
# Report of Task 3

## Introduction

In the realm of computer vision and image processing, feature extraction and matching are fundamental tasks that pave the way for more complex applications such as object recognition, image stitching, and 3D reconstruction. This report presents an in-depth analysis of a series of tasks performed on a given set of images, both grayscale and color.

The tasks encompass the extraction of unique features in all images using the Harris operator and λ- followed by the generation of feature descriptors using Scale Invariant Feature Transform (SIFT) The extracted features are then matched using two different methods: Sum of Squared Differences (SSD) and Normalized Cross Correlations.

Each task is accompanied by a detailed account of the computation times, providing valuable insights into the efficiency of these methods. The report aims to offer a comprehensive understanding of these techniques and their performance in the context of the given image set.

## I. Feature Extraction

### 1. Harris Operator Approach

The Harris corner detection algorithm is a widely used technique in computer vision for identifying corners in images. It operates by analyzing intensity changes in all directions, typically indicating corners or junctions. One of its key aspects is the computation of the "corner response function" or "f value," which approximates the lambda minus approach. This approximation reflects on the blurring effect in the detected corners, highlighting the need for post-processing techniques like non-maximal suppression.

#### 1.1. Harris Corner Detection Algorithm Steps

**1. Convert to Grayscale Image**

Convert the input color image to grayscale to simplify the processing and focus solely on intensity variations. Grayscale images contain only intensity information, making them suitable for detecting features based on intensity changes. This

simplification reduces computational complexity and enhances the efficiency of subsequent processing steps.

**2. Compute Derivatives**

Compute the image gradients using Gaussian derivative filters in both the x and y directions. Gaussian derivative filters provide a smoothed approximation of the actual derivatives, reducing the influence of noise and minor fluctuations in the image.

**3. Convolve Image with Derivative Kernels**

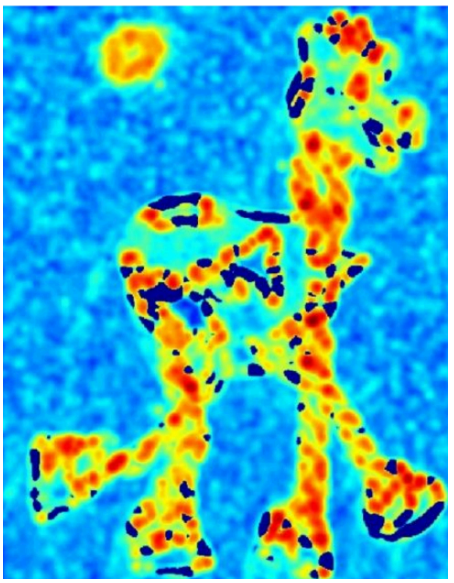
Perform convolution operations to obtain the x and y derivatives of the grayscale image. Convolution is a fundamental operation in image processing that calculates the response of a filter to the input image. By convolving the image with derivative kernels, we compute the intensity gradients at each pixel location, which are essential for detecting edges and corners.

**4. Compute Structure Tensor Components and f_value**

These tensors correspond to the elements of the H matrix, which is utilized to calculate the Harris response.
The structure tensor is a square symmetric matrix composed of second-order derivatives of image intensities. Its components represent the covariance of gradient values within the local image neighborhood. By analyzing the structure tensor components, we gain insights into the local structure of the image and extract features such as corners.
The Harris response (f value) is computed using the formula

$$\mathbf{f = det(H) / (trace(H) + \varepsilon),}$$

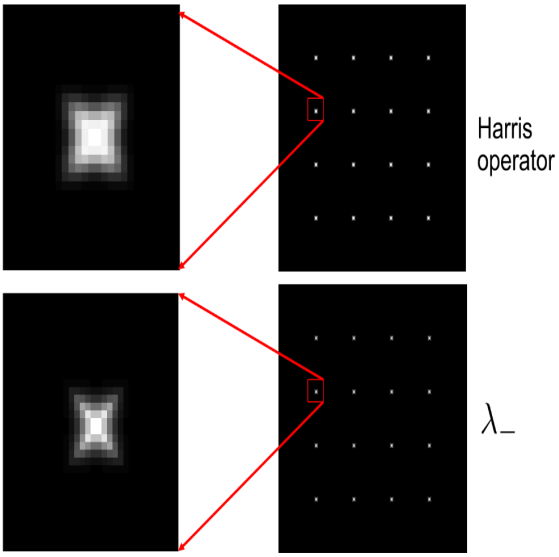
where det(H) represents the determinant of the structure tensor and trace(H) represents its trace. This ratio evaluates how much a small shift in the position of a window results in a significant change in intensity, It measures the likelihood of a pixel being part of a corner based on the local intensity variations and gradients. indicating potential corner points. The ε term is a small constant added to the denominator to prevent division by zero.



**5. Thresholding**

Apply a threshold to the Harris response to identify potential corner points. The threshold is a user-controlled step, typically specified as a percentage of the maximum Harris response. This percentage value is obtained from the user interface and multiplied by the maximum Harris response to determine the threshold value. Thresholding helps in distinguishing corners from other image features by selecting pixels with high Harris response values. It allows us to control the sensitivity of the corner detection algorithm and remove noise or weak corner candidates.

**6. Non-maximum Suppression**




Purpose: Perform non-maximum suppression to refine the detected corner points. Non-maximum suppression helps in eliminating multiple responses for the same corner and refining the localization accuracy. By selecting only the local maxima in the Harris response, we ensure that each corner is represented by a single point, enhancing the robustness of the corner detection algorithm. This step is crucial due to the blurring effect caused by the approximation used in the Harris corner detection algorithm. The approximation can lead to multiple nearby pixels having high Harris response values, resulting in multiple responses for the same corner. Non-maximum suppression addresses this issue by selecting only the strongest response in each local neighborhood, effectively reducing the blurring effect and improving the accuracy of corner localization.

**7. Get Corner Coordinates**

Retrieve the coordinates of the detected corner points. The coordinates of the detected corners represent the locations in the image where significant intensity variations occur. These corner coordinates are essential for further analysis and applications such as image stitching, object recognition, and motion tracking.

After applying thresholding to the Harris response values, we obtain a binary image where pixels with high Harris response values are marked as corner candidates. The argwhere function is then used to retrieve the coordinates of the detected corner points from the binary image. These coordinates represent the pixel locations where significant intensity variations indicative of corners are detected. The number of detected corners is printed for evaluation and further analysis.

## 1.2. Effect of Thresholding on Detected Corners

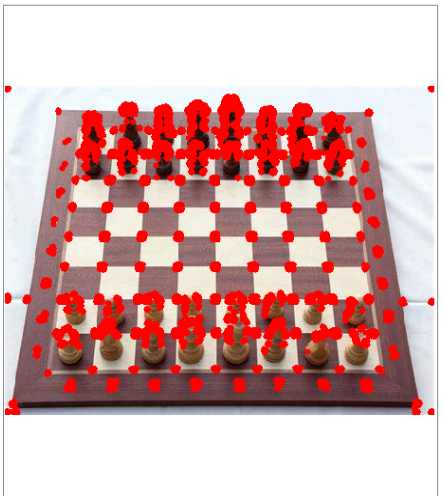
We explore the impact of different thresholding values on the detected corners in the Harris corner detection algorithm. The thresholding step plays a crucial role in distinguishing corners from other image features by selecting pixels with high Harris response values.

To analyze the effect of thresholding, we conduct experiments with varying threshold values and examine how they influence the number and distribution of detected corners.

### Experiment 1: Very Low Threshold (Threshold = 0.1)

Observation:
When applying a very low threshold value of 0.1 to the Harris response, the algorithm detects a large number of corners, resulting in an over-detection scenario.
Justification:
The selection of a very low threshold value significantly increases the sensitivity of the corner detection algorithm, leading to the identification of a large number of corners. The low threshold allows even pixels with relatively small Harris response values to be considered as corner candidates. Consequently, the algorithm detects not only genuine corners but also noise and minor intensity variations as corners, resulting in over-detection.





**Experiment 2: High Threshold (Threshold = 0.8)**

Observation:
Applying a high threshold value of 0.8 to the Harris response results in a significantly reduced number of detected corners. In this experiment, the output detected corners number is observed to be 103. Upon visual inspection of the image, only a sparse set of corners is identified, primarily in regions with pronounced intensity variations.

Justification:
The use of a high threshold value imposes strict criteria for corner detection, requiring pixels to have a substantially high Harris response to be considered corners. Consequently, only the most salient corners with pronounced intensity variations are detected, while subtle or minor variations are overlooked.



**Experiment 3: Medium Threshold (Threshold = 0.4)**

Observation:
Applying a medium threshold value of 0.4 to the Harris response results in a moderate number of detected corners. Upon visual inspection of the image, a balanced distribution of corners is identified, covering both **prominent** features and subtle intensity variations.

Justification:
The use of a medium threshold value strikes a balance between sensitivity and robustness in corner detection. By setting the threshold at 0.4, the algorithm identifies corners with moderate to high Harris response values, effectively capturing both prominent features and subtle intensity variations.

The observed distribution of detected corners in this experiment reflects the effectiveness of the medium threshold value in balancing between over-detection and

under-detection. While not as strict as a high threshold, the medium threshold effectively filters out noise and minor variations, resulting in a reduced number of false positives compared to a low threshold.



## 2. Lambda Minus Approach

In the Lambda Minus corner detection method, corners are detected by finding points where the minimum eigenvalue is greater than a certain threshold.

A structure tensor, also known as a second-moment matrix, is a matrix representation that encapsulates information about the local gradient of an image at each pixel. For a 2D image, the structure tensor is a 2x2 matrix that is defined as follows:

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

where I_x and I_y are the image derivatives in the x and y directions, respectively.

