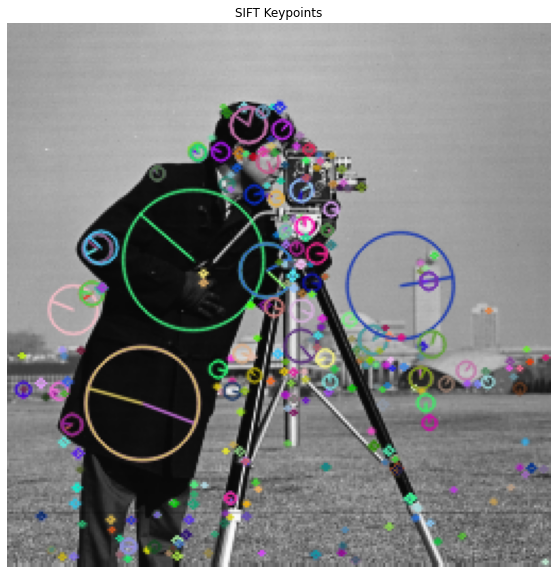
First 6 pictures using open CV built in SIFT

Original:



All Key Points:



10 Strongest Points:



Point 1: (182.26068115234375, 204.73240661621094)

Point 2: (182.26068115234375, 204.73240661621094)

Point 3: (111.76130676269531, 205.8682403564453)

Point 4: (111.76130676269531, 205.8682403564453)

Point 5: (139.72889709472656, 95.17049407958984)

Point 6: (134.54725646972656, 124.41365051269531)

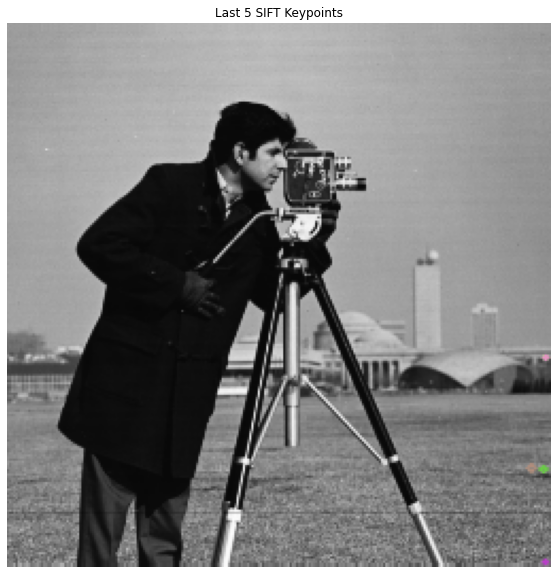
Point 7: (100.54109191894531, 173.00363159179688)

Point 8: (110.19285583496094, 155.21484375)

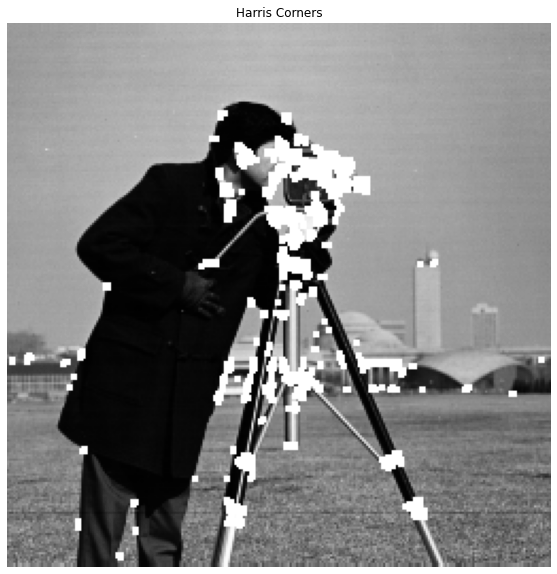
Point 9: (166.33738708496094, 76.77320098876953)

Point 10: (130.62796020507812, 113.03823852539062)

