2024

FEATURE DETECTION AND SIFT

TEAM 14

- AHMED MOHAMED ALI
- OSAMA MOHAMED ALI
- MUHANNAD ABDALLAH
- ALI SHERIF

SUBMITTED TO

- DR. AHMED M. BADAWI
- ENG. LAILA ABBAS
- ENG. OMAR DAWAH

TABLE OF CONTENTS

LAMBDA MINUS

Introduction to Lambda Minus

Eigenvalues and Eigenvectors provides a powerful framework for understanding local image structures. Given a structure tensor *M* computed from image sobel gradients, its eigenvalues represent the amount of variance or intensity change along the principal axes defined by the eigenvectors. Intuitively, eigenvalues quantify the significance of structural information within an image region, with larger eigenvalues indicating more pronounced intensity changes.

The Lambda Minus corner detection algorithm exploits the properties of eigenvalues to identify corners or regions of interest within an image. Specifically, Lambda Minus focuses on the minimum eigenvalue (λmin) derived from the structure tensor, which reflects the least amount of intensity variation in a local neighborhood. In areas with sharp intensity transitions, such as corners, the smallest eigenvalue tends to be substantially larger compared to regions with gradual intensity changes.

Algorithm Overview

It lies in the principles of differential geometry and local image structure analysis. By computing gradients using operators like the Sobel filter, the algorithm captures directional changes in image intensity. These gradients are then used to construct the structure tensor M, which characterizes the local spatial distribution of intensity gradients.
Eigenvalue analysis of M provides insight into the dominant directions of intensity variation within each image region. The minimum eigenvalue λmin serves as a sensitive indicator of corner presence due to its responsiveness to abrupt changes in image structure. In essence, corners exhibit a unique pattern of intensity variation across multiple directions, resulting in relatively large λmin values.
By setting an appropriate threshold based on a percentage of the maximum λmin, the algorithm distinguishes corners from other image features. Non-maximum suppression further refines the corner detection process by selecting local maxima in λmin within a defined window, ensuring that only the most salient corners are retained.

Steps

1. Compute image gradients using the Sobel operator to obtain Ix and Iy.
2. Compute elements of the structure tensor M using *Ix*, *Iy*, and their products.
3. Calculate eigenvalues and eigenvectors of *M* to characterize local image structures.
4. Extract λmin as the minimum eigenvalue representing corner strength.
5. Apply thresholding to select corners based on a percentage of the maximum λmin.
6. Perform non-maximum suppression to identify local maxima as corner candidates.

LAMBDA MINUS

```python
def lambda_minus_corner_detection(img, window_size=5, th_percentage=0.01):
    # Convert image to grayscale
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Compute gradients using Sobel operator
    Ix = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
    Iy = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)

    # Compute elements of the structure tensor M
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)
    Ixy = np.multiply(Ix, Iy)

    # Compute eigenvalues and eigenvectors
    M = np.stack((Ix2, Ixy, Ixy, Iy2), axis=-1).reshape((-1, 2, 2))
    eigenvals, _ = np.linalg.eig(M)

    # Reshape eigenvalues to image shape
    eigenvals = eigenvals.reshape(img.shape[0], img.shape[1], 2)
    # Compute lambda minus
    lambda_minus = np.minimum(eigenvals[:, :, 0], eigenvals[:, :, 1])

    # Threshold for selecting corners
    threshold = th_percentage * np.max(lambda_minus)

    # Find local maxima using non-maximum suppression
    corners = []
    # In regions with corners or sharp changes in intensity
    # at least one of the eigenvalues will be large, resulting in a large lambda minus.
    for y in range(window_size, lambda_minus.shape[0] - window_size):
        for x in range(window_size, lambda_minus.shape[1] - window_size):
            if lambda_minus[y, x] > threshold:
                # selects a square window centered around the pixel (x,y)
                window = lambda_minus[y - window_size:y + window_size + 1,
                         x - window_size:x + window_size + 1]
                # first index refers to the row number (y-coordinate) and the second index refers to
the column number (x-coordinate).
                if lambda_minus[y, x] == np.max(window):
                    corners.append((x, y))

    return corners
```

The code defines a function named `lambda_minus_corner_detection`, which implements the Lambda-Minus corner detection algorithm . It accepts three parameters: `img`, representing the input image as a NumPy array, `window_size` (optional), specifying the size of the window used for non-maximum suppression defined by user using slider in the UI , and `th_percentage` (optional), indicating the percentage of the maximum eigenvalue used for thresholding defined by a slider in UI. The function returns a list of tuples representing the coordinates of the detected corners in the image.
Within the function, the input image is first converted to grayscale using OpenCV's `cvtColor` function with the `COLOR_BGR2GRAY` conversion code.
Gradients of the grayscale image are then computed using the Sobel operator to obtain the derivative in the x and y directions (stored in variables `Ix` and `Iy`, respectively).
The elements of the structure tensor M are computed using the gradients I_x and I_y , which involves calculating I_x^2 , I_y^2 , and (I_x . I_y) .
Subsequently, the eigenvalues and eigenvectors of the structure tensor M are computed using Numpy library using the `np.linalg.eig` function, representing the local structure of the image.
The eigenvalues are reshaped to match the shape of the input image, and lambda minus “lambda_minus” is computed as the minimum eigenvalue at each pixel location.
A threshold is then computed as a percentage (`th_percentage`) of the maximum lambda minus value.
Non-maximum suppression is performed to find local maxima in the lambda minus image. This process involves iterating over each pixel in the lambda minus image and comparing it with its neighboring pixels within a window. If a pixel's lambda minus value is greater than the threshold and is the maximum within its local window, it is considered a corner.
Finally, the function returns a list of tuples containing the coordinates of the detected corners in the image.

