Experiment Report on SIFT Algorithm Implementation

Wu Han
Class: SEIEE 2301

December 8, 2024

1 Experiment Overview

This experiment provides an introduction to the SIFT (Scale-Invariant Feature Transform) algorithm, which was proposed by David Lowe in 1999 and further refined in 2004. The SIFT algorithm is used to detect and describe local features in images, making it robust against transformations such as translation, rotation, and affine changes. The SIFT algorithm has many key properties, like invariance, distinctiveness, multiplicity and scalability.

In OpenCV, there are pre-built functions for feature extraction and matching using SIFT. However, in this experiment, we attempt to implement our own version of a SIFT class to understand the underlying process.

The SIFT function implementation in this experiment follows these main steps:

1. Keypoint detection and localization.
2. Image preprocessing with Gaussian blur, followed by gradient magnitude and orientation computation.
3. Computation of the dominant orientation for each keypoint.
4. SIFT descriptor calculation for each keypoint.
5. Feature matching and visualization of results.

2 Keypoint Detection and Localization

For keypoint detection, we use the Harris corner detection method, which is implemented as follows:

```python
def build_image_pyramid(image, levels=3):
    """
    Construct an image pyramid by resizing the image to multiple scales.
    """
    pyramid = [image]
    for _ in range(1, levels):
```

```
        image = cv2.pyrDown(image)  # Downscale the image
        pyramid.append(image)
    return pyramid

def compute_harris_corners(image, block_size=2, ksize=3, k=0.04):
    """
    Use Harris corner detection to find keypoints with an adaptive threshold.
    """
    corners = cv2.cornerHarris(image, blockSize=block_size, ksize=ksize, k=k)
    corners = cv2.dilate(corners, None)  # Enhance corner points
    adaptive_thresh = 0.25 * corners.max()  # Compute adaptive threshold based on maximum response
    keypoints = np.argwhere(corners > adaptive_thresh)
    keypoints = [cv2.KeyPoint(float(p[1]), float(p[0]), 1) for p in keypoints]
    return keypoints
```

The first function generates an image pyramid, which includes the original image and scaled-down versions. The pyramid helps detect features at multiple scales. **Key Points:**

- `cv2.pyrDown(image)` reduces the resolution by half in each step.
- The function returns a list of images, including the original and scaled-down versions.

The second function uses the Harris corner detection method to identify keypoints. An adaptive threshold selects strong corners.

3 Gaussian Blur and Gradient Calculation

The image is processed with Gaussian blur to smooth it. Then, the gradient magnitude and orientation are computed using the following formulas:

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y-1) - L(x, y+1))^2}$$

$$\theta(x, y) = \arctan\left(\frac{L(x+1, y) - L(x-1, y)}{L(x, y-1) - L(x, y+1)}\right)$$

The gradients are computed in both directions, and any negative angles are adjusted to the range of $[0, 2\pi]$.

```
descriptors = []
    image = cv2.GaussianBlur(image, (5, 5), 1.6)  # Gaussian smoothing to reduce noise.

    for kp in keypoints:
        x, y = int(kp.pt[0]), int(kp.pt[1])  # Keypoint coordinates.

        # Extract the 16x16 neighborhood around the keypoint.
```

```
        patch = image[max(0, y - patch_size // 2):y + patch_size
           // 2,
                      max(0, x - patch_size // 2):x + patch_size
                         // 2]

        # Skip incomplete patches near image boundaries.
        if patch.shape[0] != patch_size or patch.shape[1] !=
           patch_size:
            continue

        # Compute gradients using Sobel operators.
        gx = cv2.Sobel(patch, cv2.CV_32F, 1, 0, ksize=3)
        gy = cv2.Sobel(patch, cv2.CV_32F, 0, 1, ksize=3)
        magnitude, angle = cv2.cartToPolar(gx, gy, angleInDegrees
           =True)
```

- The image is smoothed using a Gaussian filter with `cv2.GaussianBlur` to reduce noise.
- Gradients are calculated for the 16x16 neighborhood around each keypoint using Sobel operators:
  - `gx` represents the horizontal gradient.
  - `gy` represents the vertical gradient.
- Gradient magnitudes and orientations are computed using `cv2.cartToPolar`.

4 Dominant Orientation Calculation

To calculate the dominant orientation of each keypoint, a region around each keypoint is sampled. The region is weighted based on the distance to the center, and a histogram of gradients is computed. The dominant orientation is the one with the highest value in the histogram.

