Task_3

Team :19

name	sec	BN
Mariam mounier	**2**	**35**
Abdelrahman Sameh	**1**	**53**
Osamah Faisal	**1**	**11**
Sara Tarek	**1**	**40**
Beshara Safawt	**1**	**22**
Shuaib Abdulsalam	**1**	**48**

1.Harris function

The harris_corners function takes an input image as a Num.Py array and returns a list of corner coordinates. The k parameter controls the sensitivity of the algorithm, and the threshold parameter determines the minimum value of the Harris response function required to be considered a corner. The window_size parameter specifies the size of the window used to compute the second moment matrix elements.

First, we define a function called harris_corners that takes an input image as a NumPy array and several optional parameters. The first thing we do is check if the input image is in color format or grayscale format. If it's in color format, we convert it to grayscale using OpenCV's cvtColor function.

Next, we calculate the image gradients using the Sobel operator. We use the cv2.Sobel function from OpenCV, which takes the input image, the data type of the output image (in this case cv2.CV_64F for a 64-bit floating-point image), the order of the derivative in the x and y directions (1 and 0, respectively, for sobelx, and 0 and 1, respectively, for sobely), and the kernel size (ksize=3 for a 3x3 kernel).

We then compute the second moment matrix elements over a window using the box filter. The cv2.boxFilter function takes the input image, the output data type (-1 to use the same data type as the input image), and the kernel size ((window_size, window_size)).

Next, we compute the Harris response function using the second moment matrix elements. We first calculate the determinant and trace of the second moment matrix, and then compute the response using the formula R = det - k * trace^2.

Finally, we find the corners with high enough response by thresholding the response image and finding the nonzero pixels. We first scale the threshold by the maximum response value, and then set all response values below the threshold to 0. We then use the np.argwhere function to find the coordinates of the nonzero pixels, which correspond to the corner locations.

Code:

```
from flask import Flask, render_template, request
from flask_cors import CORS
import os
import base64  # convert from string to  bits
import json
import cv2
import numpy as np
import time
import calendar
import image as img1
import harrisoperator as Harris
import matplotlib.pyplot as plt
import json
import math
import cv2


app = Flask(__name__)
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
SECRET_KEY = os.urandom(32)
app.config['SECRET_KEY'] = SECRET_KEY

CORS(app)


@app.route("/", methods=["GET", "POST"])
def main():
    return render_template("harrisoperator.html")


@app.route("/harris", methods=["GET", "POST"])
def harris():
    if request.method == "POST":
        image_data = base64.b64decode(
            request.form["image_data"].split(',')[1])
```

Activate Windows
Go to Settings to activate

2

```python
@app.route("/harris", methods=["GET", "POST"])
def harris():
    if request.method == "POST":
        image_data = base64.b64decode(
            request.form["image_data"].split(',')[1])

        img_path = img1.saveImage(image_data, "harris_img")
        # img_binary = img1.readImg(img_path)

        img = cv2.imread(img_path)
        imggray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

        # img = cv2.imread(filepath1)
        # imggray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        t1 = time.time()
        r = Harris.getHarrisRespone(imggray)
        corners = Harris.getHarrisIndices(r)
        cornerImg = np.copy(img)
        cornerImg[corners == 1] = [255, 0, 0]
        t2 = time.time()

        # final_img = './static/images/output/output.jpg'
        # plt.savefig(final_img)

        plt.imsave('./static/images/output/output.jpg', cornerImg)
        # return "./static/images/output/output.jpg", t2-t1

        computationTime = t2 - t1

        current_GMT = time.gmtime()
        time_stamp = calendar.timegm(current_GMT)

        output_path = './static/images/output/output.jpg'
```

