SIFT

September 5, 2024

1 st124092

2 Kaung Htet Cho

1. **Feature detection (extraction) stage**:

- each image is searched for locations that are likely to match well in other images.

2. **Feature description stage**:

- each region around detected keypoint locations is converted into a more com- pact and stable (invariant) descriptor that can be matched against other descriptors

3. **Feature matching stage**:

- searches for likely matching candidates in other images.

4. **Feature tracking stage**:

- is an alternative to the third stage that only searches a small neighborhood around each detected feature and is therefore more suitable for video processing.

2.1 3. SIFT (Scale-Invariant Feature Transform)

SIFT detects keypoints and computes descriptors, which are invariant to scale and rotation. The steps involve: 1. `Detecting scale-space extrema`: - SIFT algorithm uses Difference of Gaussians which is an approximation of LoG (Laplacian of Gaussian). Difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different , let it be and k . This process is done for different octaves of the image in Gaussian Pyramid. - Once this DoG are found, images are searched for local extrema over scale and space. For eg, one pixel in an image is compared with its 8 neighbours as well as 9 pixels in next scale and 9 pixels in previous scales. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale.

2. `Keypoint localization and filtering`:

- Taylor series expansion of scale space is used to get more accurate location of extrema, and if the intensity at this extrema is less than a threshold value (0.03 as per the paper), it is rejected.
- DoG has higher response for edges, so edges also need to be removed. For this, a concept similar to Harris corner detector is used. So it eliminates any low-contrast keypoints and edge keypoints and what remains is strong interest points.

3. `Assigning orientation and computing descriptors`:

   Now an orientation is assigned to each keypoint to achieve invariance to image rotation

- neighbourhood is taken around the keypoint location depending on the scale, and the gradient magnitude and direction is calculated in that region.
- An orientation histogram with 36 bins covering 360 degrees is created (It is weighted by gradient magnitude and gaussian-weighted circular window with equal to 1.5 times the scale of keypoint).
- The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation.
- It creates keypoints with same location and scale, but different directions. It contribute to stability of matching.

4. `Keypoint Matching`:

- Keypoints between two images are matched by identifying their nearest neighbours.
- But in some cases, the second closest-match may be very near to the first. It may happen due to noise or some other reasons. In that case, `ratio of closest-distance to second-closest distance` is taken. If it is greater than 0.8, they are rejected. It eliminates around 90% of false matches while discards only 5% correct matches, as per the paper.

References: - https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html - https://en.wikipedia.org/wiki/Scale-invariant_feature_transform

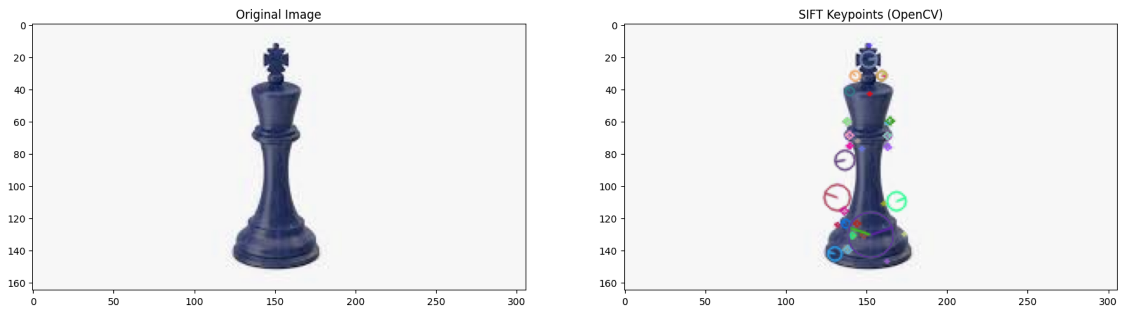
[ ]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
```

2.1.1 OpenCV SIFT

Here, we will use OpenCV's `cv2.SIFT_create()` function to detect keypoints and compute descriptors.

[ ]:
```
# OpenCV Implementation
img = cv.imread('assets/king.jpg')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Create SIFT detector
sift = cv.SIFT_create()
# Detect keypoints and descriptors
keypoints, descriptors = sift.detectAndCompute(gray, None)

