

Team 10
Computer Vision Task 3
Report

Name	Sec	BN
Abram Gad	1	1
Mahmoud Hamdy Mahmoud	2	24
Mariam Hossam	2	31
Mariam Mohamed	2	34
Mena Safwat	2	

I. The Harris corner detection algorithm:

It is a popular method for detecting corners or interest points in images. The algorithm is based on the observation that corners can be identified as points in the image where the intensity changes significantly in different directions. The Harris algorithm computes a corner response function, which measures the likelihood of a pixel being a corner based on the local image structure.

The Harris corner response function is computed using the eigenvalues of the matrix M , which is based on the derivatives of the image intensity with respect to x and y . The matrix M is defined as:

$$M = \begin{bmatrix} S_x^2 & S_{xy} \\ S_{xy} & S_y^2 \end{bmatrix}$$

where S_x and S_y are the derivatives of the image intensity with respect to x and y , respectively, and S_{xy} is their cross-product. The elements of M are then smoothed using a Gaussian filter to reduce noise.

The eigenvalues of M are computed for each pixel, and the corner response function R is defined as the smaller eigenvalue λ_1 minus a constant k times the larger eigenvalue λ_2 :

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Pixels with high corner response values are likely to be corners. The constant k is a parameter that controls the trade-off between corner detection sensitivity and selectivity.

The Harris corner detection algorithm is widely used in computer vision applications such as object detection, tracking, and image stitching. It is robust to changes in lighting conditions and is relatively insensitive to noise. However, it is sensitive to changes in scale and orientation, and it may produce multiple responses for a single corner, leading to a high number of false positives.

In summary, the Harris corner detection algorithm is a popular and effective method for detecting corners in images. It is based on the observation that corners can be identified by significant changes in intensity in different directions. The algorithm computes a corner response function based on the eigenvalues of a matrix computed from image derivatives, and pixels with high response values are considered to be corners. While it has some limitations, the Harris algorithm remains a widely used and valuable tool in computer vision.

```

def harris_corner_detection(image, block_size=2, ksize=3, k=0.04, threshold=0.2):
    # Convert image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Compute the derivative of the image in x and y direction
    dx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=ksize)
    dy = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=ksize)

    # Compute the elements of the Harris matrix
    Ixx = cv2.pow(dx, 2)
    Ixy = cv2.multiply(dx, dy)
    Iyy = cv2.pow(dy, 2)

    # Compute the sum of the elements in the local neighborhood of each pixel
    Sxx = cv2.boxFilter(Ixx, -1, (block_size, block_size))
    Sxy = cv2.boxFilter(Ixy, -1, (block_size, block_size))
    Syy = cv2.boxFilter(Iyy, -1, (block_size, block_size))

    # Compute the determinant and trace of the Harris matrix
    det = cv2.subtract(cv2.multiply(Sxx, Syy), cv2.pow(Sxy, 2))
    trace = cv2.add(Sxx, Syy)

    # Compute the Harris response for each pixel
    harris = cv2.divide(det, trace + k)

    # Threshold the Harris response
    harris = cv2.threshold(harris, threshold *
        | | | | | harris.max(), 255, cv2.THRESH_BINARY)[1]

    # Convert the Harris response to a uint8 image
    harris = np.uint8(harris)

    # Find the coordinates of the corners
    coords = np.column_stack(np.where(harris > 0))

    return coords

```

The given code is a Python implementation of the Harris corner detection algorithm, which is used to detect corners or interest points in an image. The algorithm is based on the idea that a corner can be identified by looking for a significant change in intensity in all directions. The Harris corner detection algorithm is widely used in computer vision applications such as object detection, tracking, and image stitching.

