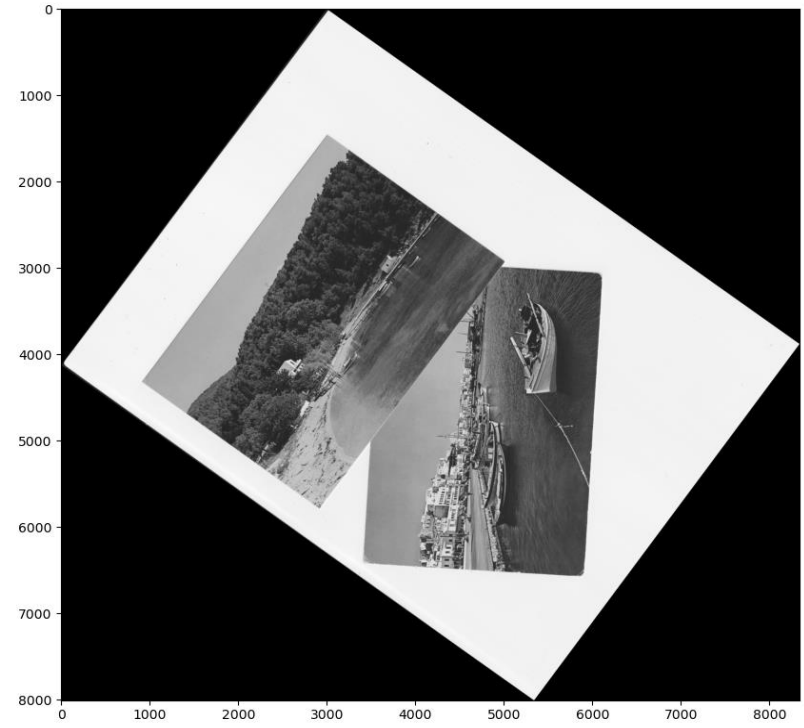


Figure 11 Περιστροφή κατά 54 μοίρες της Grayscale εικόνας

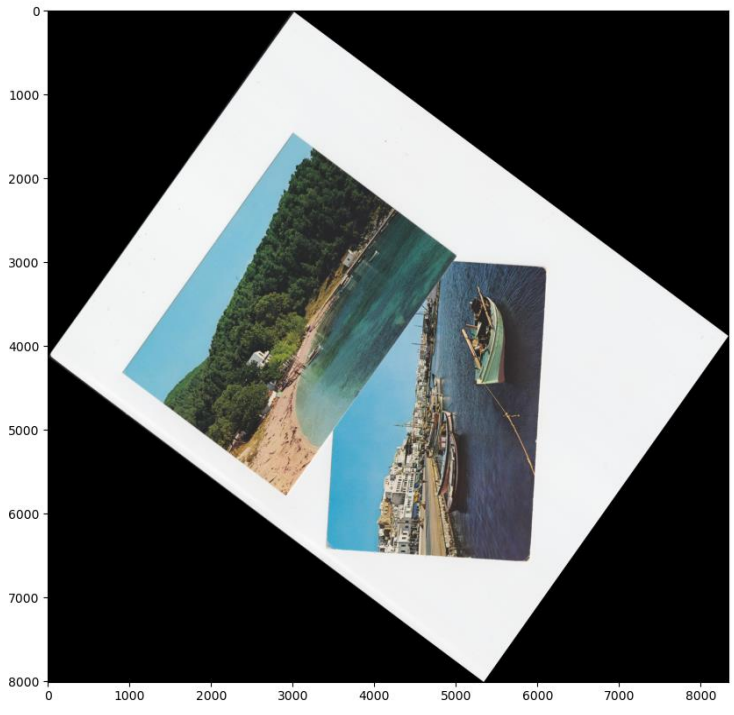


Figure 12 Περιστροφή κατά 54 μοίρες της RGB εικόνας

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

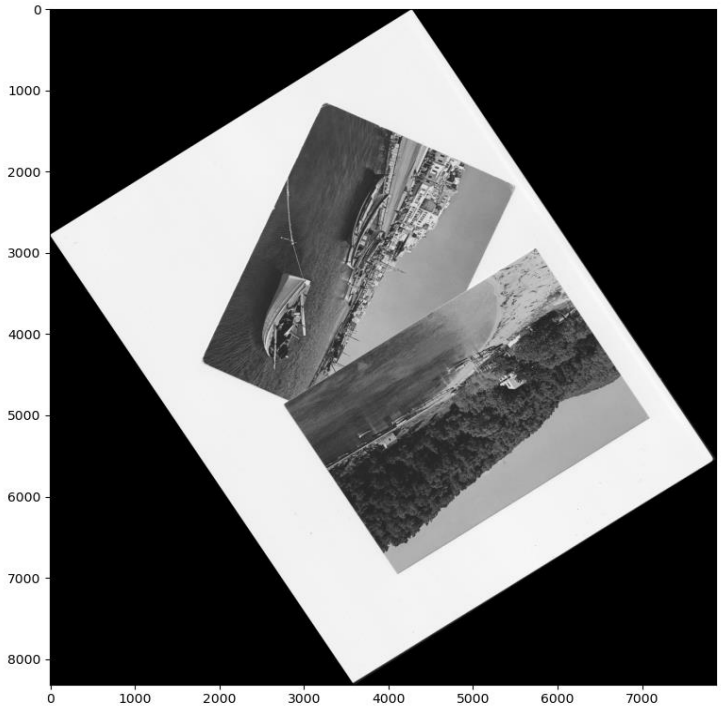


Figure 13 Περιστροφή κατά 213 μοίρες της Grayscale εικόνας

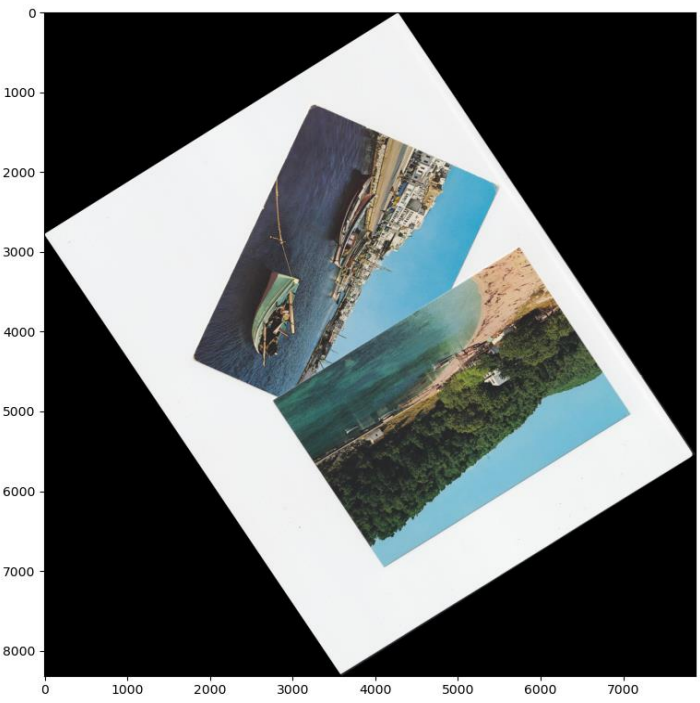


Figure 15 Περιστροφή κατά 213 μοίρες της RGB εικόνας

Χρήστος-Αλέξανδρος Δαρδαμπούνης ΑΕΜ : 10335

Τέλος, ακολουθεί μια σύντομη εξήγηση του κώδικα σε Python μέσω της οποίας πραγματοποιείται η υλοποίηση του rotation.

```python
def my_img_rotation(img, angle):
    h, w = img.shape[:2]

    # Determine the number of channels
    if len(img.shape) == 2:
        dim = 1
    else:
        dim = img.shape[2]

    sin_theta = np.sin(angle) # find the value of sin(theta)
    cos_theta = np.cos(angle) # find the value of cos(theta)

    # Calculate new image dimensions based on the rotation angle we want to perform
    new_h = int(np.abs(h * cos_theta) + np.abs(w * sin_theta))
    new_w = int(np.abs(w * cos_theta) + np.abs(h * sin_theta))

    # Initialize the rotated image
    if dim == 1:
        rot_img = np.zeros((new_h, new_w), dtype=img.dtype) # if image is Grayscale we don't need a third dimension
    else:
        rot_img = np.zeros((new_h, new_w, dim), dtype=img.dtype)


    new_center_x, new_center_y = new_w // 2, new_h // 2 # calculating the centers of the new image
    center_x, center_y = w // 2, h // 2 #  calculating the centers of the original image

    # Initializing the coordinates of the rotated image, as seen by the rotated frame of reference
    y_indices, x_indices = np.indices((new_h, new_w))

    # finding the coordinates of the original Image assuming we know the cordinates of pixels of the rotated Image
    x_coords = (x_indices - new_center_x) * cos_theta - (y_indices - new_center_y) * sin_theta + center_x
    y_coords = (x_indices - new_center_x) * sin_theta + (y_indices - new_center_y) * cos_theta + center_y

    x_coords_flat = x_coords.flatten()
    y_coords_flat = y_coords.flatten()

    # Pad the image to avoid out-of-bound errors, accordingly for Grayscale Image and for RGB image