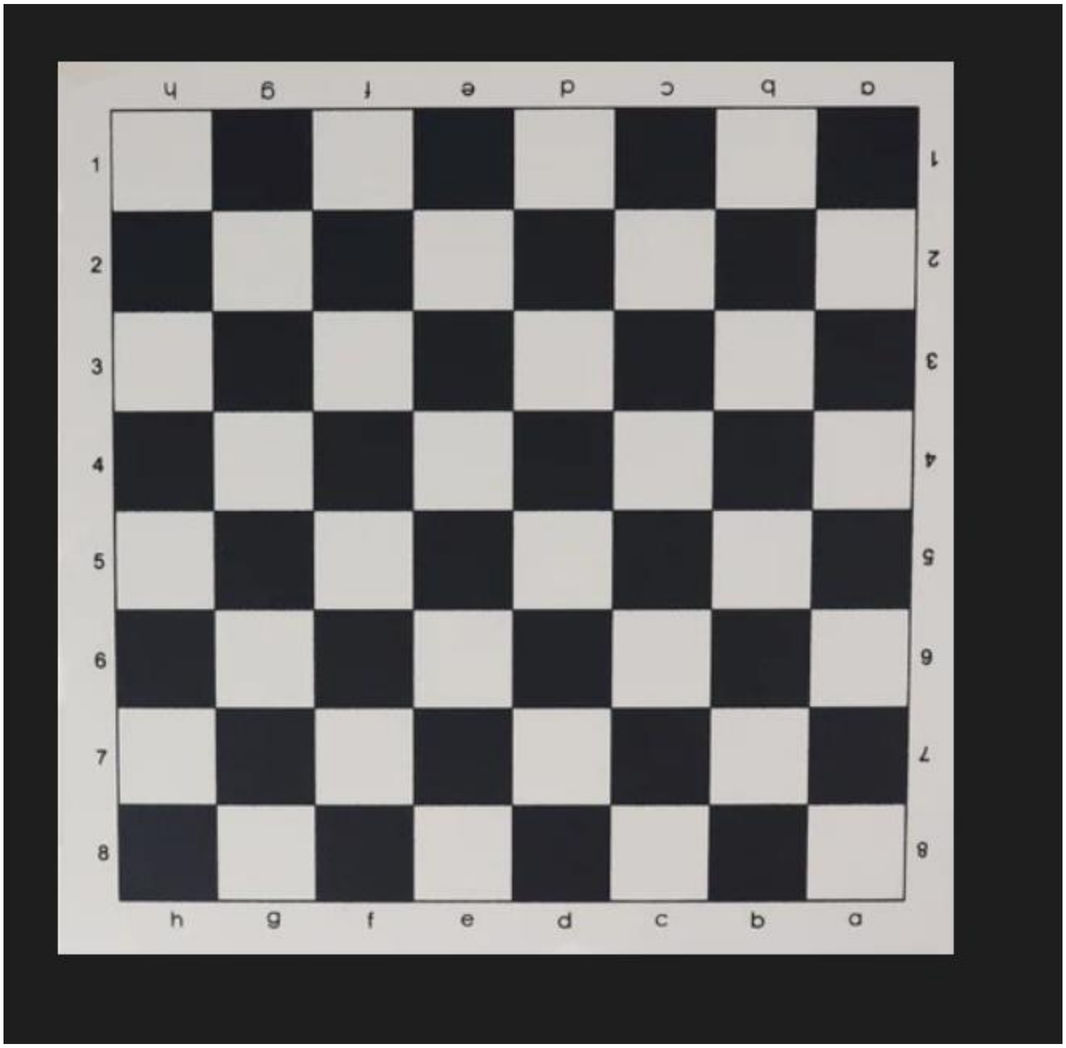
soperator.py > ...