The function `harris_corner_detection` takes an input image and several optional parameters as input and returns the coordinates of the detected corners. The input image is first converted to grayscale, and the derivative of the image in the x and y direction is computed using the Sobel operator. Then, the elements of the Harris matrix are computed and the sum of the elements in the local neighborhood of each pixel is computed using a

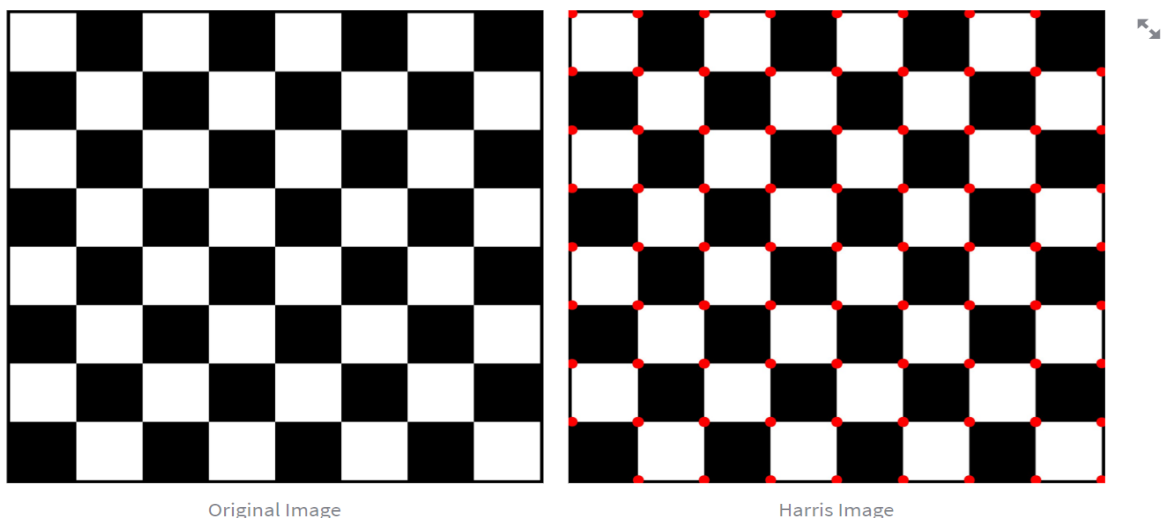
box filter. The determinant and trace of the Harris matrix are then computed, and the Harris response for each pixel is computed using the formula $\det / (\text{trace} + k)$, where k is a constant parameter.

The Harris response is then thresholded to obtain a binary image, where pixel values above a certain threshold are considered to be corners. The threshold is set based on the maximum value of the Harris response multiplied by a user-defined threshold parameter. Finally, the coordinates of the corners are extracted from the binary image.

The optional parameters of the `harris_corner_detection` function allow the user to customize the size of the local neighborhood used to compute the sum of the elements of the Harris matrix (`block_size`), the size of the Sobel kernel used to compute the derivative of the image (`ksize`), the constant parameter used in the Harris response formula (k), and the threshold used to determine which pixels are corners (`threshold`).

To use this code to extract unique features in all images, one would need to apply the `harris_corner_detection` function to each image and store the resulting corner coordinates. These corner coordinates can then be used as unique features to compare and match images.

As for the computation times, it would depend on the size of the input image and the chosen values of the optional parameters. The Sobel operator and box filter operations are computationally intensive, especially for large images and large neighborhood sizes. Therefore, the computation time may increase significantly for larger images and larger neighborhood sizes. The user can measure the computation time using Python's time module or other available profiling tools.



II. SIFT:

What is SIFT (Scale-Invariant Feature Transform): It is a computer vision algorithm that detects, describes, and matches local features in images. It identifies key-points that are distinctive across an image's width, height, and scale. When considering scale, SIFT identifies key-points that remain stable even when the template changes size, when the image quality becomes better or worse, or when the template undergoes changes in viewpoint or aspect ratio. Each key-point has an associated orientation that makes SIFT features invariant to template rotations. Finally, SIFT generates a descriptor for each key-point, which is a 128-length vector that allows key-points to be compared.

How It works:

We 1st call `computeKeypointsAndDescriptors()` function which returns the key-points then we call `generateBaseImage()` to get the base image which is the image with different scales to form our scale space. Then we get the number of octaves by calling `computeNumberOfOctaves()`.

