Task -3-

Computer vision

Under supervision of Dr. Ahmed M. Badawi

Team 21 members:

Name	***Section***	***B.N***
Amira Mohamed Abd El-Fattah	*1*	*15*
Doha Eid	*1*	*49*
Maha Medhat Fathy	*2*	*38*
Mayar Ehab Mohamed	*2*	*41*
Mariam Mohamed Ezzat	*2*	*33*

Overview:

This project is an educational desktop application that performs many functions on uploaded images and visualizing the effect of each of these functions by displaying the output and input images to show difference between them all in one page .

All functions in our application are written using C++ language without using any built-in functions of the OpenCV library.

Features:

- ***Extract the unique features using Harris operator:***
  which is a popular corner detection algorithm in computer vision and image processing. The Harris corner detector is based on the observation that corners can be detected by looking for significant changes in all directions around a pixel. The Harris corner detector works by computing the Harris response function for each pixel in an image. The Harris response function measures the amount of corner-like structure at each pixel. Pixels with high Harris response values are considered to be corners. The corners can be detected by thresholding the Harris response function and finding the local maxima in the response map. The Harris corner detector can be used for tasks such as feature detection, feature matching, and object recognition.

- ***Extract the unique features using min. Eigenvalue operator:***
  The minimum eigenvalue is commonly used in features extraction and it can be used in corner detection by using it with Harris corner detector. In these algorithms, the minimum eigenvalue is used to compute a corner response function that measures the amount of corner-like structure at each pixel. Pixels with high corner response values are considered to be corners.

- ***Generate feature descriptors using the scale invariant features (SIFT):***
  The Scale-Invariant Feature Transform (SIFT) is a popular feature extraction algorithm used in computer vision and image processing. It works by detecting and extracting key points from an input image, using a scale-space representation to make the key points scale-invariant. SIFT computes a descriptor for each key point that encodes the local gradient orientations of the image patch around the key point, making it robust to changes in the scale, orientation, and viewpoint of the object being detected.

- ***Matching of the image features using sum of squared differences (SSD) and normalized cross correlations.***

Implemented algorithms steps:

Harris function:

1. Define a class named "Harris".
2. Define a function named "cornerHarris_self" that takes a double value k as a parameter.
3. Create Mat objects to hold the absolute gradient in the x direction, absolute gradient in the y direction, x^2, y^2, xy, x^2 Gaussian, y^2 Gaussian, and xy Gaussian.
4. Set the scale and delta values for the Sobel function.
5. Use the Sobel function to calculate the x and y derivatives of the input grayscale image.
6. Calculate the other three images in M (x^2, y^2, and xy) using the x and y derivatives.
7. Apply Gaussian blur to each of the three images in M.
8. Calculate the Harris response function R using the formula $(x^2\, y^2 - xy^2) - k(x^2 + y^2)^2$.
9. Define a function named "cornerHarris_demo" that takes a Mat object "src" and two void pointers as parameters.
10. Call the "cornerHarris_self" function with a value of k.
11. Normalize the Harris response function values between 0 and 255.
12. Convert the normalized Harris response function values to integer format.
13. Create a copy of the input image to draw the detected corners on.
14. Loop through all pixels in the normalized Harris response function matrix.
15. If the Harris response value at the pixel is above a specified threshold, draw a circle around the pixel on the output image.
16. Return the output image with detected corners drawn on it.

Minimum Eigen value with Harris operator:

1. Define a function named detectCorners that takes an input image and a threshold value as parameters.
2. Convert the input image to grayscale using the cvtColor function from OpenCV.
3. Compute the gradients in the x and y directions using the Sobel function from OpenCV.
4. Compute the second-order partial derivatives of the image intensity using the Sobel gradients to calculate the Hessian matrix at each pixel.

5. Compute the Harris response function for each pixel using the eigenvalues of the Hessian matrix.
6. If the Harris response is above the specified threshold, add the pixel to the list of detected corners.
7. Return a vector of Point2f objects representing the locations of the detected corners.
8. In the main function, load an input image using the imread function from OpenCV.
9. Call the detectCorners function to detect corners in the image with a threshold value of 100000.
10. Visualize the detected corners by drawing circles at their locations using the circle function from OpenCV.
11. Display the resulting image in a window using the imshow function from OpenCV.