The eigenvalues of this matrix represent the magnitude of the edges in the image in two orthogonal directions. In the context of corner detection, one of these directions will correspond to the edge along one direction of the corner, and the other will correspond to the edge along the other direction of the corner.

The minimum eigenvalue (Lambda Minus) of the structure tensor is the smaller of these two eigenvalues. It represents the edge strength in the direction of the corner where the edge is weaker. In other words, it gives us a measure of the smallest gradient at each point, which is useful for detecting corners: a corner will have a large gradient (and thus a large eigenvalue) in both directions, so the minimum eigenvalue will also be large at corners.

The eigenvalues of this matrix, often denoted as λ1 and λ2, can be calculated using the following formulas derived from the characteristic equation of the matrix:

$$\lambda_{1,2} = \frac{Tr(M) \pm \sqrt{Tr(M)^2 - 4Det(M)}}{2}$$

where Tr(M) is the trace of the matrix (the sum of the diagonal elements, I_x^2 + I_y^2 in this case), and Det(M) is the determinant of the matrix (I_x^2 * I_y^2 - (I_x * I_y)^2 in this case).

The term under the square root, Tr(M)^2 - 4Det(M), is called the discriminant. The discriminant tells us about the nature of the eigenvalues. If the discriminant is positive, the matrix has two distinct real eigenvalues. If it's zero, the matrix has one repeated real eigenvalue. If it's negative, the matrix has two complex conjugate eigenvalues.

In the context of corner detection, we're typically interested in the minimum eigenvalue (λ-), which corresponds to the weaker edge at each point in the image. This is calculated as:

$$\lambda_{-} = \frac{Tr(M) - \sqrt{Tr(M)^2 - 4Det(M)}}{2}$$

### 2.1. Function Signature

The function `lambda_minus_corner_detection` is a method of a class (not specified in the code snippet). It takes three optional arguments:

- `corner_color`: A list representing the RGB color to mark the corners. The default color is red ([0, 0, 255]).

- `blockSize`: An integer representing the size of the neighborhood considered for corner detection. It's used in the computation of the sum of products of derivatives over a local window.
- `ksize`: An integer representing the size of the Sobel kernel used to compute image derivatives. The Sobel operator is used in the function to compute the derivatives Ix and Iy.

### 2.2. Function Operation

The function operates by performing the following steps:

1. Image Preparation: The function first creates a copy of the current image to avoid modifying the original image. If the image is not grayscale (has 3 dimensions: height, width, and channels), it converts the image to grayscale.
2. Derivative Computation: The function computes the derivatives `Ix` and `Iy` using the Sobel operator.
3. Product of Derivatives: The function then computes the products of these derivatives (`Ixx`, `Iyy`, and `Ixy`).
4. Sum of Products: The function computes the sum of products of derivatives over a local window (`Sxx`, `Syy`, and `Sxy`).
5. Determinant and Trace: The function computes the determinant and trace of the matrix.
6. Minimum Eigenvalue (Lambda Minus): The function computes the minimum eigenvalue (Lambda Minus) of the 2x2 structure tensor.
7. Corner Detection: The function finds the indices where Lambda Minus is greater than a threshold. These indices represent the corners in the image.
8. Corner Marking: If the image is an RGB image, the corners are marked with the user-specified color. If the image is grayscale, the corners are marked with the mean intensity of the user-specified color.
9. Computation Time: The function measures the computation time for calculating Lambda Minus and prints it out.

### 2.3. Function Output

The function does not return any value (`None`). Instead, it modifies the input image by marking the detected corners and displays the modified image. It also prints the computation time for calculating Lambda Minus.

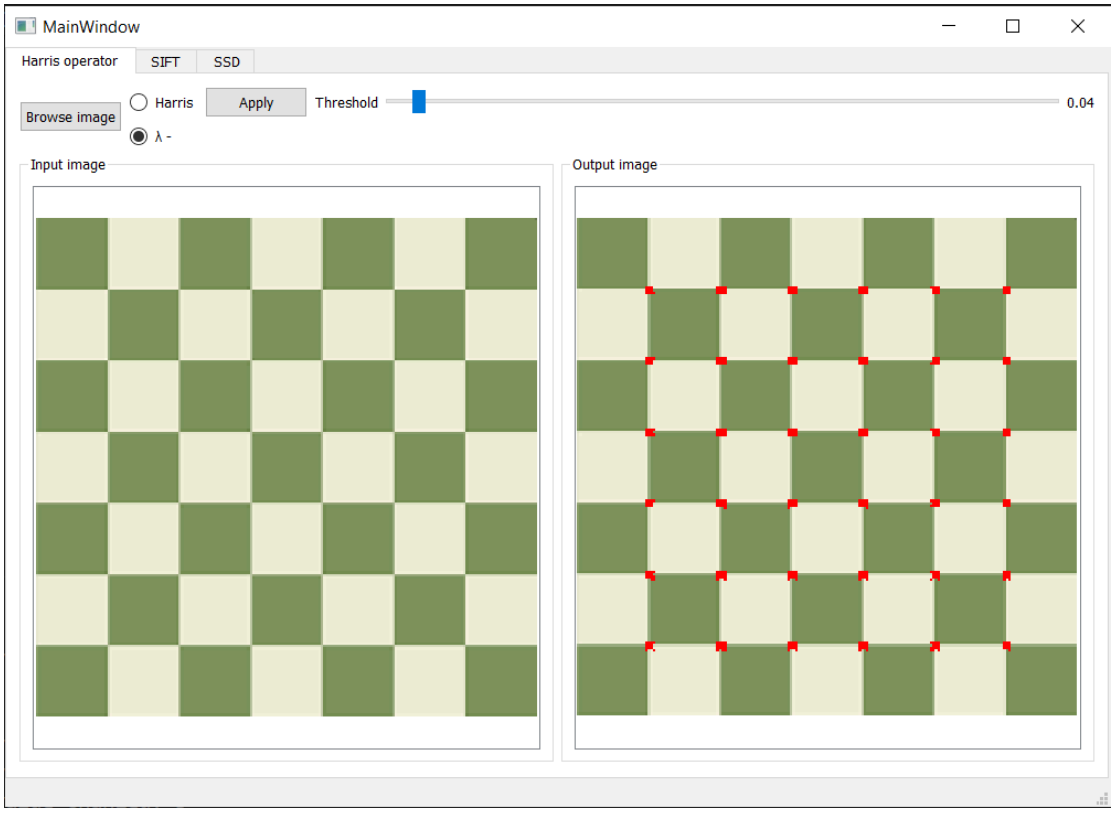
This function is part of a larger image processing pipeline, specifically designed for corner detection using the Lambda Minus method. It provides a detailed and efficient

way to detect and mark corners in an image, which can be useful in various computer vision tasks such as feature extraction, image alignment, and object recognition.

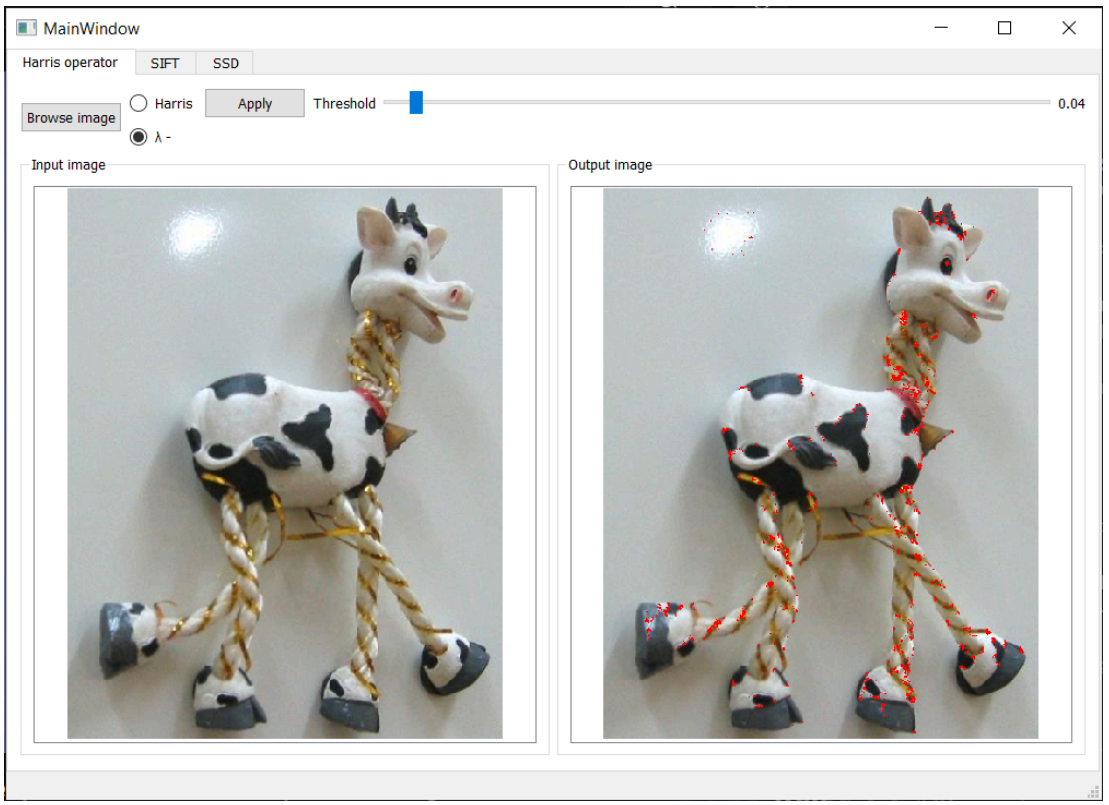
2.4. Findings with varying the window function, blockSize and ksize