Each keypoint's descriptor is computed by creating a 16x16 block around the keypoint. This block is divided into 16 smaller 4x4 regions. For each region, bilinear interpolation is used to distribute gradient magnitudes across bins and cells. The descriptor is a 128-dimensional vector formed by the weighted sum of gradient magnitudes in each direction.

```
# Adjust angles relative to the dominant keypoint orientation.
        main_orientation = kp.angle if kp.angle else 0
        adjusted_angle = (angle - main_orientation) % 360
```

- The dominant orientation of the keypoint is calculated from the gradient data.
- Angles are adjusted relative to this orientation to ensure rotational invariance.

```
# Initialize a 4x4 grid with 8 bins per cell (128-dimensional
   descriptor).
        cell_size = patch_size // 4  # Each cell is 4x4 pixels.
```

```python
        descriptor = np.zeros((4, 4, 8), dtype=np.float32)

        # Iterate over all pixels in the 16x16 patch.
        for i in range(patch_size):
            for j in range(patch_size):
                # Compute relative coordinates within the grid.
                patch_x = j + 0.5  # Pixel center adjustment
                patch_y = i + 0.5

                cell_x = patch_x / cell_size  # Grid row index
                cell_y = patch_y / cell_size  # Grid column index

                # Identify the 4 nearest grid cells.
                x0, y0 = int(cell_x), int(cell_y)
                x1, y1 = min(x0 + 1, 3), min(y0 + 1, 3)  # Ensure within 4x4 grid boundaries.

                # Compute interpolation weights.
                dx1, dy1 = cell_x - x0, cell_y - y0
                dx2, dy2 = 1 - dx1, 1 - dy1

                # Quantize angle into 8 bins (each 45 degrees).
                bin_idx = int(adjusted_angle[i, j] // 45) % 8
                magnitude_value = magnitude[i, j]

                # Bilinearly interpolate the gradient magnitude into the 4 nearest cells.
                descriptor[y0, x0, bin_idx] += magnitude_value * dx2 * dy2
                descriptor[y0, x1, bin_idx] += magnitude_value * dx1 * dy2
                descriptor[y1, x0, bin_idx] += magnitude_value * dx2 * dy1
                descriptor[y1, x1, bin_idx] += magnitude_value * dx1 * dy1
```

- The 16x16 patch is divided into a 4x4 grid, with each cell being 4x4 pixels.
- For each cell, an 8-bin histogram is generated to represent gradient orientations.
- Bilinear interpolation is used to distribute gradient magnitudes across bins and cells.
- The resulting histograms are concatenated to form a 128-dimensional descriptor.
- The descriptor is normalized to unit length, clipped to 0.2 to suppress extreme values, and re-normalized.

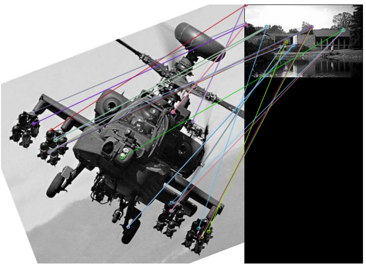
5 Feature Matching

To match descriptors between two images, we calculate the Euclidean distance between their descriptors. If the distance between two descriptors is small, we consider them as matching features.

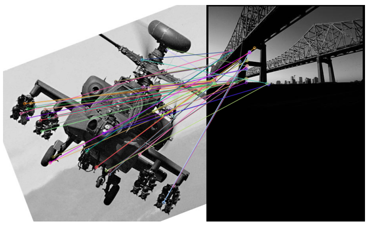
```python
def match_features(descriptors1, descriptors2):
    """
    Match features between two sets of descriptors using a brute-
        force matcher.
    """
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(descriptors1, descriptors2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches
```

6 Results

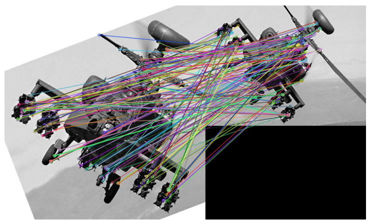
look at Figure 1 below, you will find that there are many arrows in Image3, but nearly no arrows and lines in other images.So we can say that our self-designed SIFT eligible for using.



