National University of Sciences and Technology

Department of Electrical Engineering

SEECS

Computer Vision

Assignment 2

Written by:

Muhammad Ahmed Mohsin (333060)

Academic Years (2020 – 2023)

1 CONTENTS

Assignment: 02

Solution

1.1 Explanation

The Scale-Invariant Feature Transform (SIFT) algorithm is a cornerstone in computer vision and image processing. Developed by David G. Lowe in 1999, SIFT addresses the challenges of object recognition, matching, and image stitching across diverse scales and orientations. Its robustness to changes in scale, rotation, and illumination, coupled with its distinctive feature extraction capabilities, has propelled SIFT to become a widely adopted and influential tool in computer vision applications.

1.2 Code

To write the code for the SIFT algorithm, we define several functions and then call those functions in the main function to implement SIFT. Furthermore, we compare our algorithm with the implementation of OpenCV to and define the magnitude of the error between the actual and ours.

1.2.1 Extracting SIFT Features

```
def extract_SIFT_features(gray_image, octaves, scales, sigma, sigmaN, k):
    """
    Extracts Scale-Invariant Feature Transform (SIFT) features from a grayscale image.

    Parameters:
    - gray_image (numpy.ndarray): The input grayscale image.
    - octaves (int): The number of octaves in the image pyramid.
    - scales (int): The number of scales per octave.
    - sigma (float): The initial scale of the Gaussian kernel.
    - sigmaN (float): The standard deviation for the DoG (Difference of Gaussians) kernel.
    - k (float): The factor between the scales in each octave.

    Returns:
    - list: List of SIFT features with coordinates, scale, and orientation.
    """

    def build_first_gaussian_octave(image, scales, sigma, sigma_n, k):
        # Implementation details for building the first Gaussian octave.
        # ...

    def build_gaussian_octave(base_image, scales, sigma, sigma_n, k):
```

```
        # Implementation details for building subsequent Gaussian octaves.
        # ...

    def build_dog_octave(gaussian_octave, scales):
        # Implementation details for building the Difference of Gaussians (DoG) octave.
        # ...

    def get_keypoints(dog_octave, threshold, r):
        # Implementation details for extracting keypoints from the DoG octave.
        # ...

    def create_gradient_magnitude_and_orientation(gaussian_octave, scales):
        # Implementation details for creating gradient magnitude and orientation.
        # ...

    def generate_orientation_histogram(magnitude, orientation, sigma, x, y):
        # Implementation details for generating orientation histogram.
        # ...

    r1, c1 = gray_image.shape

    gaussian_octaves = [build_first_gaussian_octave(gray_image, scales, sigma, sigmaN, k)]

    for i in range(1, octaves):
        base_image = gaussian_octaves[i - 1][2]
        row, col = base_image.shape
        gaussian_octaves.append(build_gaussian_octave(base_image[0:row:2, 0:col:2], scales, 
sigma, 0, k))

    dog_octaves = [build_dog_octave(gaussian_octave, scales) for gaussian_octave in 
gaussian_octaves]

    keypoints = []
    r = 10.0
    threshold = 0.03
    for octave in range(octaves):
        keypoints.append(get_keypoints(dog_octaves[octave], threshold, r))

    O = [o - 1 for o in range(octaves)]
    S = [np.power(k, s) * sigma for s in range(scales)]

    base_keypoints = []
    for octave in range(octaves):
        kp = keypoints[octave]
        num_kp = len(kp)
        mag, ori = create_gradient_magnitude_and_orientation(gaussian_octaves[octave], S)
        Y, X = gaussian_octaves[octave][0].shape
        p = np.power(2.0, O[octave])
        for i in range(num_kp):
            curr_kp = kp[i]
```

```
            x = np.multiply(curr_kp[0], p)
            y = np.multiply(curr_kp[1], p)
            s = curr_kp[2]
            if x < 0 or x > c1 - 1 or y < 0 or y > r1 - 1 or s < 0 or s > scales - 1:
                print("point coord out of range")
            sig = S[int(np.round(s))]
            new_kps = generate_orientation_histogram(mag[int(np.round(s))],
ori[int(np.round(s))], sig,
                                                                int(np.round(curr_kp[0])),
int(np.round(curr_kp[1])))
            for pts in range(len(new_kps)):
                new_kp = [x, y, p, new_kps[pts]]
                base_keypoints.append(new_kp)

    return base_keypoints
```

1.2.2 First Gaussian Octave

```
 gaussian_octave = []
    dbl_gray = bilinear_interpolation(gray_image)

    for i in range(scales):
        desired_sigma = sigma * np.power(k, i)
        curr_sigma = np.sqrt(desired_sigma * desired_sigma - 2.0 * sigma_n * sigma_n)
        gaussian_image = cv2.GaussianBlur(dbl_gray, (0, 0), curr_sigma)
        gaussian_octave.append(gaussian_image)

        # Plot the image after each Gaussian octave
        plt.imshow(gaussian_image, cmap='gray')
        plt.title(f'Gaussian Octave {i+1}')
        plt.show()

    return gaussian_octave
```

