CV TA-2

Name : Priyank Maheshwari

Section : A

Roll No. : 52

Github link : https://github.com/Priyank2310/Computer-Vision-Assignment

1. **Implement the SIFT algorithm to detect and match key points between two images.**

   Steps:

   1. Load Images & Convert to Grayscale
   2. Detect Keypoints & Descriptors using SIFT
      - SIFT finds distinctive, scale- and rotation-invariant keypoints.
      - Each keypoint has a 128-dimensional descriptor.
   3. Match Descriptors
      - Use Brute-Force or FLANN matcher to compare descriptors.
      - Apply Lowe's ratio test to keep only good matches (ratio < 0.75).
   4. Draw Matches
      - Use cv2.drawMatches() to visualize matching keypoints between images.

   Applications:
   Used in image stitching, object recognition, and 3D reconstruction.

Outputs :

```
sift = cv2.SIFT_create()
```

```
kp1 , ds1 = sift.detectAndCompute(img1,None)
kp2 , ds2 = sift.detectAndCompute(img2,None)
```

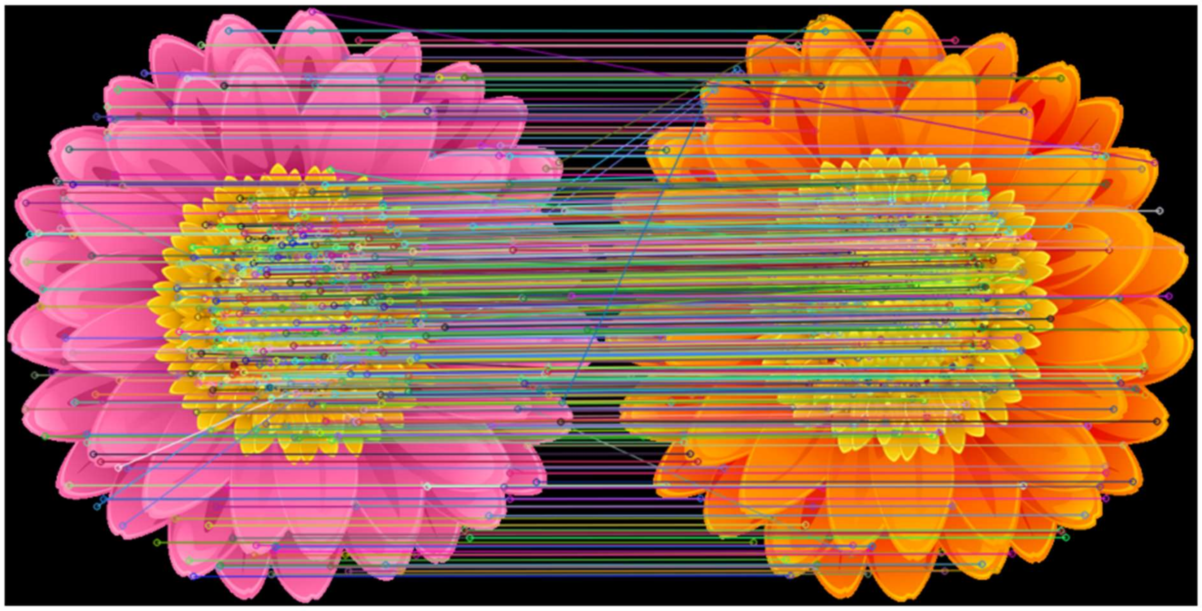
```
bf = cv2.BFMatcher()
matches = bf.knnMatch(ds1,ds2,k=2)
```

```
good_matches = []

for m,n in matches :
    if m.distance < 0.6 * n.distance :
        good_matches.append(m)
```

```
img_matched = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

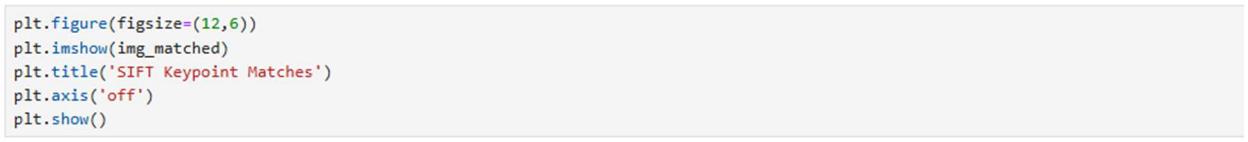
```
plt.figure(figsize=(12,6))
plt.imshow(img_matched)
plt.title('SIFT Keypoint Matches')
plt.axis('off')
plt.show()
```

SIFT Keypoint Matches



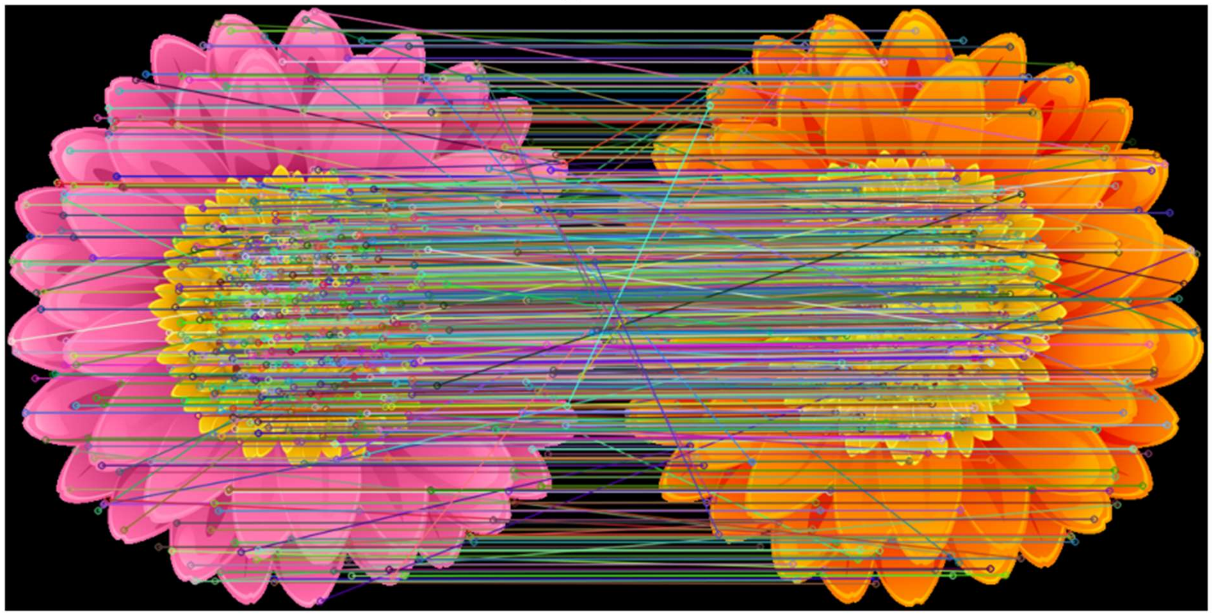
```
good_matches = []