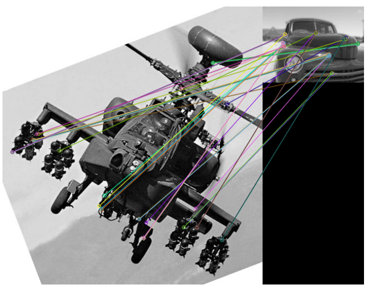
(a) Image 1



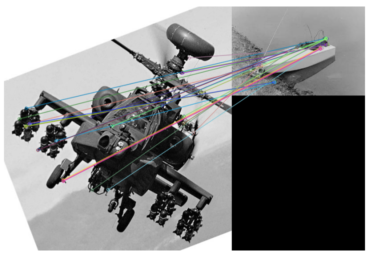
(b) Image 2



(c) Image 3



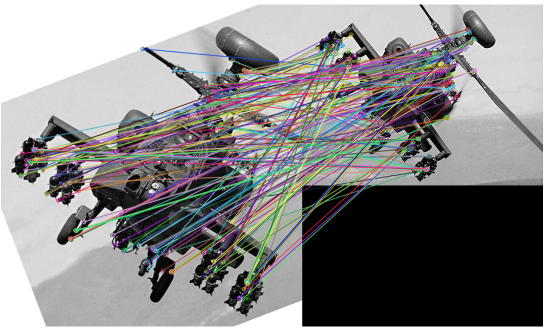
(d) Image 4



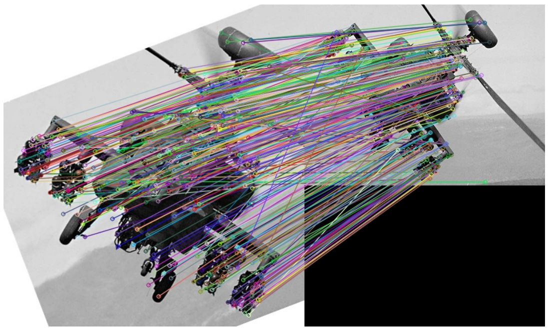
(e) Image 5

Figure 1: Result of Self Designed SIFT

Then we compare the self-designed result with the pre-built functions.Just look at Figure 2: The details are Shown in the table:



(a) self-designed SIFT



(b) pre-built SIFT

Figure 2: Comparison Of Different SIFT

Figure	self-designed	pre-built
1.jpg	32	31
2.jpg	54	35
3.jpg	241	399
4.jpg	33	31
5.jpg	26	18

We may cannot see the differences from Figure 2, but we can see the differences the table above. It's obvious that the pre-built function is better than our self-designed function.From my perspective, the reason is that it's hard to adjust exactly the parameters of each function in our self-designed function. But for the pre-built function, maybe it has already used the best hyperparameters from the experiment.

7 Discussion

7.1 Why Delete Matched Points?

If matched points are not deleted, the same point may be matched to multiple points. Additionally, since the traversal is not area-based, points in different columns may remain.

7.2 Why Use Gaussian Blur?

Gaussian blur is applied to prevent sharp edges from causing distortion and inaccuracies in the results.

7.3 Challenges with Harris Corner Detection

Compared to direct SIFT, Harris corner detection produces many feature points, including many unnecessary ones. Using a Gaussian pyramid may mitigate this issue.

8 Reflections on the Lab

This lab provided significant opportunities for me to improve my skills in multiple areas, including mathematics, logical reasoning, and coding abilities. By implementing a simplified version of the SIFT algorithm, I gained a deeper understanding of mathematical concepts such as gradient calculations, Gaussian smoothing, and histogram interpolation.

From a logical thinking perspective, breaking down the SIFT descriptor into smaller steps, such as Gaussian blur, gradient computation, and descriptor construction, allowed me to approach the problem systematically.

Moreover, my coding abilities advanced significantly throughout the lab. Writing clean, efficient Python code while debugging errors and optimizing performance helped me build confidence in handling computer vision tasks.

Overall, this lab was a valuable experience that fostered growth in my mathematical understanding, logical thinking, and programming skills, laying a solid foundation for future exploration in computer vision and related fields.