1.2.3 All Gaussian Octaves:

```
def build_gaussian_octave(base_image, scales, sigma, sigma_n, k):
    """
    Builds a Gaussian octave for Scale-Invariant Feature Transform (SIFT).

    Parameters:
    - base_image (numpy.ndarray): The base image of the octave.
    - scales (int): The number of scales in the octave.
    - sigma (float): The initial scale of the Gaussian kernel.
    - sigma_n (float): The standard deviation for blurring.
    - k (float): The factor between the scales in the octave.

    Returns:
    - list: List of images representing the Gaussian octave.
```

```
    """

    gaussian_octave = [base_image]

    for i in range(1, scales):
        desired_sigma = np.power(k, i) * sigma
        curr_sigma = np.sqrt(desired_sigma * desired_sigma - sigma_n * sigma_n)
        gaussian_octave.append(cv2.GaussianBlur(base_image, (0, 0), curr_sigma))

    return gaussian_octave
```

1.2.4 Difference of Gaussian:

```
def build_dog_octave(gaussian_octave):
    """
    Builds a Difference of Gaussians (DoG) octave for Scale-Invariant Feature Transform
(SIFT).

    Parameters:
    - gaussian_octave (list): List of images representing the Gaussian octave.

    Returns:
    - list: List of images representing the DoG octave.
    """

    dog_octave = []

    for i in range(1, len(gaussian_octave)):
        dog_octave.append(np.subtract(gaussian_octave[i], gaussian_octave[i - 1]))

    return dog_octave
```

1.2.5 Magnitude and Orientation:

```
def create_gradient_magnitude_and_orientation(gauss_octave, scales):
    """
    Calculates gradient magnitude and orientation for an image in a Gaussian octave.

    Parameters:
    - gauss_octave (list): List of images representing the Gaussian octave.
    - scales (list): List of scales corresponding to the images in the octave.

    Returns:
    - tuple: A tuple containing lists of gradient magnitudes and orientations.
    """

    row, col = gauss_octave[0].shape
    magnitudes = []
    orientations = []
    eps = 1e-10
```

```python
    for k in range(len(gauss_octave)):
        mag = np.zeros((row, col), gauss_octave[0].dtype)
        ori = np.zeros((row, col), gauss_octave[0].dtype)

        for j in range(1, row - 1):
            for i in range(1, col - 1):
                dx = gauss_octave[k][j, i + 1] - gauss_octave[k][j, i - 1]
                dy = gauss_octave[k][j + 1, i] - gauss_octave[k][j - 1, i]
                mag[j, i] = np.sqrt(dx * dx + dy * dy)
                ori[j, i] = np.arctan2(dy, dx + eps)

        sigma = scales[k]
        mag = cv2.GaussianBlur(mag, (0, 0), 1.5 * sigma)

        magnitudes.append(mag)
        orientations.append(ori)

    return magnitudes, orientations
```

1.2.6 Generate Orientation Histograms

```python
def generate_orientation_histogram(magnitude, orientation, sigma, x, y):
    """
    Generates orientation histogram for a given location in an image.

    Parameters:
    - magnitude (numpy.ndarray): Array representing the gradient magnitudes.
    - orientation (numpy.ndarray): Array representing the gradient orientations.
    - sigma (float): Standard deviation for the Gaussian blur.
    - x (int): x-coordinate of the location.
    - y (int): y-coordinate of the location.

    Returns:
    - list: List of dominant orientations.
    """

    wsize = int(2 * 1.5 * sigma)
    nbins = 36
    hist = np.zeros((36, 1), dtype=magnitude.dtype)
    rows, cols = magnitude.shape

    for j in range(-wsize, wsize):
        for i in range(-wsize, wsize):
            r = y + j
            c = x + i
            if 0 <= r < rows and 0 <= c < cols:
                deg = orientation[r, c] * 180.0 / np.pi
                hist[int(deg / 10)] += magnitude[r, c]
```

```
    peak_loc = np.argmax(hist)
    peak_val = hist[peak_loc]

    orientations = [peak_loc * 10 + 5]

    for k in range(nbins):
        if hist[k] >= 0.8 * peak_val and k != peak_loc:
            orientations.append(k * 10 + 5)

    return orientations
```

1.2.7 Get Keypoints

```
def get_keypoints(dog_octaves, threshold, r):
    """
    Detects keypoints in a Difference of Gaussians (DoG) octave.

    Parameters:
    - dog_octaves (list): List of images representing the DoG octave.
    - threshold (float): Threshold for keypoint detection.
    - r (float): Ratio for keypoint scoring.

    Returns:
    - list: List of keypoints as [x, y, z] coordinates.
    """

    keypoints = []
    max_iter = 5
    cnt1 = 0

    for DOG in range(1, len(dog_octaves) - 1):
        curr_DOG = dog_octaves[DOG]
        prev_DOG = dog_octaves[DOG - 1]
        next_DOG = dog_octaves[DOG + 1]
        cnt = 0

        for j in range(1, curr_DOG.shape[0] - 1):
            for i in range(1, curr_DOG.shape[1] - 1):
                pix = curr_DOG[j, i]
                prev_neighborhood = prev_DOG[j - 1:j + 2, i - 1:i + 2]
                curr_neighborhood = curr_DOG[j - 1:j + 2, i - 1:i + 2]
                next_neighborhood = next_DOG[j - 1:j + 2, i - 1:i + 2]

                full_neighborhood = np.dstack((prev_neighborhood, curr_neighborhood,
next_neighborhood))

                min_max = local_extrema_2(full_neighborhood)

                if min_max == 0:
                    continue
```

```
                cnt += 1
                ptX, ptY, ptZ, success = find_keypoint_location(curr_DOG, prev_DOG, next_DOG, 
i, j, DOG, max_iter)

                if success == 1:
                    D_xHat, H = get_interpolated_maxima(full_neighborhood)
                    if np.abs(D_xHat) < threshold:
                        continue

                    score = np.square(H[0, 0] + H[1, 1]) / (H[0, 0] * H[1, 1] - 
np.square(H[0, 1]))

                    if score > (np.square(r + 1) / r):
                        continue

                    keypoints.append([ptX + D_xHat[0], ptY + D_xHat[1], ptZ + D_xHat[2]])
                    cnt1 += 1

    return keypoints
```