Then The image pyramid can be built by using the `generateGaussianKernels()` function to create a list of scales (gaussian kernel sizes) that is passed to `generateGaussianImages()`. This function repeatedly blurs and downsamples the base image. The adjacent pairs of gaussian images are then subtracted to form a pyramid of difference-of-Gaussian "DOG" images. The final DOG image pyramid is used to identify keypoints using `findScaleSpaceExtrema()`. These keypoints are cleaned up by removing duplicates and converting them to the input image size. Finally, descriptors are generated for each keypoint via `generateDescriptors()`.

The Code:

So, in this code we double the input image in size and apply Gaussian blur on it aka the base image

```
def generateBaseImage(image, sigma, assumed_blur):  
    """  
    Generate base image from input image by upsampling by 2 in both directions and blurring  
    """  
    #Here we are doubling the input image  
    image = resize(image, (0, 0), fx=2, fy=2, interpolation=INTER_LINEAR)  
    sigma_diff = sqrt(max((sigma ** 2) - ((2 * assumed_blur) ** 2), 0.01))  
    #Applying Gaussian blur by sigma difference which is (sigma**2 - assumed_sigma**2)**0.5  
    #So the image blur is now sigma instead of assumed_blur  
    return GaussianBlur(image, (0, 0), sigmaX=sigma_diff, sigmaY=sigma_diff)
```

Then we have the `computeNumberOfOctaves()` where we calculate the number of times where we can halve the image until it becomes indivisible then we round the image to the nearest integer to have an integer number of layers in our image.

Then use `generateGaussianKernels()` to create a list of the amount of blur for each image in a particular layer.

We can observe that it has `numOctaves` layers and each layer consists of `numIntervals + 3` images. Which is required to cover one blur value to twice that value. We have another `2` for one blur step before the first image in the layer and another blur step after the last image in the layer.

```
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.  
    """  
    """Why do we add 3 to the calculated num_intervals?  
    ->To cover one blur value to twice that value. We have another 2 for  
    one blur step before the first image in the  
    layer and another blur step after the last image in the layer.  
    """  
    num_images_per_octave = num_intervals + 3  
    k = 2 ** (1. / num_intervals)  
    # scale of gaussian blur necessary to go from one blur scale to the next within an octave  
    gaussian_kernels = 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] = sqrt(sigma_total ** 2 - sigma_previous ** 2)  
    return gaussian_kernels
```

Now we will proceed in generating the image pyramid so we start with the base image and then we continue blurring according to the gaussian kernel

By using `generateGaussianImages()` function. Then we use the resulted images to get the DOG images aka apply edge detection using `generateDoGImages()` function.

```
def generateGaussianImages(image, num_octaves, gaussian_kernels):
    """Generate scale-space pyramid of Gaussian images
    We get the blurred images according to the gaussian kernel we calculated
    """
    gaussian_images = []

    for octave_index in range(num_octaves):
        gaussian_images_in_octave = []
        gaussian_images_in_octave.append(image) # first image in octave already has the correct blur
        for gaussian_kernel in gaussian_kernels[1:]:
            image = 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 = resize(octave_base, (int(octave_base.shape[1] / 2), int(octave_base.shape[0] / 2)), interpolation=INTER_NEAREST)
    return array(gaussian_images, dtype=object)

def generateDoGImages(gaussian_images):
    """Generate Difference-of-Gaussians image pyramid
    Get the DOG images aka edge detection
    """
    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:]):
            # ordinary subtraction will not work because the images are unsigned integers
            dog_images_in_octave.append(subtract(second_image, first_image))
        dog_images.append(dog_images_in_octave)
    return array(dog_images, dtype=object)
```

Then we Get the features of the objects in the image by using Scale space extrema detection step.

