# SIFT-2

September 13, 2024

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.

## 0.1 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

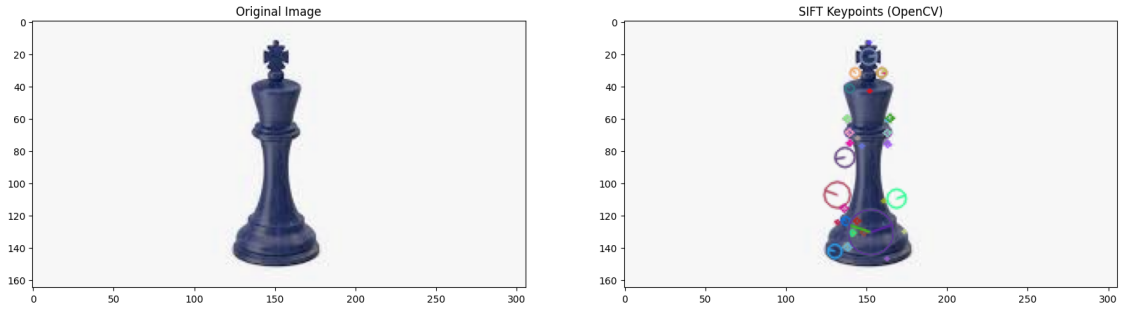
[1]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
```

### 0.1.1 OpenCV SIFT

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

[3]:
```
# 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, keypoints, None, flags=cv.
 ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Displaying using matplotlib
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()
```




[4]:
```
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.]
```

[5]:
```
# Finding Keypoint information
index = 2
print(f'location:{keypoints[index].pt}')
print(f'angle:{keypoints[index].angle}')
print(f'response:{keypoints[index].response}')
```

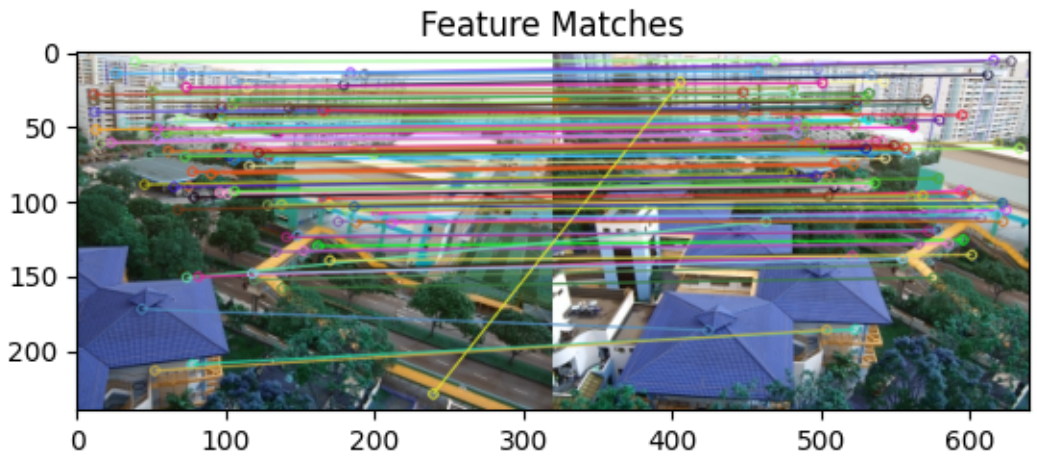
```
location:(131.6247100830078, 107.39229583740234)
angle:200.12496948242188
response:0.04201919585466385
```

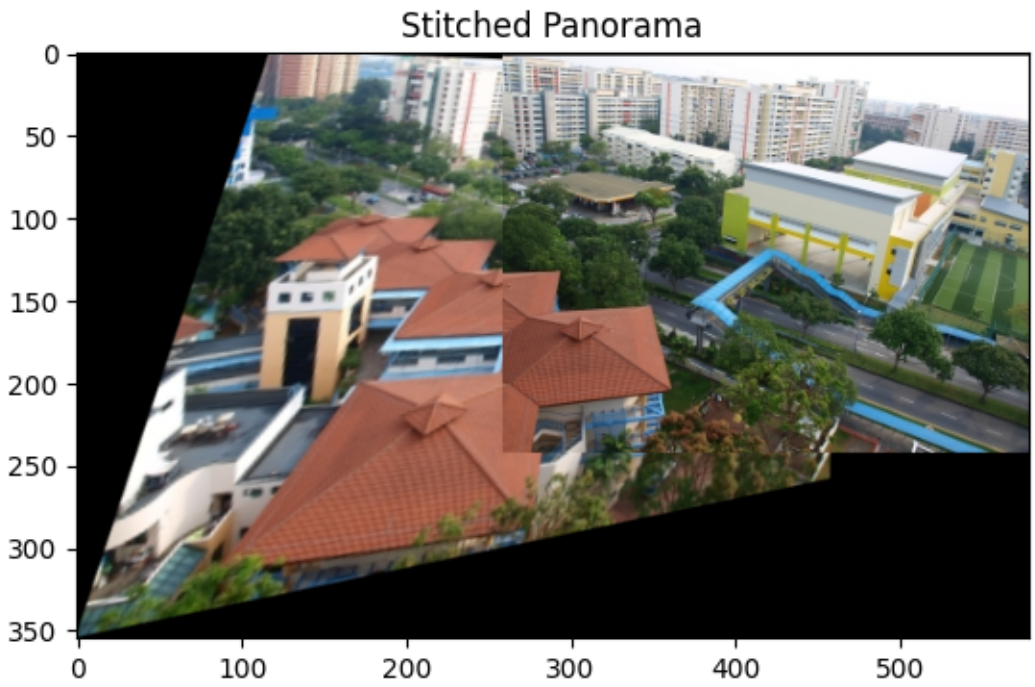
## 0.2 Panorama Stitching using SIFT

[6]:
```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()
```






# 1 TAKE HOME EXERCISE

## 1.1 Hybrid Algorithm Exercise: [20 points]

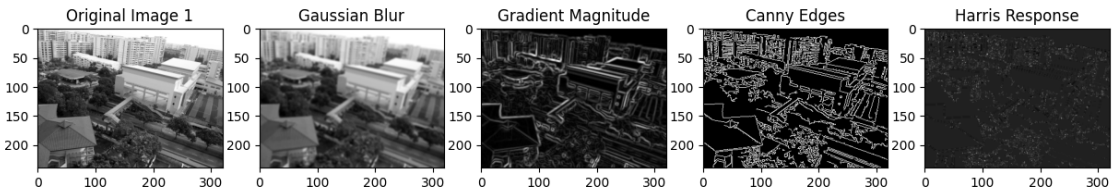
Combine Canny edge detection and Harris corner detection to first detect edges and then identify corners on those edges. This approach can be useful for detecting specific types of features in an image. Use these for feature detection in street signs, buildings, or other structures in urban images.

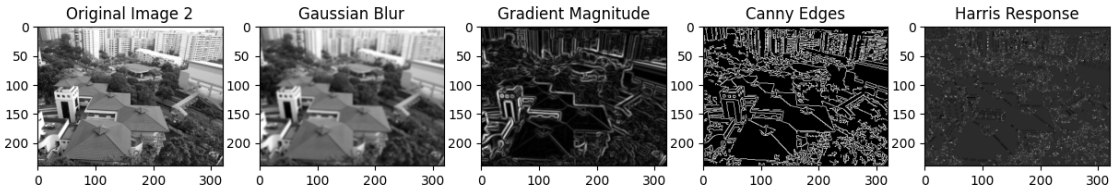
Hint: Apply the Canny detector, then run the Harris corner detector on the resulting edge map instead of the original image.

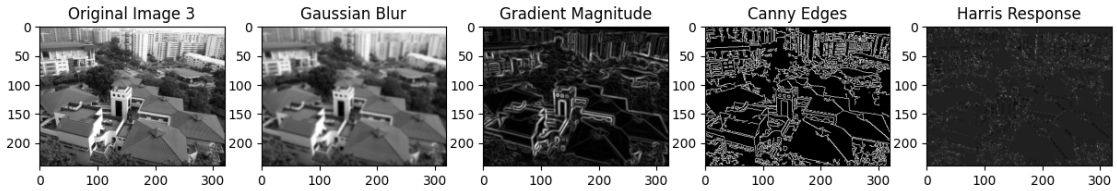
[7]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt

class EdgeCornerDetector:
    def __init__(self, images, low_thresh, high_thresh, block_size=2, ksize=3,␣
 ↪k=0.04):
        self.images = images
        self.low_thresh = low_thresh
        self.high_thresh = high_thresh
        self.block_size = block_size
        self.ksize = ksize
        self.k = k
```

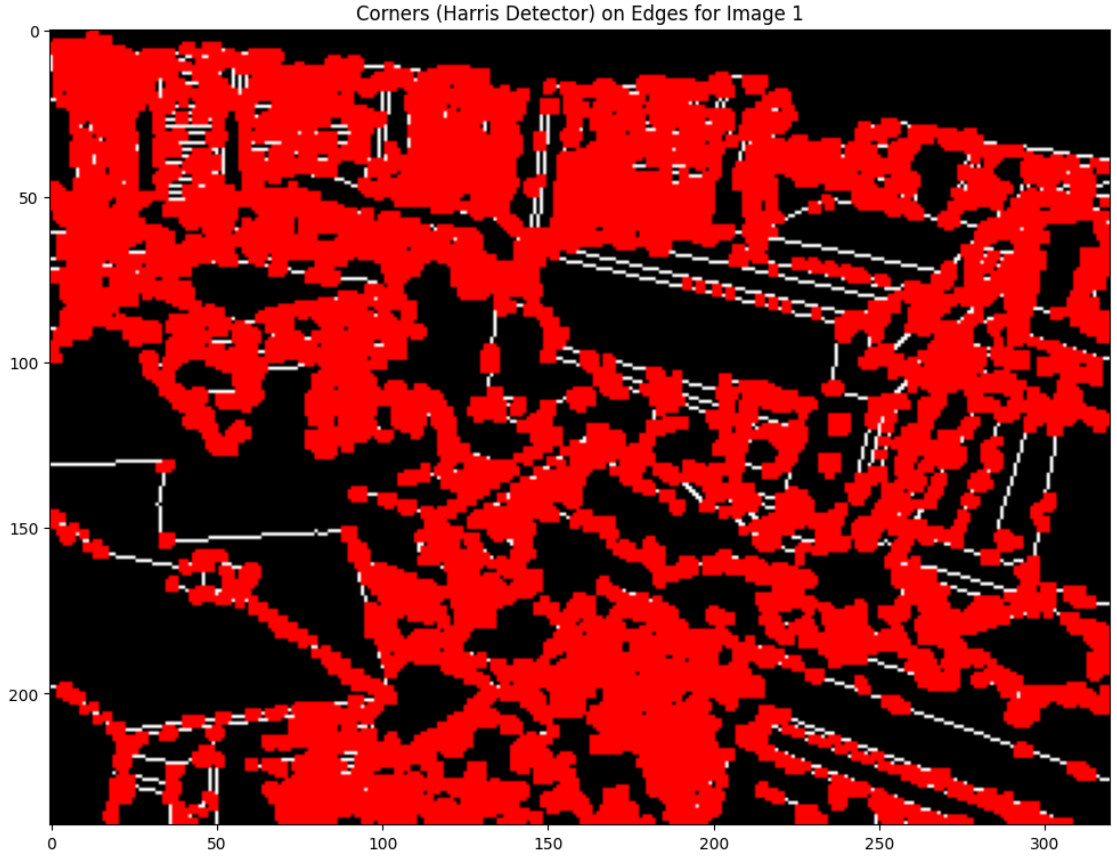
```
        self.results = []

    def process_images(self):
        for img in self.images:
            result = {}
            img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
            result['img_gray'] = img_gray

            gaussian_blur = cv.GaussianBlur(img_gray, (5, 5), 1.4)
            result['gaussian_blur'] = gaussian_blur

            gx = cv.Sobel(gaussian_blur, cv.CV_64F, 1, 0, ksize=3)
            gy = cv.Sobel(gaussian_blur, cv.CV_64F, 0, 1, ksize=3)
            gradient_magnitude = np.sqrt(gx**2 + gy**2)
            result['gradient_magnitude'] = gradient_magnitude

            edges = cv.Canny(img_gray, self.low_thresh, self.high_thresh)
            result['edges'] = edges

            harris_response = cv.cornerHarris(edges, self.block_size, self.
↪ksize, self.k)
            result['harris_response'] = harris_response

            self.results.append(result)

    def plot_intermediate_steps(self):
        for i, result in enumerate(self.results):
            fig, ax = plt.subplots(1, 5, figsize=(15, 5))
            ax[0].imshow(result['img_gray'], cmap='gray')
            ax[0].set_title(f'Original Image {i+1}')
            ax[1].imshow(result['gaussian_blur'], cmap='gray')
            ax[1].set_title('Gaussian Blur')
            ax[2].imshow(result['gradient_magnitude'], cmap='gray')
            ax[2].set_title('Gradient Magnitude')
            ax[3].imshow(result['edges'], cmap='gray')
            ax[3].set_title('Canny Edges')
            ax[4].imshow(result['harris_response'], cmap='gray')
            ax[4].set_title('Harris Response')
            plt.show()

    def mark_corners(self, threshold=0.01):
        for i, result in enumerate(self.results):
            img_with_corners = cv.cvtColor(np.copy(result['edges']), cv.
↪COLOR_GRAY2BGR)
            harris_response = result['harris_response']

            # Dilate the Harris response to mark the corners more clearly
```

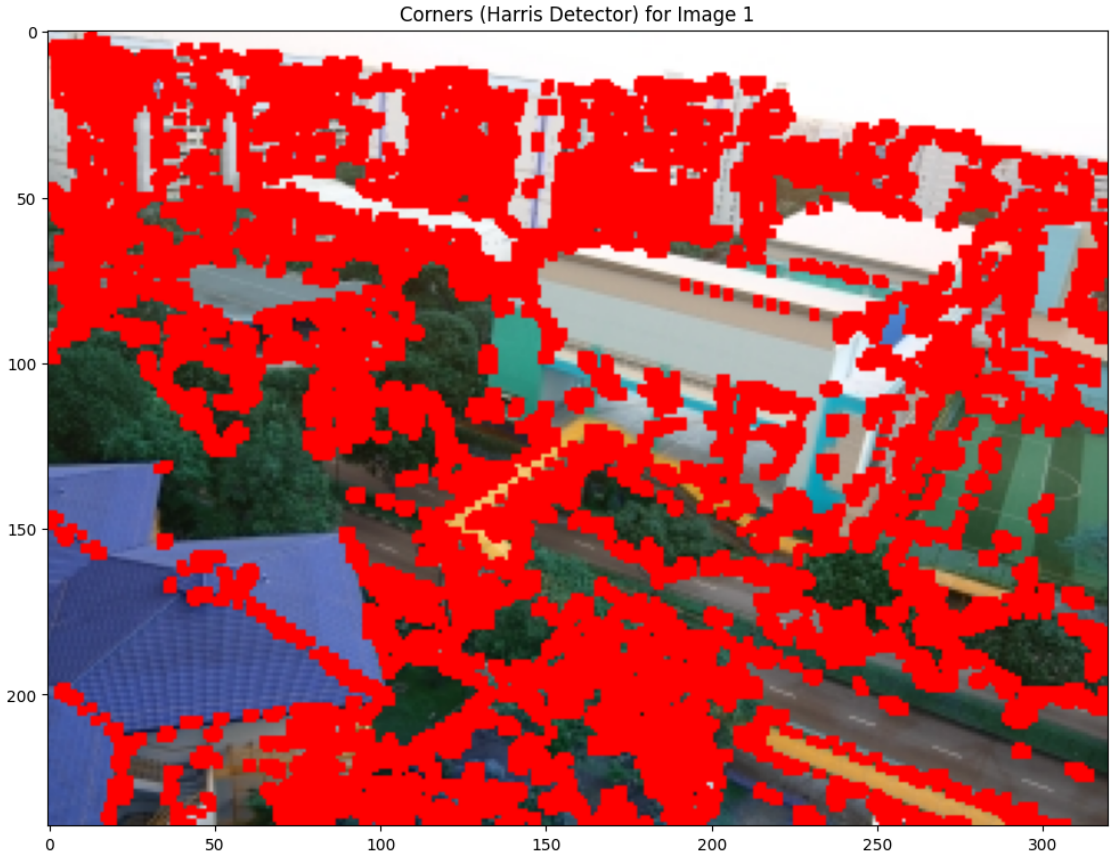
```
            harris_response = cv.dilate(harris_response, None)

            # Mark the corners in red where the Harris response exceeds the␣
 ↪threshold
            img_with_corners[harris_response > threshold * harris_response.
 ↪max()] = [255, 0, 0]

            plt.figure(figsize=(12, 12))
            plt.imshow(img_with_corners)
            plt.title(f'Corners (Harris Detector) on Edges for Image {i+1}')
            plt.show()

            img_copy = np.copy(self.images[i])
            img_copy[harris_response > threshold * harris_response.max()] =␣
 ↪[255, 0, 0]

            plt.figure(figsize=(12, 12))
            plt.imshow(img_copy)
            plt.title(f'Corners (Harris Detector) for Image {i+1}')
            plt.show()

            self.results[i]['img_with_corners'] = img_with_corners

# Example usage with multiple images
images = [cv.imread('assets/a.jpg'), cv.imread('assets/b.jpg'), cv.
 ↪imread('assets/c.jpg')]  # Add more images as needed
edge_corner = EdgeCornerDetector(images, 100, 250)
edge_corner.process_images()
edge_corner.plot_intermediate_steps()  # Plot for all images
edge_corner.mark_corners()             # Mark corners for all images
```

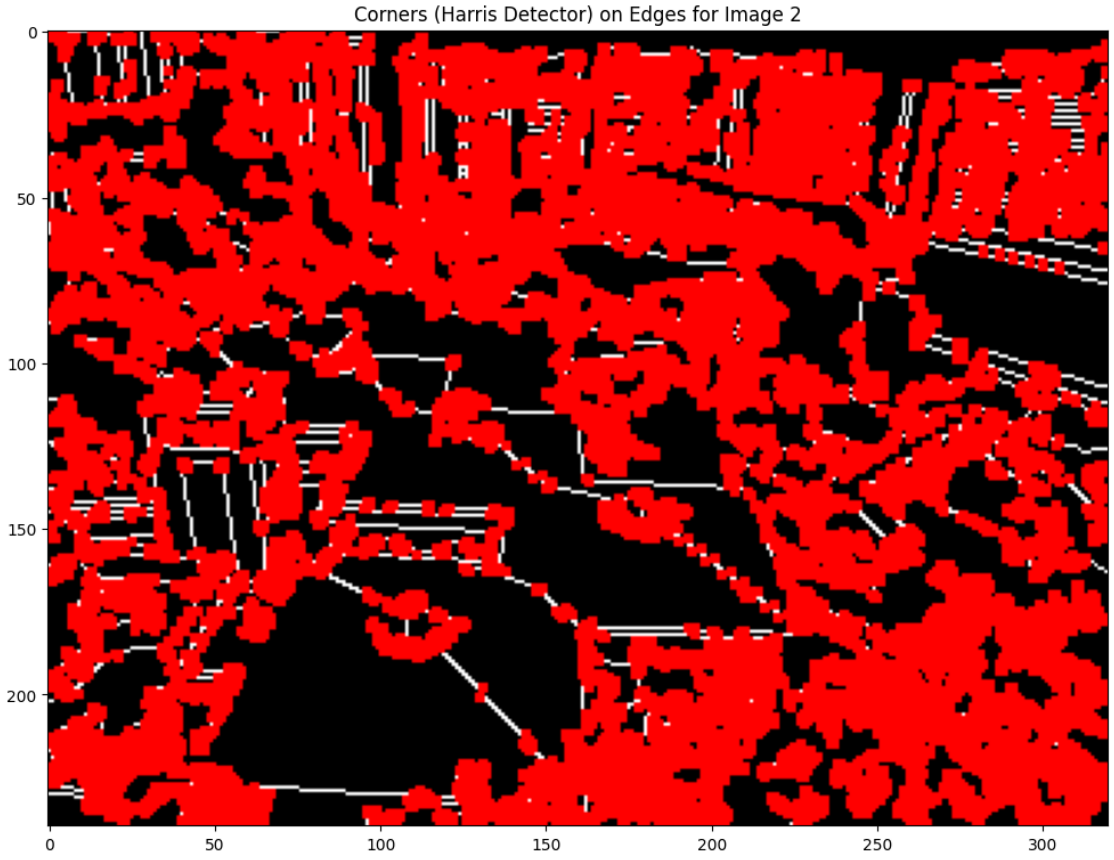


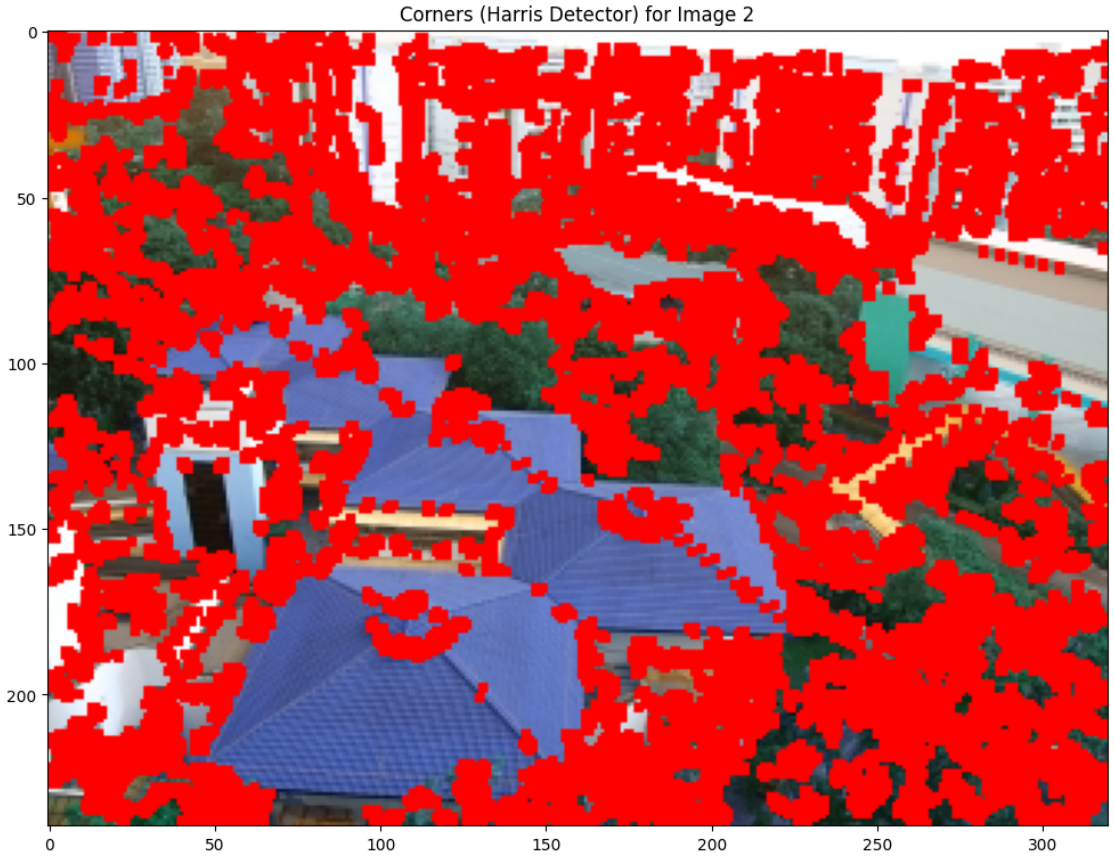



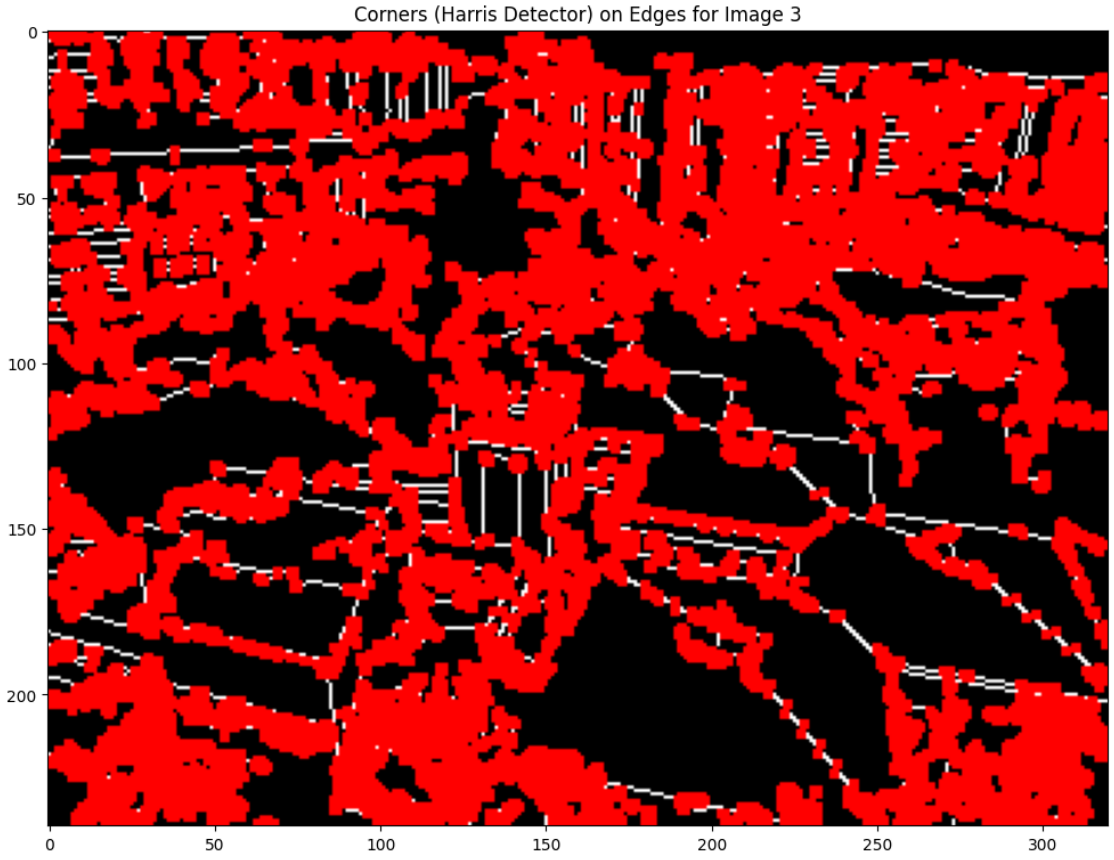

Original Image 3
Gaussian Blur
Gradient Magnitude
Canny Edges
Harris Response

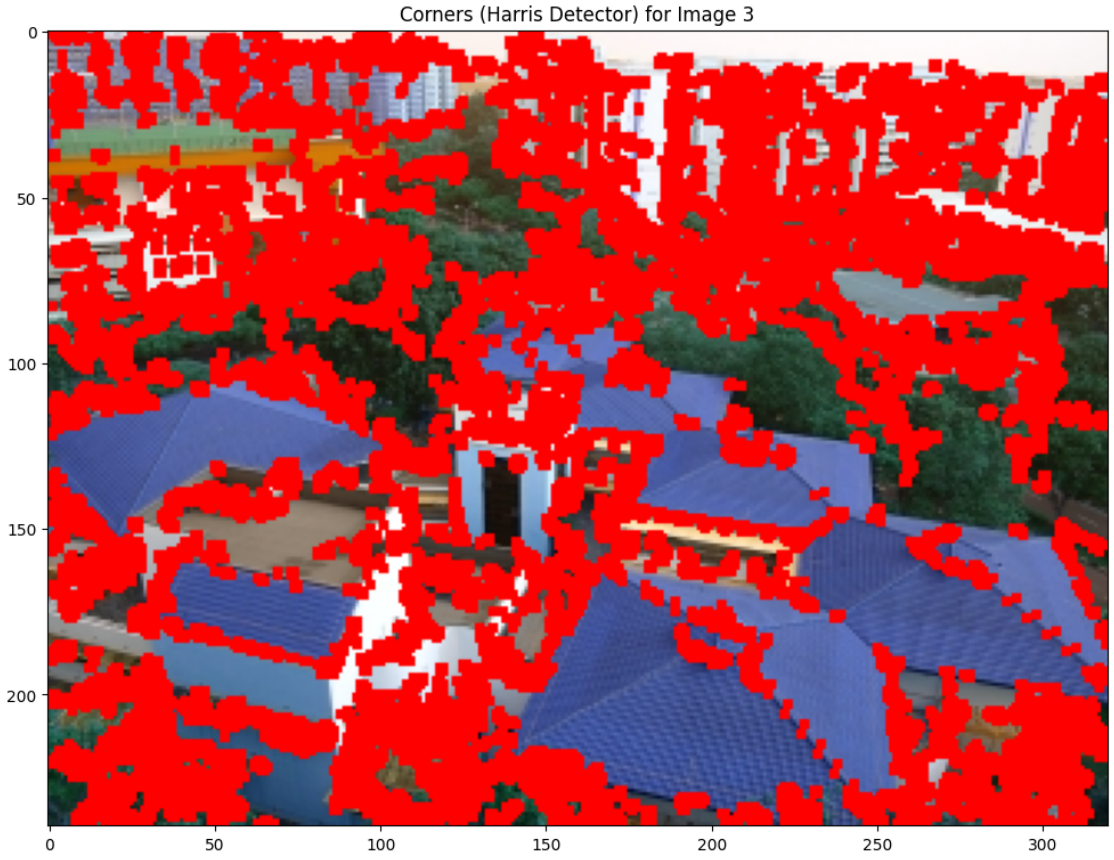

Corners (Harris Detector) on Edges for Image 1


Corners (Harris Detector) for Image 1
0
50
100
150
200
0
50
100
150
200
250
300


Corners (Harris Detector) on Edges for Image 2
0
50
100
150
200
0
50
100
150
200
250
300


Corners (Harris Detector) for Image 2
0
50
100
150
200
0
50
100
150
200
250
300


Corners (Harris Detector) on Edges for Image 3
0
50
100
150
200
0
50
100
150
200
250
300




### 1.2 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

[8]:
```
## your code goes here!!
```

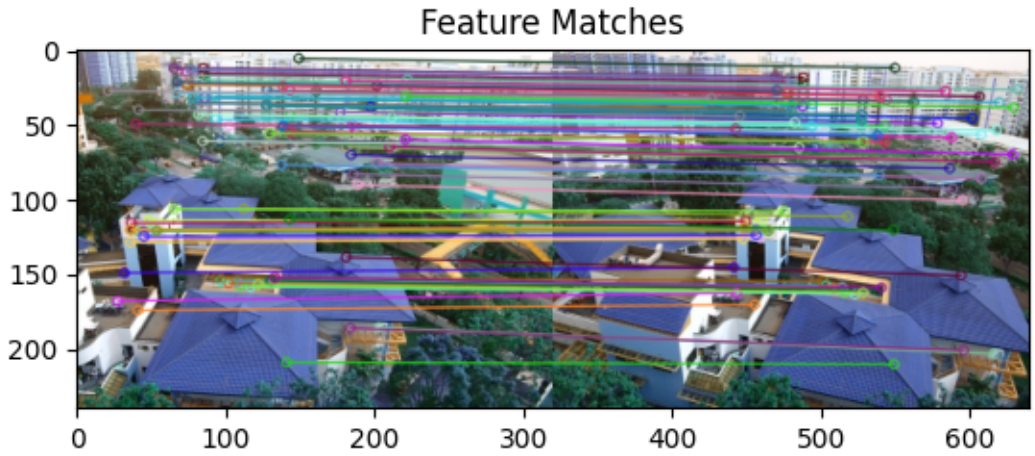
[9]:
```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

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 visualize_matches(self, img1, img2, keypoints1, keypoints2, matches):
        """
        Visualize the feature matches between two images.
        """
        img_matches = cv.drawMatches(img1, keypoints1, img2, keypoints2,␣
↪matches[:100], None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

        # Show the matches
        plt.imshow(img_matches)
        plt.title("Feature Matches")
        plt.show()

    def stitch_pair(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
        self.visualize_matches(img1, img2, keypoints1, keypoints2, matches)

        # 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

    def stitch_three_images(self, img1, img2, img3):
        """
        Stitch three images together to create a panoramic image.
        """
        print("Stitching img1 and img2...")
        result1_2 = self.stitch_pair(img1, img2)

        print("Stitching (img1 + img2) and img3...")
        panorama = self.stitch_pair(result1_2, img3)

        return panorama


img1 = cv.imread('assets/a.jpg')
img2 = cv.imread('assets/b.jpg')
img3 = cv.imread('assets/c.jpg')

# Initialize the panorama stitcher
stitcher = PanoramaStitcher()

# Stitch the three images together
panorama = stitcher.stitch_three_images(img2, img3, img1)

# Show the final stitched panorama
plt.imshow(cv.cvtColor(panorama, cv.COLOR_BGR2RGB))
plt.title("Stitched Panorama with 3 Images")
plt.show()
```

```
Stitching img1 and img2...
```



```
Stitching (img1 + img2) and img3…
```