# Draw keypoints on the image
img_with_keypoints = cv.drawKeypoints(img, keypoints, None, flags=cv.
 ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
# kp_img = cv.drawKeypoints(img, kp, None, flags=cv.
 ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Displaying using matplotlib
```

```python
plt.figure(figsize=(20, 18))
plt.subplot(121),plt.imshow(img,'gray')
plt.title('Original Image')
plt.subplot(122),plt.imshow(img_with_keypoints, cmap='gray')
plt.title('SIFT Keypoints (OpenCV)')
plt.show()
```



[ ]:
```python
print(f'The no of keypoints:{len(keypoints)}')
print(f'The shape of descriptor:{descriptors.shape})')

print("Visualise Randow idx")
print(descriptors[0])
```

```
The no of keypoints:43
The shape of descriptor:(43, 128))
Visualise Randow idx
[ 11.   6.   5.  11.   2.   6.  10.  21. 129.  69.   0.   0.   1.  12.
  19.  10. 163. 133.   0.   0.   0.   0.   0.   1.  17.   7.   0.   0.
   0.   0.   0.   0.   2.   0.   0.   0.   0.  34. 124.  33.  79.  13.
   1.   4.   1.  22.  76.  32. 163.  57.   0.   0.   0.   0.  10.  65.
  54.   6.   0.   0.   0.   0.   0.   4.   0.   0.   0.   0.   0. 109.
 163.   0.  10.   0.   0.   1.   0.  56. 163.  75. 102.   1.   0.   0.
   0.   0.  99. 163.  19.   0.   0.   0.   0.   0.   0.  14.   0.   0.
   0.   0.   0.   9.  25.   0.   0.   0.   0.   0.   0.   5.  62.   7.
   0.   0.   0.   0.   0.   0.  20.  16.   0.   0.   0.   0.   0.   0.
   0.   1.]
```

[ ]:
```python
# Finding Keypoint information
index = 2
print(f'location:{keypoints[index].pt}')
print(f'angle:{keypoints[index].angle}')
print(f'response:{keypoints[index].response}')
```

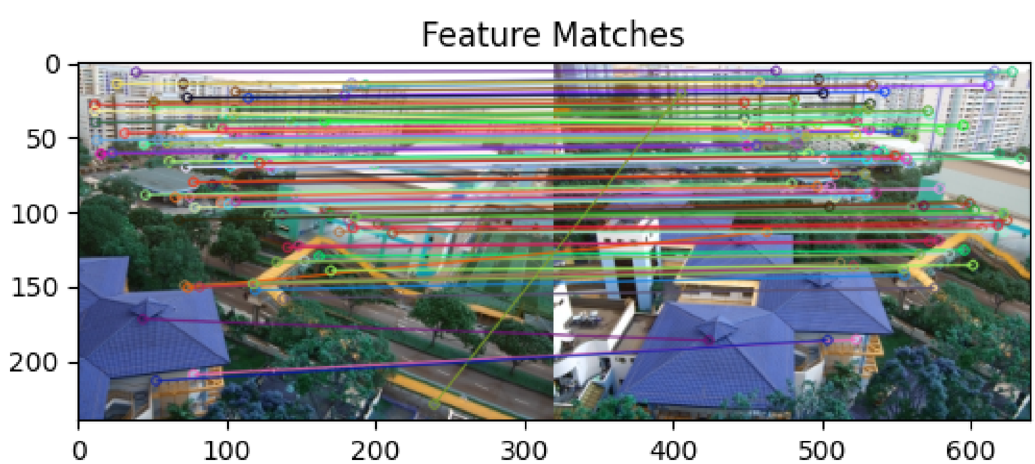
```
location:(131.6247100830078, 107.39229583740234)
angle:200.12496948242188
response:0.04201919585466385
```

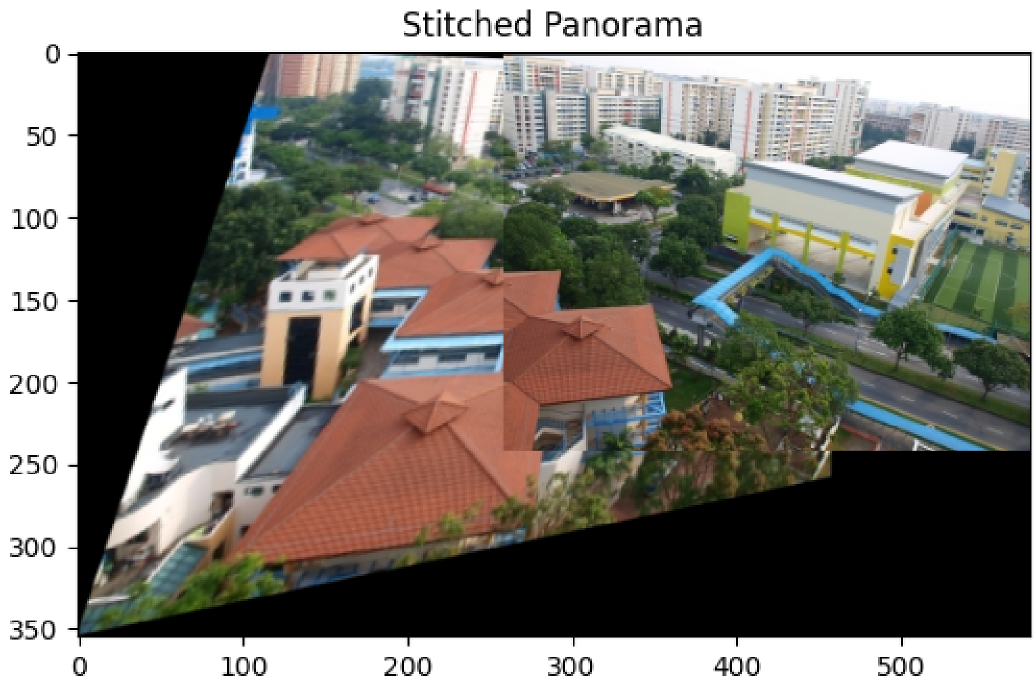
2.2 Panorama Stitching using SIFT

[ ]:
```python
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

class PanoramaStitcher:
    def __init__(self):
        # Initialize SIFT detector
        self.sift = cv.SIFT_create()

    def detect_and_compute(self, img):
        """
        Detect keypoints and compute descriptors for the given image.
        """
        # Convert the image to grayscale
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        # Detect keypoints and compute descriptors
        keypoints, descriptors = self.sift.detectAndCompute(gray, None)

        # Return keypoints and descriptors
        return keypoints, descriptors

    def match_features(self, descriptors1, descriptors2):
        """
        Match features between two sets of descriptors.
        """
        # Initialize BFMatcher with default parameters
        bf = cv.BFMatcher(cv.NORM_L2, crossCheck=True)

        # Match descriptors
        matches = bf.match(descriptors1, descriptors2)

        # Sort them in the order of their distance (best matches first)
        matches = sorted(matches, key=lambda x: x.distance)

        return matches

    def stitch_images(self, img1, img2):
        """
        Stitch two images together based on feature matching.
        """
        # Detect and compute keypoints and descriptors for both images
        keypoints1, descriptors1 = self.detect_and_compute(img1)
        keypoints2, descriptors2 = self.detect_and_compute(img2)
```

```
        # Match features between the two sets of descriptors
        matches = self.match_features(descriptors1, descriptors2)

        # Visualize matches between the two images
        img_matches = cv.drawMatches(img1, keypoints1, img2, keypoints2,␣
↪matches[:100], None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        plt.imshow(img_matches)
        plt.title("Feature Matches")
        plt.show()