9 Full code

pre built.py:

```python
import cv2
import numpy as np
import os

def build_image_pyramid(image, levels=3):
    """
    Construct an image pyramid by resizing the image to multiple scales.
    """
    pyramid = [image]
    for _ in range(1, levels):
        image = cv2.pyrDown(image)  # Downscale the image
        pyramid.append(image)
    return pyramid

def compute_harris_corners(image, block_size=2, ksize=3, k=0.04):
    """
    Use Harris corner detection to find keypoints with an adaptive threshold.
    """
    corners = cv2.cornerHarris(image, blockSize=block_size, ksize=ksize, k=k)
    corners = cv2.dilate(corners, None)  # Enhance corner points
    adaptive_thresh = 0.25 * corners.max()  # Compute adaptive threshold based on maximum response
    keypoints = np.argwhere(corners > adaptive_thresh)
    keypoints = [cv2.KeyPoint(float(p[1]), float(p[0]), 1) for p in keypoints]
    return keypoints


def compute_sift_descriptor(image, keypoints, patch_size=16):
    """
    Accurately compute SIFT-like descriptors with bilinear interpolation for keypoints.
    Each descriptor uses a 16x16 neighborhood divided into 4x4
```

```
        cells,
    with an 8-bin histogram for each cell (128-dimensional descriptor).
    """
    descriptors = []
    image = cv2.GaussianBlur(image, (5, 5), 1.6)  # Gaussian smoothing to reduce noise.

    for kp in keypoints:
        x, y = int(kp.pt[0]), int(kp.pt[1])  # Keypoint coordinates.

        # Extract the 16x16 neighborhood around the keypoint.
        patch = image[max(0, y - patch_size // 2):y + patch_size // 2,
                      max(0, x - patch_size // 2):x + patch_size // 2]

        # Skip incomplete patches near image boundaries.
        if patch.shape[0] != patch_size or patch.shape[1] != patch_size:
            continue

        # Compute gradients using Sobel operators.
        gx = cv2.Sobel(patch, cv2.CV_32F, 1, 0, ksize=3)
        gy = cv2.Sobel(patch, cv2.CV_32F, 0, 1, ksize=3)
        magnitude, angle = cv2.cartToPolar(gx, gy, angleInDegrees=True)

        # Adjust angles relative to the dominant keypoint orientation.
        main_orientation = kp.angle if kp.angle else 0
        adjusted_angle = (angle - main_orientation) % 360

        # Initialize a 4x4 grid with 8 bins per cell (128-dimensional descriptor).
        cell_size = patch_size // 4  # Each cell is 4x4 pixels.
        descriptor = np.zeros((4, 4, 8), dtype=np.float32)

        # Iterate over all pixels in the 16x16 patch.
        for i in range(patch_size):
            for j in range(patch_size):
                # Compute relative coordinates within the grid.
                patch_x = j + 0.5  # Pixel center adjustment
                patch_y = i + 0.5

                cell_x = patch_x / cell_size  # Grid row index
                cell_y = patch_y / cell_size  # Grid column index

                # Identify the 4 nearest grid cells.
                x0, y0 = int(cell_x), int(cell_y)
```

```python
                x1, y1 = min(x0 + 1, 3), min(y0 + 1, 3)  # Ensure within 4x4 grid boundaries.

                # Compute interpolation weights.
                dx1, dy1 = cell_x - x0, cell_y - y0
                dx2, dy2 = 1 - dx1, 1 - dy1

                # Quantize angle into 8 bins (each 45 degrees).
                bin_idx = int(adjusted_angle[i, j] // 45) % 8
                magnitude_value = magnitude[i, j]

                # Bilinearly interpolate the gradient magnitude into the 4 nearest cells.
                descriptor[y0, x0, bin_idx] += magnitude_value * dx2 * dy2
                descriptor[y0, x1, bin_idx] += magnitude_value * dx1 * dy2
                descriptor[y1, x0, bin_idx] += magnitude_value * dx2 * dy1
                descriptor[y1, x1, bin_idx] += magnitude_value * dx1 * dy1

        # Flatten the 4x4x8 grid into a 128-dimensional vector.
        descriptor = descriptor.flatten()

        # Normalize the descriptor to unit length (L2 normalization).
        descriptor /= (np.linalg.norm(descriptor) + 1e-7)

        # Clip values to 0.2 to suppress extreme gradients and re-normalize.
        descriptor = np.clip(descriptor, 0, 0.2)
        descriptor /= (np.linalg.norm(descriptor) + 1e-7)

        descriptors.append(descriptor)

    return np.array(descriptors, dtype=np.float32)


def match_features(descriptors1, descriptors2):
    """
    Match features between two sets of descriptors using a brute-force matcher.
    """
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(descriptors1, descriptors2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches

def main():
    target_path = "target.jpg"
```

```
    dataset_folder = "dataset"
    output_folder = "output_matches"

    # Ensure the output folder exists
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Load target image and create image pyramid
    target_image = cv2.imread(target_path, cv2.IMREAD_GRAYSCALE)
    target_pyramid = build_image_pyramid(target_image, levels=3)

    # Compute Harris corners and SIFT descriptors for target
       image
    target_keypoints = compute_harris_corners(target_pyramid[0])
    target_descriptors = compute_sift_descriptor(target_pyramid
       [0], target_keypoints)

    for image_name in os.listdir(dataset_folder):
        image_path = os.path.join(dataset_folder, image_name)
        dataset_image = cv2.imread(image_path, cv2.
           IMREAD_GRAYSCALE)
        dataset_pyramid = build_image_pyramid(dataset_image,
           levels=3)

        # Compute Harris corners and SIFT descriptors for dataset
            image
        dataset_keypoints = compute_harris_corners(
           dataset_pyramid[0])
        dataset_descriptors = compute_sift_descriptor(
           dataset_pyramid[0], dataset_keypoints)

        # Match features
        matches = match_features(target_descriptors,
           dataset_descriptors)


        print(f"{image_name}: {len(matches)} matches")

        # Draw matches
        result_image = cv2.drawMatches(target_pyramid[0],
           target_keypoints,
                                        dataset_pyramid[0],
                                           dataset_keypoints,
                                        matches, None,
                                        flags=cv2.
                                           DrawMatchesFlags_NOT_DRAW_SINGLE_P
                                           )


        output_path = os.path.join(output_folder, f"matches_{
           image_name}.png")
```

```
        cv2.imwrite(output_path, result_image)

if __name__ == "__main__":
    main()
```