LAMBDA MINUS

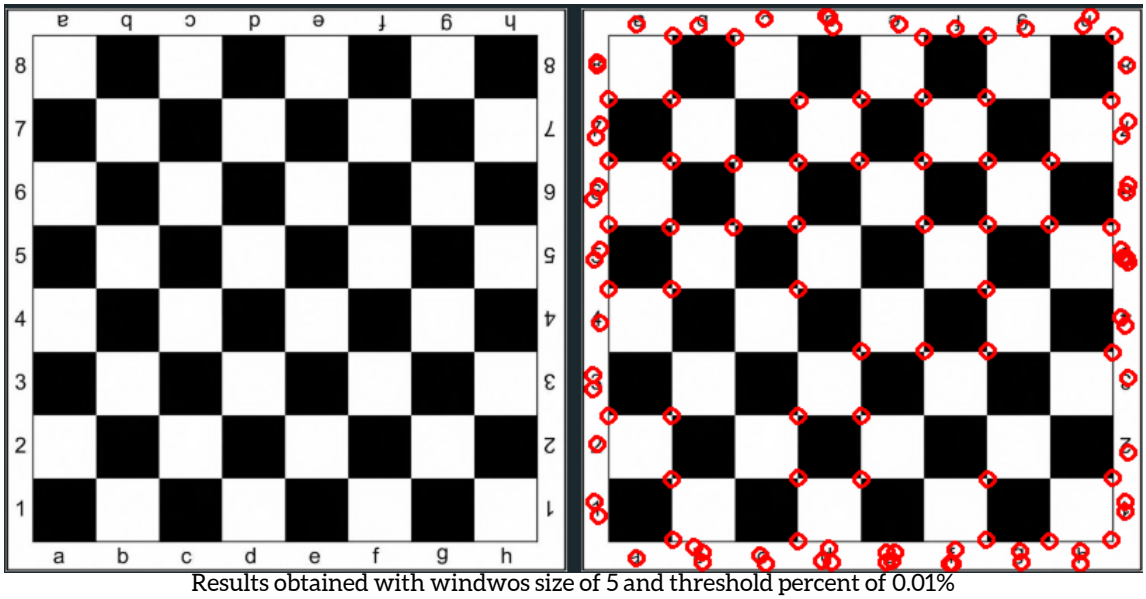
```python
def draw_corners_eigenvalues(self):

        # Start the timer
        start_time = time.time()
        img = self.image['feature_detection_1']
        window_size = int(self.window_size_slider.value())
        th_percentage = float(self.th_percentage_slider.value() / 100)
        corners = feature_extraction.lambda_minus_corner_detection(img, window_size, th_percentage)
        # Draw corners on the original image
        img_with_corners = img.copy()
        for corner in corners:
            cv2.circle(img_with_corners, corner, 5, (0, 0, 255), 2)  # Red color for corners
        self.display_image(img_with_corners, self.image_after_harris)
        # End the timer
        end_time = time.time()

        # Calculate the execution time
        execution_time = end_time - start_time  # in seconds
        self.comp_time_lbl.setText(f"Computation Time: {execution_time * 1000:.2f}ms")
```

This method is responsible for drawing corners detected on the image using the Lambda Minus Corner Detection algorithm.
This method begins by starting a timer to measure the execution time of the function. It retrieves the image from the 'feature_detection_1' key in the 'image' dictionary which is set by user in UI. It also retrieves the values of the 'window_size' and 'th_percentage' parameters from sliders and converts them to appropriate data types and ranges.
Next, the method calls the lambda_minus_corner_detection function, passing the image, window size, and threshold percentage as arguments.

After obtaining the corners, the method creates a copy of the original image and draws circles at the detected corner locations using OpenCV's cv2.circle function. These circles are drawn in red color to highlight the corners.
The resulting image with the drawn corners is then displayed using another method called display_image, passing the modified image and a reference to where it should be displayed.
Subsequently, the timer is stopped, and the execution time is calculated by subtracting the start time from the end time. The execution time is then converted to milliseconds and displayed in a label named 'comp_time_lbl', indicating the time taken for the computation.

LAMBDA MINUS

Results and Observations



Results obtained with windwos size of 5 and threshold percent of 0.01%

The comparison between the results obtained using the Lambda-Minus corner detection algorithm and the Harris Corner Detection algorithm reveals significant differences. Firstly, the Lambda-Minus algorithm exhibits a notably slower execution time, approximately three times longer than the Harris algorithm. This suggests that while the Lambda-Minus algorithm may offer certain advantages, such as simplicity or computational efficiency in specific scenarios, it comes at the cost of increased processing time. Secondly, despite the longer computation time, the Lambda-Minus algorithm yields less accurate corner detection results compared to the Harris algorithm. This implies that the corners identified using the Lambda-Minus algorithm may not be as precise or reliable as those detected using the Harris algorithm. These observations suggest a potential trade-off between speed and accuracy in corner detection algorithms. While the Lambda-Minus algorithm may be suitable for applications where speed is critical and approximate corner detection is acceptable, such as real-time processing in certain contexts, more accurate methods like the Harris Corner Detection algorithm may be preferred in scenarios where accuracy is paramount.Therefore, the choice of corner detection algorithm should be made based on the specific requirements and constraints of the application at hand.

HARRIS OPERATOR

Introduction

Harris Corner Detection is a fundamental algorithm in computer vision used to identify corners or interest points within images. It operates on the principle that corners exhibit significant intensity variations in multiple directions, making them distinguishable from other image features.

Algorithm Overview

Harris Corner Detection is grounded in the computation of image gradients and the analysis of local intensity changes. The algorithm proceeds through the following steps:

1. Gradient Computation: Compute the image gradients using operators like the Sobel filter to obtain Ix and Iy, representing horizontal and vertical intensity changes, respectively.
2. Structure Tensor Construction: Utilize the gradients Ix and Iy to construct the elements of the structure tensor M. This tensor characterizes the local spatial distribution of intensity gradients.
3. Corner Response Calculation: Compute the Harris corner response R for each pixel using the elements of the structure tensor. The response is defined as the determinant of M minus the k times the trace of M, where k is a constant parameter.
4. Corner Candidate Selection: Threshold the Harris response to identify potential corner candidates. Pixels with responses above a certain threshold are considered corner candidates due to significant intensity variations.
5. Non-Maximum Suppression: Perform non-maximum suppression to refine the corner detection process. Local maxima in the Harris response within a defined window are retained as the final corner points, ensuring that only the most salient corners are selected.