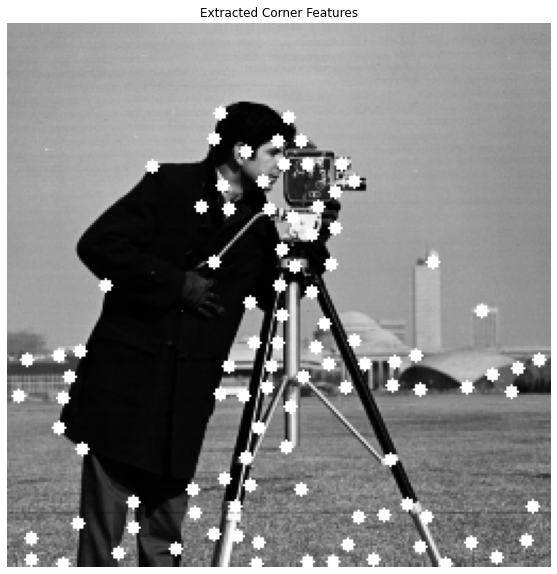
5 weakest points:



Harris Corners:



Extracted Corner Features:



Detected ALL Keypoints by implementing SIFT manually, not built in SIFT (Second code)



**Code using built in SIFT**

import cv2

import matplotlib.pyplot as plt

import numpy as np

# Load the image

image = cv2.imread("C:\\Users\\zezva\\Desktop\\Lab#7\\cameraman.tif", cv2.IMREAD\_GRAYSCALE)

# Create SIFT detector object

sift = cv2.SIFT\_create()

# Detect SIFT features

keypoints, descriptors = sift.detectAndCompute(image, None)

# Draw the keypoints on the image

image\_with\_keypoints = cv2.drawKeypoints(image, keypoints, None, flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

# Display the image with keypoints

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_keypoints, cmap='gray')

plt.title('SIFT Keypoints')

plt.axis('off')

plt.show()

#############################  Display the 10 strongest points

# Sort keypoints based on their response (strength)

keypoints = sorted(keypoints, key=lambda x: -x.response)

# Display the 10 strongest points

strongest\_keypoints = keypoints[:10]

image\_with\_strongest\_keypoints = cv2.drawKeypoints(image, strongest\_keypoints, None, flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

# Print the coordinates of the strongest keypoints

print("Coordinates of the 10 strongest keypoints:")

for i, kp in enumerate(strongest\_keypoints):

    print(f"Point {i+1}: ({kp.pt[0]}, {kp.pt[1]})")

# Display the image with the 10 strongest keypoints

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_strongest\_keypoints, cmap='gray')

plt.title('10 Strongest SIFT Keypoints')

plt.axis('off')

plt.show()

########################### Detect and display the last 5 SIFT features

# Display the last 5 detected points

last\_keypoints = keypoints[-5:]

image\_with\_last\_keypoints = cv2.drawKeypoints(image, last\_keypoints, None, flags=cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)

# Display the image with the last 5 keypoints

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_last\_keypoints, cmap='gray')

plt.title('Last 5 SIFT Keypoints')

plt.axis('off')

plt.show()

###########################   Detect Harris corner features and extract them

# Detect Harris corners

harris\_corners = cv2.cornerHarris(image, blockSize=2, ksize=3, k=0.04)

# Dilate corner image to enhance corner points

harris\_corners = cv2.dilate(harris\_corners, None)

# Threshold for an optimal value, it may vary depending on the image

threshold = 0.01 \* harris\_corners.max()

image\_with\_harris = image.copy()

image\_with\_harris[harris\_corners > threshold] = 255

# Display the Harris corners

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_harris, cmap='gray')

plt.title('Harris Corners')

plt.axis('off')

plt.show()

#########################   Extract Harris features:

# Find and extract corner features from the image

corners = cv2.goodFeaturesToTrack(image, maxCorners=100, qualityLevel=0.01, minDistance=10)

corners = np.int0(corners)

# Draw corners on the image

image\_with\_corners = image.copy()

for corner in corners:

    x, y = corner.ravel()

    cv2.circle(image\_with\_corners, (x, y), 3, 255, -1)

# Display the corners

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_corners, cmap='gray')

plt.title('Extracted Corner Features')

plt.axis('off')

plt.show()

**Code implementing “building” SIFT**

import cv2

import numpy as np

import logging

from functools import cmp\_to\_key

# Initialize logging

logger = logging.getLogger(\_\_name\_\_)