1) Using a uniform window function

- blockSize = 2, ksize = 3

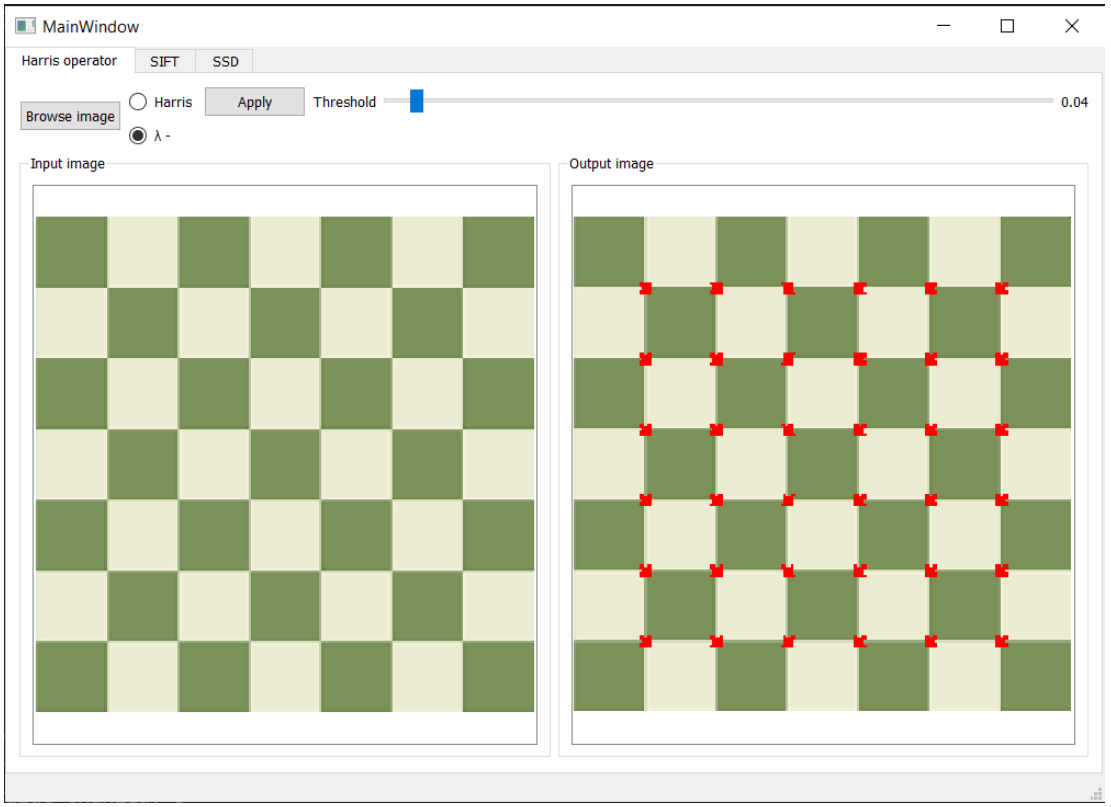



Computation time: 0.004015207290649414



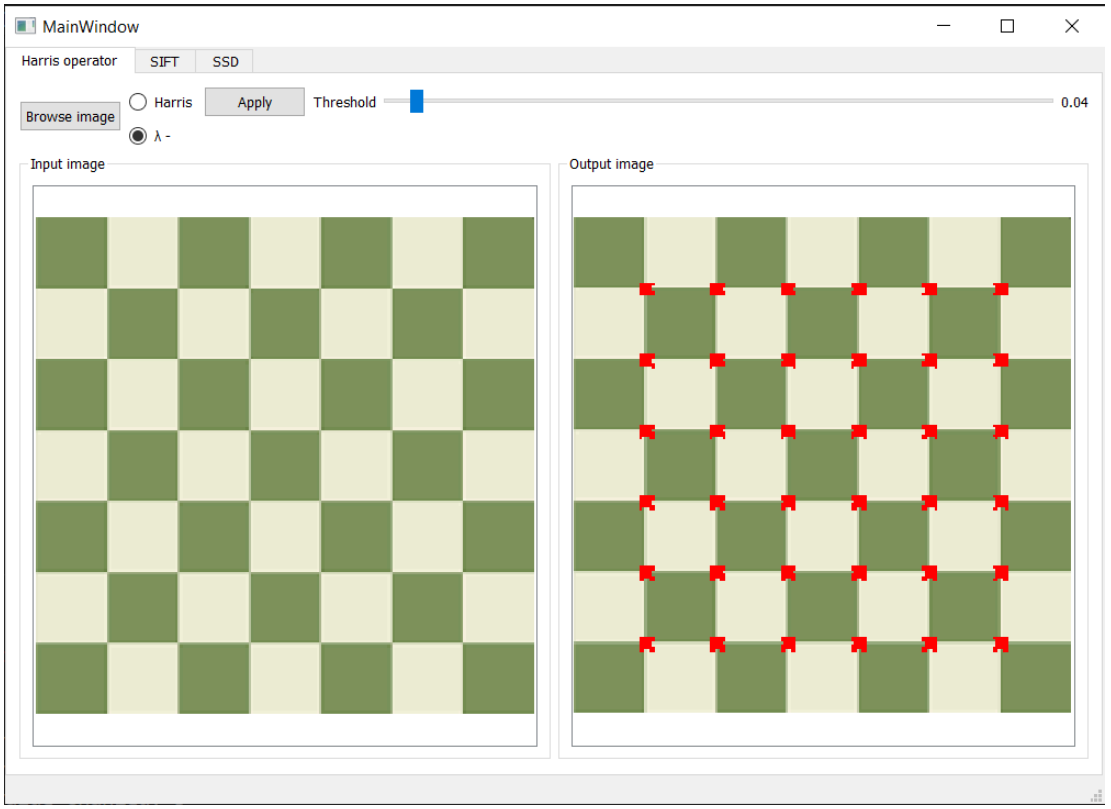

Computation time: 0.03145337104797363

- blockSize = 2, ksize = 5



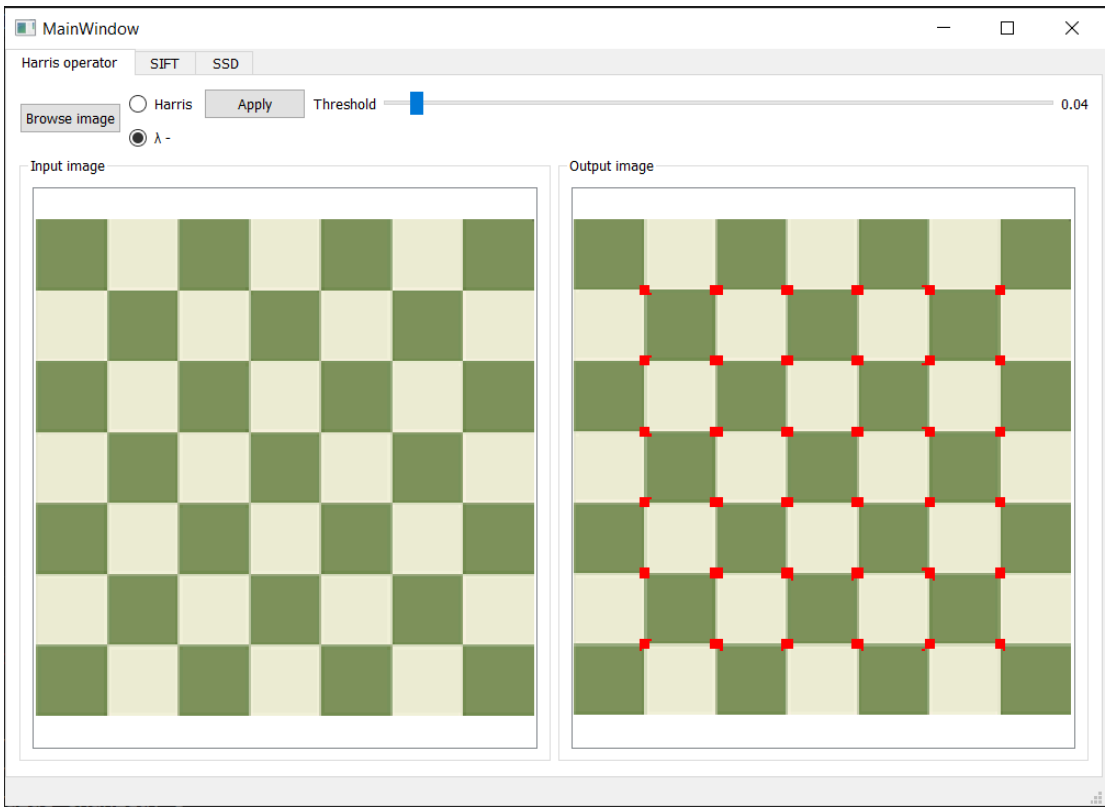
Computation time: 0.004987001419067383

- blockSize = 2, ksize = 7



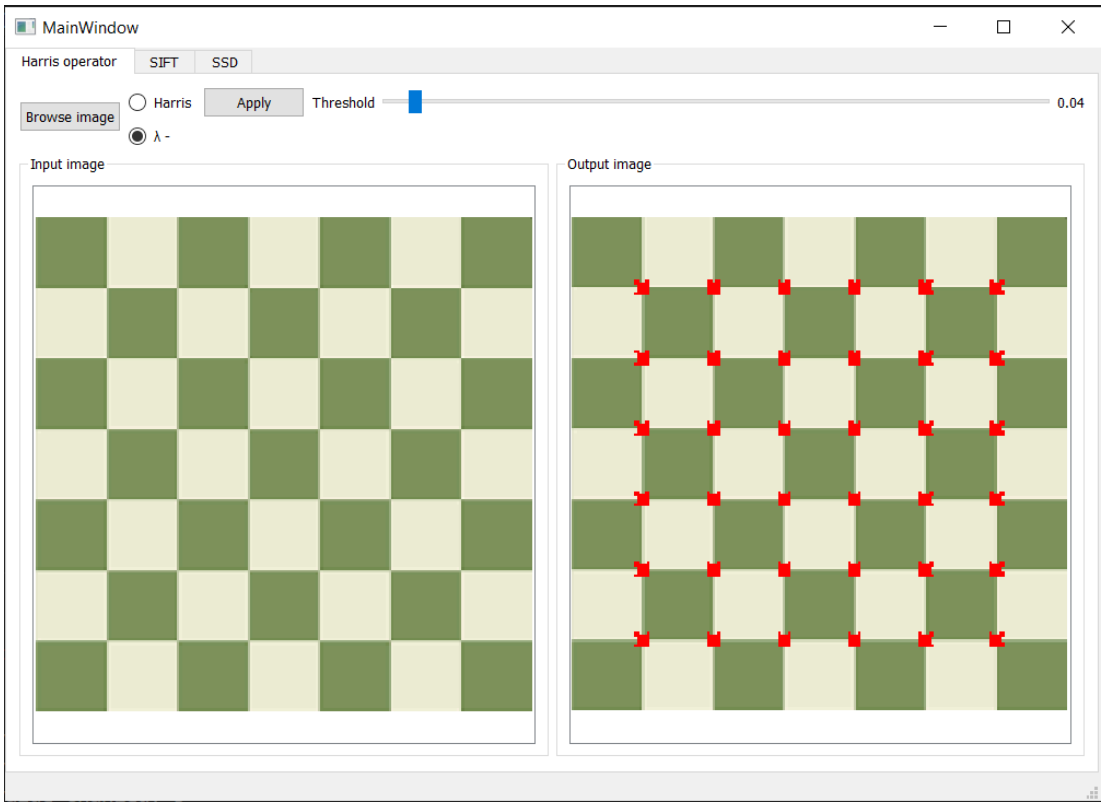

Computation time: 0.00398564338684082

- blockSize = 3, ksize = 3



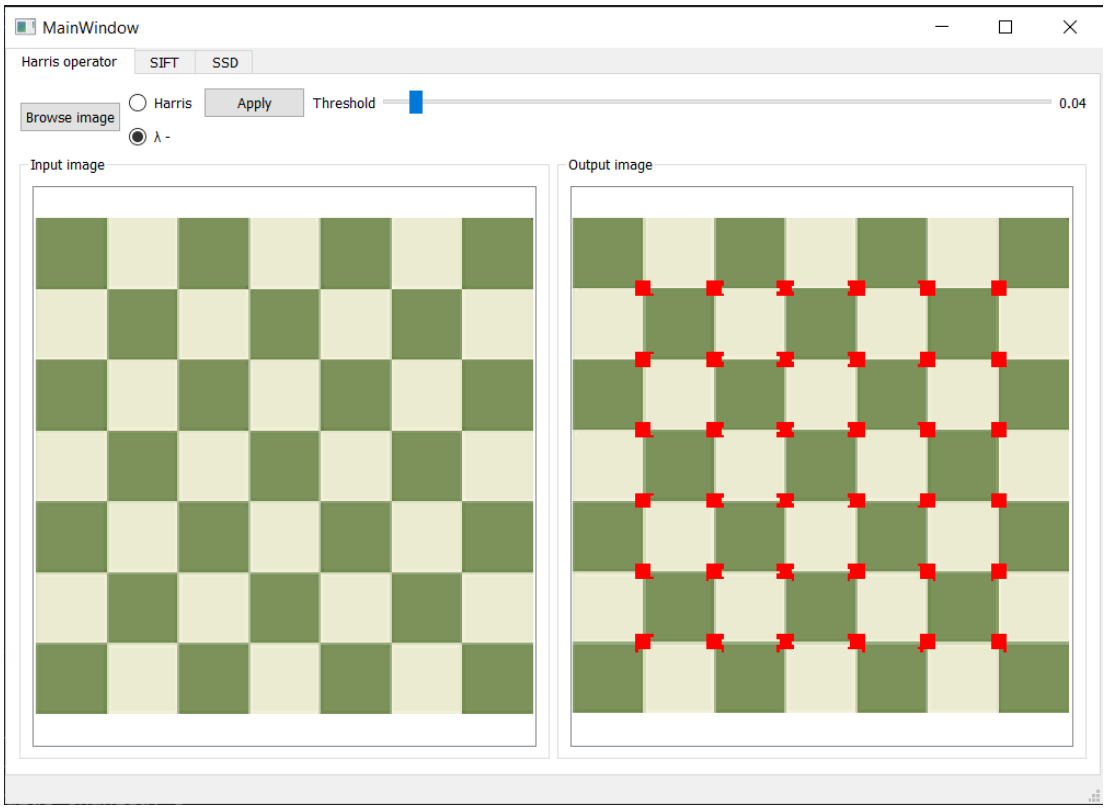

Computation time: 0.005076408386230469