By following these steps, the Harris Corner Detection algorithm effectively identifies corners in images, enabling various computer vision applications such as feature matching, object recognition, and image registration.

Parameters and Thresholding

The Harris Corner Detection algorithm offers several parameters that can be adjusted to control its sensitivity and accuracy. Key parameters include the window size for local gradient computation, the constant k for corner response calculation, and the threshold for corner candidate selection. Adjusting these parameters allows users to fine-tune the algorithm's performance for specific applications and image characteristics.

HARRIS OPERATOR

```python
def harris_corner_detection(img, window_size=5, k=0.04, th_percentage=0.01):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    Ix = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
    Iy = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
    # Compute elements of the structure tensor
    Ix2 = Ix ** 2
    Iy2 = Iy ** 2
    Ixy = Ix * Iy
    # Compute sums of structure tensor elements over a local window
    Sx2 = cv2.boxFilter(Ix2, -1, (window_size, window_size))
    Sy2 = cv2.boxFilter(Iy2, -1, (window_size, window_size))
    Sxy = cv2.boxFilter(Ixy, -1, (window_size, window_size))
    # Compute Harris response for each pixel
    R = (Sx2 * Sy2 - Sxy ** 2) - k * (Sx2 + Sy2) ** 2

    # Threshold the Harris response to obtain corner candidates
    threshold = th_percentage * np.max(R)
    corners = np.argwhere(R > threshold)  # Extract corner coordinates

    return corners
```

Harris Corner Detection function and its procedure step by step:

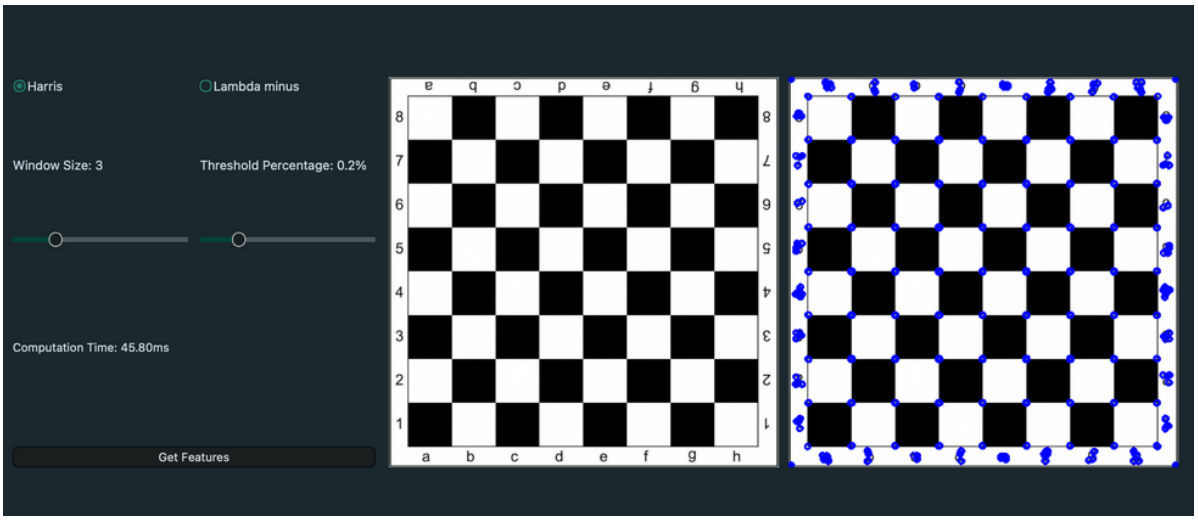
1. Convert Image to Grayscale:
   - The function starts by converting the input image img from BGR color space to grayscale using OpenCV's cv2.cvtColor function.
2. Compute Image Gradients:
   - Sobel operators are applied to the grayscale image to compute the first-order image derivatives Ix and Iy. These derivatives represent the horizontal and vertical gradients, respectively.
   - Ix = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3) computes Ix.
   - Iy = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3) computes Iy.
3. Compute Structure Tensor Elements:
   - Using the computed gradients Ix and Iy, the elements of the structure tensor are calculated:
   - IxIx, IyIy, and · Ix · Iy.
4. Compute Sums of Structure Tensor Elements:
   - Box filters are applied to sum the structure tensor elements over a local window of size window_size.
   - cv2.boxFilter function is used to compute the sums of IxIx, IyIy, and · Ix · Iy using the specified window size.
5. Compute Harris Response:
   - The Harris corner response R is calculated for each pixel using the summed structure tensor elements:
   - $R=(Sx2 \cdot Sy2)-(Sxy)2-k \cdot (Sx2+Sy2)2$
6. Thresholding:
   - The Harris response map is thresholded to identify corner candidates.
   - A threshold value is computed based on a percentage (th_percentage) of the maximum response value.
   - Pixels with response values above this threshold are considered corner candida
7. Extract Corner Coordinates:
   - The coordinates of pixels with response values above the threshold are extracted as corner candidates.