# Global variables

float\_tolerance = 1e-7

def computeKeypointsAndDescriptors(image, sigma=1.6, num\_intervals=3, assumed\_blur=0.5, image\_border\_width=5):

    """Compute SIFT keypoints and descriptors for an input image

    """

    image = image.astype('float32')

    base\_image = generateBaseImage(image, sigma, assumed\_blur)

    num\_octaves = computeNumberOfOctaves(base\_image.shape)

    gaussian\_kernels = generateGaussianKernels(sigma, num\_intervals)

    gaussian\_images = generateGaussianImages(base\_image, num\_octaves, gaussian\_kernels)

    dog\_images = generateDoGImages(gaussian\_images)

    keypoints = findScaleSpaceExtrema(gaussian\_images, dog\_images, num\_intervals, sigma, image\_border\_width)

    keypoints = removeDuplicateKeypoints(keypoints)

    keypoints = convertKeypointsToInputImageSize(keypoints)

    descriptors = generateDescriptors(keypoints, gaussian\_images)

    return keypoints, descriptors

def generateBaseImage(image, sigma, assumed\_blur):

    """Generate base image from input image by upsampling by 2 in both directions and blurring

    """

    logger.debug('Generating base image...')

    image = cv2.resize(image, (0, 0), fx=2, fy=2, interpolation=cv2.INTER\_LINEAR)

    sigma\_diff = np.sqrt(max((sigma \*\* 2) - ((2 \* assumed\_blur) \*\* 2), 0.01))

    return cv2.GaussianBlur(image, (0, 0), sigmaX=sigma\_diff, sigmaY=sigma\_diff)

def computeNumberOfOctaves(image\_shape):

    """Compute number of octaves in image pyramid as function of base image shape (OpenCV default)

    """

    return int(round(np.log(min(image\_shape)) / np.log(2) - 1))

def generateGaussianKernels(sigma, num\_intervals):

    """Generate list of gaussian kernels at which to blur the input image. Default values of sigma, intervals, and octaves follow section 3 of Lowe's paper.

    """

    logger.debug('Generating scales...')

    num\_images\_per\_octave = num\_intervals + 3

    k = 2 \*\* (1. / num\_intervals)

    gaussian\_kernels = np.zeros(num\_images\_per\_octave)

    gaussian\_kernels[0] = sigma

    for image\_index in range(1, num\_images\_per\_octave):

        sigma\_previous = (k \*\* (image\_index - 1)) \* sigma

        sigma\_total = k \* sigma\_previous

        gaussian\_kernels[image\_index] = np.sqrt(sigma\_total \*\* 2 - sigma\_previous \*\* 2)

    return gaussian\_kernels

def generateGaussianImages(image, num\_octaves, gaussian\_kernels):

    """Generate scale-space pyramid of Gaussian images

    """

    logger.debug('Generating Gaussian images...')

    gaussian\_images = []

    for octave\_index in range(num\_octaves):

        gaussian\_images\_in\_octave = []

        gaussian\_images\_in\_octave.append(image)

        for gaussian\_kernel in gaussian\_kernels[1:]:

            image = cv2.GaussianBlur(image, (0, 0), sigmaX=gaussian\_kernel, sigmaY=gaussian\_kernel)

            gaussian\_images\_in\_octave.append(image)

        gaussian\_images.append(gaussian\_images\_in\_octave)

        octave\_base = gaussian\_images\_in\_octave[-3]

        image = cv2.resize(octave\_base, (int(octave\_base.shape[1] / 2), int(octave\_base.shape[0] / 2)), interpolation=cv2.INTER\_NEAREST)

    return np.array(gaussian\_images, dtype=object)

def generateDoGImages(gaussian\_images):

    """Generate Difference-of-Gaussians image pyramid

    """

    logger.debug('Generating Difference-of-Gaussian images...')

    dog\_images = []

    for gaussian\_images\_in\_octave in gaussian\_images:

        dog\_images\_in\_octave = []

        for first\_image, second\_image in zip(gaussian\_images\_in\_octave, gaussian\_images\_in\_octave[1:]):

            dog\_images\_in\_octave.append(cv2.subtract(second\_image, first\_image))

        dog\_images.append(dog\_images\_in\_octave)

    return np.array(dog\_images, dtype=object)

def findScaleSpaceExtrema(gaussian\_images, dog\_images, num\_intervals, sigma, image\_border\_width, contrast\_threshold=0.04):