Which is simply we iterate over three successive images at a time we then look at the middle pixels and compare it with the 8 pixels around it and with the 9 pixels before it and the 9 pixels after it then we check whether this pixel extremum or not using the function `isPixelAnExtremum()` which detects whether the pixel is maximum or minimum we then use

`localizeExtremumViaQuadraticFit()` to localize the extremum position's

The procedure involves fitting a quadratic model to the input keypoint pixel and all 26 of its neighboring pixels (called a pixel_cube). The keypoint's position is then updated with the subpixel-accurate extremum estimated from this model. The process is iterated at most 5 times until the next update moves the keypoint less than 0.5 in any of the three directions. This means that the quadratic model has converged to one pixel location.

The two helper functions `computeGradientAtCenterPixel()` and `computeHessianAtCenterPixel()` implement second-order central finite difference approximations of the gradients and Hessians in all three dimensions. Ordinary quadratic interpolation that you may have done in calculus class won't

work well here because we're using a uniform mesh (a grid of evenly spaced pixels). Finite difference approximations consider this discretization to produce more accurate extrema estimates.

After this method we proceed to **Orientation assignment** to compute key points orientations by drawing their histogram and then removing the duplicate key points and compute the descriptor which have then information about the key point's neighborhood to allow comparison between the key points

This happens by using `computeKeypointsWithOrientations()` to compute a histogram of gradients for pixels around the key point's neighborhood. Which happens by computing the magnitude and orientation of the 2D gradient at each pixel in this neighborhood. We create a 36-bin histogram for the orientations '10 degrees per bin.' The orientation of a particular pixel **tells us which histogram bin to choose** 'Usually the max value in the histogram', but the actual value we place in that bin is that pixel's gradient magnitude with a Gaussian weighting. This makes pixels farther from the key point have less of an influence on the histogram. We repeat this procedure for all pixels in the neighborhood, accumulating our results into the same 36-bin histogram.

Then we sort and remove the duplicates by using `removeDuplicateKeypoints()`



Figure 1 Input Image to SIFT

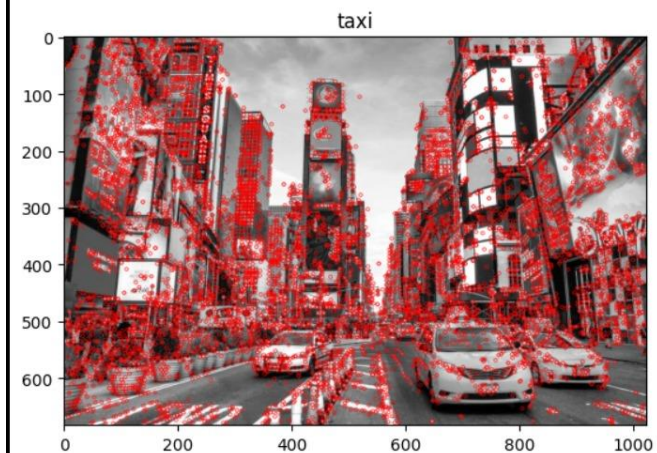


Figure 2 Output Image from SIFT

III. Feature Detection:

Sum of Squared Differences (SSD) and Normalized Cross Correlation (NCC) are two popular techniques used for measuring the similarity between two images or image patches. SSD is a simple and fast technique that calculates the sum of the squared differences between the corresponding pixel values of the two images or image patches. The smaller the SSD value, the more similar the images or patches are. However, SSD is sensitive to changes in lighting and contrast, which can affect the overall intensity of the images or patches.

On the other hand, NCC is a more robust technique that calculates the correlation between the pixel values of the two images or image patches, while also taking into account their mean and standard deviation. The resulting value ranges between -1 and 1, where a value of 1 indicates a perfect match, 0 indicates no correlation, and -1 indicates a perfect anti-correlation. NCC is less affected by changes in lighting and contrast, as it normalizes the pixel values by their means and standard deviations. However, NCC is more computationally expensive than SSD, as it involves calculating the means and standard deviations of the pixel values for each image or patch.