for m,n in matches :
    if m.distance < 0.8 * n.distance :
        good_matches.append(m)
```

```
img_matched = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```



```
plt.figure(figsize=(12,6))
plt.imshow(img_matched)
plt.title('SIFT Keypoint Matches')
plt.axis('off')
plt.show()
```

SIFT Keypoint Matches



Conclusion :

As the ratio threshold was increased from 0.60 to 0.80, a noticeable rise in the number of matched keypoints was observed. While the higher threshold allowed for detecting more correspondences, it also introduced several incorrect or weak matches, resulting in a denser but less reliable visualization. On the other hand, the stricter threshold (0.60) yielded fewer matches but with better precision, highlighting only the most confident keypoint pairs. This demonstrates the trade-off between match quantity and match quality when tuning the ratio parameter in SIFT.

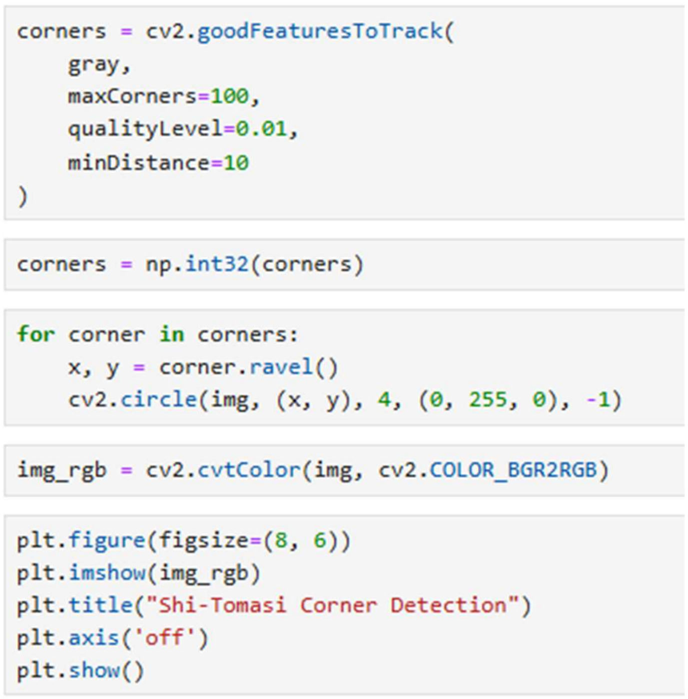
2. Use the Shi-Tomasi corner detector to identify and mark corner points in an image.

Steps:

1. Load Image & Convert to Grayscale
   - Corner detection works best on single-channel (grayscale) images.
2. Detect Corners using Shi-Tomasi
   - Use cv2.goodFeaturesToTrack() function.
   - Based on the minimum eigenvalue of the image patch's gradient matrix.
   - Detects strong, stable corners.
3. Mark the Detected Corners
   - Draw circles or dots at detected corner positions using cv2.circle() or similar.

Applications:
Used in object tracking, motion detection, video surveillance, and feature matching in computer vision tasks.

Outputs :



```
corners = cv2.goodFeaturesToTrack(
    gray,
    maxCorners=100,
    qualityLevel=0.01,
    minDistance=10
)
```

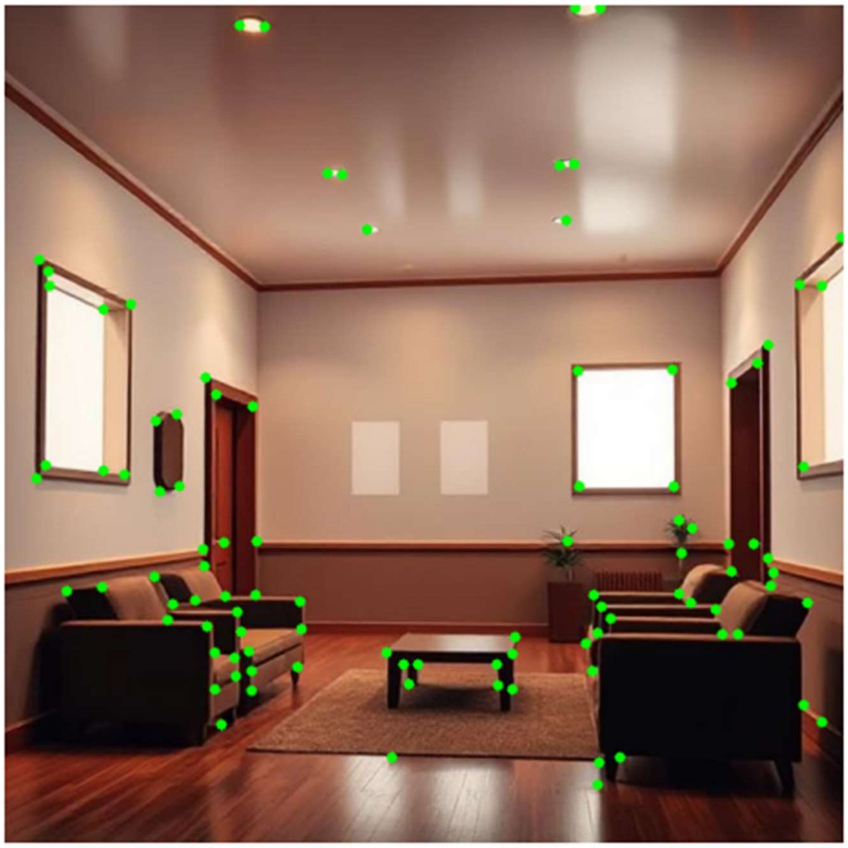
```
corners = np.int32(corners)
```

```
for corner in corners:
    x, y = corner.ravel()
    cv2.circle(img, (x, y), 4, (0, 255, 0), -1)
```

```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
plt.figure(figsize=(8, 6))
plt.imshow(img_rgb)
plt.title("Shi-Tomasi Corner Detection")
plt.axis('off')
plt.show()
```