Generate features using the scale invariant features (SIFT):

1- The generate_gaussian_pyramid() function takes an input image and generates a Gaussian pyramid with a minimum sigma value, number of octaves, and scales per octave.

2- The function resizes the input image to twice its original size and applies Gaussian smoothing to reach the required base sigma.

3- It determines a list of sigma values for blurring and creates a scale space pyramid of Gaussian images using these sigma values.

4- Each octave in the pyramid is half the size of the previous one.

5- The generate_dog_pyramid() function takes the Gaussian pyramid generated by generate_gaussian_pyramid() and generates a Difference of Gaussians (DoG) pyramid.

6- The DoG pyramid is created by subtracting each image in a given octave from the previous image in the same octave.

7- The resulting pyramid has the same number of octaves as the Gaussian pyramid but one fewer image per octave.

8- The point_is_extremum() function checks if a given pixel is an extremum (a local minimum or maximum) in its 3x3 neighborhood across the octave.

9- It takes a vector of images representing an octave, the scale index of the pixel, and the pixel coordinates.

10- The function checks if the pixel value is greater or less than all of its 3x3 neighbors in the previous, current, and next scales in the octave.

11- If the pixel value is neither a local minimum nor maximum, the function returns false, indicating that the pixel is not an extremum.

12- The fit_quadratic function fits a quadratic surface to the neighborhood of a given keypoint in an image, estimates the offset of the interpolated extremum from the discrete extremum, and returns the offsets as a tuple.

13- The point_is_on_edge function checks if a given keypoint is located on an edge (i.e., a region with high gradient magnitude) by computing the Hessian matrix at the keypoint and evaluating the ratio of its trace to determinant.

14- The find_input_img_coords function calculates the scale, x, and y coordinates of a given keypoint in the original input image, given the octave, offsets, and other parameters.

15- The refine_or_discard_keypoint() function iteratively refines the position of a given keypoint() by calling fit_quadratic(), checking the contrast and edge conditions using point_is_on_edge(), and updating the keypoint coordinates using find_input_img_coords(). The loop continues until the maximum number of iterations is reached or the keypoint is deemed valid based on the conditions.

16- find_keypoints() function:

- Takes a scale-space pyramid of difference of Gaussian (DoG) images, a contrast threshold, and an edge threshold as input.
- Finds local extrema in scale and space in each image of the pyramid.
- Refines keypoint locations and discards keypoints with low contrast or that are on edges. Returns a vector of Keypoint structs representing the detected keypoints.

17-generate_gradient_pyramid() function:

- Takes a scale-space pyramid of images as input.
- Computes the gradient (x and y derivatives) for each image in the pyramid. Returns a new scale-space pyramid where each image represents the gradient of the corresponding image in the input pyramid.

18- smooth_histogram() function:

- Takes an array of histogram bins as input.
- Applies a smoothing filter to the histogram using a 6x box filter.
- Updates the input histogram with the smoothed values.

19- find_keypoint_orientations() function:

- Takes a Keypoint struct, a gradient pyramid, and two parameters lambda_ori and lambda_desc as input.
- Computes a set of reference orientations for the keypoint based on the gradient orientations in a circular patch around the keypoint.
- Smoothes the histogram of gradient orientations using smooth_histogram.
- Extracts local maxima in the smoothed histogram as reference orientations.
- Returns a vector of reference orientations.

20- update_histograms() function:

- Takes a 3D histogram array, coordinates of a sample, a sample contribution, the reference orientation of the keypoint, and the descriptor width parameter as input.
- Computes weights for the sample based on its distance from the center of the histogram bin and its orientation difference from the reference orientation. Updates the histogram with the weighted contribution of the sample.

21- hists_to_vec() function:

- Takes a 3D histogram array and converts it to a feature vector.
- Normalizes the histogram by the L2 norm of its values.
- Quantizes the normalized histogram values into 8-bit integers.

22- compute_keypoint_descriptor() function:

- Takes a Keypoint object, a reference orientation, a gradient pyramid, and the descriptor width parameter as input.
- Computes the start and end coordinates of the patch in the image based on the keypoint position and descriptor width.
- Accumulates samples within the patch into histograms using update_histograms().
- Converts the histograms to a feature vector using hists_to_vec().
- Stores the resulting feature vector in the descriptor member of the Keypoint object.