HARRIS OPERATOR

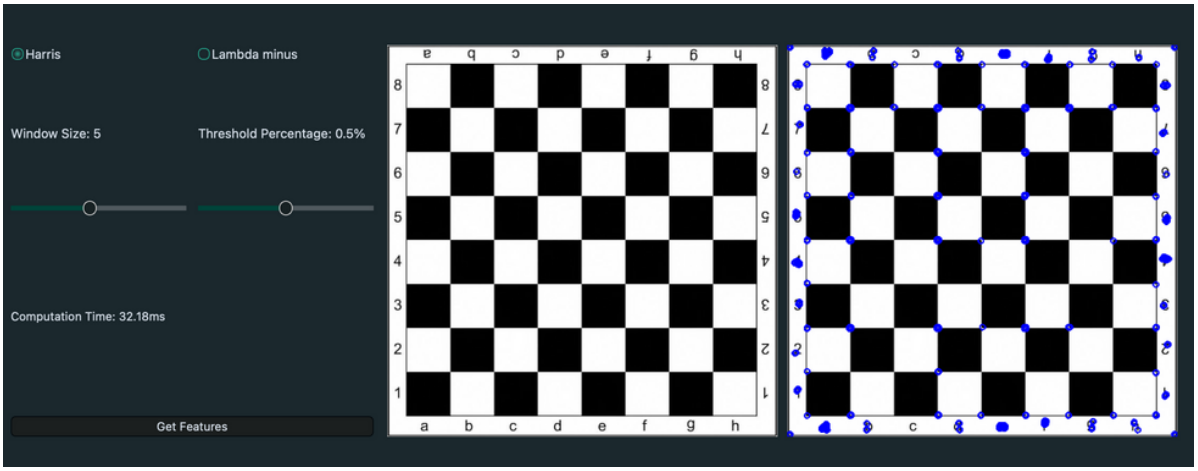
Experimental Results

In our experiments, we applied the Harris Corner Detection algorithm to various images and observed its effectiveness in identifying corners. We found that adjusting parameters such as the threshold and window size had a significant impact on the detected corners' quality and quantity. By carefully tuning these parameters, we were able to enhance the algorithm's performance and extract meaningful corner features from different types of images.

And here are different results based on different thresholds, and we can observe how the the corners are effectively extracted and the computational time is varying.



Results obtained with windwos size of 3 and threshold percent of 0.2



Results obtained with windwos size of 5 and threshold percent of 0.5

Overall, the Harris Corner Detection algorithm proved to be a valuable tool for corner detection tasks in computer vision. Its robustness, coupled with parameter adjustability, makes it suitable for a wide range of applications, from image stitching to motion tracking. Further research and experimentation could focus on optimizing parameter selection methodologies and exploring advanced corner detection techniques to improve algorithm performance and versatility.

SIFT

introduction

SIFT (Scale-Invariant Feature Transform) is a widely-used algorithm in computer vision for detecting and describing local features in images. It was introduced by David Lowe in 1999 and has since become a fundamental tool in various computer vision tasks such as object recognition, image stitching, and 3D reconstruction.
The key feature of SIFT is its ability to detect and describe keypoints that are invariant to changes in scale, rotation, and illumination. This robustness to transformations makes it particularly useful in scenarios where the appearance of objects may vary significantly across different images or viewpoints.

Algorithm overview

1. **Gaussian and Gaussian Difference Calculation:** In the initial stages of SIFT, Gaussian images are generated by convolving the input image with Gaussian kernels at multiple scales. These Gaussian images serve as the basis for constructing the scale-space pyramid. Additionally, the difference of Gaussian (DoG) images is computed by subtracting adjacent Gaussian images within each octave. These DoG images highlight regions of significant intensity change across scales and are instrumental in keypoint detection.
2. **Scale-space Extrema Detection**: SIFT generates a scale-space representation of the input image by convolving it with Gaussian kernels at different scales. Keypoints are then identified as local extrema in this scale-space.
3. **Keypoint Localization**: Once potential keypoints are detected, SIFT refines their locations and eliminates low-contrast keypoints and keypoints located on edges.
4. **Orientation Assignment**: SIFT assigns a canonical orientation to each keypoint based on local image gradient directions. This step ensures rotational invariance of the keypoints.
5. **Descriptor Generation**: Finally, SIFT computes a descriptor for each keypoint, which encodes information about the local image region surrounding the keypoint. These descriptors are highly distinctive and invariant to changes in scale, rotation, and illumination, making them suitable for matching keypoints across different images.

Overall, SIFT provides a powerful framework for extracting and matching local image features, enabling tasks such as object recognition and image alignment in various computer vision applications.

SIFT

```
 def generateDoGImages(self, gaussian_images):
        # Generate Difference-of-Gaussian's image pyramid
        dog_images = []

        for gaussian_images_in_octave in gaussian_images:
            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
                images_in_octave.append(subtract(second_image, first_image))
            dog_images.append(images_in_octave)

        return array(dog_images, dtype=object)
```

After generating Gaussian images for each octave by convolving the base image with Gaussian kernels, the code proceeds to compute the difference of Gaussian (DoG) images. These DoG images are obtained by taking the pixel-wise difference between adjacent Gaussian images within each octave. The resulting DoG images are then stored in an array for further processing.

```
    def findScaleSpaceExtrema(self, gaussian_images, dog_images, num_intervals, sigma,
image_border_width,
                              contrast_threshold=0.04):
        # Find pixel positions of all scale-space extrema in the image pyramid

        threshold = floor(0.5 * contrast_threshold / num_intervals * 255)  # from OpenCV implementation
        keypoints = []

        # Iterate each image in each octave in the DoG Generated Pyramid
        for octave_index, images_in_octave in enumerate(dog_images):

            # Triple Image Iteration for the Extraction of the Extrema Key Point
            for image_index, (first_image, second_image, third_image) \
                    in enumerate(zip(images_in_octave, images_in_octave[1:], images_in_octave[2:])):
                # (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 this Pixel is a Maxima of a Minima, Further Localize the Pixel location
using the
                        # Taylor Expansion, Further Optimization of the Key point is to discard the low
Contrast
                        # points and Eliminate Edge Responses.
                        if self.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 = self.localizeExtremumViaQuadraticFit(i, j,
image_index + 1,octave_index,num_intervals, images_in_octave,sigma,
contrast_threshold,image_border_width)
                            # Compute Key Point Orientation for Invariance of Key point Orientation
                            if localization_result is not None:
                                keypoint, localized_image_index = localization_result
                                keypoints_with_orientations =
self.computeKeypointsWithOrientations(keypoint,octave_index,gaussian_images[octave_index]
[localized_image_index])
                                for keypoint_with_orientation in keypoints_with_orientations:
                                    keypoints.append(keypoint_with_orientation)

        return keypoints
```

SIFT

Within the `findScaleSpaceExtrema` function, the objective is to identify keypoints within each octave of the scale-space pyramid. This process involves examining sets of three consecutive images within the octave and detecting points that exhibit significant intensity changes compared to their surrounding pixels across multiple scales. Specifically, we search for pixels that surpass a predefined threshold and demonstrate higher intensity than their neighboring pixels within a 3x3x3 cube.