- blockSize = 3, ksize = 5



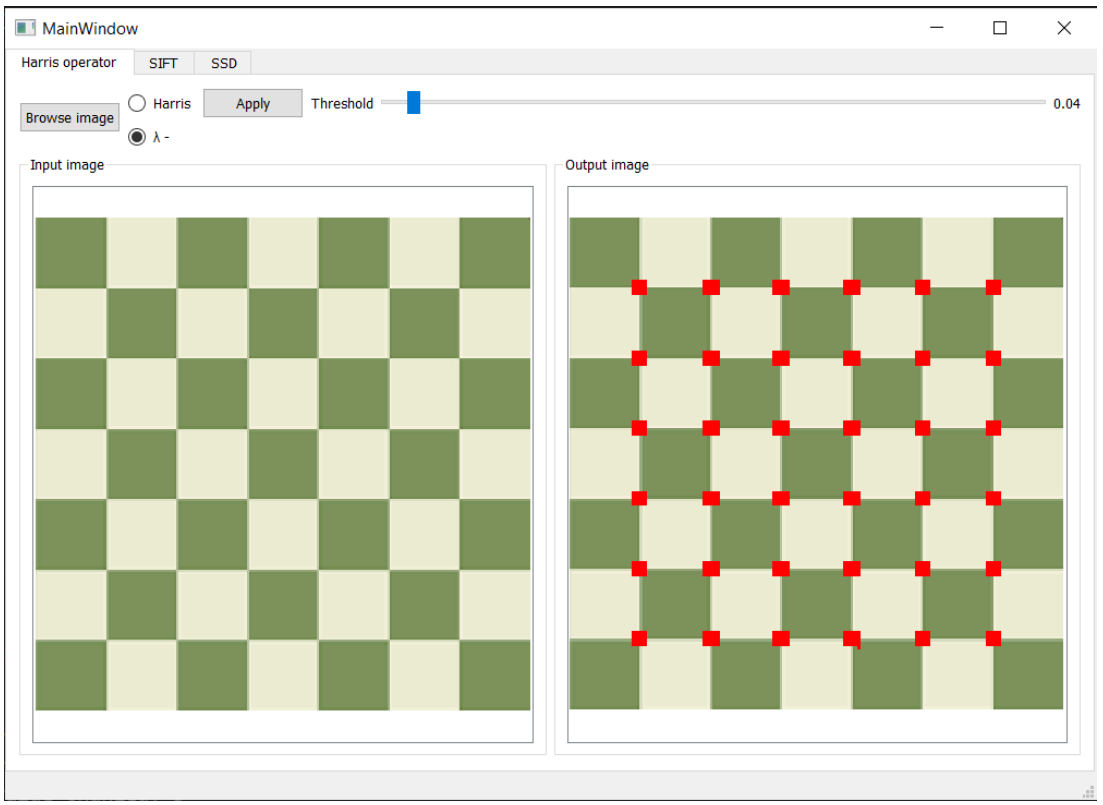
Computation time: 0.004986763000488281

- blockSize = 3, ksize = 7



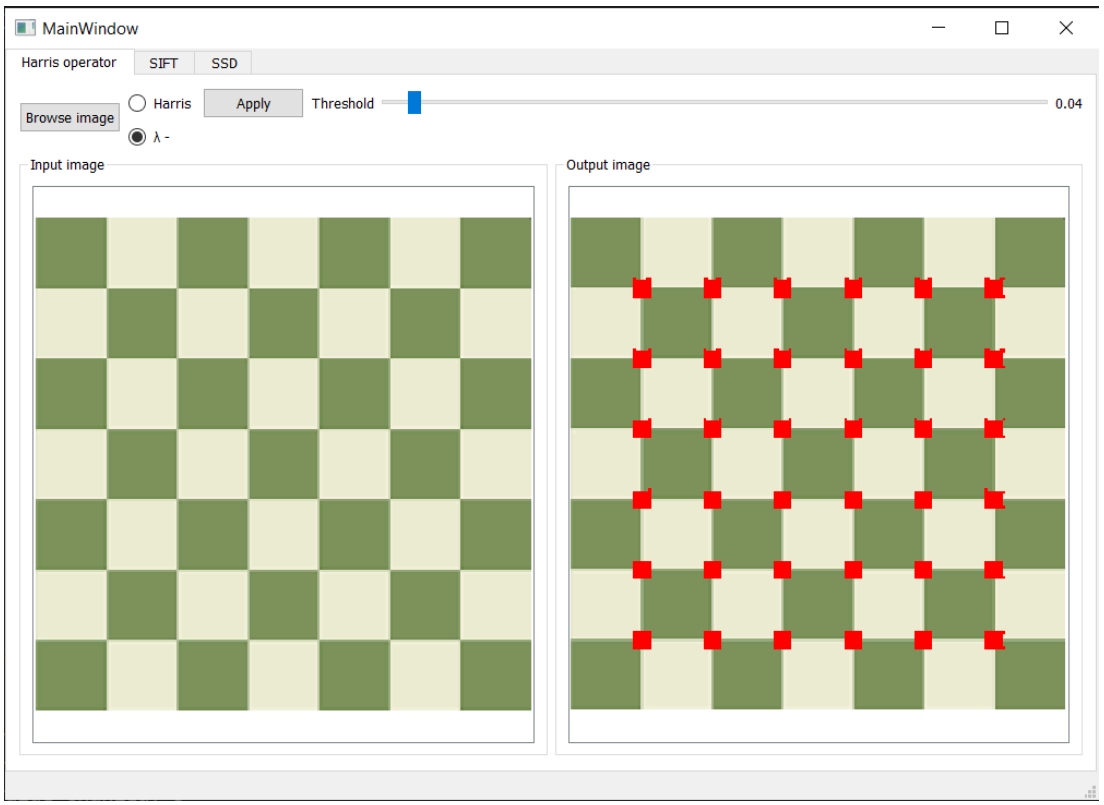

Computation time: 0.0030677318572998047

- blockSize = 5, ksize = 3



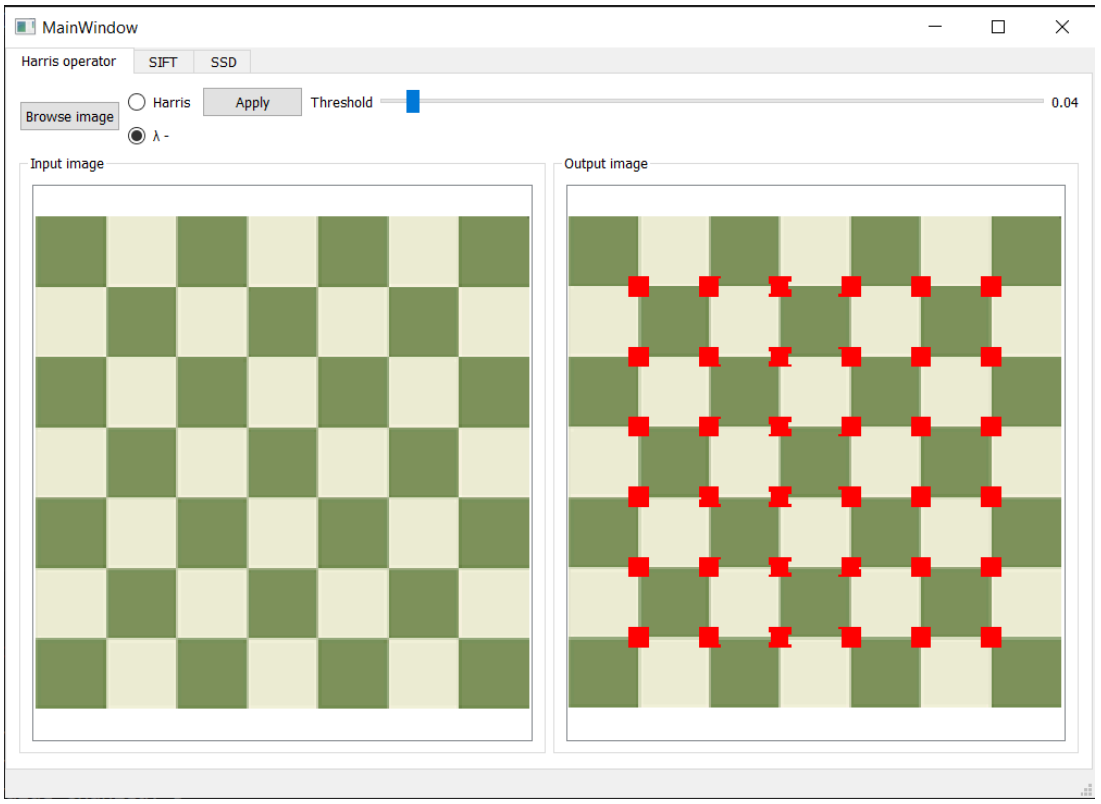

Computation time: 0.0040628910064697266

- blockSize = 5, ksize = 5



Computation time: 0.0049877166748046875

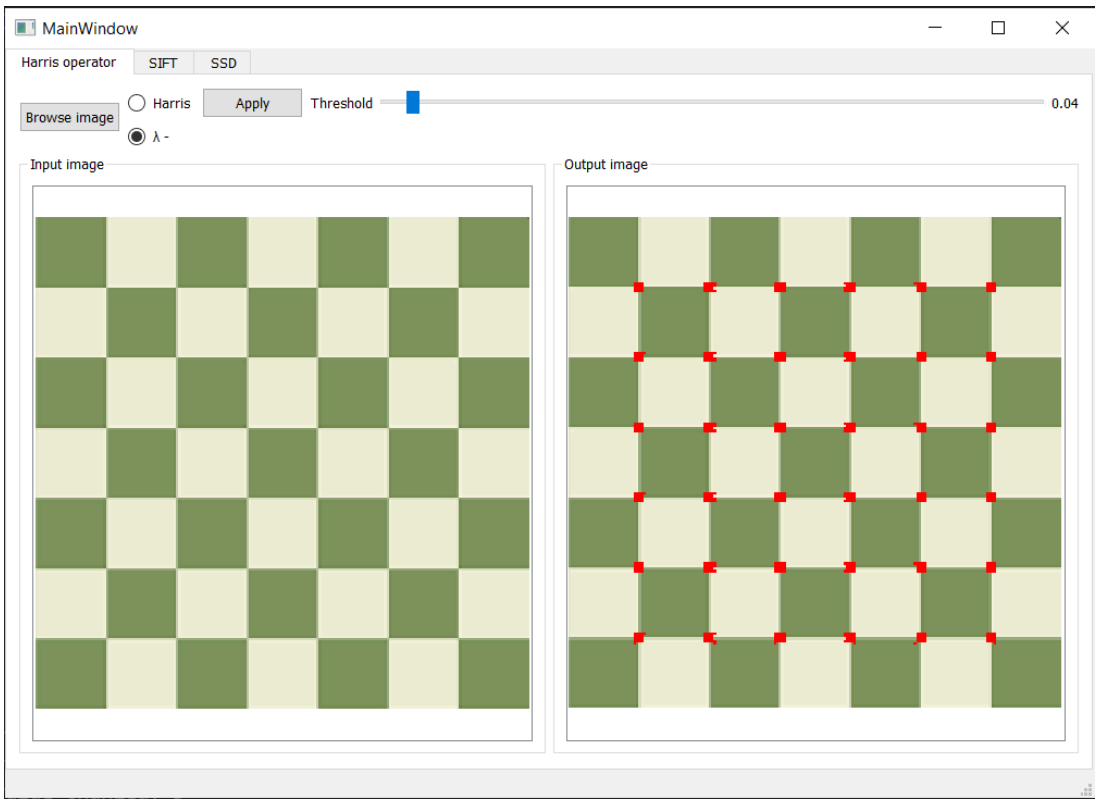
- blockSize = 5, ksize = 7



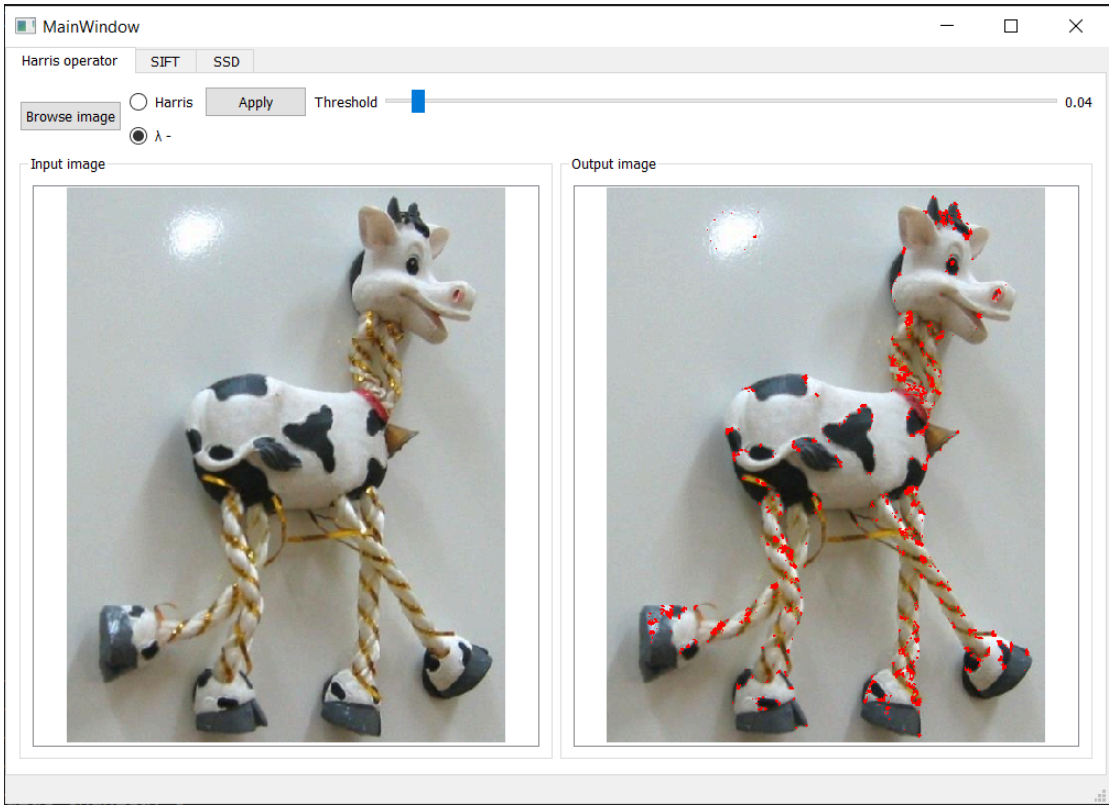

Computation time: 0.004987001419067383

2) Using a Gaussian window function

- blockSize = 3, ksize = 3




Computation time: 0.00554656982421875




Computation time: 0.029396772384643555

## 2.5. Conclusion

If you increase the blockSize, the window of the neighborhood considered for corner detection becomes larger. This means that the algorithm becomes more robust to noise, as it averages over a larger area. However, it may also result in less precise corner detection, as it could average out smaller corners and make the algorithm less sensitive to small changes.