Shi-Tomasi Corner Detection



Changing the parameters and observing changes .

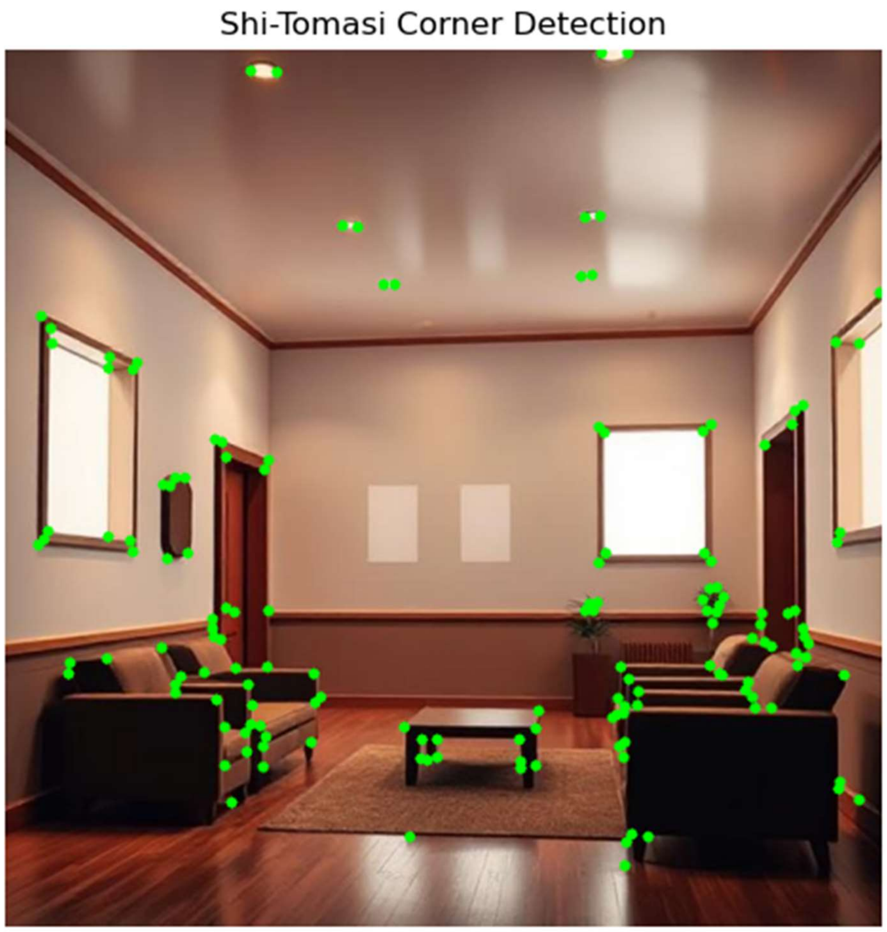
```
corners = cv2.goodFeaturesToTrack(
    gray,
    maxCorners=150,
    qualityLevel=0.02,
    minDistance=5
)
```

```
corners = np.int32(corners)
```

```
for corner in corners:
    x, y = corner.ravel()
    cv2.circle(img, (x, y), 4, (0, 255, 0), -1)
```

```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
plt.figure(figsize=(8, 6))
plt.imshow(img_rgb)
plt.title("Shi-Tomasi Corner Detection")
plt.axis('off')
plt.show()
```



Conclusion :