Once potential keypoints are identified, the localization process begins. This step aims to refine the detected keypoints' positions and ensure their robustness. Localization is achieved by fitting a quadratic function to the intensity values of the selected pixel and its immediate neighbors. This quadratic fit is performed using the method of least squares, enabling us to estimate the local extrema more accurately.

However, not all detected pixels may qualify as keypoints. To be considered valid keypoints, certain criteria must be met. One crucial criterion is ensuring that the pixel exhibits a strong response to changes in intensity. Additionally, the chosen pixel should demonstrate characteristics indicative of a prominent feature, such as a maximum of positive curvature. If the initial pixel fails to meet these criteria, the algorithm continues searching for alternative keypoints within the octave.

Following keypoint localization in the `findScaleSpaceExtrema` function, SIFT computes the orientation for each detected keypoint. This step involves analyzing the local image gradient directions around each keypoint to determine its dominant orientation. Assigning this orientation ensures rotational invariance, enabling consistent feature representation across various orientations of the same object or scene.

In essence, the `findScaleSpaceExtrema` function iteratively scans through each octave, identifying and refining keypoints to capture significant image structures at various scales. By employing robust detection and localization techniques, SIFT ensures the selection of stable and distinctive keypoints crucial for subsequent feature matching and recognition tasks.

SIFT

Within the SIFT algorithm, the `generateDescriptors` function plays a critical role in computing descriptors for each identified keypoint. These descriptors encapsulate detailed information about the local image region surrounding each keypoint, facilitating subsequent feature matching tasks. The function begins by iterating through the list of keypoints detected in the image. For each keypoint, it extracts relevant information such as the octave, layer, and scale. Utilizing this information, it retrieves the corresponding Gaussian-blurred image from the precomputed set of images at different scales.

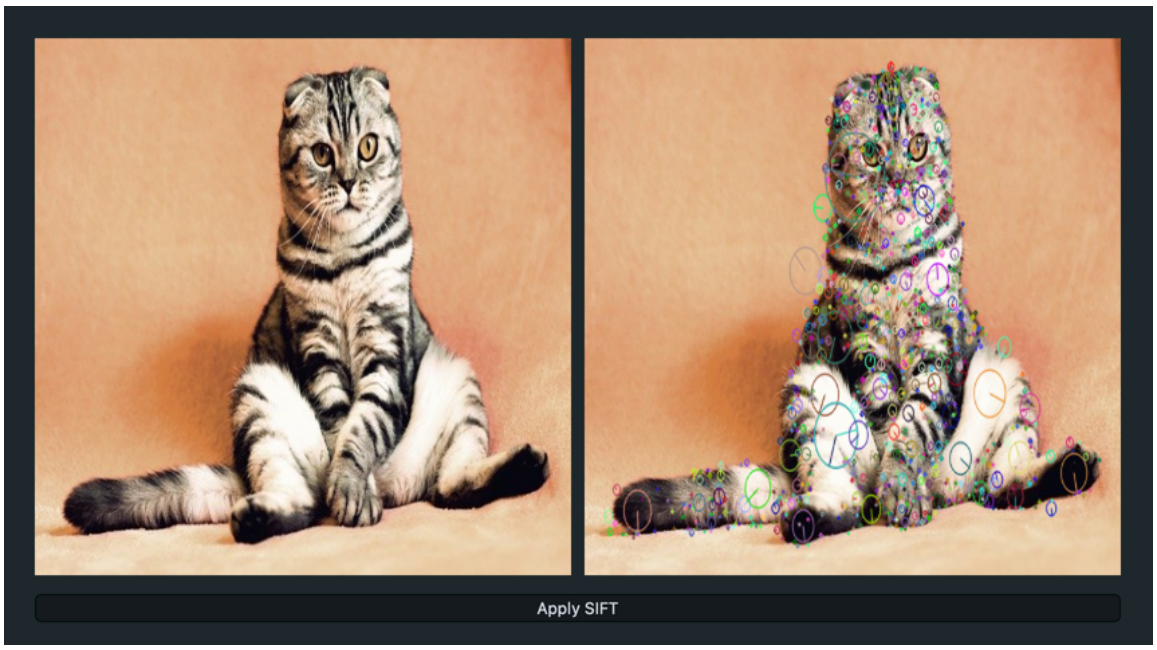
Next, the function calculates the orientation of the keypoint, an essential step in ensuring rotational invariance of the descriptors. It then initializes various lists to store gradient magnitude, orientation, and histogram bins. These bins are crucial for capturing the distribution of gradient orientations within the descriptor window. The size of this window is determined based on the scale of the keypoint, ensuring that it encompasses sufficient local image context.

Iterating over each pixel within the descriptor window, the function computes the orientation of the pixel and evaluates the gradient magnitude and orientation. It applies trilinear interpolation to distribute the contributions of gradient orientations to neighboring bins, effectively capturing the spatial distribution of gradients within the window. The histogram tensor accumulates these contributions, forming the basis of the descriptor for the keypoint.

Upon completing the computation of descriptors for all keypoints, the function normalizes the descriptor vectors and applies thresholding to enhance their robustness against variations in illumination and noise. Finally, it converts the descriptor vectors to unsigned char format, ready for further processing or storage. Overall, the `generateDescriptors` function constitutes a crucial component of the SIFT algorithm, enabling the extraction of distinctive and invariant features essential for tasks such as object recognition and image alignment.