On the other hand, if you decrease the blockSize, the window of the neighborhood considered for corner detection becomes smaller. This could make the algorithm more sensitive to small changes and potentially detect smaller corners. However, it may also make the algorithm more sensitive to noise, as it averages over a smaller area.

If you increase the ksize, the Sobel operator becomes more sensitive to larger, more prominent edges and less sensitive to smaller, finer details. This could result in fewer corners being detected, but those detected may be more robust and significant.

Conversely, if you decrease the ksize, the Sobel operator becomes more sensitive to smaller, finer details and less sensitive to larger, more prominent edges. This could result in more corners being detected, but these might include less significant corners or noise.

A uniform window gives equal weight to all pixels in the neighborhood. This can sometimes lead to abrupt changes in the boundaries of the window, which might not always be desirable. However, it can potentially provide more precise corner detection if the corners are well-defined and the image has low noise.

On the other hand, a Gaussian window gives more weight to pixels near the center of the window and less weight to pixels further away. This results in a smoother transition at the boundaries of the window, which can sometimes lead to better results, especially in the presence of noise. However, it might potentially be less precise if the corners are small or subtle, as the smoothing effect of the Gaussian window might average out these details.

In practice, the choice of window type (uniform or Gaussian), as well as the selection of parameters like `blockSize` and `ksize`, often involves a trade-off between noise robustness and precision. Larger `blockSize` and `ksize` values can make the algorithm more robust to noise but potentially less precise, while smaller values can increase precision but make the algorithm more sensitive to noise. It's also worth noting that the optimal combination of window type, `blockSize`, and `ksize` can vary depending on the specific task and the nature of the images we're working with.

To sum up, the Harris operator computes the corner response as the determinant of the structure tensor minus a constant times the square of the trace of the structure tensor. It then finds corners as the local maxima of this response. The Lambda Minus approach computes the corner response as the smaller eigenvalue (Lambda Minus) of the structure tensor. It then finds corners as the points where this response is greater than a threshold.

In other words, while the Harris operator looks for local maxima in the corner response, the Lambda Minus approach looks for points where the corner response exceeds a certain threshold. This can make the Lambda Minus approach more sensitive to corners, but it might also make it more sensitive to noise.

## II. Feature Descriptors Generation

### 1. Introduction

The Scale-Invariant Feature Transform (SIFT) algorithm is a powerful method for detecting and describing local features in images. It is widely used in computer vision tasks such as object recognition, image stitching, and 3D reconstruction. SIFT features are invariant to image scale, and rotation, and partially invariant to changes in illumination and viewpoint.

### 2. Key Components

#### a. Base Image Generation

The algorithm starts by creating a base image from the input image. This base image is generated by convolving the input image with Gaussian filters of different scales. The purpose of this step is to create an image pyramid that represents the image at multiple scales.

#### b. Scale Space Representation

The scale space representation is created by applying a series of Gaussian blurs to the base image. This generates a pyramid of blurred images at different scales, where each level in the pyramid represents a different degree of image smoothing. The scale space representation allows the algorithm to detect features at different scales.

#### c. Difference of Gaussians (DoG)

The DoG images are created by taking the difference between adjacent levels in the scale space representation. These DoG images highlight areas of high contrast and are used to identify potential keypoint locations.

**d. Keypoint Detection**

Keypoints are identified as local extrema in the DoG images. A keypoint represents a distinctive feature in the image, such as a corner or a blob. The algorithm uses a method called "find_scale_space_extrema" to locate these keypoints.

**e. Keypoint Localization**

Once keypoints are detected, the algorithm performs localization to improve their accuracy. This involves refining the keypoint location based on the gradient information around the keypoint. The function "localizeExtremumViaQuadraticFit" is typically used for this purpose, but it is not explicitly defined in the provided code.

**f. Orientation Assignment**

Each keypoint is assigned an orientation based on the local gradient information around the keypoint. This step ensures that the descriptors are invariant to image rotation. The function "compute_keypoints_with_orientations" performs orientation assignment for each keypoint.

**g. Descriptor Generation**

Descriptors are generated for each keypoint to capture the local image information surrounding the keypoint. These descriptors are robust to changes in scale, rotation, and illumination. The "generate_descriptors" function computes descriptors based on gradient information within a descriptor window.

**3. Algorithm Workflow**

1. Image Preprocessing: Convert the input image to a float32 format and generate a base image.
2. Scale Space Construction: Generate a scale space representation by applying Gaussian blurs to the base image.
3. DoG Image Computation: Compute the difference of Gaussians (DoG) images from the scale-space representation.
4. Keypoint Detection: Identify local extrema in the DoG images as keypoints.
5. Keypoint Localization: Refine the keypoint locations for accuracy (not explicitly defined in the provided code).
6. Orientation Assignment: Assign orientations to keypoints based on local gradient information.
7. Descriptor Generation: Generate descriptors for each keypoint to describe the local image region.

**4. Code Functionality**

- sift(image, sigma, num_intervals, assumed_blur, image_border_width): Main function that orchestrates the SIFT algorithm. It takes an input image and returns keypoints and descriptors.

- generate_base_image(image, sigma, assumed_blur): Function to generate a base image for the SIFT algorithm.
- generate_gaussian_kernels(sigma, num_intervals): Function to generate Gaussian kernels for image blurring.
- generate_gaussian_images(base_image, num_octaves, gaussian_kernels): Function to generate Gaussian-blurred images for scale-space representation.
- generate_DoG_images(gaussian_images): Function to compute the difference of Gaussians (DoG) images.
- find_scale_space_extrema(gaussian_images, dog_images, num_intervals, sigma, image_border_width): Function to detect keypoints as local extrema in the DoG images.
- compute_keypoints_with_orientations(keypoint, octave_index, gaussian_image, radius_factor, num_bins, peak_ratio, scale_factor): Function to compute orientations for keypoints based on gradient information.
- generate_descriptors(keypoints, gaussian_images, window_width, num_bins, scale_multiplier, descriptor_max_value): Function to generate descriptors for keypoints.

## 5. The Parameters Chosen

### 1. Sigma (sigma)

- Definition: Sigma represents the standard deviation of the Gaussian kernel used for blurring the input image.
- Purpose: Sigma controls the scale of features that the algorithm detects. Higher sigma values result in smoother images, which are more robust to noise but may miss fine details. Lower sigma values capture finer details but are more sensitive to noise.
- Chosen Value: The default value of 1.6 is commonly used in SIFT implementations and provides a good balance between noise robustness and feature sensitivity.

### 2. Number of Intervals (num_intervals)

- Definition: The number of intervals in each octave of the scale space representation.
- Purpose: Increasing the number of intervals increases the number of DoG images generated, allowing for more precise localization of keypoints.
- Chosen Value: The default value of 3 is often used as it provides a sufficient number of intervals to capture scale variations in the image without significantly increasing computational cost.

### 3. Assumed Blur (assumed_blur)

- Definition: The assumed blur of the input image.

- Purpose: Specify the amount of pre-blurring applied to the input image before constructing the scale space pyramid. This helps in reducing noise and simplifying subsequent computations.
- Chosen Value: The default value of 0.5 is commonly used and represents a moderate level of blur assumed to be present in the input image.

**4. Image Border Width (image_border_width)**

- Definition: The width of the border around the image where keypoints are not detected.
- Purpose: Helps to avoid edge effects and keypoints being located close to image borders, where gradient computation may be unreliable.
- Chosen Value: The default value of 5 ensures that keypoints are not detected near the image borders, reducing potential artifacts.

**5. Window Width (window_width)**

- Definition: The width of the descriptor window used for computing descriptors around each keypoint.
- Purpose: Determines the size of the local region around each keypoint used for computing feature descriptors. Larger windows capture more spatial information but may be less distinctive.
- Chosen Value: The default value of 4 is commonly used and provides a good balance between descriptor size and computational efficiency.

**6. Number of Bins (num_bins)**

- Definition: The number of bins in the orientation histogram used for computing keypoint orientations and descriptors.
- Purpose: Determines the granularity of orientation information captured in the descriptors. More bins result in finer orientation resolution but may increase descriptor dimensionality.
- Chosen Value: The default value of 8 is often used as it provides sufficient orientation resolution while keeping descriptor dimensionality manageable.

**7. Scale Multiplier (scale_multiplier)**

- Definition: A multiplier used to scale the descriptor size relative to the keypoint size.
- Purpose: Allows the descriptors to capture spatial information at a scale larger than the keypoint size, improving robustness to scale variations.
- Chosen Value: The default value of 3 is commonly used and provides a reasonable scale factor for descriptor size relative to the keypoint size.

**8. Descriptor Max Value (descriptor_max_value)**

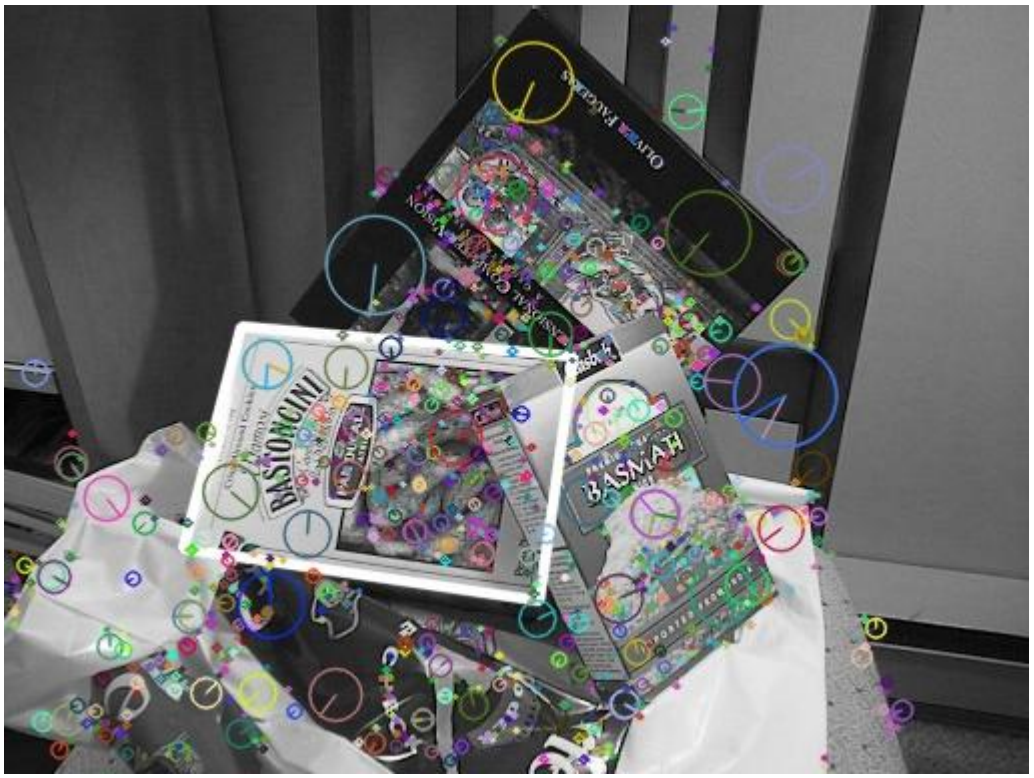
- Definition: The maximum value allowed for descriptor elements.
- Purpose: Helps to normalize descriptor values and prevent large variations in descriptor magnitudes.
- Chosen Value: The default value of 0.2 is often used to limit the range of descriptor values and improve descriptor robustness.

**6. Conclusion**

The SIFT algorithm is a robust method for detecting and describing local features in images. It provides a reliable way to match keypoints across different images, making it invaluable for various computer vision applications.

- **Result of SIFT algorithm:**

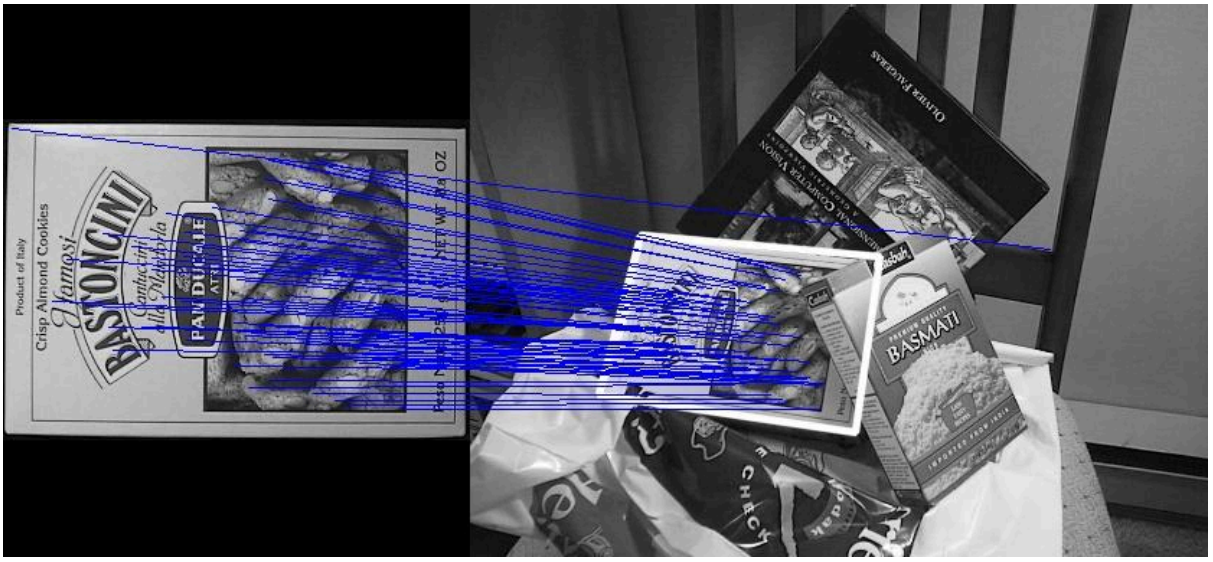
**query image**




**train image**


**Observation: what we see in the above images are are the Keypoints with their orientation.**
**computation_time: 129.2920205593109 sec**

- **Regardless the methods will be mentioned after like SSD and NCC for matching I tried another technique to ensure that the SIFT algorithm works well, which is the <u>KNN technique.</u>**
  **After computing SIFT keypoints and descriptors for both images, potential matches were identified using a nearest-neighbor approach and filtered using Lowe's ratio test. A homography matrix was then estimated to align the query image with the train image, allowing for visualization of matched keypoints and their spatial correspondence.**

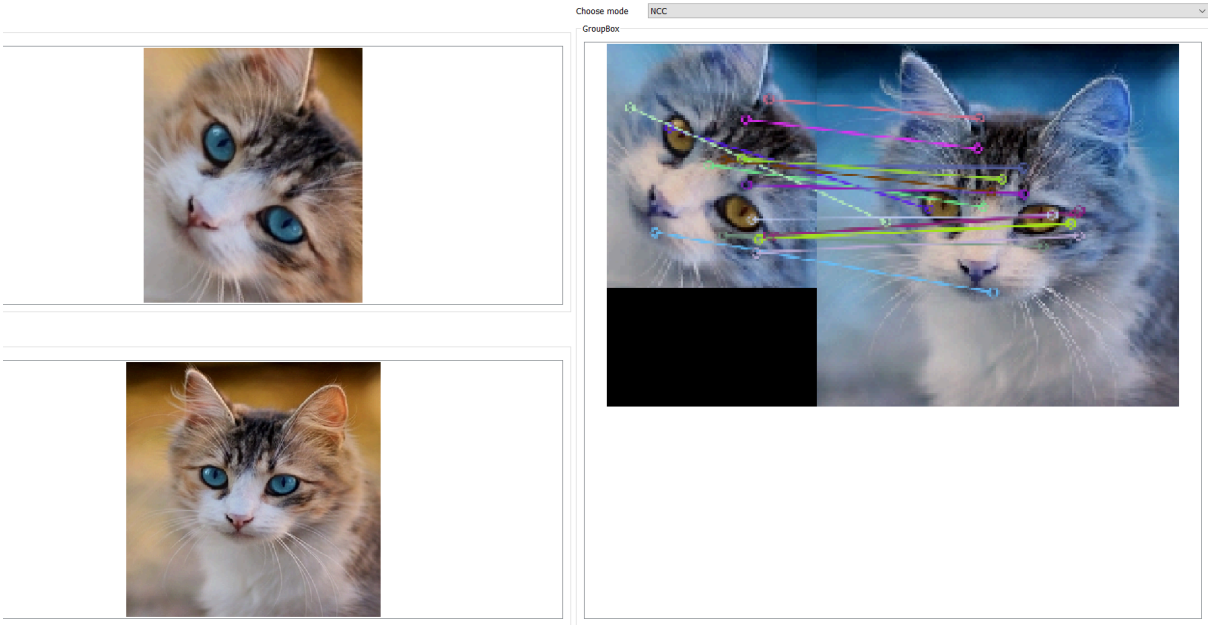



**Observation: The matching process demonstrated the ability of the SIFT algorithm to robustly identify and match distinctive features across images.**
**computation_time: 75.58000254631042 sec**

## III. Feature Matching

1. Normalized Cross Correlations
   NCC is a similarity measure, it measures the similarity between the descriptors of keypoints in two images.
   It computes the correlation coefficient between descriptor sets of keypoints in two images, normalizing them to zero mean and unit standard deviation.
   Higher NCC values indicate stronger similarity between keypoints, hence considered as good matches.