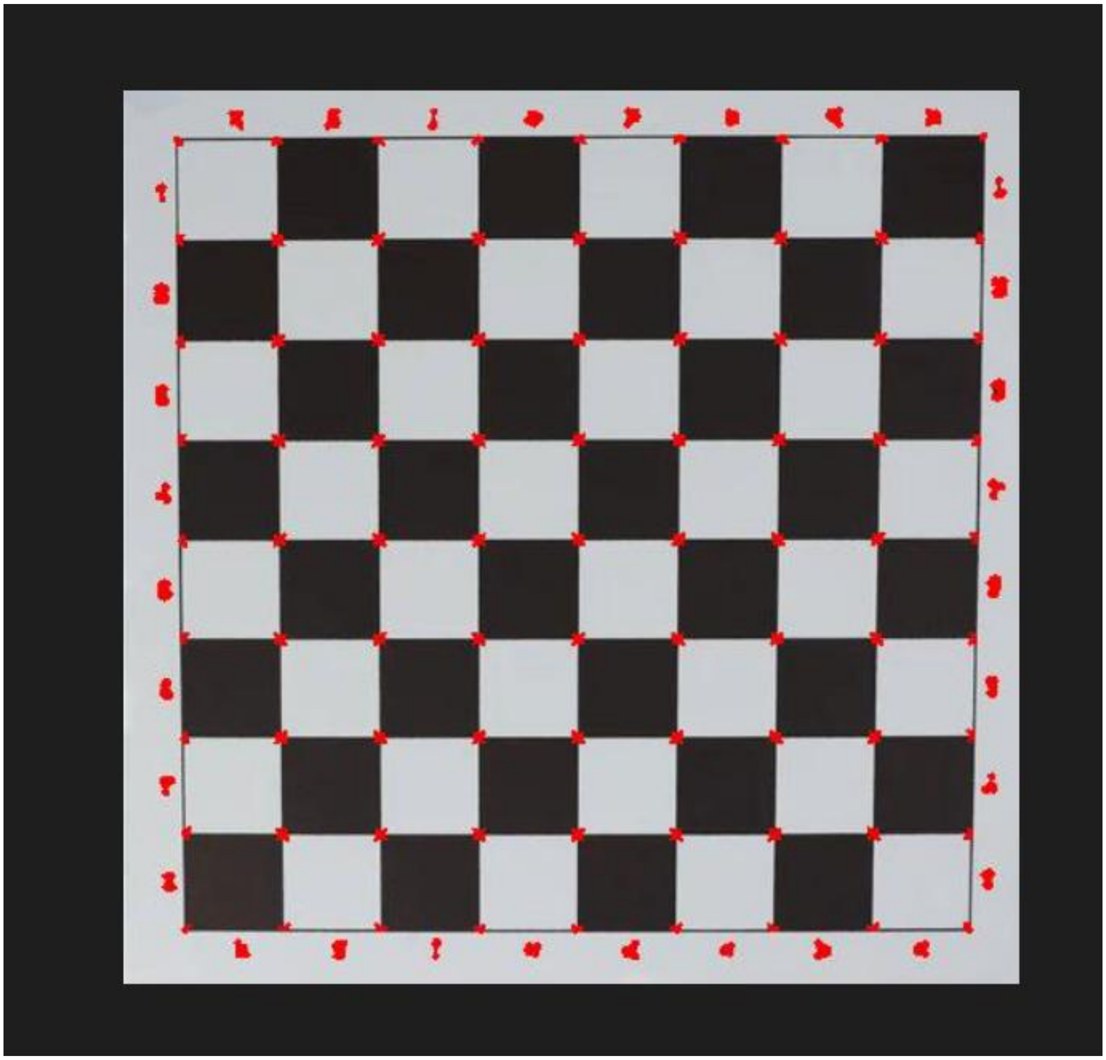
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt


def getHarrisRespone(imggray):
    # Calculation of Sobelx
    sobelx = cv2.Sobel(imggray, cv2.CV_64F, 1, 0, ksize=5)
    # Calculation of Sobely
    sobely = cv2.Sobel(imggray, cv2.CV_64F, 0, 1, ksize=5)
    # Apply GaussianBlue for noise cancellation
    Ixx = cv2.GaussianBlur(src=sobelx ** 2, ksize=(5, 5), sigmaX=0)
    Ixy = cv2.GaussianBlur(src=sobely * sobelx, ksize=(5, 5), sigmaX=0)
    Iyy = cv2.GaussianBlur(src=sobely ** 2, ksize=(5, 5), sigmaX=0)

    k = 0.05
    # determinant
    detA = Ixx * Iyy - Ixy ** 2
    # trace
    traceA = Ixx + Iyy
    # get r
    harris_response = detA - k * traceA ** 2
    return harris_response


def getHarrisIndices(harrisRes):
    # find edges and corners using r
    #Edge : r < 0
    #Corner : r > 0
    #Flat: r = 0
    threshold = 0.01
    harrisRecsopy = np.copy(harrisRes)
    rMatrix = cv2.dilate(harrisRecsopy, None)
    rMax = np.max(rMatrix)
    corner = np.array(harrisRes > (rMax*threshold), dtype="int8")
    return corner
```

image.py > ...

```python
import numpy as np
import cv2


# this function takes the image data that sent from js code and new image name
# then saves the image to the input folder and returns its path
def saveImage(imgData, imgName):
    path = f'./static/images/input/{imgName}.jpg'
    with open(path, 'wb') as f:
        f.write(imgData)

    return path


# this function takes the image path
# then reads the image as grayscale image and resize it and returns the result
def readImg(path):
    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    img = cv2.resize(img, (600, 600))
    return img
```

Input&output:



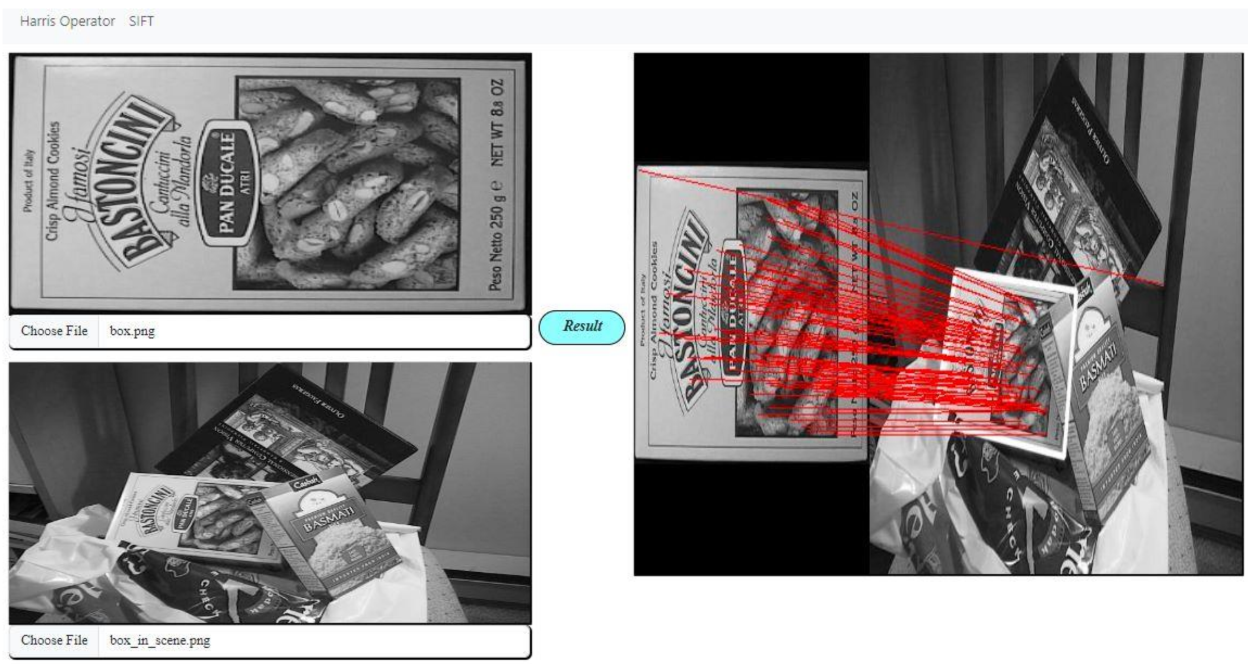


2.scale invariant features(SIFT).

SIFT (Scale-Invariant Feature Transform) is a computer vision algorithm to detect, describe, and match local features in images1. It was invented by David Lowe in 19991. There are mainly four steps involved in SIFT algorithm23:

- Scale-space extrema detection
- Keypoint localization
- Orientation assignment
- Keypoint descriptor generation.

There is no picture of the code because it is 450 lines



1.main function