As the maxCorners was increased from 100 to 150 and minDistance was decreased from 10 to 5, a noticeable increase in the number of detected corner points was observed. The updated parameters enabled detection of more densely packed corners, especially in regions with fine textures and repeated structures. However, while the higher corner count offered a richer set of features, it also introduced more closely located and potentially redundant points, which may not contribute significantly to uniqueness or robustness in further processing. Conversely, the original settings produced fewer but more spatially distinct corners, favoring precision and clarity in detection. This highlights the trade-off between feature density and feature distinctiveness when tuning parameters in Shi-Tomasi corner detection.¶

3 . Detect and mark corner points in an image using the FAST corner detection algorithm.

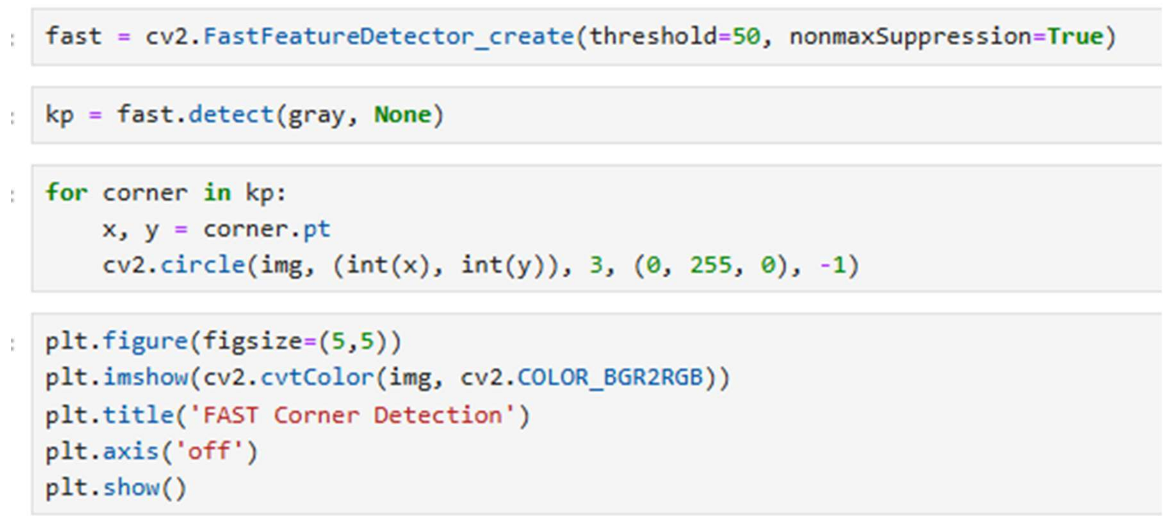
Steps:

1. Load Image & Convert to Grayscale
   - FAST (Features from Accelerated Segment Test) works on grayscale images for better performance.
2. Detect Corners using FAST
   - Use cv2.FastFeatureDetector_create() in OpenCV.
   - FAST detects corners by comparing the intensity of a pixel to a circle of 16 pixels around it.
   - A corner is detected if a set number of contiguous pixels in the circle are all brighter or all darker than the center pixel.
3. Mark the Detected Corners
   - Use cv2.drawKeypoints() to visualize corner points on the image.

Applications:

Used in real-time vision tasks like SLAM (Simultaneous Localization and Mapping), mobile robotics, and tracking—especially where speed is critical.

Outputs :