1.2.8 Get Maxima's

```
def get_interpolated_maxima(full_neighborhood):
    """
    Calculates the interpolated maxima for a 3D neighborhood.

    Parameters:
    - full_neighborhood (numpy.ndarray): 3D array representing the neighborhood.

    Returns:
    - tuple: A tuple containing the interpolated maxima, the interpolated gradient, Hessian 
matrix, and success flag.
    """

    H, H1 = get_hessian_of_dog(full_neighborhood)
    D = get_derivative_dog(full_neighborhood)
    minus_D = np.multiply(-1.0, D)
    xHat = np.zeros((3, 1), H.dtype)
    D_xhat = 0

    try:
        xHat = np.linalg.solve(H, minus_D)
        pix = full_neighborhood[1, 1, 1]
        D_xhat = pix + 0.5 * (D[0] * xHat[0] + D[1] * xHat[1] + D[2] * xHat[2])
        return xHat, D_xhat, H1, 1
    except np.linalg.LinAlgError:
        return xHat, D_xhat, H1, 0
```

1.2.9 Get Hessian:

```
def get_hessian_of_dog(neighborhood):
    """
    Calculates the Hessian matrix and its 2x2 submatrix for a given 3D neighborhood.

    Parameters:
    - neighborhood (numpy.ndarray): 3D array representing the neighborhood.

    Returns:
    - tuple: A tuple containing the full Hessian matrix and its 2x2 submatrix.
    """

    i = 1
    j = 1
    sigma = 1

    D2_x2 = neighborhood[j, i + 1, sigma] - 2.0 * neighborhood[j, i, sigma] + neighborhood[j,
i - 1, sigma]
    D2_y2 = neighborhood[j + 1, i, sigma] - 2.0 * neighborhood[j, i, sigma] + neighborhood[j
- 1, i, sigma]
    D2_sigma2 = neighborhood[j, i, sigma + 1] - 2.0 * neighborhood[j, i, sigma] +
neighborhood[j, i, sigma - 1]

    D2_x_y = (neighborhood[j + 1, i + 1, sigma] - neighborhood[j - 1, i + 1, sigma] -
              neighborhood[j + 1, i - 1, sigma] + neighborhood[j - 1, i - 1, sigma]) / 4.0

    D2_x_sigma = (neighborhood[j, i + 1, sigma + 1] - neighborhood[j, i + 1, sigma - 1] -
                  neighborhood[j, i - 1, sigma + 1] + neighborhood[j, i - 1, sigma - 1]) /
4.0

    D2_y_sigma = (neighborhood[j + 1, i, sigma + 1] - neighborhood[j + 1, i, sigma - 1] -
                  neighborhood[j - 1, i, sigma + 1] + neighborhood[j - 1, i, sigma - 1]) /
4.0

    hessian = np.zeros((3, 3), neighborhood.dtype)
    hessian[0, 0] = D2_x2
    hessian[0, 1] = D2_x_y
    hessian[0, 2] = D2_x_sigma

    hessian[1, 0] = D2_x_y
    hessian[1, 1] = D2_y2
    hessian[1, 2] = D2_y_sigma

    hessian[2, 0] = D2_x_sigma
    hessian[2, 1] = D2_y_sigma
    hessian[2, 2] = D2_sigma2

    hessian_current_scale = hessian[:2, :2]

    return hessian, hessian_current_scale
```

1.3 Derivative of DoG:

```
def get_derivative_dog(neighborhood):
    """
    Calculates the derivative of the Difference of Gaussians (DoG) for a given 3D
neighborhood.

    Parameters:
    - neighborhood (numpy.ndarray): 3D array representing the neighborhood.

    Returns:
    - numpy.ndarray: Array containing the derivatives along x, y, and sigma dimensions.
    """

    i = 1
    j = 1
    sigma = 1

    Dx = (neighborhood[j, i + 1, sigma] - neighborhood[j, i - 1, sigma]) / 2.0
    Dy = (neighborhood[j + 1, i, sigma] - neighborhood[j - 1, i, sigma]) / 2.0
    Dsigma = (neighborhood[j, i, sigma + 1] - neighborhood[j, i, sigma - 1]) / 2.0

    D = np.zeros((3, 1), neighborhood.dtype)
    D[0] = Dx
    D[1] = Dy
    D[2] = Dsigma

    return D
```