        # Extract location of good matches (keypoints from both images)
        points1 = np.zeros((len(matches), 2), dtype=np.float32)
        points2 = np.zeros((len(matches), 2), dtype=np.float32)

        for i, match in enumerate(matches):
            points1[i, :] = keypoints1[match.queryIdx].pt
            points2[i, :] = keypoints2[match.trainIdx].pt

        # Calculate the Homography matrix between the two sets of points
        H, mask = cv.findHomography(points2, points1, cv.RANSAC)

        # Warp the second image to align with the first image
        height1, width1, _ = img1.shape
        height2, width2, _ = img2.shape

        # Determine the size of the final panorama by warping img2 and␣
↪calculating dimensions
        corners_img2 = np.array([[0, 0], [0, height2], [width2, height2],␣
↪[width2, 0]], dtype=np.float32)
        warped_corners_img2 = cv.perspectiveTransform(np.array([corners_img2]),␣
↪H)[0]

        # Calculate the size of the resulting stitched image
        min_x = int(min(0, warped_corners_img2[:, 0].min()))
        min_y = int(min(0, warped_corners_img2[:, 1].min()))
        max_x = int(max(width1, warped_corners_img2[:, 0].max()))
        max_y = int(max(height1, warped_corners_img2[:, 1].max()))

        # Offset translation to shift everything to positive coordinates if␣
↪needed
        translation = np.array([[1, 0, -min_x], [0, 1, -min_y], [0, 0, 1]])

        # Adjust the homography to consider the translation
        H_translation = np.dot(translation, H)

        # Warp img2 to create the panorama
        panorama_width = int(max_x - min_x)
```

```
        panorama_height = int(max_y - min_y)
        img2_aligned = cv.warpPerspective(img2, H_translation, (panorama_width,␣
↪panorama_height))