SIFT

Results and observation



In the resulting image, the key points are visually represented as circles of varying sizes, overlaid on the image of the cat. These circles correspond to the detected keypoints at different octaves and scales. Within each circle, a line indicates the orientation associated with the keypoint. This orientation information is crucial for achieving both scale and orientation invariance, ensuring that the keypoints can accurately represent local image features across different scales and orientations. Overall, the visualization of keypoints and their orientations provides valuable insight into the distribution and characteristics of features identified by the SIFT algorithm within the image.

FEATURE MATCHING USING SSD, NCC, AND SIFT

Introduction

Feature matching plays a crucial role in computer vision tasks, allowing us to identify corresponding points or regions between images. In this report, we explore three common techniques for feature matching: Sum of Squared Differences (SSD), Normalized Cross-Correlation (NCC), and Scale-Invariant Feature Transform (SIFT). Each technique offers unique advantages and is suited to different scenarios.

SSD (Sum of Squared Differences)

Algorithm Overview

The SSD algorithm measures the similarity between two image patches by computing the sum of squared differences between their pixel intensities. Lower SSD values indicate higher similarity.

Equation: $SSD = \sum_{i,j}(I(i,j) - T(i,j))^2$

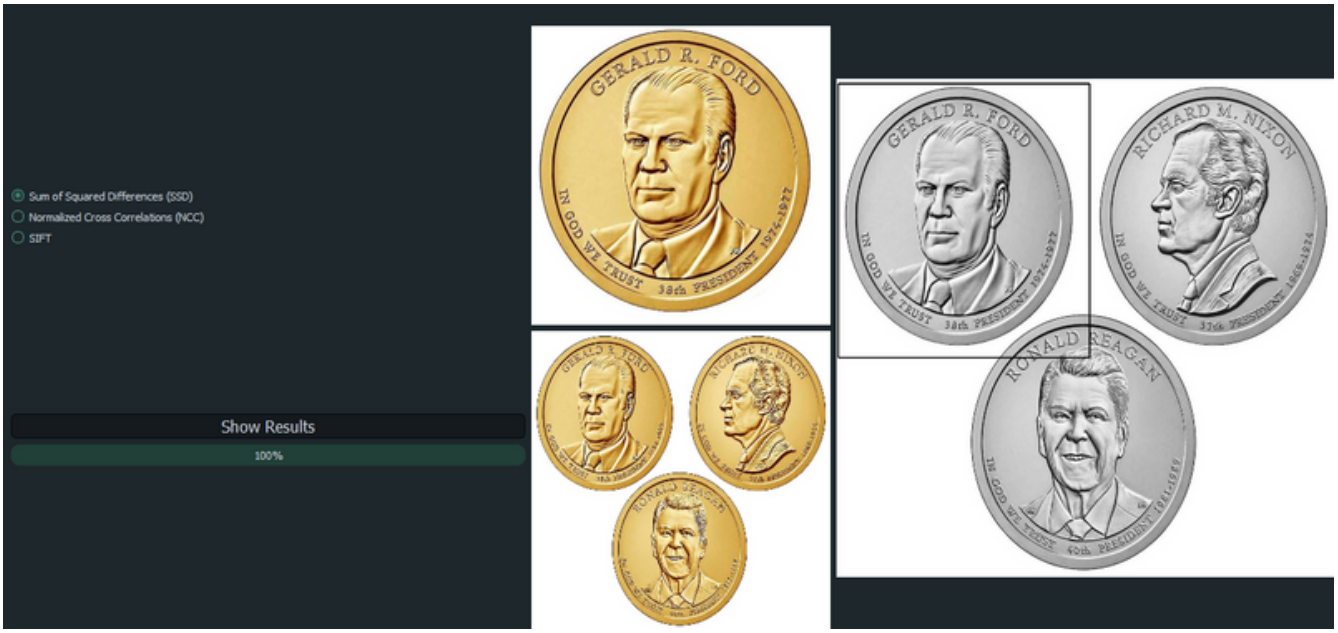
Steps

1. Compute the SSD between the template image and each possible position in the original image.
2. Select the position with the minimum SSD as the matched region.

```python
def template_matching_sqdiff(original_img, template_img):
    # Get dimensions of original and template images
    original_h, original_w = original_img.shape[:2]
    template_h, template_w = template_img.shape[:2]

    # Calculate SSD for each possible position of the template within the original ima
    min_ssd = float('inf')
    min_loc = (0, 0)

    for y in range(original_h - template_h + 1):
        for x in range(original_w - template_w + 1):
            patch = original_img[y:y + template_h, x:x + template_w]
            ssd = np.sum((patch - template_img) ** 2)
            if ssd < min_ssd:
                min_ssd = ssd
                min_loc = (x, y)

    return min_loc
```

FEATURE MATCHING USING SSD, NCC, AND SIFT

Results and Observations



SSD values range from 0 to infinity, where lower values indicate better matches. However, SSD is sensitive to changes in scale and orientation, making it less effective in scenarios where these factors vary significantly between images.

NCC (Normalized Cross-Correlation)

Algorithm Overview

NCC measures the similarity between two image patches by computing the normalized cross-correlation coefficient. Higher NCC values indicate higher similarity.

Equation: $$NCC = \frac{\sum_{i,j}(I(i,j)-\mu_I)\cdot(T(i,j)-\mu_T)}{\sigma_I \cdot \sigma_T}$$