    if dim == 1:
        img = np.pad(img, 1, mode='constant', constant_values=0)
    else:
        img = np.pad(img, ((1, 1), (1, 1), (0, 0)), mode='constant', constant_values=0)

    # creating a mask which holds the indices of the  x, y coordinates on which we are going to perform bilinear interpolation
    # to find the rotated Image
    valid_mask = (x_coords_flat >= 0) & (x_coords_flat < w) & (y_coords_flat >= 0) & (y_coords_flat < h)
```

```python
    valid_x = x_coords_flat[valid_mask]
    valid_y = y_coords_flat[valid_mask]

    valid_x_int = valid_x.astype(int)
    valid_y_int = valid_y.astype(int)

    if dim == 1:
        # Bilinear interpolation for grayscale images
        p2 = img[valid_y_int + 1, valid_x_int].astype(float)
        p8 = img[valid_y_int - 1, valid_x_int].astype(float)
        p4 = img[valid_y_int, valid_x_int - 1].astype(float)
        p6 = img[valid_y_int, valid_x_int + 1].astype(float)
        rot_values = (p2 + p4 + p6 + p8) / 4

        rot_img_flat = rot_img.flatten() # reducing the dimension of the Image to 1
        rot_img_flat[valid_mask] = rot_values # passing the values of the rotated pixels to the reduced Image
        rot_img = rot_img_flat.reshape((new_h, new_w)) # reshaping the array in order to correspond to a Grayscale Image
    else:
        # Bilinear interpolation for RGB images
        p2 = img[valid_y_int + 1, valid_x_int, :].astype(float)
        p8 = img[valid_y_int - 1, valid_x_int, :].astype(float)
        p4 = img[valid_y_int, valid_x_int - 1, :].astype(float)
        p6 = img[valid_y_int, valid_x_int + 1, :].astype(float)
        rot_values = (p2 + p4 + p6 + p8) / 4

        rot_img_flat = rot_img.reshape(-1, dim) # reducing the dimensions from 3D to a 2D array where every row of the array
        # represents a pixel and every column represents the corresponding value of each of the 3 channels
        rot_img_flat[valid_mask] = rot_values # assigning the corresponding rotation values to every pixel for all of the 3 channels
        rot_img = rot_img_flat.reshape((new_h, new_w, dim)) # reshaping the array to get the RGB Image

    return rot_img
```

