First 6 pictures using open CV built in SIFT

Original:

Keypoints:

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)

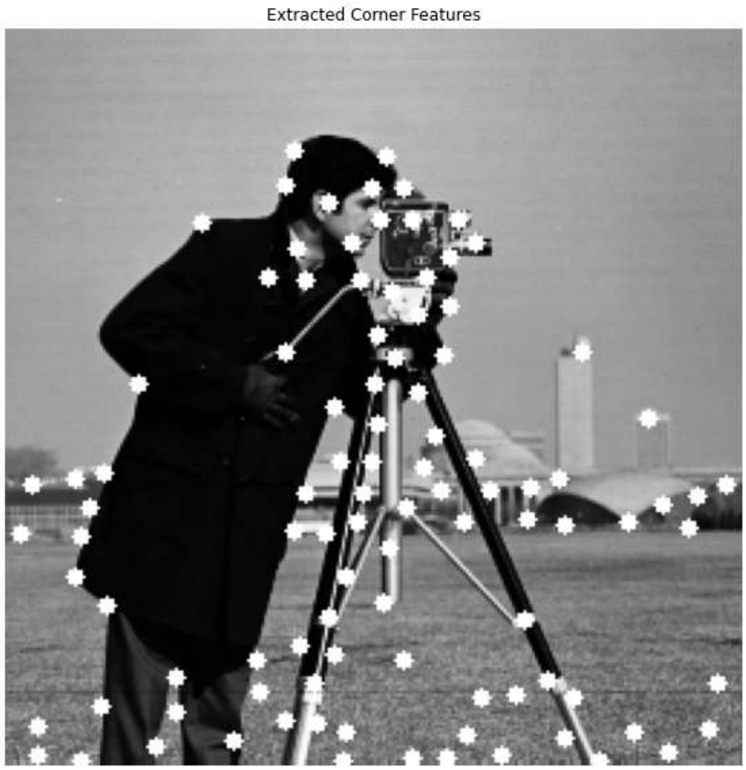
Point 9: (166.33738708496094, 76.77320098876953)

Point 10: (130.62796020507812, 113.03823852539062)

5 last SIFT keypoints:

Harris Corners:

Extracted Corner Features:



Detected Keypoints by implementing SIFT manually, instead of python built in SIFT through CV:

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')
```

```python
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()
```