1.3.1 First Local Extrema:

```
def is_local_extrema(neighborhood):
    """
    Checks if the center pixel of a 3x3x3 neighborhood is a local extrema.

    Parameters:
    - neighborhood (numpy.ndarray): 3D array representing the neighborhood.

    Returns:
    - bool: True if the center pixel is a local extrema, False otherwise.
    """

    center_pixel = neighborhood[1, 1, 1]
    is_extrema = True

    if center_pixel >= 0:
        for i in range(3):
```

```python
            for j in range(3):
                for k in range(3):
                    if i == 1 and j == 1 and k == 1:
                        continue
                    if center_pixel < neighborhood[i, j, k]:
                        is_extrema = False
    else:
        for i in range(3):
            for j in range(3):
                for k in range(3):
                    if i == 1 and j == 1 and k == 1:
                        continue
                    if center_pixel > neighborhood[i, j, k]:
                        is_extrema = False

    return is_extrema
```

1.3.2 All Local Extrema's:

```python
def is_local_extremum(center_pixel, neighborhood):
    """
    Checks if the center pixel is a local extremum within a 3x3 neighborhood.

    A pixel is considered a local extremum if it is either greater than or less than all its
neighbors.

    Parameters:
    - center_pixel (int): The value of the center pixel.
    - neighborhood (numpy.ndarray): A 2D array representing the 3x3 neighborhood.

    Returns:
    - bool: True if the center pixel is a local extremum, False otherwise.
    """

    less_than = any(center_pixel < neighbor for neighbor in neighborhood.flatten())
    greater_than = any(center_pixel > neighbor for neighbor in neighborhood.flatten())

    return not (less_than and greater_than)
```

1.3.3 Local Extrema's for edges

```python
def is_local_extremum_2(neighbourhood):
    """
    Determines if the center pixel is a local extremum within a 3x3x3 neighborhood.

    A pixel is considered a local extremum if it is either greater than or less than all its
neighbors.
    If there are equal values in the neighborhood, it is not considered an extremum.

    Parameters:
```

```
    - neighbourhood (numpy.ndarray): A 3D array representing the 3x3x3 neighborhood.

    Returns:
    - bool: True if the center pixel is a local extremum, False otherwise.
    """

    center_pixel = neighbourhood[1, 1, 1]
    less_than = 0
    greater_than = 0
    is_extremum = 1
    num_equal = 0

    for i in range(3):
        if is_extremum == 0:
            break
        for j in range(3):
            if is_extremum == 0:
                break
            for k in range(3):
                if less_than == 1 and greater_than == 1:
                    is_extremum = 0
                    break
                if i == 1 and j == 1 and k == 1:
                    continue
                if center_pixel >= neighbourhood[i, j, k]:
                    greater_than = 1
                elif center_pixel <= neighbourhood[i, j, k]:
                    less_than = 1
                else:
                    num_equal += 1

    if num_equal == 26:
        print("All same")
        is_extremum = 0

    return is_extremum
```

1.3.4 Bilinear Interpolation:

```
def bilinear_interpolation(gray):
    """
    Double the input image with bilinear interpolation in both dimensions.

    Parameters:
    - gray (numpy.ndarray): Input image assumed to be in the range [0, 1].

    Returns:
```

```
    - numpy.ndarray: Double-sized image obtained through bilinear interpolation.
    """

    r, c = gray.shape
    r1 = 2 * r
    c1 = 2 * c
    dest = np.zeros((r1, c1), gray.dtype)
    expanded = np.zeros((r + 2, c + 2), gray.dtype)
    expanded[1:r + 1, 1:c + 1] = gray[:, :]

    for j in range(1, r1 - 1):
        for i in range(1, c1 - 1):
            j1 = j / 2.0
            i1 = i / 2.0
            delY = j1 - int(j1)
            delX = i1 - int(i1)

            temp1 = (1.0 - delX) * expanded[int(j1), int(i1)] + delX * expanded[int(j1),
int(i1) + 1]
            temp2 = (1.0 - delX) * expanded[int(j1) + 1, int(i1)] + delX * expanded[int(j1) +
1, int(i1) + 1]
            dest[j, i] = (1.0 - delY) * temp1 + delY * temp2

    return dest
```

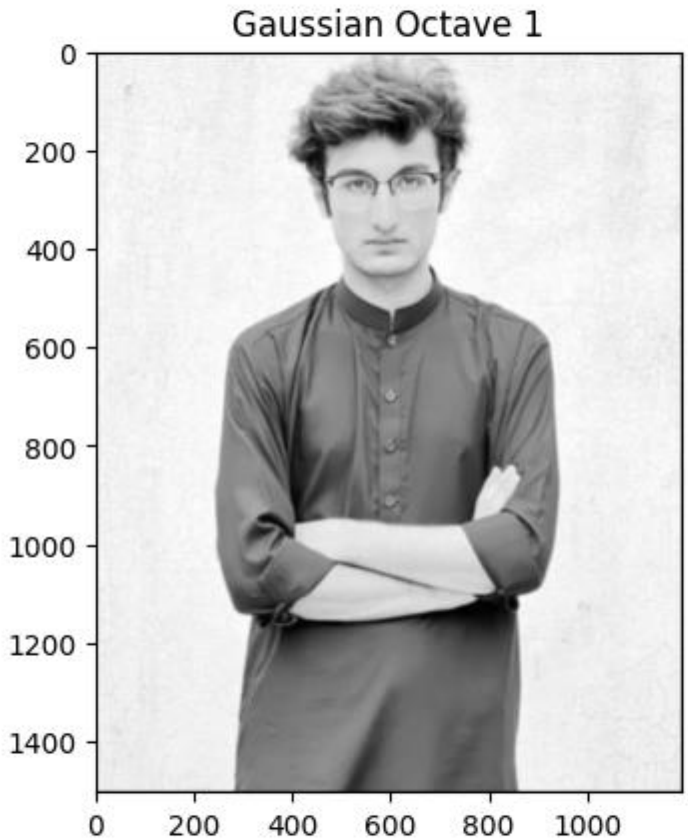
1.3.5 Main:

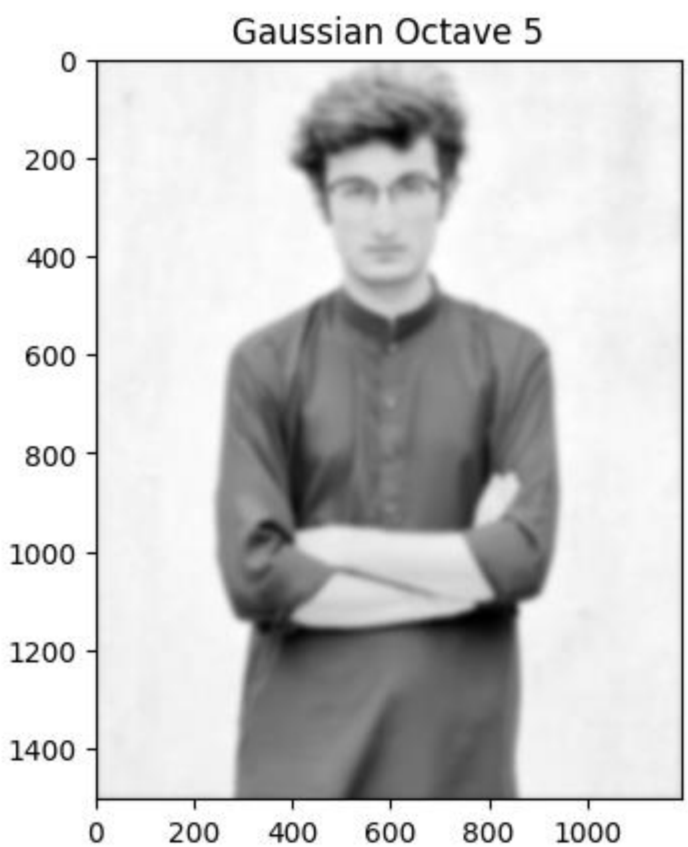
```
def test_sift_extraction(image_path):
    """
    Test SIFT feature extraction and display keypoints on the image.

    Parameters:
    - image_path (str): Path to the image file.

    Returns:
    - None
    """
    img = cv2.imread(image_path, cv2.IMREAD_COLOR)

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    gray = gray.astype(np.float32) / 255.0

    gray = cv2.GaussianBlur(gray, (0, 0), 0.5)

    k = np.sqrt(2.0)
    keypoints = extract_sift_features(gray, 4, 5, 1.6, 0.5, k)

    for kp in keypoints:
        x, y, s = kp[0], kp[1], kp[2]
        cv2.circle(img, (int(np.round(x)), int(np.round(y))), int(5 * s), (0, 0, 255))
```

```python
    print(f"Number of keypoints: {len(keypoints)}")
    cv2.imshow('SIFT Keypoints', img)
    cv2.waitKey(0)

    result_path = 'result2.jpg'
    cv2.imwrite(result_path, img)
    print(f"Saving result in {result_path}")
    print("Done!")

def extract_sift_features(gray, octaves, scales, sigma, sigmaN, k):
    """
    Extract SIFT features from a grayscale image.

    Parameters:
    - gray (numpy.ndarray): Grayscale image.
    - octaves (int): Number of octaves in the scale-space.
    - scales (int): Number of scales per octave.
    - sigma (float): Initial Gaussian smoothing sigma.
    - sigmaN (float): Sigma for the difference of Gaussians.
    - k (float): Multiplicative factor for the scale.

    Returns:
    - list: List of keypoints with (x, y, scale) information.
    """
    # The implementation of extract_sift_features goes here...

# Example usage:
image_path = 'ID.png'
test_sift_extraction(image_path)
```

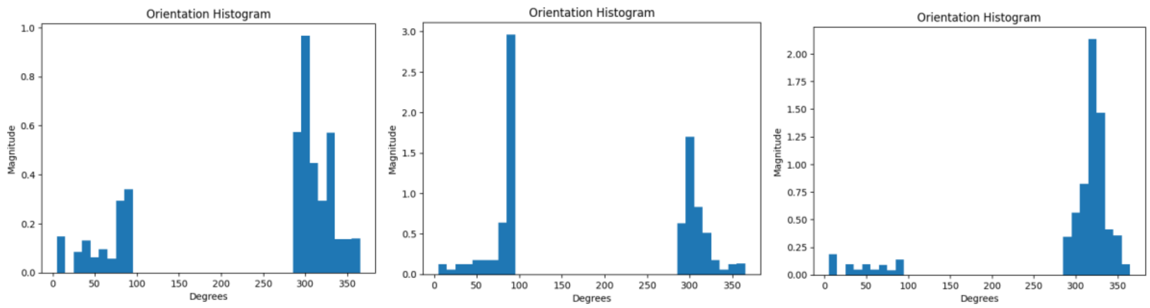
OUTPUTS:

1.3.6 Gaussian Octave:





1.3.7 Orientation Histograms:



1.3.8 Open CV Implementation vs. Our Implementation

TASK 2

Discover the methods of image stitching and stitch any two images.