```python
def computeKeypointsAndDescriptors(image, sigma=1.6, num_intervals=3, assumed_blur=0.5, image_border_width=5):

    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
```

2.localization

```python
def localizeExtremumViaQuadraticFit(i, j, image_index, octave_index, num_intervals, dog_images_in_octave, sigma, contrast_threshold, image_border_width, eigenvalue_ratio=10, num_attempt

    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 = 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 = -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]))
        # make sure the new pixel_cube will lie entirely within the image
        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 > nu
            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 * dot(gradient, extremum_update)
    if abs(functionValueAtUpdatedExtremum) * num_intervals >= contrast_threshold:
        xy_hessian = hessian[:2, :2]
        xy_hessian_trace = trace(xy_hessian)
        xy_hessian_det = det(xy_hessian)
        if xy_hessian_det > 0 and eigenvalue_ratio * (xy_hessian_trace ** 2) < ((eigenvalue_ratio + 1) ** 2) * xy_hessian_det:
            # Contrast check passed -- construct and return OpenCV KeyPoint object
            keypoint = 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]) / float32(num_intervals))) * (2 ** (octave_index + 1))  # octave_index + 1 because the input image was doub
            keypoint.response = abs(functionValueAtUpdatedExtremum)
            return keypoint, image_index
    return None
```

```python
def removeDuplicateKeypoints(keypoints):

    if len(keypoints) < 2:
        return keypoints

    keypoints.sort(key=cmp_to_key(compareKeypoints))
    unique_keypoints = [keypoints[0]]

    for next_keypoint in keypoints[1:]:
        las (variable) last_unique_keypoint: Any 1]
        if                                       point.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
```

```python
def findScaleSpaceExtrema(gaussian_images, dog_images, num_intervals, sigma, image_border_width, contrast_threshold=0.04):
    logger.debug('Finding scale-space extrema...')
    threshold = floor(0.5 * contrast_threshold / num_intervals * 255)  # from OpenCV implementation
    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_oct
            # (i, j) is the center of the 3x3 array
            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_octa
                        if localization_result is not None:
                            keypoint, localized_image_index = localization_result
                            keypoints_with_orientations = computeKeypointsWithOrientations(keypoint, octave_index, gaussian_images[octave_index][loc
                            for keypoint_with_orientation in keypoints_with_orientations:
                                keypoints.append(keypoint_with_orientation)
    return keypoints
```

3.sum of squared differences (SSD) normalized cross correlations.

In digital image processing, template matching is a process to determine the location of sub image inside an image. The sub image, which is called template, usually has similarity with a part of the image. The template can be in different size, color or form. Template matching is famously used in image registration and object recognition. In this paper, we focus on the performance of the Sum of Squared Differences (SSD) and Normalized Cross Correlation (NCC)as the techniques that used in image registration for matching the template with an image. This experiment is aiming to compare the ability of both techniques in term of quality of output image as well as the time taken in execution process. Furthermore, it also to test the effect of template image to output image when there is noise and rotation. Finally, the performance of these methods is tested by making comparison based on the value of correlation coefficient that produced from different image templates.