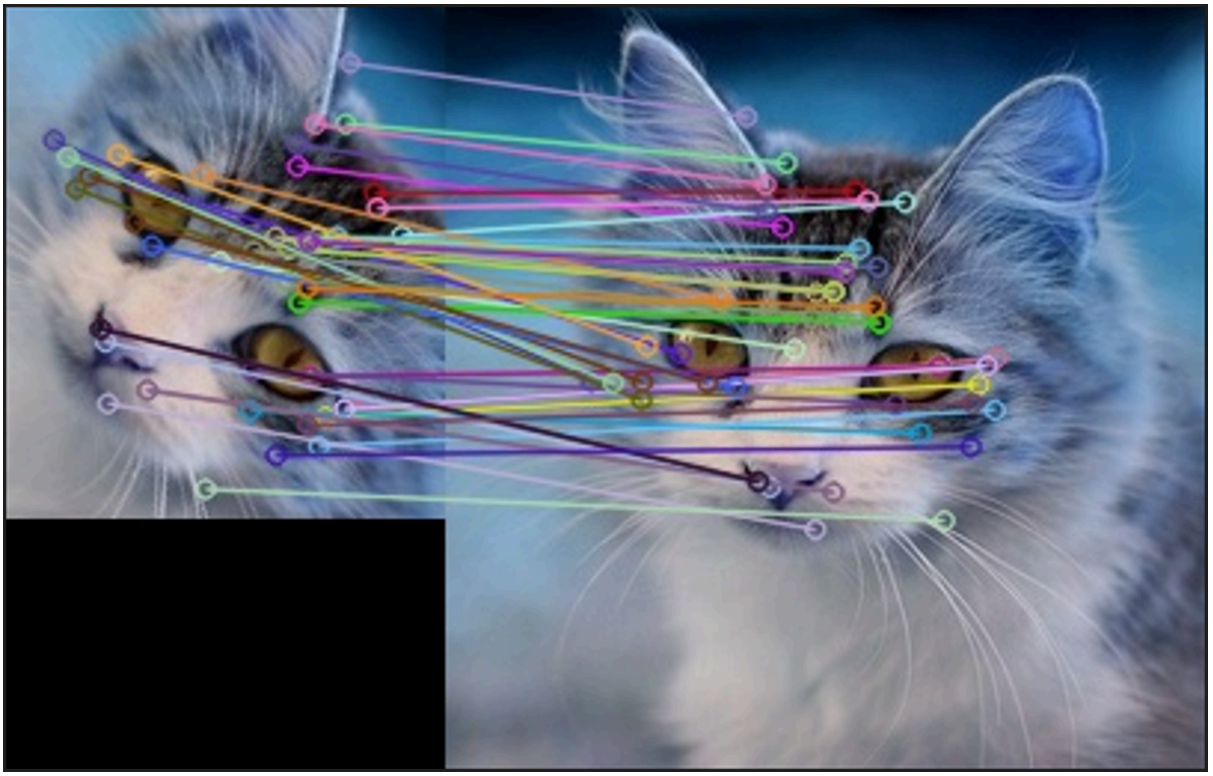
   **Results:**




**Matching computation time: 4.801262855529785 seconds**

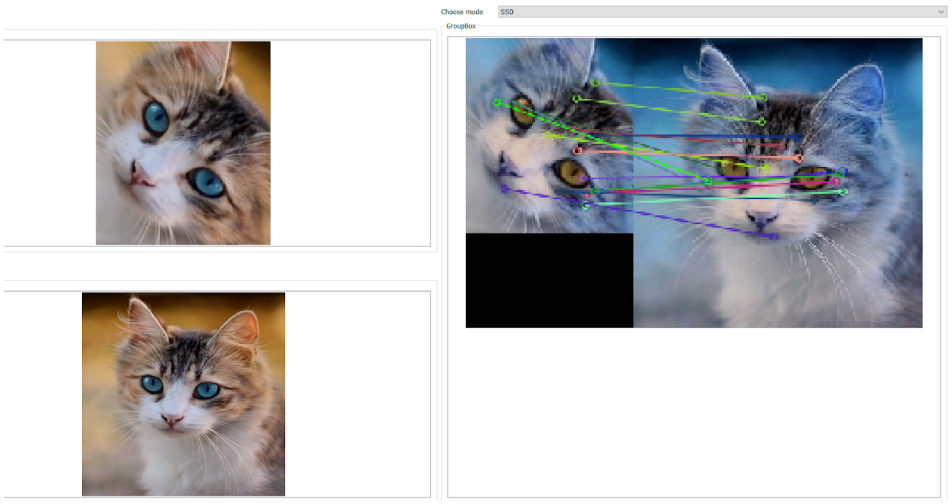
The result depends on the count of matched (number of matches) features or key points between two images in a feature-matching process.

Increasing number of matches number of matches :

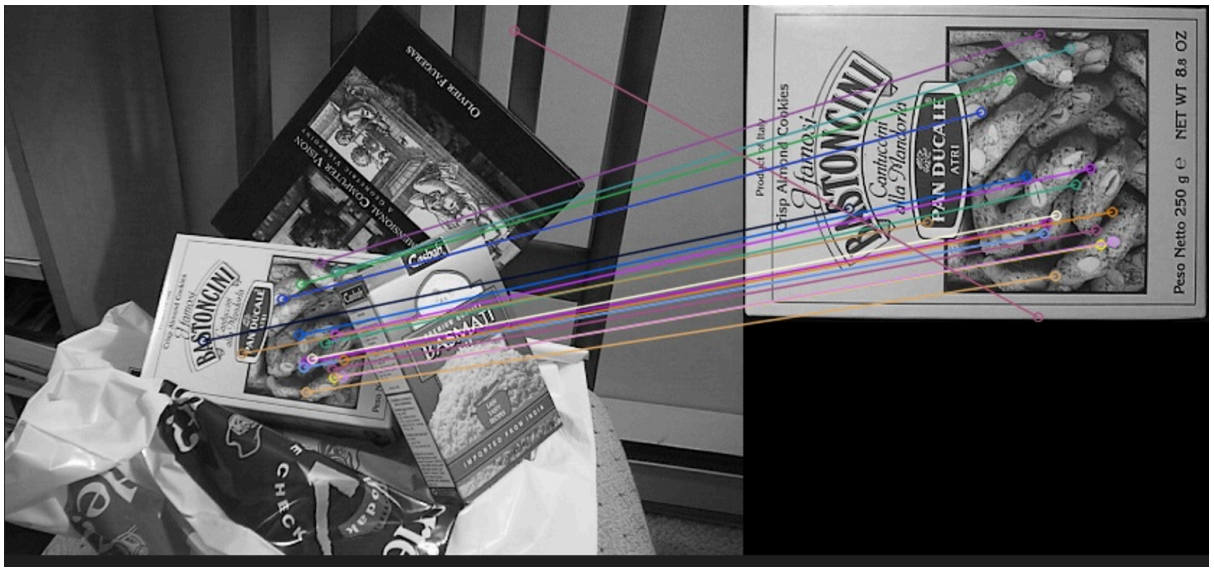


**Matching computation time: 7.177109956741333 seconds**

2. Sum of Square Differences (SSD) measures the dissimilarity between descriptor sets of keypoints in two images by computing the sum of squared differences. Lower SSD values indicate greater similarity between keypoints, hence considered as good matches."
   **Results:**

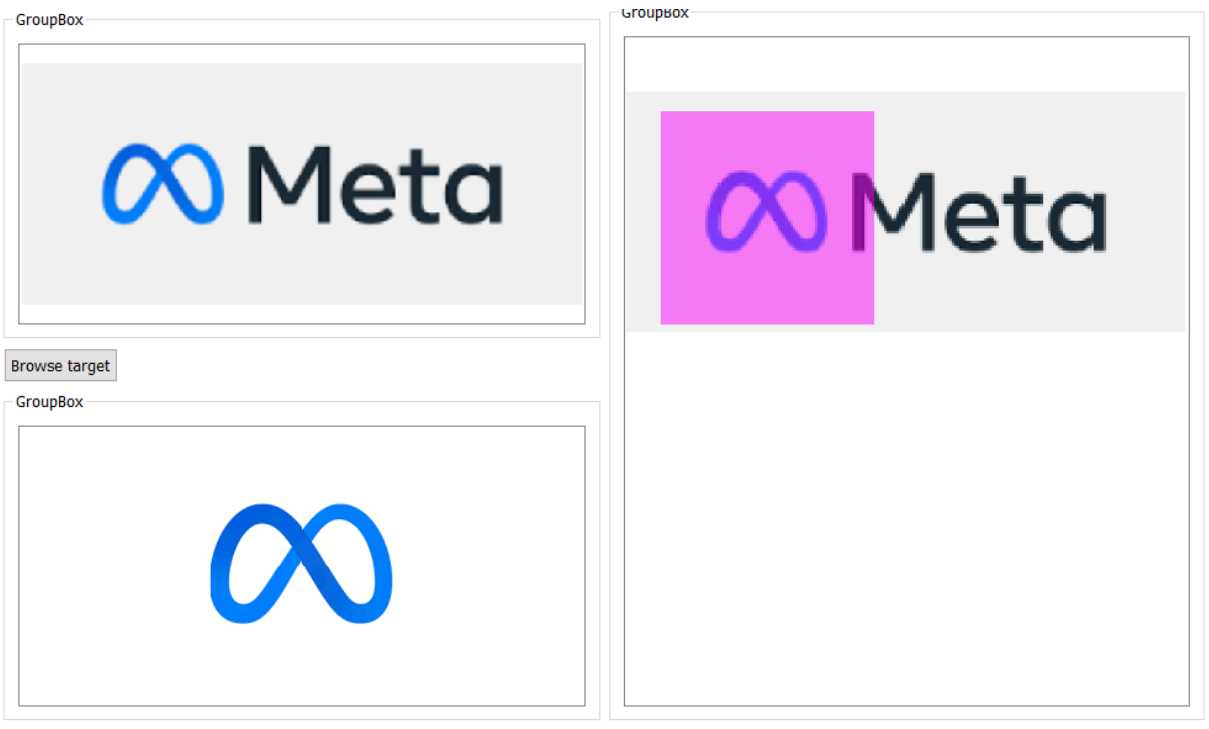



**Matching computation time: 0.3231074810028076 seconds**




**Matching computation time: 3.611819267272949 seconds**

3. NCC for detecting objects :
   NCC helps in detecting the presence and location of a specific pattern (the target image) within a larger image (the whole image) by measuring the similarity between them.
   Steps:
   1. Comparison: It compares each possible position in the whole image with the target image to see how similar they are.
   2. Normalization: It normalizes the correlation values to ensure that they are not affected by the intensity or contrast differences between the images.
   3. Scoring: It produces a correlation map where each point represents the correlation coefficient between the target image and the corresponding region in the whole image.
   4. Detection: The position with the highest correlation coefficient indicates the most likely location of the target image within the whole image.




**Matching computation time: 50.111819267272949 seconds**