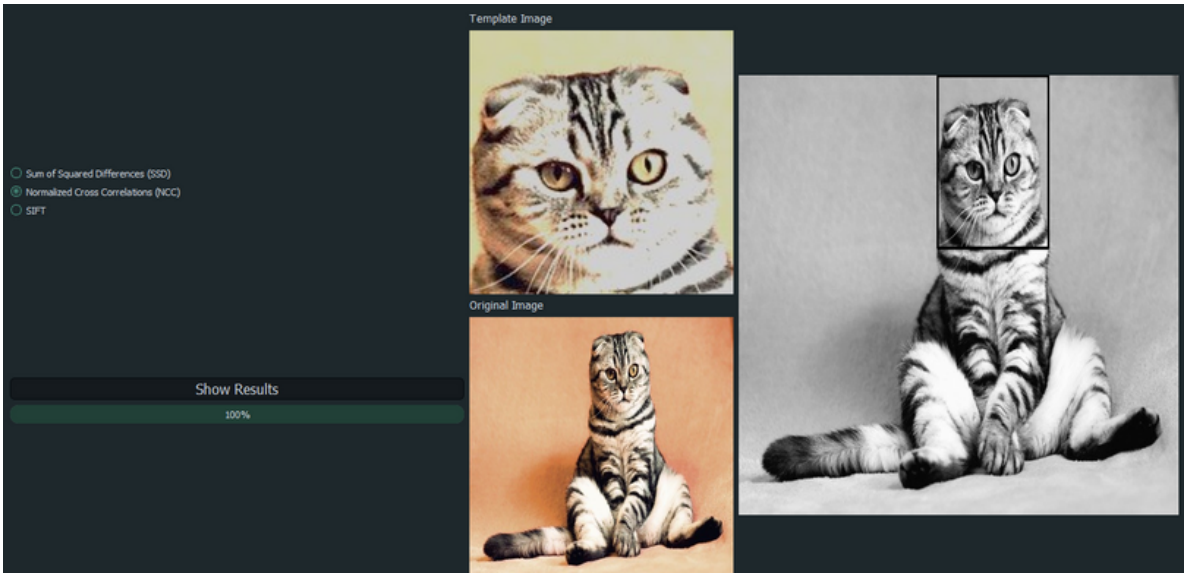
Steps

1. Compute the SSD between the template image and each possible position in the original image.
2. Select the position with the minimum SSD as the matched region.

FEATURE MATCHING USING SSD, NCC, AND SIFT

```python
def template_matching_ncc(original_img, template_img):
    """
    # Get dimensions of original and template images
    original_h, original_w = original_img.shape[:2]
    template_h, template_w = template_img.shape[:2]

    # Calculate mean and standard deviation of template image
    template_mean = np.mean(template_img)
    template_std = np.std(template_img)

    # Initialize variables for max NCC and corresponding location
    max_ncc = -np.inf
    max_loc = (0, 0)

    # Slide the template over the original image and calculate NCC
    for y in range(original_h - template_h + 1):
        for x in range(original_w - template_w + 1):
            patch = original_img[y:y + template_h, x:x + template_w]

            # Calculate mean and standard deviation of patch
            patch_mean = np.mean(patch)
            patch_std = np.std(patch)

            if patch_std > 0:
                # Calculate NCC between patch and template
                ncc = np.sum((patch - patch_mean) * (template_img - template_mean)) / (
                            patch_std * template_std * np.prod(template_img.shape))

                # Update max NCC and corresponding location if current NCC is higher
                if ncc > max_ncc:
                    max_ncc = ncc
                    max_loc = (x, y)

    return max_loc
```

Results and Observations



NCC values range from -1 to 1, where 1 indicates a perfect match and -1 indicates a perfect anti-match. Unlike SSD, NCC is less sensitive to changes in scale and orientation, making it more robust in scenarios where these factors vary.

FEATURE MATCHING USING SSD, NCC, AND SIFT

SIFT (Scale-Invariant Feature Transform)

Algorithm Overview

SIFT detects and describes local features that are invariant to scale and rotation changes. It extracts keypoints and generates descriptors to represent local image structures.