23- find_keypoints_and_descriptors function:

- Takes an input image, various parameters for creating a scale-space pyramid, and parameters for keypoint detection and descriptor computation. Generates scale-space pyramids for the input image and its gradient, and uses them to detect keypoints and compute descriptors.

- Returns a vector of Keypoint objects, each representing a detected keypoint with its descriptor.

24- SSD() function:

Takes two std::array<uint8_t, 128> arrays as input, representing the descriptors of two keypoints. Computes the Euclidean distance between the two arrays.

25- normalized_cross_correlation() function:

Takes two std::array<uint8_t, 128> arrays as input, representing the descriptors of two keypoints. Computes the normalized cross-correlation between the two arrays.

26- find_keypoint_matches() function:

- Takes two vectors of Keypoint objects, representing the keypoints and descriptors of two images, and two threshold parameters for matching.
- Computes matches between the keypoints using the Euclidean distance and normalized cross-correlation metrics.
- Returns a vector of pairs of indices into the two input vectors, indicating which keypoints match.

27- draw_keypoints() function:

Takes an image and a vector of Keypoint objects as input. Draws circles at the locations of the keypoints in the image. Returns a new image with the keypoints drawn.

28- draw_matches() function:

- Takes two images, two vectors of Keypoint objects representing the keypoints and descriptors of the two images, and a vector of matches between keypoints as input. Draws lines between the matching keypoints in the two images.
- Returns a new image with the matches drawn.

Features extraction algorithms:

Using cross correlation:

In our code the function named "euclidean_dist" has been renamed to "normalized_cross_correlation". This function mainly calculates the mean of the two input arrays, then computes the numerator and denominator of the normalized cross-correlation formula separately. The function then checks if the denominator is equal to zero (to avoid division by zero), and returns the resulting normalized cross-correlation value.

It is important that the normalized cross-correlation between two arrays always falls within the range of -1 to 1, where a value of 1 indicates a **perfect match**, 0 indicates **no correlation**, and -1 indicates a **perfect anti-correlation.**

Using sum of squared differences (SSD):

The function now simply returns the sum of squared differences between the two arrays, which is equivalent to the SSD.

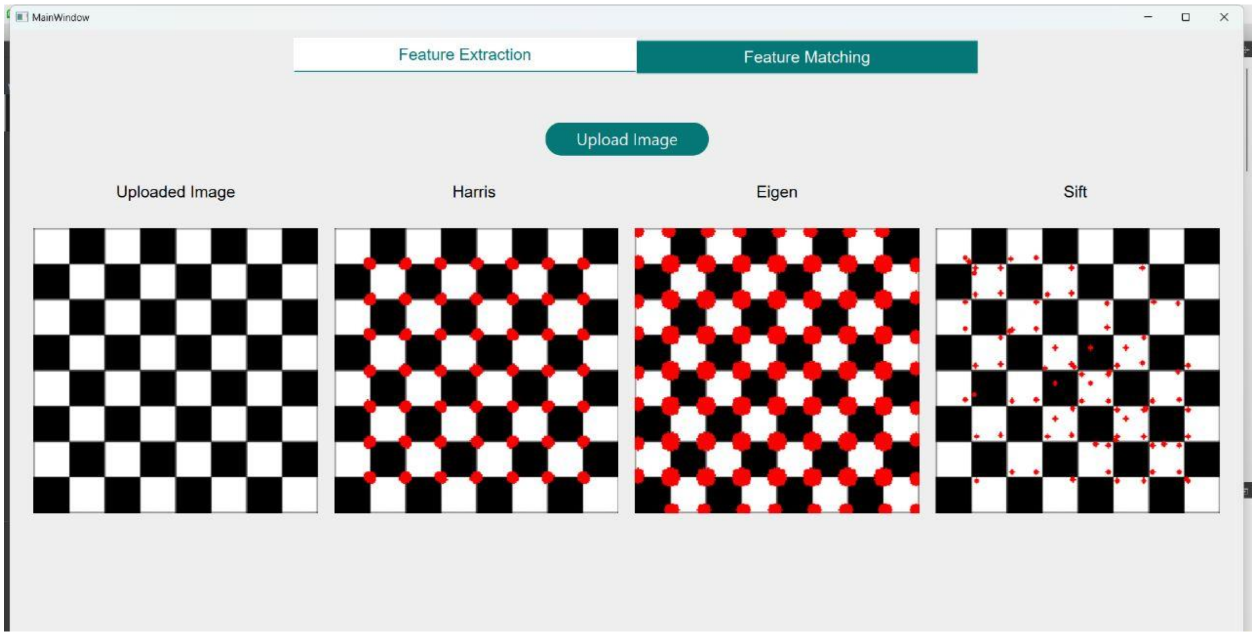
Computation time of our algorithms:

- computation time of SIFT function = 7303 milliseconds.
- computation time of SSD =7863 milliseconds.
- computation time of normalized cross correlations = 11339 milliseconds.
- computation time for features extraction using eigen values and Harris operator = 26 milliseconds.
- Harris computation time = 1180 milliseconds.

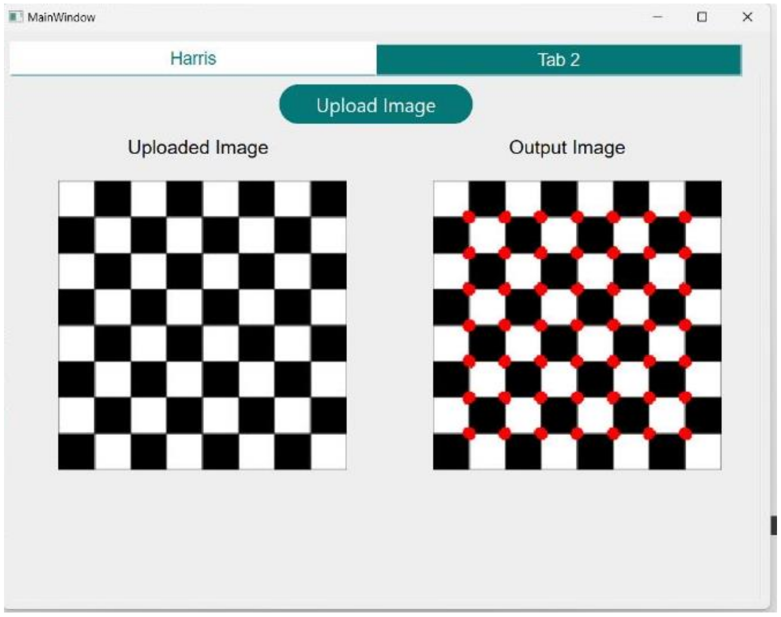
Outputs obtained using our application:

The first page in our application contain:

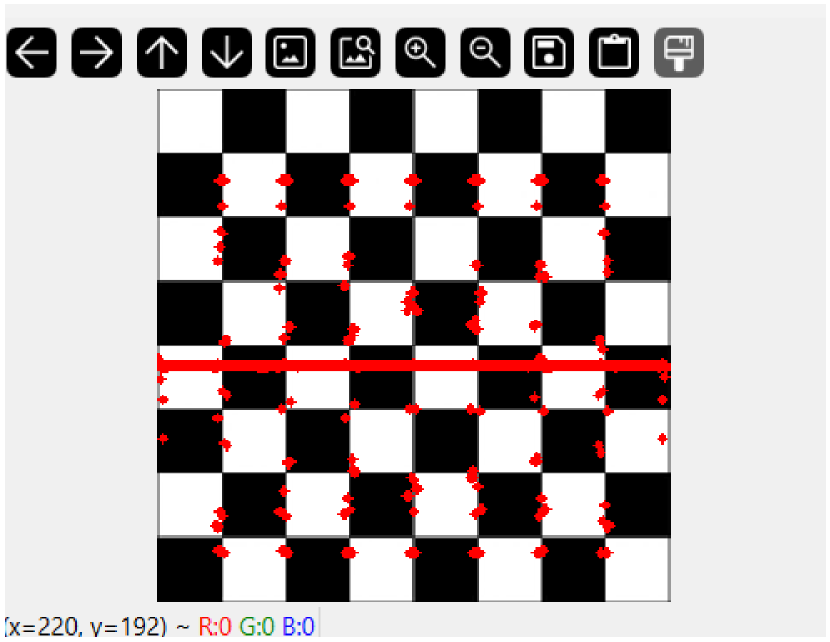
- The uploaded image.
- Output after applying the Harris operator function.
- The output after using the minimum eigen values for features detection with the Harris operator.
- The features plotting obtained from the sift function.



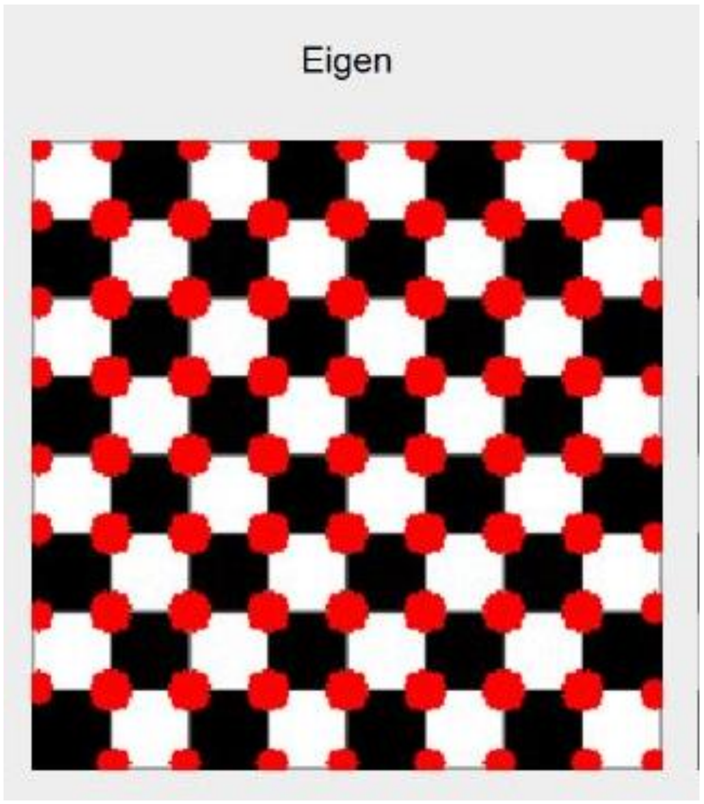
__Input and output of Harris operator function:__



__Output of features plotting by using eigen values and eigen vectors:__

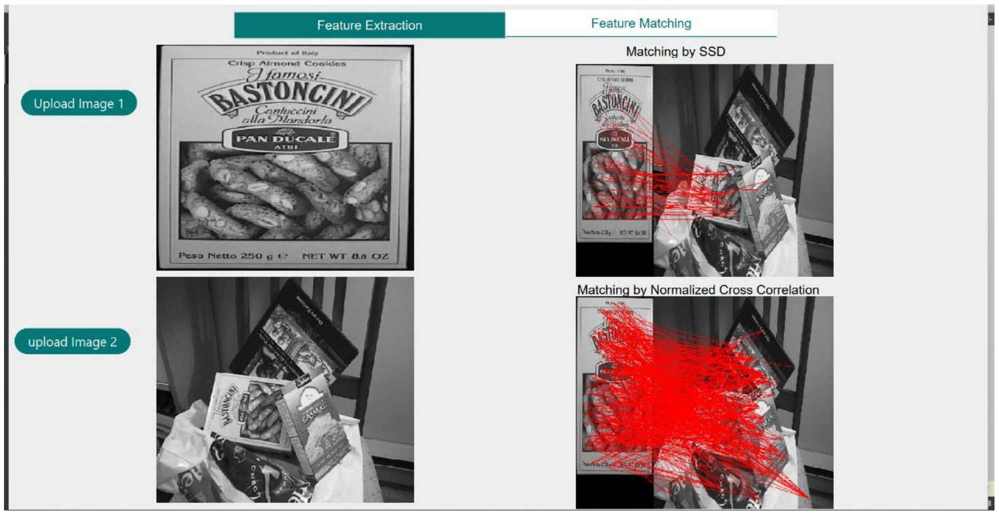


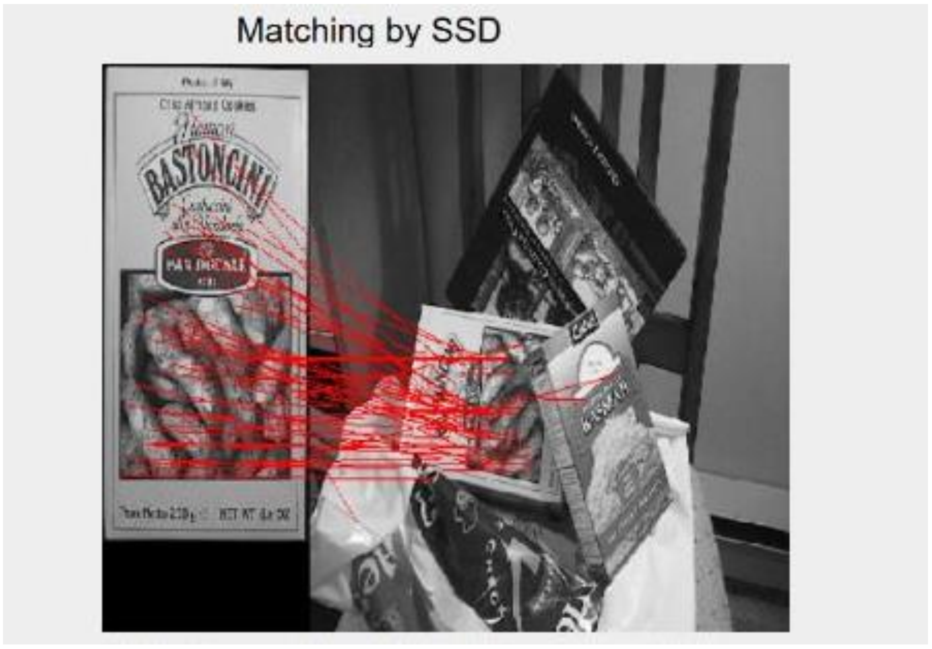
Input and output of features extraction using minimum Eigen value with the Harris operators:

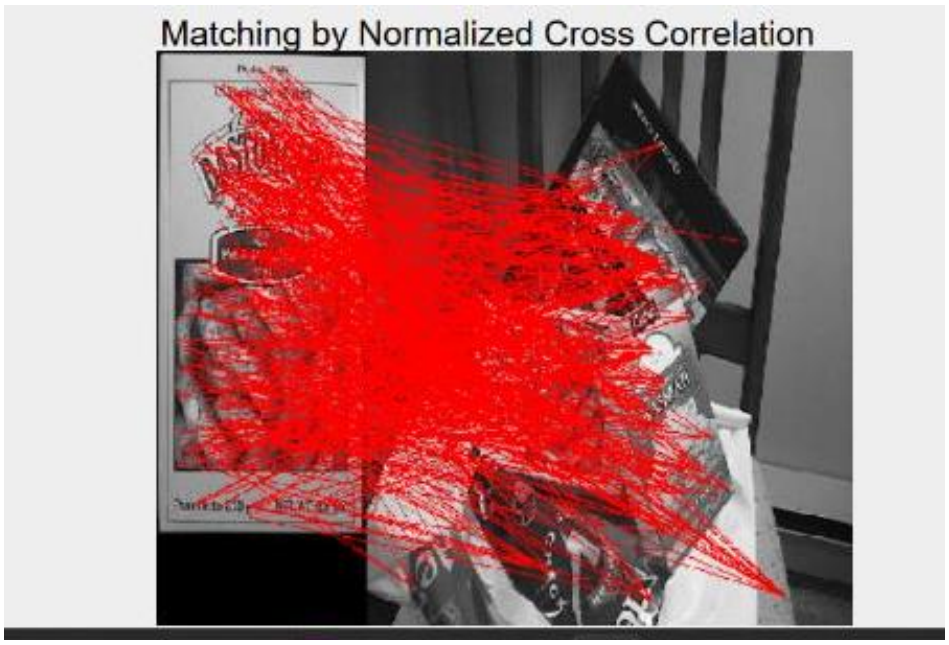


Second tab in our application:

- Features matching using SSD.
- Features matching using by normalized cross correlation.






Matching by Normalized Cross Correlation