    """Find pixel positions of all scale-space extrema in the image pyramid

    """

    logger.debug('Finding scale-space extrema...')

    threshold = np.floor(0.5 \* contrast\_threshold / num\_intervals \* 255)

    keypoints = []

    for octave\_index, dog\_images\_in\_octave in enumerate(dog\_images):

        for image\_index, (first\_image, second\_image, third\_image) in enumerate(zip(dog\_images\_in\_octave, dog\_images\_in\_octave[1:], dog\_images\_in\_octave[2:])):

            for i in range(image\_border\_width, first\_image.shape[0] - image\_border\_width):

                for j in range(image\_border\_width, first\_image.shape[1] - image\_border\_width):

                    if isPixelAnExtremum(first\_image[i-1:i+2, j-1:j+2], second\_image[i-1:i+2, j-1:j+2], third\_image[i-1:i+2, j-1:j+2], threshold):

                        localization\_result = localizeExtremumViaQuadraticFit(i, j, image\_index + 1, octave\_index, num\_intervals, dog\_images\_in\_octave, sigma, contrast\_threshold, image\_border\_width)

                        if localization\_result is not None:

                            keypoint, localized\_image\_index = localization\_result

                            keypoints\_with\_orientations = computeKeypointsWithOrientations(keypoint, octave\_index, gaussian\_images[octave\_index][localized\_image\_index])

                            for keypoint\_with\_orientation in keypoints\_with\_orientations:

                                keypoints.append(keypoint\_with\_orientation)

    return keypoints

def isPixelAnExtremum(first\_subimage, second\_subimage, third\_subimage, threshold):

    """Return True if the center element of the 3x3x3 input array is strictly greater than or less than all its neighbors, False otherwise

    """

    center\_pixel\_value = second\_subimage[1, 1]

    if abs(center\_pixel\_value) > threshold:

        if center\_pixel\_value > 0:

            return np.all(center\_pixel\_value >= first\_subimage) and \

                   np.all(center\_pixel\_value >= third\_subimage) and \

                   np.all(center\_pixel\_value >= second\_subimage[0, :]) and \

                   np.all(center\_pixel\_value >= second\_subimage[2, :]) and \

                   center\_pixel\_value >= second\_subimage[1, 0] and \

                   center\_pixel\_value >= second\_subimage[1, 2]

        elif center\_pixel\_value < 0:

            return np.all(center\_pixel\_value <= first\_subimage) and \

                   np.all(center\_pixel\_value <= third\_subimage) and \

                   np.all(center\_pixel\_value <= second\_subimage[0, :]) and \

                   np.all(center\_pixel\_value <= second\_subimage[2, :]) and \

                   center\_pixel\_value <= second\_subimage[1, 0] and \

                   center\_pixel\_value <= second\_subimage[1, 2]

    return False

def localizeExtremumViaQuadraticFit(i, j, image\_index, octave\_index, num\_intervals, dog\_images\_in\_octave, sigma, contrast\_threshold, image\_border\_width, eigenvalue\_ratio=10, num\_attempts\_until\_convergence=5):

    """Iteratively refine pixel positions of scale-space extrema via quadratic fit around each extremum's neighbors

    """

    logger.debug('Localizing scale-space extrema...')

    extremum\_is\_outside\_image = False

    image\_shape = dog\_images\_in\_octave[0].shape

    for attempt\_index in range(num\_attempts\_until\_convergence):

        first\_image, second\_image, third\_image = dog\_images\_in\_octave[image\_index-1:image\_index+2]

        pixel\_cube = np.stack([first\_image[i-1:i+2, j-1:j+2],

                               second\_image[i-1:i+2, j-1:j+2],

                               third\_image[i-1:i+2, j-1:j+2]]).astype('float32') / 255.

        gradient = computeGradientAtCenterPixel(pixel\_cube)

        hessian = computeHessianAtCenterPixel(pixel\_cube)

        extremum\_update = -np.linalg.lstsq(hessian, gradient, rcond=None)[0]

        if abs(extremum\_update[0]) < 0.5 and abs(extremum\_update[1]) < 0.5 and abs(extremum\_update[2]) < 0.5:

            break

        j += int(round(extremum\_update[0]))

        i += int(round(extremum\_update[1]))

        image\_index += int(round(extremum\_update[2]))

        if i < image\_border\_width or i >= image\_shape[0] - image\_border\_width or j < image\_border\_width or j >= image\_shape[1] - image\_border\_width or image\_index < 1 or image\_index > num\_intervals:

            extremum\_is\_outside\_image = True

            break

    if extremum\_is\_outside\_image:

        logger.debug('Updated extremum moved outside of image before reaching convergence. Skipping...')

        return None

    if attempt\_index >= num\_attempts\_until\_convergence - 1:

        logger.debug('Exceeded maximum number of attempts without reaching convergence for this extremum. Skipping...')

        return None

    functionValueAtUpdatedExtremum = pixel\_cube[1, 1, 1] + 0.5 \* np.dot(gradient, extremum\_update)

    if abs(functionValueAtUpdatedExtremum) \* num\_intervals >= contrast\_threshold:

        xy\_hessian = hessian[:2, :2]

        xy\_hessian\_trace = np.trace(xy\_hessian)

        xy\_hessian\_det = np.linalg.det(xy\_hessian)

        if xy\_hessian\_det > 0 and eigenvalue\_ratio \* (xy\_hessian\_trace \*\* 2) < ((eigenvalue\_ratio + 1) \*\* 2) \* xy\_hessian\_det:

            keypoint = cv2.KeyPoint()

            keypoint.pt = ((j + extremum\_update[0]) \* (2 \*\* octave\_index), (i + extremum\_update[1]) \* (2 \*\* octave\_index))

            keypoint.octave = octave\_index + image\_index \* (2 \*\* 8) + int(round((extremum\_update[2] + 0.5) \* 255)) \* (2 \*\* 16)

            keypoint.size = sigma \* (2 \*\* ((image\_index + extremum\_update[2]) / np.float32(num\_intervals))) \* (2 \*\* (octave\_index + 1))

            keypoint.response = abs(functionValueAtUpdatedExtremum)

            return keypoint, image\_index

    return None

def computeGradientAtCenterPixel(pixel\_array):

    """Approximate gradient at center pixel [1, 1, 1] of 3x3x3 array using central difference formula of order O(h^2), where h is the step size

    """

    dx = 0.5 \* (pixel\_array[1, 1, 2] - pixel\_array[1, 1, 0])

    dy = 0.5 \* (pixel\_array[1, 2, 1] - pixel\_array[1, 0, 1])

    ds = 0.5 \* (pixel\_array[2, 1, 1] - pixel\_array[0, 1, 1])

    return np.array([dx, dy, ds])

def computeHessianAtCenterPixel(pixel\_array):

    """Approximate Hessian at center pixel [1, 1, 1] of 3x3x3 array using central difference formula of order O(h^2), where h is the step size

    """

    center\_pixel\_value = pixel\_array[1, 1, 1]

    dxx = pixel\_array[1, 1, 2] - 2 \* center\_pixel\_value + pixel\_array[1, 1, 0]

    dyy = pixel\_array[1, 2, 1] - 2 \* center\_pixel\_value + pixel\_array[1, 0, 1]

    dss = pixel\_array[2, 1, 1] - 2 \* center\_pixel\_value + pixel\_array[0, 1, 1]

    dxy = 0.25 \* (pixel\_array[1, 2, 2] - pixel\_array[1, 2, 0] - pixel\_array[1, 0, 2] + pixel\_array[1, 0, 0])

    dxs = 0.25 \* (pixel\_array[2, 1, 2] - pixel\_array[2, 1, 0] - pixel\_array[0, 1, 2] + pixel\_array[0, 1, 0])

    dys = 0.25 \* (pixel\_array[2, 2, 1] - pixel\_array[2, 0, 1] - pixel\_array[0, 2, 1] + pixel\_array[0, 0, 1])

    return np.array([[dxx, dxy, dxs],

                     [dxy, dyy, dys],

                     [dxs, dys, dss]])

def computeKeypointsWithOrientations(keypoint, octave\_index, gaussian\_image, radius\_factor=3, num\_bins=36, peak\_ratio=0.8, scale\_factor=1.5):

    """Compute orientations for each keypoint

    """

    logger.debug('Computing keypoint orientations...')

    keypoints\_with\_orientations = []

    image\_shape = gaussian\_image.shape

    scale = scale\_factor \* keypoint.size / np.float32(2 \*\* (octave\_index + 1))

    radius = int(round(radius\_factor \* scale))

    weight\_factor = -0.5 / (scale \*\* 2)

    raw\_histogram = np.zeros(num\_bins)

    smooth\_histogram = np.zeros(num\_bins)

    for i in range(-radius, radius + 1):

        region\_y = int(round(keypoint.pt[1] / np.float32(2 \*\* octave\_index))) + i

        if region\_y > 0 and region\_y < image\_shape[0] - 1:

            for j in range(-radius, radius + 1):

                region\_x = int(round(keypoint.pt[0] / np.float32(2 \*\* octave\_index))) + j

                if region\_x > 0 and region\_x < image\_shape[1] - 1:

                    dx = gaussian\_image[region\_y, region\_x + 1] - gaussian\_image[region\_y, region\_x - 1]

                    dy = gaussian\_image[region\_y - 1, region\_x] - gaussian\_image[region\_y + 1, region\_x]

                    gradient\_magnitude = np.sqrt(dx \* dx + dy \* dy)

                    gradient\_orientation = np.rad2deg(np.arctan2(dy, dx))

                    weight = np.exp(weight\_factor \* (i \*\* 2 + j \*\* 2))

                    histogram\_index = int(round(gradient\_orientation \* num\_bins / 360.))

                    raw\_histogram[histogram\_index % num\_bins] += weight \* gradient\_magnitude

    for n in range(num\_bins):

        smooth\_histogram[n] = (6 \* raw\_histogram[n] + 4 \* (raw\_histogram[n - 1] + raw\_histogram[(n + 1) % num\_bins]) + raw\_histogram[n - 2] + raw\_histogram[(n + 2) % num\_bins]) / 16.

    orientation\_max = max(smooth\_histogram)

    orientation\_peaks = np.where(np.logical\_and(smooth\_histogram > np.roll(smooth\_histogram, 1), smooth\_histogram > np.roll(smooth\_histogram, -1)))[0]

    for peak\_index in orientation\_peaks:

        peak\_value = smooth\_histogram[peak\_index]

        if peak\_value >= peak\_ratio \* orientation\_max:

            left\_value = smooth\_histogram[(peak\_index - 1) % num\_bins]

            right\_value = smooth\_histogram[(peak\_index + 1) % num\_bins]

            interpolated\_peak\_index = (peak\_index + 0.5 \* (left\_value - right\_value) / (left\_value - 2 \* peak\_value + right\_value)) % num\_bins

            orientation = 360. - interpolated\_peak\_index \* 360. / num\_bins

            if abs(orientation - 360.) < float\_tolerance:

                orientation = 0

            new\_keypoint = cv2.KeyPoint(\*keypoint.pt, keypoint.size, orientation, keypoint.response, keypoint.octave)

            keypoints\_with\_orientations.append(new\_keypoint)

    return keypoints\_with\_orientations

def compareKeypoints(keypoint1, keypoint2):

    """Return True if keypoint1 is less than keypoint2

    """

    if keypoint1.pt[0] != keypoint2.pt[0]:

        return keypoint1.pt[0] - keypoint2.pt[0]

    if keypoint1.pt[1] != keypoint2.pt[1]:

        return keypoint1.pt[1] - keypoint2.pt[1]

    if keypoint1.size != keypoint2.size:

        return keypoint2.size - keypoint1.size

    if keypoint1.angle != keypoint2.angle:

        return keypoint1.angle - keypoint2.angle

    if keypoint1.response != keypoint2.response:

        return keypoint2.response - keypoint1.response

    if keypoint1.octave != keypoint2.octave:

        return keypoint2.octave - keypoint1.octave

    return keypoint2.class\_id - keypoint1.class\_id

def removeDuplicateKeypoints(keypoints):