self designed.py:

```
import cv2
import os
import numpy as np

# Path to the dataset folder and target image
dataset_path = "dataset"
target_image_path = "target.jpg"
output_folder = "system_shift"

# Ensure the output folder exists
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# Load the target image
target_image = cv2.imread(target_image_path, cv2.IMREAD_GRAYSCALE)
if target_image is None:
    print(f"Error: Could not load target image '{target_image_path}'")
    exit()

# Initialize SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints in the target image
target_keypoints = sift.detect(target_image, None)

# Compute descriptors for the detected keypoints
_, target_descriptors = sift.compute(target_image, target_keypoints)

# Draw keypoints for the target image (optional visualization)
target_keypoint_image = cv2.drawKeypoints(target_image, target_keypoints, None)
cv2.imwrite(os.path.join(output_folder, "target_keypoints.jpg"), target_keypoint_image)

# Set up FLANN-based matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

best_match_count = 0
best_match_image = None
```

```
best_match_keypoints = None
best_match_good_matches = None

# Iterate through all images in the dataset folder
for image_name in os.listdir(dataset_path):
    image_path = os.path.join(dataset_path, image_name)

    # Load the current image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print(f"Warning: Could not load image '{image_path}'")
        continue

    # Detect keypoints in the current image
    keypoints = sift.detect(image, None)

    # Compute descriptors for the detected keypoints
    _, descriptors = sift.compute(image, keypoints)

    # Use FLANN matcher to find matches between the target and current image
    if descriptors is not None and target_descriptors is not None:
        matches = flann.knnMatch(target_descriptors, descriptors, k=2)

        # Apply Lowe's ratio test to find good matches
        good_matches = []
        for m, n in matches:
            if m.distance < 0.7 * n.distance:
                good_matches.append(m)

        # Output the number of good matches for the current image
        print(f"Image '{image_name}' has {len(good_matches)} good matches.")

        # Draw keypoints for the current image (optional visualization)
        keypoint_image = cv2.drawKeypoints(image, keypoints, None)
        cv2.imwrite(os.path.join(output_folder, f"keypoints_{image_name}"), keypoint_image)

        # Draw matches for the current image and save
        result_image = cv2.drawMatches(target_image, target_keypoints, image, keypoints, good_matches,
                                        None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_P
                                        )
        output_path = os.path.join(output_folder, f"matches_{image_name}")
```

```python
        cv2.imwrite(output_path, result_image)

        # Update the best match if the current image has more good matches
        if len(good_matches) > best_match_count:
            best_match_count = len(good_matches)
            best_match_image = image
            best_match_keypoints = keypoints
            best_match_good_matches = good_matches

# Draw and save the best match result if found
if best_match_image is not None:
    best_match_result = cv2.drawMatches(target_image, target_keypoints, best_match_image, best_match_keypoints,
                                        best_match_good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_
                                        )
    cv2.imwrite(os.path.join(output_folder, "best_match_result.jpg"), best_match_result)
    print(f"Best match saved as 'best_match_result.jpg' with {best_match_count} good matches.")
else:
    print("No good matches found in the dataset.")
```