        # Paste img1 into the resulting panorama
        img2_aligned[-min_y:height1 - min_y, -min_x:width1 - min_x] = img1

        # Return the stitched image (panorama)
        return img2_aligned


# Load your two images with overlapping fields of view
img1 = cv.imread('assets/a.jpg')
img2 = cv.imread('assets/b.jpg')

# Initialize the panorama stitcher
stitcher = PanoramaStitcher()

# Stitch the two images together
panorama = stitcher.stitch_images(img1, img2)

# Show the final stitched panorama
plt.imshow(cv.cvtColor(panorama, cv.COLOR_BGR2RGB))
plt.title("Stitched Panorama")
plt.show()
```





3 TAKE HOME EXERCISE

3.1 Hybrid Algorithm Exercise: [20 points]

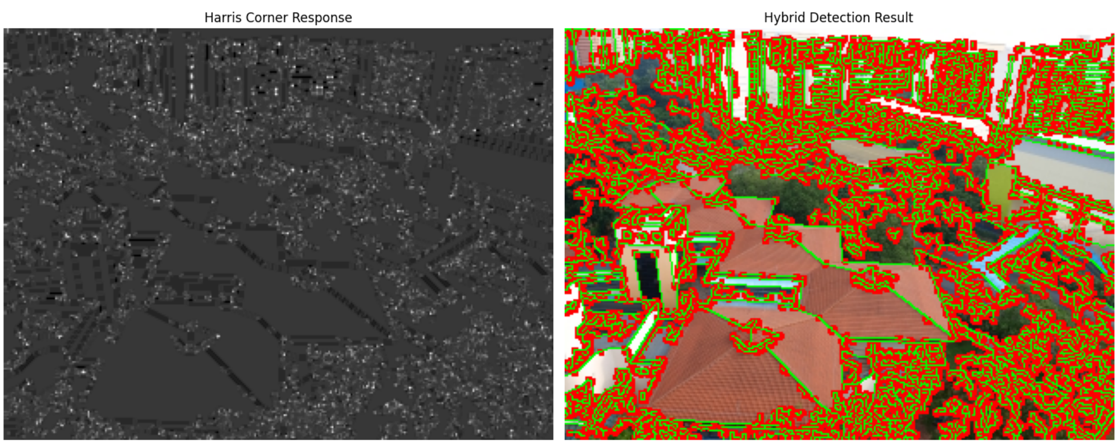
Combine Canny edge detection and Harris corner detection to first detect edges and then identify corners on those edges. This approach can be useful for detecting specific types of features in an image. Use these for feature detection in street signs, buildings, or other structures in urban images.

Hint: Apply the Canny detector, then run the Harris corner detector on the resulting edge map instead of the original image.

[ ]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt

class HybridCannyHarrisDetector:
    def __init__(self, img, canny_low_thresh=100, canny_high_thresh=200,
 ↪harris_block_size=2, harris_ksize=3, harris_k=0.05):
        self.original_img = img
        self.gray_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        self.canny_low_thresh = canny_low_thresh
        self.canny_high_thresh = canny_high_thresh
        self.harris_block_size = harris_block_size
        self.harris_ksize = harris_ksize
```

```python
        self.harris_k = harris_k
        self.edges = None
        self.corners = None

    def detect_edges(self):
        self.edges = cv.Canny(self.gray_img, self.canny_low_thresh, self.
↪canny_high_thresh)
        return self.edges

    def detect_corners(self):
        edges_float = self.edges.astype(np.float32)

        # Apply Harris corner detection on the edge map
        self.corners = cv.cornerHarris(edges_float, self.harris_block_size,␣
↪self.harris_ksize, self.harris_k)
        return self.corners

    def apply_hybrid_detection(self):
        self.detect_edges()
        self.detect_corners()

    def visualize_results(self, corner_threshold=0.01):
        # Create a copy of the original image for visualization
        result_img = self.original_img.copy()

        # Dilate corner response for better visibility
        dilated_corners = cv.dilate(self.corners, None)

        # Threshold for corner detection
        threshold = corner_threshold * dilated_corners.max()

        # Mark corners on the image
        result_img[dilated_corners > threshold] = [0, 0, 255]  # Red color for␣
↪corners

        # Overlay edges in green
        result_img[self.edges != 0] = [0, 255, 0]  # Green color for edges

        # Plot results
        fig, axs = plt.subplots(2, 2, figsize=(15, 15))
        axs[0, 0].imshow(cv.cvtColor(self.original_img, cv.COLOR_BGR2RGB))
        axs[0, 0].set_title('Original Image')
        axs[0, 1].imshow(self.edges, cmap='gray')
        axs[0, 1].set_title('Canny Edges')
        axs[1, 0].imshow(self.corners, cmap='gray')
        axs[1, 0].set_title('Harris Corner Response')
        axs[1, 1].imshow(cv.cvtColor(result_img, cv.COLOR_BGR2RGB))
```

```
        axs[1, 1].set_title('Hybrid Detection Result')

        for ax in axs.flat:
            ax.axis('off')

        plt.tight_layout()
        plt.show()

        return result_img

    def mark_corners(self, threshold=0.01):
        # Create a copy of the original image for marking corners
        img_with_corners = np.copy(self.original_img)

        # Dilate corner response for marking
        dilated_corners = cv.dilate(self.corners, None)

        # Threshold for corner detection
        corner_threshold = threshold * dilated_corners.max()

        # Mark corners on the image
        img_with_corners[dilated_corners > corner_threshold] = [255, 0, 0]  #
 ↪Red color for corners

        return img_with_corners

img = cv.imread('assets/b.jpg')
hybrid_detector = HybridCannyHarrisDetector(img)
hybrid_detector.apply_hybrid_detection()
result = hybrid_detector.visualize_results()
```



3.2 Panoroma Stiching: [20 points]

Combine 3 images, modify the above class to accept 3 image and perform stiching operation to get a panoromic image.

Use images - assests/a.jpg, - assets/b.jpg, - assets/c.jpg

[ ]:
```
## your code goes here!!
```

[ ]:
```

```