In the algorithm, we start by calling the driver function `call_matching()` which divides the function depending on the chosen method, before drawing the matches of each algorithm using `cv2.drawMatches()` and calculating the time.

```
def call_matching (original_img ,original_template,method,threshold,resize,Sift_builtIn):
    t0 = time.time()
    if resize:
        original_img = cv2.resize(original_img,(150,150))
        original_template = cv2.resize(original_template,(150,150))
    img = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
    template = cv2.cvtColor(original_template, cv2.COLOR_BGR2GRAY)

    if Sift_builtIn:
        sift = cv2.SIFT_create()

        key_pts1, descriptor1 = sift.detectAndCompute(img, None)
        key_pts2, descriptor2 = sift.detectAndCompute(template, None)
    else:
        key_pts1, descriptor1 = computeKeypointsAndDescriptors(img)
        key_pts2, descriptor2 = computeKeypointsAndDescriptors(template)

    if method=="Normalized cross correlation":
        matches = normalized_cross_correlation(key_pts1,key_pts2,descriptor1, descriptor2, threshold/100)
    elif method=="SSD":
        matches = SSD(descriptor1, descriptor2, threshold)
    print(matches)
    img_matches = cv2.drawMatches(original_img, key_pts1, original_template, key_pts2, matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    img_matches = cv2.resize(img_matches,(500,300))

    cv2.imwrite('template/result.png', img_matches)

    t1 = time.time()
    return t1-t0
```

Sum of Squared Differences:

First, the images are turned into grayscale before applying the SIFT algorithm to both images before calling `SSD()`. The function is given the descriptor of both of the images alongside the threshold. The function loops the values of both descriptors while calling `calculateSSD()` which returns the sum of squared differences. Then if the returned value is less than the threshold it is added to the matches which are then returned to be drawn.

```
def calculateSSD(desc_image1, desc_image2):
    ssd = 0.0
    if len(desc_image1) != len(desc_image2):
        # Return -1 to indicate error (features have different sizes)
        return -1
    for i in range(len(desc_image1)):
        ssd += (desc_image1[i] - desc_image2[i])**2
    final_ss = np.sqrt(ssd)
    return final_ss

def SSD(descriptor1, descriptor2, threshold):
    KeyPoints1 = descriptor1.shape[0]
    KeyPoints2 = descriptor2.shape[0]
    matches = []
    for kp1 in range(KeyPoints1):
        best_ss = float('inf')
        best_index = -1
        for Kp2 in range(KeyPoints2):
            ssd = calculateSSD(descriptor1[kp1], descriptor2[Kp2])
            if ssd < best_ss:
                best_ss = ssd
                best_index = Kp2
        if best_ss <= threshold:
            feature = cv2.DMatch()
            # The index of the feature in the first image
            feature.queryIdx = kp1
            # The index of the feature in the second image
            feature.trainIdx = best_index
            # The distance between the two features
            feature.distance = best_ss
            matches.append(feature)
    return matches
```

Normalized cross-correlation:

For the NCC algorithm, in addition to turning images to greyscale, we resize the images then following the same steps as the SSD, we calculate key points and descriptors using SIFT algorithm the calling `normalized_cross_corelation()`. In the function, it loops through the values of descriptors of both images while calling `calculate_NCC()` which returns the normalized cross-correlation value. Then if the returned value is less than the threshold it is added to the matches which are then returned to be drawn.

```

def calculate_NCC(desc_image1, desc_image2):
    difference1=(desc_image1 - np.mean(desc_image1))
    difference2=(desc_image2 - np.mean(desc_image2))
    correlation_vector = np.multiply(difference1, difference2)

    normalized_output = correlation_vector / np.sqrt(difference1**2,difference2**2)
    NCC = float(np.mean(normalized_output))

    return NCC

def normalized_cross_corelation(key_pts_1, key_pts_2, desc1, desc2, threshold):

    matches = []

    for index1 in range(len(desc1)):
        for index2 in range(len(desc2)):
            out1_norm = (desc1[index1] - np.mean(desc1[index1])) / (np.std(desc1[index1]))
            out2_norm = (desc2[index2] - np.mean(desc2[index2])) / (np.std(desc2[index2]))
            corr_vector = np.multiply(out1_norm, out2_norm)
            corr = float(np.mean(corr_vector))
            # only taking above threshold
            if corr > threshold:
                matches.append([index1, index2, corr])

    output = []
    for index in range(len(matches)):
        dis = np.linalg.norm(np.array(key_pts_1[matches[index][0]].pt) - np.array(key_pts_2[matches[index][1]].pt))
        output.append(cv2.DMatch(matches[index][0], matches[index][1], dis))
    return output