    """Sort keypoints and remove duplicate keypoints

    """

    if len(keypoints) < 2:

        return keypoints

    keypoints.sort(key=cmp\_to\_key(compareKeypoints))

    unique\_keypoints = [keypoints[0]]

    for next\_keypoint in keypoints[1:]:

        last\_unique\_keypoint = unique\_keypoints[-1]

        if last\_unique\_keypoint.pt[0] != next\_keypoint.pt[0] or \

           last\_unique\_keypoint.pt[1] != next\_keypoint.pt[1] or \

           last\_unique\_keypoint.size != next\_keypoint.size or \

           last\_unique\_keypoint.angle != next\_keypoint.angle:

            unique\_keypoints.append(next\_keypoint)

    return unique\_keypoints

def convertKeypointsToInputImageSize(keypoints):

    """Convert keypoint point, size, and octave to input image size

    """

    converted\_keypoints = []

    for keypoint in keypoints:

        keypoint.pt = tuple(0.5 \* np.array(keypoint.pt))

        keypoint.size \*= 0.5

        keypoint.octave = (keypoint.octave & ~255) | ((keypoint.octave - 1) & 255)

        converted\_keypoints.append(keypoint)

    return converted\_keypoints

def unpackOctave(keypoint):

    """Compute octave, layer, and scale from a keypoint

    """

    octave = keypoint.octave & 255

    layer = (keypoint.octave >> 8) & 255

    if octave >= 128:

        octave = octave | -128

    scale = 1 / np.float32(1 << octave) if octave >= 0 else np.float32(1 << -octave)

    return octave, layer, scale

def generateDescriptors(keypoints, gaussian\_images, window\_width=4, num\_bins=8, scale\_multiplier=3, descriptor\_max\_value=0.2):

    """Generate descriptors for each keypoint

    """

    logger.debug('Generating descriptors...')

    descriptors = []

    for keypoint in keypoints:

        octave, layer, scale = unpackOctave(keypoint)

        gaussian\_image = gaussian\_images[octave + 1, layer]

        num\_rows, num\_cols = gaussian\_image.shape

        point = np.round(scale \* np.array(keypoint.pt)).astype('int')

        bins\_per\_degree = num\_bins / 360.

        angle = 360. - keypoint.angle

        cos\_angle = np.cos(np.deg2rad(angle))

        sin\_angle = np.sin(np.deg2rad(angle))

        weight\_multiplier = -0.5 / ((0.5 \* window\_width) \*\* 2)

        row\_bin\_list = []

        col\_bin\_list = []

        magnitude\_list = []

        orientation\_bin\_list = []

        histogram\_tensor = np.zeros((window\_width + 2, window\_width + 2, num\_bins))

        hist\_width = scale\_multiplier \* 0.5 \* scale \* keypoint.size

        half\_width = int(round(hist\_width \* np.sqrt(2) \* (window\_width + 1) \* 0.5))

        half\_width = int(min(half\_width, np.sqrt(num\_rows \*\* 2 + num\_cols \*\* 2)))

        for row in range(-half\_width, half\_width + 1):

            for col in range(-half\_width, half\_width + 1):

                row\_rot = col \* sin\_angle + row \* cos\_angle

                col\_rot = col \* cos\_angle - row \* sin\_angle

                row\_bin = (row\_rot / hist\_width) + 0.5 \* window\_width - 0.5

                col\_bin = (col\_rot / hist\_width) + 0.5 \* window\_width - 0.5

                if row\_bin > -1 and row\_bin < window\_width and col\_bin > -1 and col\_bin < window\_width:

                    window\_row = int(round(point[1] + row))

                    window\_col = int(round(point[0] + col))

                    if window\_row > 0 and window\_row < num\_rows - 1 and window\_col > 0 and window\_col < num\_cols - 1:

                        dx = gaussian\_image[window\_row, window\_col + 1] - gaussian\_image[window\_row, window\_col - 1]

                        dy = gaussian\_image[window\_row - 1, window\_col] - gaussian\_image[window\_row + 1, window\_col]

                        gradient\_magnitude = np.sqrt(dx \* dx + dy \* dy)

                        gradient\_orientation = np.rad2deg(np.arctan2(dy, dx)) % 360

                        weight = np.exp(weight\_multiplier \* ((row\_rot / hist\_width) \*\* 2 + (col\_rot / hist\_width) \*\* 2))