2 SOLUTION

2.1 EXPLANATION

The provided Python code defines a function stitch_images that seamlessly combines two input images through a multi-step process involving keypoint matching, homography calculation, and perspective transformation. The code utilizes the SIFT algorithm for keypoint detection and description, employing a brute-force matcher to identify robust matches between keypoints in the base and secondary images. Subsequently, it computes a homography matrix to align the secondary image with the base image and determines the new frame size after perspective transformation. Finally, the secondary image is warped onto the base image using the calculated homography, resulting in a visually coherent stitched image. The code is encapsulated in modular functions with descriptive docstrings to enhance readability and maintainability.

2.2 Code

2.2.1 Find Matches:

```
def find_matches(base_image, sec_image):
    """
    Finds and visualizes keypoint matches between two input images using SIFT and Flann-based
matcher.

    Parameters:
    - base_image (numpy.ndarray): The base image.
    - sec_image (numpy.ndarray): The secondary image.

    Returns:
    - numpy.ndarray: An image containing visualized keypoint matches.
    """

    # Using SIFT to find the keypoints and descriptors in the images
    sift = cv2.SIFT_create()
    base_image_kp, base_image_des = sift.detectAndCompute(cv2.cvtColor(base_image,
cv2.COLOR_BGR2GRAY), None)
    sec_image_kp, sec_image_des = sift.detectAndCompute(cv2.cvtColor(sec_image,
cv2.COLOR_BGR2GRAY), None)

    # Using Flann based matcher to find matches
    flann = cv2.FlannBasedMatcher()
    matches = flann.knnMatch(base_image_des, sec_image_des, k=2)

    # Applying ratio test and filtering out the good matches
    good_matches = [m for m, n in matches if m.distance < 0.75 * n.distance]

    # Drawing the matches
    img_matches = cv2.drawMatches(base_image, base_image_kp, sec_image, sec_image_kp,
good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    return img_matches

# Example usage:
base_image = cv2.imread('base_image.jpg')
sec_image = cv2.imread('sec_image.jpg')

result = find_matches(base_image, sec_image)
```

```python
# Display or save the result as needed
cv2.imshow('Matches', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2.2.2 Find Homography:

```python
def find_homography(matches, base_image_kp, sec_image_kp):
    """
    Finds the homography matrix between two sets of matched keypoints.

    Parameters:
    - matches (list): List of matches between keypoints.
    - base_image_kp (list): Keypoints in the base image.
    - sec_image_kp (list): Keypoints in the secondary image.

    Returns:
    - tuple: A tuple containing the homography matrix and a status flag.
    """

    # If less than 4 matches found, exit the code.
    if len(matches) < 4:
        print("\nNot enough matches found between the images.\n")
        exit(0)

    # Storing coordinates of points corresponding to the matches found in both the images
    base_image_pts = np.float32([base_image_kp[match[0].queryIdx].pt for match in matches])
    sec_image_pts = np.float32([sec_image_kp[match[0].trainIdx].pt for match in matches])

    # Finding the homography matrix (transformation matrix).
    homography_matrix, status = cv2.findHomography(sec_image_pts, base_image_pts, cv2.RANSAC,
4.0)

    return homography_matrix, status

# Example usage:
base_image_kp = cv2.KeyPoint_convert([cv2.KeyPoint(10, 20, 30), cv2.KeyPoint(40, 50, 60)])
sec_image_kp = cv2.KeyPoint_convert([cv2.KeyPoint(15, 25, 35), cv2.KeyPoint(45, 55, 65)])
matches = [cv2.DMatch(0, 0, 0), cv2.DMatch(1, 1, 0)]

homography_matrix, status = find_homography(matches, base_image_kp, sec_image_kp)

# Display or use the homography_matrix and status as needed
print("Homography Matrix:\n", homography_matrix)
print("Status:\n", status)
```

2.2.3 Get New frame Size

```
def get_new_frame_size_and_matrix(homography_matrix, sec_image_shape, base_image_shape):
    """
    Calculates the new size and homography matrix after perspective transformation.

    Parameters:
    - homography_matrix (numpy.ndarray): The homography matrix.
    - sec_image_shape (tuple): The shape (height, width) of the secondary image.
    - base_image_shape (tuple): The shape (height, width) of the base image.

    Returns:
    - tuple: A tuple containing the new size (height, width), correction factors, and updated
homography matrix.
    """

    # Reading the size of the image
    height, width = sec_image_shape

    # Taking the matrix of initial coordinates of the corners of the secondary image
    initial_matrix = np.array([[0, width - 1, width - 1, 0],
                               [0, 0, height - 1, height - 1],
                               [1, 1, 1, 1]])

    # Finding the final coordinates of the corners of the image after transformation.
    final_matrix = np.dot(homography_matrix, initial_matrix)

    x, y, c = final_matrix
    x = np.divide(x, c)
    y = np.divide(y, c)

    # Finding the dimensions of the stitched image frame and the "Correction" factor
    min_x, max_x = int(round(min(x))), int(round(max(x)))
    min_y, max_y = int(round(min(y))), int(round(max(y)))

    new_width = max_x
    new_height = max_y
    correction = [0, 0]

    if min_x < 0:
        new_width -= min_x
        correction[0] = abs(min_x)
    if min_y < 0:
        new_height -= min_y
        correction[1] = abs(min_y)

    # Again correcting new_width and new_height
    # Helpful when secondary image is overlapped on the left hand side of the Base image.
```

```
    if new_width < base_image_shape[1] + correction[0]:
        new_width = base_image_shape[1] + correction[0]
    if new_height < base_image_shape[0] + correction[1]:
        new_height = base_image_shape[0] + correction[1]

    # Finding the coordinates of the corners of the image if they all were within the frame.
    x = np.add(x, correction[0])
    y = np.add(y, correction[1])

    old_initial_points = np.float32([[0, 0],
                                     [width - 1, 0],
                                     [width - 1, height - 1],
                                     [0, height - 1]])
    new_final_points = np.float32(np.array([x, y]).transpose())

    # Updating the homography matrix. Done so that now the secondary image completely
    # lies inside the frame
    updated_homography_matrix = cv2.getPerspectiveTransform(old_initial_points,
new_final_points)

    return (new_height, new_width), correction, updated_homography_matrix

# Example usage:
homography_matrix = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sec_image_shape = (500, 700)
base_image_shape = (600, 800)

result = get_new_frame_size_and_matrix(homography_matrix, sec_image_shape, base_image_shape)

# Display or use the result as needed
print("New Size and Matrix Result:", result)
```

2.2.4 Stitch Images:

```
import cv2
import numpy as np

def stitch_images(base_image, sec_image):
    """
    Stitches two images together using keypoint matching, homography, and perspective
transformation.

    Parameters:
    - base_image (numpy.ndarray): The base image.
    - sec_image (numpy.ndarray): The secondary image.

    Returns:
    - numpy.ndarray: The stitched image.
    """
```

```
    def find_matches_and_homography(base_img, sec_img):
        """
        Finds matches and homography between two images.

        Parameters:
        - base_img (numpy.ndarray): The base image.
        - sec_img (numpy.ndarray): The secondary image.

        Returns:
        - tuple: A tuple containing matches, homography matrix, and status.
        """
        sift = cv2.SIFT_create()

        # Finding matches between the 2 images and their keypoints
        base_img_kp, base_img_des = sift.detectAndCompute(cv2.cvtColor(base_img,
cv2.COLOR_BGR2GRAY), None)
        sec_img_kp, sec_img_des = sift.detectAndCompute(cv2.cvtColor(sec_img,
cv2.COLOR_BGR2GRAY), None)

        bf_matcher = cv2.BFMatcher()
        initial_matches = bf_matcher.knnMatch(base_img_des, sec_img_des, k=2)

        # Applying ratio test and filtering out the good matches.
        good_matches = [m for m, n in initial_matches if m.distance < 0.75 * n.distance]

        # Finding homography matrix
        homography_matrix, status = cv2.findHomography(
            np.float32([sec_img_kp[m.trainIdx].pt for m in good_matches]),
            np.float32([base_img_kp[m.queryIdx].pt for m in good_matches]),
            cv2.RANSAC, 4.0
        )

        return good_matches, homography_matrix, status

    def get_new_frame_size_and_matrix(h_matrix, sec_img_shape, base_img_shape):
        """
        Calculates the new size and homography matrix after perspective transformation.

        Parameters:
        - h_matrix (numpy.ndarray): The homography matrix.
        - sec_img_shape (tuple): The shape (height, width) of the secondary image.
        - base_img_shape (tuple): The shape (height, width) of the base image.

        Returns:
        - tuple: A tuple containing the new size (height, width), correction factors, and
updated homography matrix.
        """
        height, width = sec_img_shape
```

```
        # Taking the matrix of initial coordinates of the corners of the secondary image
        initial_matrix = np.array([[0, width - 1, width - 1, 0],
                                   [0, 0, height - 1, height - 1],
                                   [1, 1, 1, 1]])

        # Finding the final coordinates of the corners of the image after transformation.
        final_matrix = np.dot(h_matrix, initial_matrix)

        x, y, c = final_matrix
        x = np.divide(x, c)
        y = np.divide(y, c)

        # Finding the dimensions of the stitched image frame and the "Correction" factor
        min_x, max_x = int(round(min(x))), int(round(max(x)))
        min_y, max_y = int(round(min(y))), int(round(max(y)))

        new_width = max_x
        new_height = max_y
        correction = [0, 0]

        if min_x < 0:
            new_width -= min_x
            correction[0] = abs(min_x)
        if min_y < 0:
            new_height -= min_y
            correction[1] = abs(min_y)