Αρχικά ελέγχω στον κώδικα άμα η εικόνα εισόδου είναι Grayscale ή RGB και αφότου υπολογίσω το ημίτονο και το συνημίτονο για την γωνία περιστροφής, βρίσκω τις νέες διαστάσεις της με βάση αυτά και η καινούργια εικόνα που θα προκύψει θα έχει προφανώς μεγαλύτερες διαστάσεις προκειμένου να χωρέσει ολόκληρη την περιεστρεμμένη εικόνα. Στην συνέχεια, βρίσκω τα κέντρα της καινούργιας και της παλιάς εικόνας μέσω των οποίων θα μπορέσω να κάνω αντιστοίχιση των pixels της μετασχηματισμένης εικόνας με τα pixel της αρχικής. Ο τρόπος με τον οποίο το κάνω αυτό είναι χρησιμοποιώντας τον πίνακα στροφής για μια βασική στροφή γύρω από τον άξονα z στο 2D επίπεδο πάνω στις συντεταγμένες της

μετασχηματισμένης εικόνας με αναφορά το κέντρο της και μετατοπίζοντας κατάλληλα το κάθε ως προς το κέντρο της αρχικής εικόνας προκειμένου να λάβω τις συντεταγμένες των pixels της αρχικής εικόνας ως προς το πλαίσιο αναφοράς της περιεστρεμμένης. Αφού το κάνω αυτό κάνω κατάλληλο padding την εικόνα έτσι ώστε να μπορέσω να εφαρμόσω την διγραμμική παρεμβολή η οποία θα εφαρμοστεί μόνο στα pixels τα οποία βρίσκονται μέσα στα όρια της αρχικής εικόνας διότι έχει νόημα μόνο για αυτά να γίνει αυτή η διαδικασία. Τέλος, εφαρμόζω την διγραμμική παρεμβολή στα επιλεγμένα pixels της αρχικής εικόνας και με αυτό τον τρόπο υπολογίζω τις τιμές των pixels της μετασχηματισμένης.

ΠΡΟΒΛΗΜΑ

Όσον αφορά το κομμάτι του προβλήματος δεν κατάφερα ούτε να αποκόπτω τις σωστές εικόνες αλλά ούτε και να βρίσκω τον σωστό προσανατολισμό τους. Ωστόσο, το καλύτερο που κατάφερα είναι να εντοπίζω μέσω του Hough ορισμένες ευθείες, να βρίσκω τα σημεία τομής τους και μετά να κόβω ορισμένα ορθογώνια από αυτές τις εικόνες. Επειδή ο κώδικας μου δεν λειτουργεί όπως ζητείται από την εκφώνηση, δεν θα προβώ στην εξήγηση του και στην περαιτέρω ανάλυση του. Παρακάτω φαίνονται τα αποτελέσματα που παράγει μόνο για την εικόνα im2.jpg: