UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

COMPUTER VISION

Report Lab 02
Harris Key Point Detection

Lecturers	Phạm Minh Hoàng
	Nguyễn Trọng Việt
	Võ Hoài Việt
Student	Võ Nguyễn Hoàng Kim
	21127090

CONTENT

A. SELF-ASSESSMENT FORM

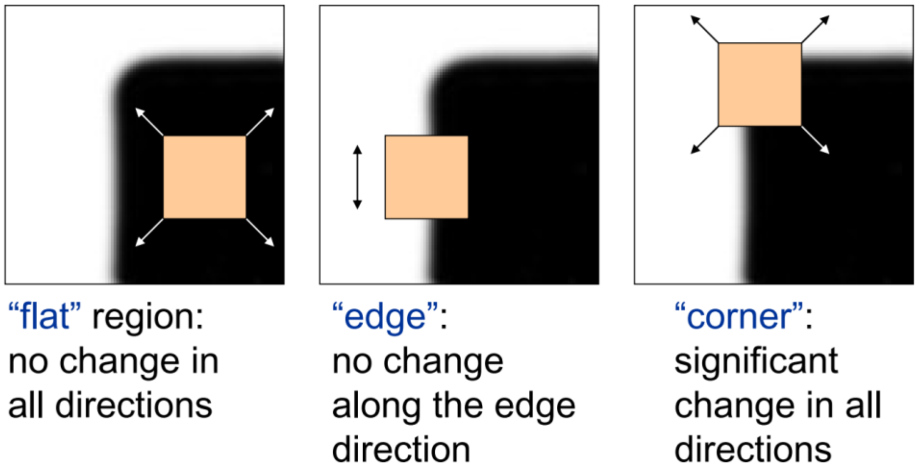
Algorithm	Requirement	Level of completion
Harris key point detection	Detect and highlight key points by Harris.	100%

B. ANALYSIS AND RESULT

I. The Harris detector

1. Corner

- In images, a corner is an area where two edges meet together, it has the following characteristics: [1]
    - Discreteness: Occupying a unique and small space, not forming a continuous strip.
    - Reliability: Likely to consistently appear from one frame to another.
    - Meaningfulness: Strong corners often originate from real objects rather than shadows or lighting.
- The following image illustrates the differences between flat, edge, and corner: [2]


"flat" region: no change in all directions
"edge": no change along the edge direction
"corner": significant change in all directions

2. Harris Theory

a. Theory [2]

- A formula to calculate the change in brightness intensity when shifting by [u, v] as follows:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

where:
    - $w(x,y)$: Window function
    - $I(x+u,y+v)$: Shifted intensity.
    - $I(x,y)$: Intensity.
- By Taylor, expansion of $I(x+u\,,y+v)$:

$$I(x+u,y+v) \approx I(x,y) + I_u(x,y)u + I_v(x,y)v$$

$$E(u,v) \approx \sum_{x,y} w(x,y)[I_u(x,y)u + I_v(x,y)v]^2$$

- The bilinear approximation simplifies to

$$E(u,v) \approx [u \quad v]\, M \begin{bmatrix} u \\ v \end{bmatrix}$$

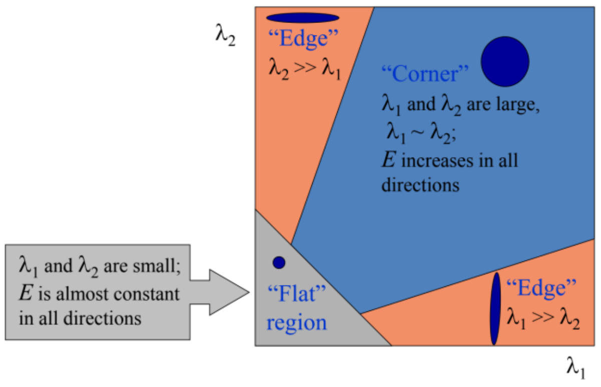
- Where M is 2×2 matrix computed from image derivatives:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix}$$

- Since M is symmetric, we have:

$$M = R^{-1} \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} R$$

- Classification of image points using eigenvalues of M:


λ_2
"Edge"
$\lambda_2 >> \lambda_1$
"Corner"
λ_1 and λ_2 are large,
$\lambda_1 \sim \lambda_2$;
E increases in all directions
λ_1 and λ_2 are small;
E is almost constant in all directions
"Flat" region
"Edge"
$\lambda_1 >> \lambda_2$
λ_1

- Corner response function:

$$R = det(M) - \alpha\, trace(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$$

- α: is a constant with the value from 0.04 to 0.06.

b. Algorithm [2]

- Filter image with Gaussian.
- Compute magnitude of the x and y gradients at each pixel.
- Construct M in a window around each pixel (using a Gaussian window).
- Compute λs of M.
- Compute $R = det(M) - \alpha\, trace(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$
- If $R > T$:
  - A corner is detected.
  - Retain point of local maxima

II. Experiment

1. Input and Output

Input

- To implement this algorithm, we use an input image as a color image.

Output

- At the end of the program, the output image remains a color image. However, the detected corners will be highlighted with red circles.

2. Helper functions

a. Matrix computation functions

Multiply

- Name of function: **multiplyMat**
- Purpose: This function is used to multiply corresponding elements between two matrices, resulting in the output matrix.
- Parameters: Two matrices *matrixA* and *matrixB*
- Implement:
    - Use *for* loops to access each element [i, j] of the matrices.
    - With each element at position [i, j], perform the multiplication between the corresponding elements of the *matrixA* and *matrixB*.
    - Save the result of this multiplication into the corresponding element [i, j] of the output matrix *resultMatrix*.

Sum

- Name of function: **sumMat**
- Purpose: This function is used to sum corresponding elements between two matrices, resulting in the output matrix.
- Parameters: Two matrices *matrixA* and *matrixB*
- Implement:
    - Use *for* loops to access each element [i, j] of the matrices.
    - With each element at position [i, j], perform the addition between the corresponding elements of the *matrixA* and *matrixB*.
    - Save the result of this addition into the corresponding element [i, j] of the output matrix *resultMatrix*.

Subtract

- Name of function: **subtractMat**
- Purpose: This function is used to calculate the convolution between two input matrices and save the result into the output matrix.
- Parameters: Two matrices *matrixA* and *matrixB*
- Implement:
    - Use *for* loops to access each element [i, j] of the matrices.
    - With each element at position [i, j], perform the subtraction between the corresponding elements of the *matrixA* and *matrixB*.
    - Save the result of this subtraction into the subtraction element [i, j] of the output matrix *resultMatrix*.

b. Gradient computation functions

Convolution

- Name of function: **myConvolution**

- Purpose: This function is used to calculate the convolution between an input matrix and a kernel with a size of $k \times k$.
- Parameters: A matrix *matrix*, a kernel *kernel* with *k* size
- Implement:
    - Use *for* loops to iterate through each element of the matrix.
    - For each element [i, j] in the matrix, perform the convolution operation with the kernel.
    - Record the result at the corresponding position [i, j] in the output matrix.
    - ***Note:*** For the boundary elements, due to the lack of neighboring elements, convolution cannot be performed effectively. To ensure the convolution task is carried out, we accept skipping these edge elements, meaning we only copy the values from the original matrix.

Gradient

- Name of function: **gradientBySobel**
- Purpose: This function is used to compute the gradient between an input matrix and a kernel with a size of $k \times k$.
- Parameters: A input matrix *matrix*, another matrix to contain the result *resMatrix*, flags (*axisX*, *axisY*) are used to mark which direction the derivative function should be performed.
- Implement:
    - In this program, we use the Sobel operator to calculate derivatives.
    - Perform convolution between the input matrix and the Sobel X mask to calculate the derivative along the X-direction (flag *axisX = 1*, *axisY = 0*).
    - Perform convolution between the input matrix and the Sobel Y mask to calculate the derivative along the Y-direction (flag *axisX = 0*, *axisY = 1*).
    - The convolution operation is carried out using the **myConvolution** function mentioned earlier. The Sobel masks are initialized with fixed values in the program.

c. Checking function

Checking the local maxima

- Purpose: This function is used to check an element whether it is a local maximum in an area with the size of $k \times k$.
- Parameters: A matrix *matrix*, index of the element *rowIndex, colIndex*, value of the checking element *checkingValue*.
- Implement:
    - Utilize a 3x3 filter window to check if an element is a local maximum within its neighborhood.
    - If there exists an element greater than *checkingValue*, the function immediately returns False.
    - If, after traversing the neighborhood, no value greater is found, it is determined to be the local maximum, and the function returns True.
    - ***Note:*** For the boundary elements, due to the lack of neighboring elements, the checking cannot be performed effectively. To ensure the checking task is carried out, we accept skipping these edge elements.

3. Harris Corner Detection

Based on the theory of the Harris corner detection algorithm, we proceed to implement it with the following steps:

Step 1: Preprocess image

- **Convert to grayscale**: The algorithm is implemented to process grayscale images. However, the input images received by the program are RGB color images. Therefore, it is necessary to perform the conversion from RGB to Grayscale for the algorithm to work accurately.
    - This task is implemented by using **cv::cvtColor()** function.
- **Noise reduction:** Factors such as noise can impact the search results of the algorithm. Therefore, noise reduction is one of the important tasks that helps improve the final accuracy of the results.
    - This task is implemented by using **cv::GaussianBlur()** function with the *kernel size* is 5 × 5.

Step 2: Compute the gradient

- **Compute the gradient:** To compute the gradient of pixels in the image, we use the derivative method along both the x and y directions to record the gradient results separately. These results are then stored in two distinct matrices respectively I_x, I_y.
    - We use **gradientBySobel()** with flag *axisX = 1*, *axisY = 0* to compute the derivative along the X-direction.
    - We use **gradientBySobel()** with flag *axisX = 0*, *axisY = 1* to compute the derivative along the X-direction.

Step 3: Construct M

- **Compute factors:** There are 3 crucial factors that need to construct the M matrix: $(I_x)^2, (I_y)^2, I_{xy}$
    - To compute $(I_x)^2$: we use **multiplytMat()** function to perform the multiplication between I_x and I_x
    - Perform a similar operation to calculate $(I_y)^2$ and I_{xy} using the **multiplytMat()** function.
    - These results are stored in 3 variable A, B, C respectively.
- **Apply window function:** To complete the construction of the M matrix, a **window function** is applied to weigh the influence of surrounding pixels. This process enhances sensitivity to corners while reducing sensitivity in less crucial regions. In this program, we use a Gaussian window.
    - Apply **cv::GaussianBlur()** function (*kernel size* is 5 × 5) to three variables A, B, and C (the results of the previous calculations). Still store the results of this step in those three variables.
- Finally, we have M as below:

$$M = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$$

Step 4: Compute R

- **Find R:** After constructing M, we proceed to calculate relevant values to find R, based on the formula:

$$R = det(M) - \alpha\, trace(M)^2$$

  where:

    - α: is a constant that determines the influence of the trace term in the overall corner response calculation. The choice of α depends on the specific requirements of application. The common values for α range between 0.04 and 0.06. In this program, we use the $\alpha = 0.04$.
    - det(M): is the determinant of M. It is calculated based on the formula:
        - $det(M) = A \times C - B^2$
            - Firstly, we calculate the value of expression $A \times C$ using the **multiplyMat()** function.
            - Then, continue to use **multiplyMat()** function to calculate B^2

            - Finally, use the **subtractMat()** function to find the difference between these two values, following the provided formula. The result is the determinant of M.
    - trace(M): is the trace of M, which is calculated based on the formula:
        - $trace(M) = (A + C)^2$
            - Firstly, use the **sumMat()** function to calculate the value of the expression $A + C$.
            - Then, call **multiplyMat()** function to compute its square $(A + C)^2$. The result is the trace of M.
    - To calculate R follow the above formula:
        - Using **multiplyMat()** function to compute squared of $trace(M)$. The $trace(M)^2$ is used to multiply with α and store the result in a temporary variable.
        - Finally, use the **subtractMat()** function to compute the value of R.

Step 5: Threshold filtering

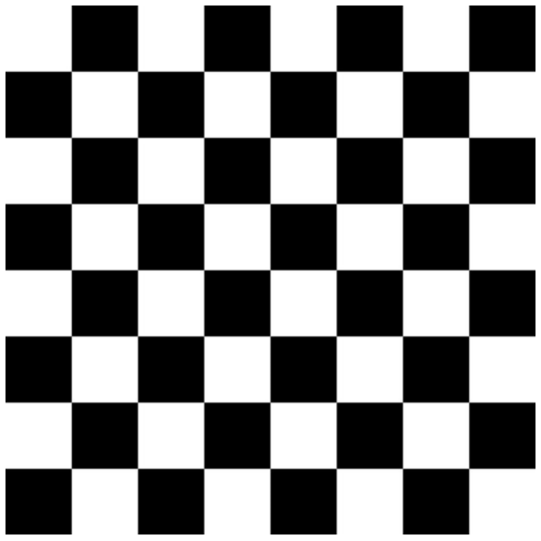
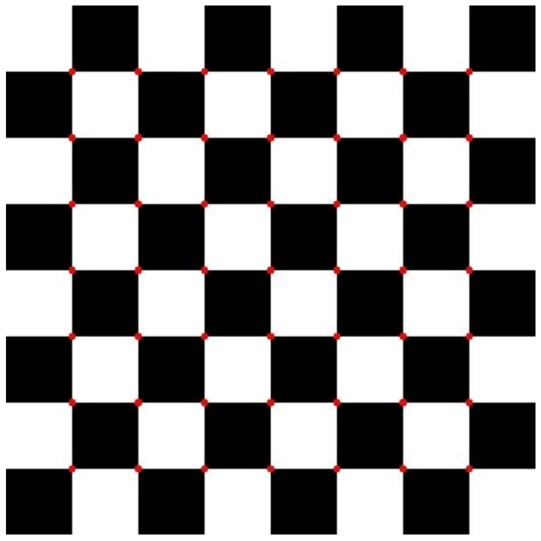
- **Apply threshold:** Threshold filtering is a step that involves applying a threshold to the computed corner response values (R). The purpose is to identify and retain only those points in the image that have a corner response above a certain threshold, discarding points with lower values. This thresholding step helps filter out weak or less significant corners, leaving behind only the strongest corners or features in the image.
    - In this program, we use the threshold value based on the corner response matrix (R): it is calculated by multiplying the maximum element value in R by a constant factor (here it is 0.01).
    - Performing threshold filtering: Use a for loop to sequentially iterate through the positions of elements in the image. At the pixel [i, j], if its value in the corner response matrix R is greater than the threshold T ($R[i,j] > T$), we identify it as a corner.
        - However, to restrict the corner points in the region, we perform a check for the local maximum value (using the **isLocalMaxima()** function): if this point is a corner and it is a local maximum in the region, we mark it. Otherwise, we skip it.
            - We use **cv::Circle()** function to mark corners on the image at the point (j, i).

4. Result

- This part will present the results (including input and output) of some example images:

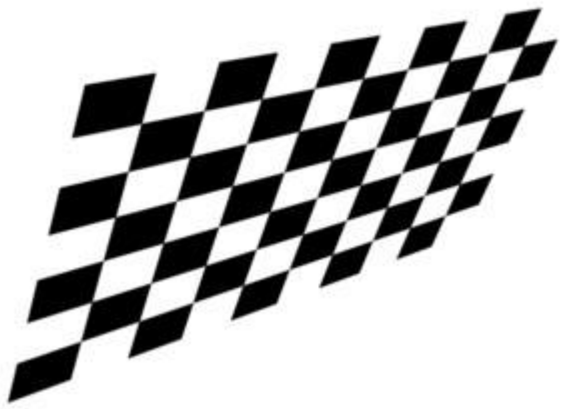
a. Example 1

Input Output

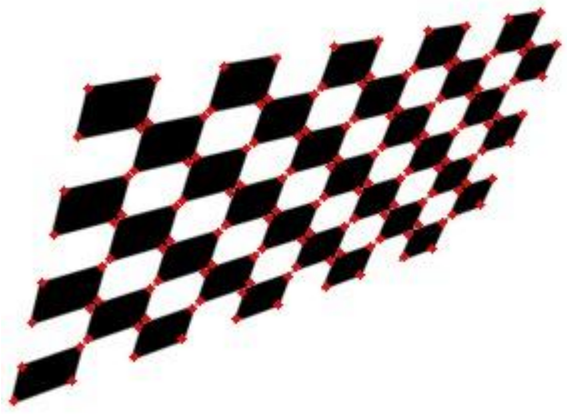



b. Example 2

Input



Output



C. USER GUIDE

I. Folder Structure

- There are 4 folders:
    - o Document: Contains the report.
    - o Data: Contains the 2 sub-folders: Input and Output, where input images and output images in.
    - o Executable: Contains executable file. The executable file is named **21127090.exe**
    - o Source: Contains all ***.cpp*** and ***.h*** files

II. How to run the source code

*The source code is implemented in **Visual Studio 2022** on the Windows operating system.*

- Run the source code, follow these steps:
    - o Step 1: Open the terminal of your browser.
    - o Step 2: Enter the path to the directory containing the source code.

- Step 3: Enter the command line based on the following structure:

  21127090.exe -harris <InputFilePath> <OutputFilePath>

  Notation:

  - 21127090.exe: the executable file
  - InputFilePath: the path of input file (that maybe ..\Data\Input\<name-of-image.png/jpg>
    - Example: ..\Data\Input\chessboard.jpg
  - OutputFilePath: the path of output file (we must assign the name for the output image)
    - Example: ..\Data\Output\chessboard.jpg

D. REFERENCES

[1] I. Berrios, “Harris Corner and Edge Detector,” 2023.

[2] Computer Vision - 21TGMT - Slide, “Corner Detection”.