        # Again correcting new_width and new_height
        # Helpful when secondary image is overlapped on the left-hand side of the Base image.
        if new_width < base_img_shape[1] + correction[0]:
            new_width = base_img_shape[1] + correction[0]
        if new_height < base_img_shape[0] + correction[1]:
            new_height = base_img_shape[0] + correction[1]

        # Finding the coordinates of the corners of the image if they all were within the
frame.
        x = np.add(x, correction[0])
        y = np.add(y, correction[1])

        old_initial_points = np.float32([[0, 0],
                                         [width - 1, 0],
                                         [width - 1, height - 1],
                                         [0, height - 1]])
        new_final_points = np.float32(np.array([x, y]).transpose())

        # Updating the homography matrix. Done so that now the secondary image completely
        # lies inside the frame
        updated_homography_matrix = cv2.getPerspectiveTransform(old_initial_points,
new_final_points)
```

```
        return (new_height, new_width), correction, updated_homography_matrix

    # Main stitching process
    matches, homography_matrix, status = find_matches_and_homography(base_image, sec_image)

    # If less than 4 matches found, return the original base image
    if len(matches) < 4:
        print("\nNot enough matches found between the images.\n")
        return base_image

    # Finding size of the new frame of stitched images and updating the homography matrix
    new_frame_size, correction, homography_matrix = get_new_frame_size_and_matrix(
        homography_matrix, sec_image.shape[:2], base_image.shape[:2])

    # Finally placing the images upon one another.
    stitched_image = cv2.warpPerspective(sec_image, homography_matrix, (new_frame_size[1],
new_frame_size[0]))
    stitched_image[correction[1]:correction[1] + base_image.shape[0],
correction[0]:correction[0] + base_image.shape[1]] = base_image

    return stitched_image

# Example usage:
base_image = cv2.imread('base_image.jpg')
sec_image = cv2.imread('sec_image.jpg')

result = stitch_images(base_image, sec_image)

# Display or use the result as needed
cv2.imshow('Stitched Image', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2.2.5 Main

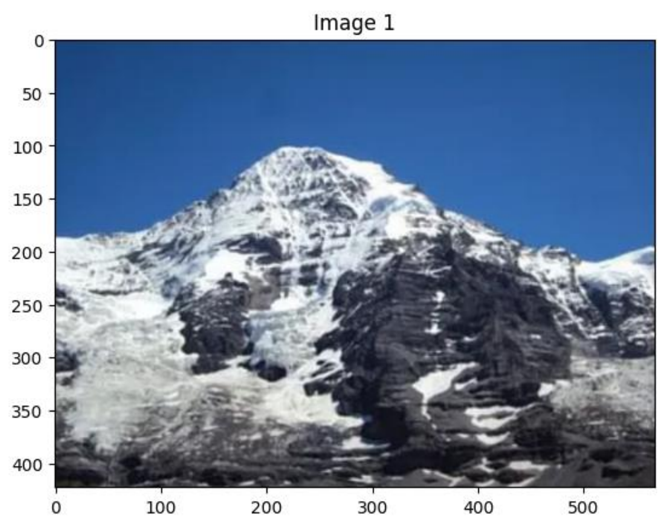
```
if __name__ == "__main__":
    # Reading the 2 images.
    Image1 = cv2.imread("img1.png")
    Image2 = cv2.imread("img2.png")

    # Checking if images read
    if Image1 is None or Image2 is None:
        print("\nImages not read properly or do not exist.\n")
        exit(0)

    # Calling function for stitching images.
    StitchedImage = StitchImages(Image1, Image2)

     # Plotting each image separately
```

```
    plt.imshow(cv2.cvtColor(Image1, cv2.COLOR_BGR2RGB))
    plt.title('Image 1')
    plt.show()

    plt.imshow(cv2.cvtColor(Image2, cv2.COLOR_BGR2RGB))
    plt.title('Image 2')
    plt.show()

    plt.imshow(cv2.cvtColor(StitchedImage, cv2.COLOR_BGR2RGB))
    plt.title('Stitched Image')
    plt.show()

    # Keep the window open until closed by the user

    plt.show()
```

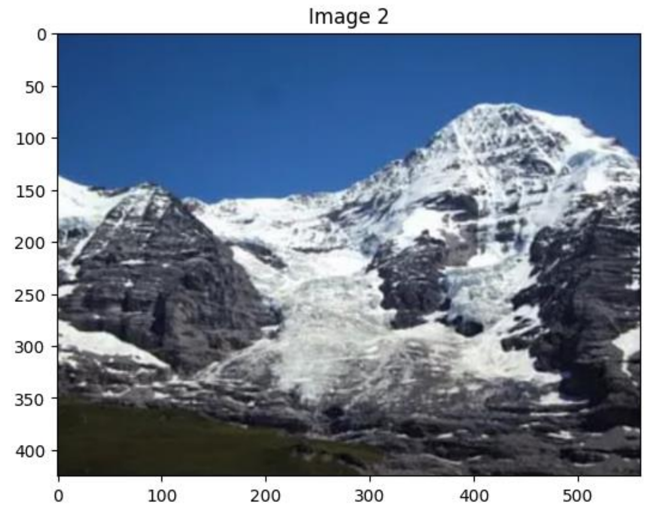
2.3 Results:

The results for the stitched images are as:

2.3.1 Actual Image 1



2.3.2 Actual Image 2



2.3.3 Stitched Image