```


Outputs:

Feature Matching

Image


[Browse files](#)


 box_in_scene.png 119.6KB



Target

[Browse files](#)

 box.png 49.5KB




Select method

☒ Normalized cross corelation

☐ SSD

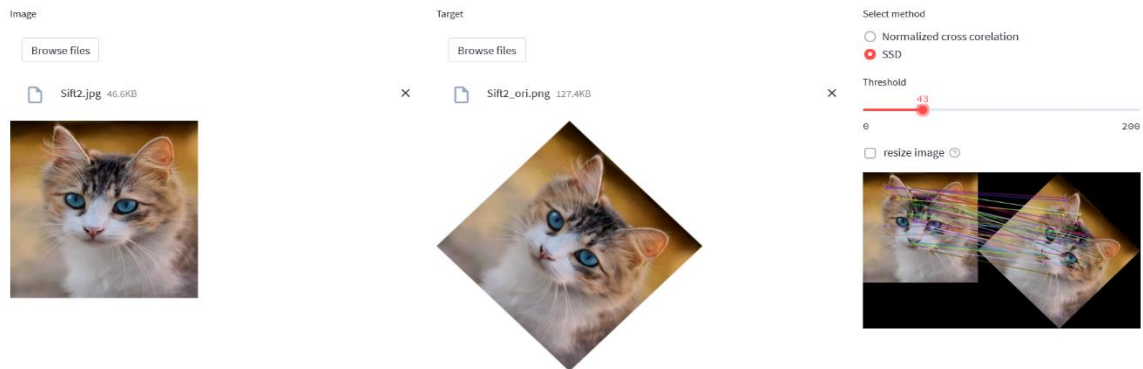
Threshold

☒ resize image



Execution Time = 5.072781085968018

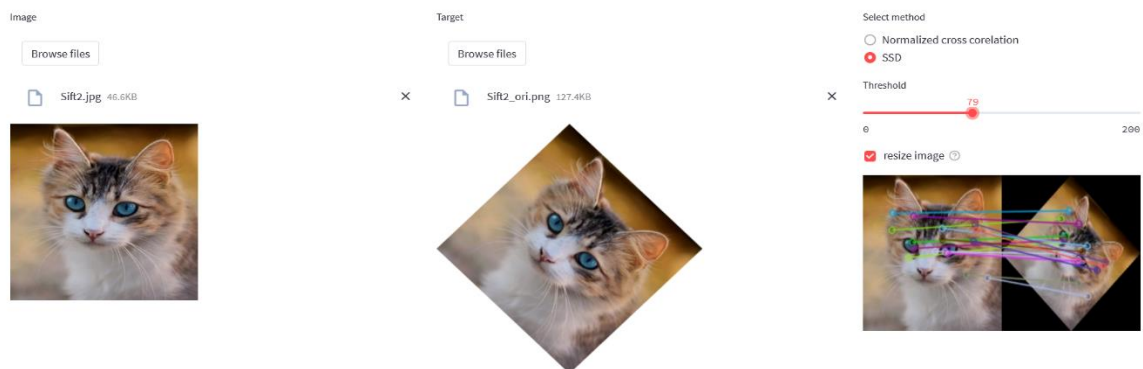
Feature Matching



Execution Time = 22.286759853363037

Made with Streamlit

Feature Matching



Execution Time = 5.747558832168579

Made with Streamlit

NOTES:

- Resize image: allows the program to resize images which significantly increases execution time but may cause inaccurate results

resize image is calculated faster but maybe inaccurate

☐ resize image ?

- Sift built-in: if checked the code uses the cv2 built-in function

☒ Built in is faster ?

☐ Sift Built In ?