                        row\_bin\_list.append(row\_bin)

                        col\_bin\_list.append(col\_bin)

                        magnitude\_list.append(weight \* gradient\_magnitude)

                        orientation\_bin\_list.append((gradient\_orientation - angle) \* bins\_per\_degree)

        for row\_bin, col\_bin, magnitude, orientation\_bin in zip(row\_bin\_list, col\_bin\_list, magnitude\_list, orientation\_bin\_list):

            row\_bin\_floor, col\_bin\_floor, orientation\_bin\_floor = np.floor([row\_bin, col\_bin, orientation\_bin]).astype(int)

            row\_fraction, col\_fraction, orientation\_fraction = row\_bin - row\_bin\_floor, col\_bin - col\_bin\_floor, orientation\_bin - orientation\_bin\_floor

            if orientation\_bin\_floor < 0:

                orientation\_bin\_floor += num\_bins

            if orientation\_bin\_floor >= num\_bins:

                orientation\_bin\_floor -= num\_bins

            c1 = magnitude \* row\_fraction

            c0 = magnitude \* (1 - row\_fraction)

            c11 = c1 \* col\_fraction

            c10 = c1 \* (1 - col\_fraction)

            c01 = c0 \* col\_fraction

            c00 = c0 \* (1 - col\_fraction)

            c111 = c11 \* orientation\_fraction

            c110 = c11 \* (1 - orientation\_fraction)

            c101 = c10 \* orientation\_fraction

            c100 = c10 \* (1 - orientation\_fraction)

            c011 = c01 \* orientation\_fraction

            c010 = c01 \* (1 - orientation\_fraction)

            c001 = c00 \* orientation\_fraction

            c000 = c00 \* (1 - orientation\_fraction)

            histogram\_tensor[row\_bin\_floor + 1, col\_bin\_floor + 1, orientation\_bin\_floor] += c000

            histogram\_tensor[row\_bin\_floor + 1, col\_bin\_floor + 1, (orientation\_bin\_floor + 1) % num\_bins] += c001

            histogram\_tensor[row\_bin\_floor + 1, col\_bin\_floor + 2, orientation\_bin\_floor] += c010

            histogram\_tensor[row\_bin\_floor + 1, col\_bin\_floor + 2, (orientation\_bin\_floor + 1) % num\_bins] += c011

            histogram\_tensor[row\_bin\_floor + 2, col\_bin\_floor + 1, orientation\_bin\_floor] += c100

            histogram\_tensor[row\_bin\_floor + 2, col\_bin\_floor + 1, (orientation\_bin\_floor + 1) % num\_bins] += c101

            histogram\_tensor[row\_bin\_floor + 2, col\_bin\_floor + 2, orientation\_bin\_floor] += c110

            histogram\_tensor[row\_bin\_floor + 2, col\_bin\_floor + 2, (orientation\_bin\_floor + 1) % num\_bins] += c111

        descriptor\_vector = histogram\_tensor[1:-1, 1:-1, :].flatten()

        threshold = np.linalg.norm(descriptor\_vector) \* descriptor\_max\_value

        descriptor\_vector[descriptor\_vector > threshold] = threshold

        descriptor\_vector /= max(np.linalg.norm(descriptor\_vector), float\_tolerance)

        descriptor\_vector = np.round(512 \* descriptor\_vector)

        descriptor\_vector[descriptor\_vector < 0] = 0

        descriptor\_vector[descriptor\_vector > 255] = 255

        descriptors.append(descriptor\_vector)

    return np.array(descriptors, dtype='float32')

# Example usage:

image = cv2.imread("C:\\Users\\zezva\\Desktop\\Lab#7\\cameraman.tif", 0)

keypoints, descriptors = computeKeypointsAndDescriptors(image)

import matplotlib.pyplot as plt

# Draw keypoints on the image

image\_with\_keypoints = cv2.drawKeypoints(image, keypoints, None, color=(0, 255, 0))

# Display the image with keypoints

plt.figure(figsize=(10, 10))

plt.imshow(image\_with\_keypoints, cmap='gray')

plt.title('Keypoints Detected')

plt.show()