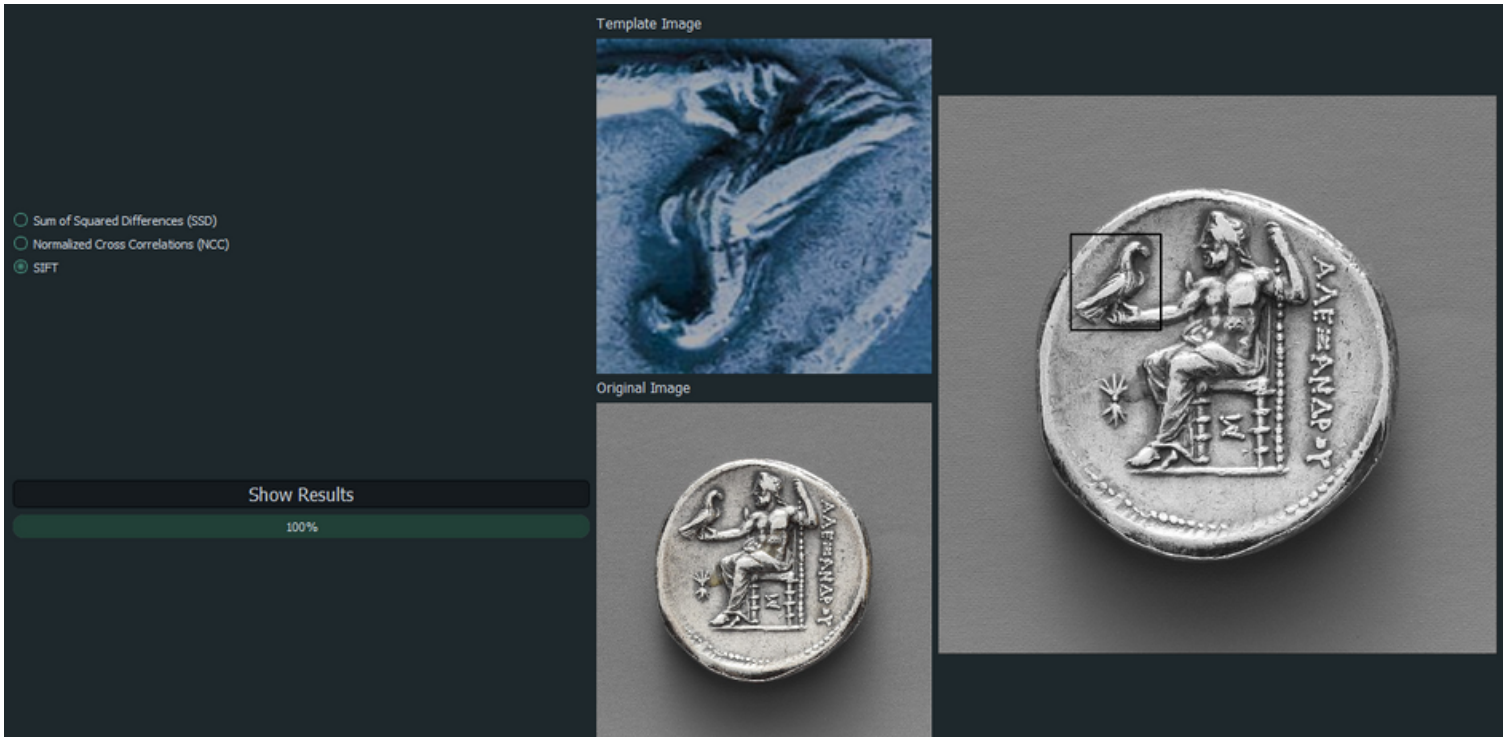
Steps

1. Detect key-points using scale-space extrema in the Difference of Gaussians (DoG) pyramid.
2. Assign orientations to key-points based on gradient orientations.
3. Compute descriptors using histograms of gradient magnitudes and orientations.

```
def template_matching_and_draw_roi(original_img, template_img, method='SSD'):
    """
    # Choose matching method
    if method == 'SSD':
        top_left = template_matching_sqdiff(original_img, template_img)
        bottom_right = (top_left[0] + template_img.shape[1], top_left[1] + template_img.shape[0])
    elif method == 'NCC':
        top_left = template_matching_ncc(original_img, template_img)
        bottom_right = (top_left[0] + template_img.shape[1], top_left[1] + template_img.shape[0])
    else:
        sift_original = siftapply(original_img)
        sift_template = siftapply(template_img)
        keypoints_original, descriptors_original = sift_original.return_keypoints(),
sift_original.return_descriptors()
        keypoints_template, descriptors_template = sift_template.return_keypoints(),
sift_template.return_descriptors()

        bf = cv2.BFMatcher()
        matches = bf.knnMatch(descriptors_template, descriptors_original, k=2)

        good_matches = []
        for m, n in matches:
            if m.distance < 0.75 * n.distance:
                good_matches.append(m)

        template_pts = np.float32([keypoints_template[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
        original_pts = np.float32([keypoints_original[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)

        M, _ = cv2.findHomography(template_pts, original_pts, cv2.RANSAC)

        h, w = template_img.shape[:2]
        template_corners = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1,
2)

        transformed_corners = cv2.perspectiveTransform(template_corners, M)

        top_left = tuple(np.int32(transformed_corners.min(axis=0).ravel()))
        bottom_right = tuple(np.int32(transformed_corners.max(axis=0).ravel()))

    # Check if top_left is None
    if top_left is None:
        return original_img  # Return original image if template not found

    # Draw ROI on the original image
    cv2.rectangle(original_img, top_left, bottom_right, (0, 255, 0), 2)

    return original_img
```

FEATURE MATCHING USING SSD, NCC, AND SIFT

Results and Observations



When we use the Scale-Invariant Feature Transform (SIFT) algorithm to match features between images, we can make some interesting observations. SIFT is well-known for its ability to handle changes in scale, rotation, and illumination, which gives it a clear advantage in feature matching tasks.

The SIFT algorithm achieves scale invariance by detecting keypoints at different scales within the image. It does this by using a scale-space representation, which allows the algorithm to identify features regardless of their size or resolution. This means that SIFT can perform consistently well, no matter the scale of the image.

Another great feature of SIFT is its ability to handle image rotation. During keypoint detection, SIFT assigns an orientation to each keypoint based on local image gradients. This ensures that the feature descriptors are not affected by image rotation, making it easier to accurately match features even in rotated images.

SIFT also encodes information about the local image neighborhood surrounding each keypoint in its feature descriptors. This, combined with its robust keypoint detection, allows SIFT to effectively match features across images, even when there are variations in brightness, contrast, and partial occlusion. But SIFT doesn't stop there. It can also handle changes in viewpoint and scene clutter, going beyond traditional geometric transformations. This makes it a great choice for real-world scenarios where images may undergo complex transformations and environmental conditions.

FEATURE MATCHING USING SSD, NCC, AND SIFT

Conclusion

We investigated three methods for matching features in computer vision: Sum of Squared Differences (SSD), Normalized Cross-Correlation (NCC), and Scale-Invariant Feature Transform (SIFT). These methods have unique benefits and drawbacks, making them suitable for various applications depending on specific needs.

Sum of Squared Differences (SSD):

- **Advantages:**
  - **Simplicity and ease of implementation.**
  - **Direct computation of pixel-wise differences.**
- **Observations:**
  - **SSD is sensitive to changes in scale and orientation, making it less robust in scenarios with varying conditions.**
  - **While computationally efficient, SSD may yield suboptimal results when dealing with complex image transformations.**

Normalized Cross-Correlation (NCC):

- **Advantages:**
  - **Robustness to changes in illumination and contrast.**
  - **Ability to handle variations in scale and rotation to some extent.**
- **Observations:**
  - **NCC provides better results compared to SSD in scenarios with varying lighting conditions.**
  - **While more robust than SSD, NCC can still be sensitive to large-scale transformations and occlusions.**

Scale-Invariant Feature Transform (SIFT):

- **Advantages:**
  - **Invariance to scale, rotation, and partial occlusion.**
  - **High level of robustness and accuracy in feature matching.**
- **Observations:**
  - **SIFT outperforms SSD and NCC in scenarios with significant variations in scale, rotation, and illumination.**
  - **Despite its computational complexity, SIFT provides superior results in challenging image matching tasks.**

Overall, the choice of feature matching method should be guided by the specific requirements of the application. For real-time applications with minimal computational resources and relatively simple transformations, SSD or NCC may suffice. However, in scenarios requiring robustness to scale, rotation, and illumination changes, SIFT emerges as the preferred choice despite its higher computational cost.