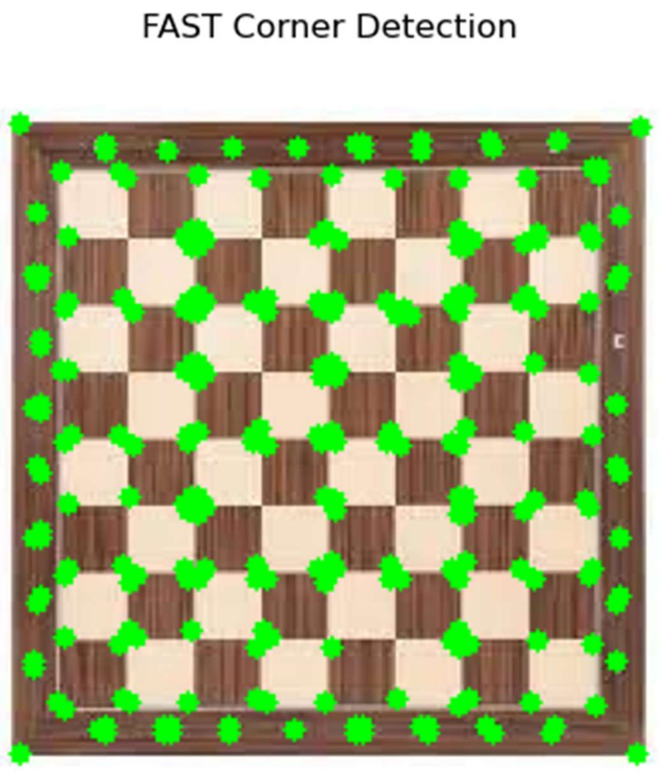


```
fast = cv2.FastFeatureDetector_create(threshold=50, nonmaxSuppression=True)
```

```
kp = fast.detect(gray, None)
```

```
for corner in kp:
    x, y = corner.pt
    cv2.circle(img, (int(x), int(y)), 3, (0, 255, 0), -1)
```

```
plt.figure(figsize=(5,5))
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('FAST Corner Detection')
plt.axis('off')
plt.show()
```



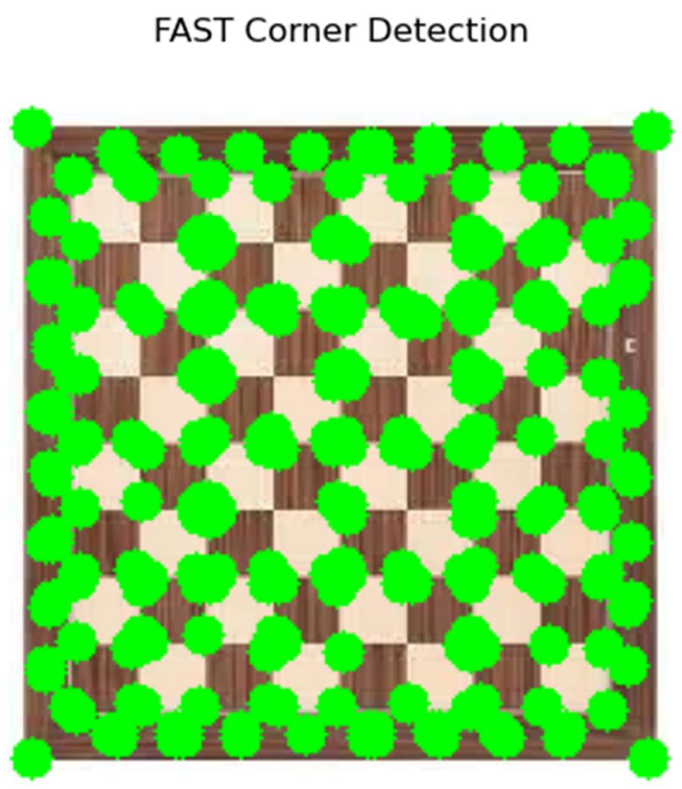
Changing the parameters and observing changes .

```
fast = cv2.FastFeatureDetector_create(threshold=50, nonmaxSuppression=True)
```

```
kp = fast.detect(gray, None)
```

```
for corner in kp:
    x, y = corner.pt
    cv2.circle(img, (int(x), int(y)), 6, (0, 255, 0), -1)
```

```
plt.figure(figsize=(5,5))
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('FAST Corner Detection')
plt.axis('off')
plt.show()
```



Conclusion :

As the circle radius used for marking keypoints was decreased from 6 to 3, a clearer and more refined visualization of corner points was achieved. The updated visualization made it easier to observe individual corner detections without excessive overlap, especially in densely packed regions such as checkerboard intersections. While the larger radius provided a more prominent and visually striking highlight of keypoints, it often obscured neighboring features, making it harder to distinguish closely located corners. In contrast, the smaller radius improved interpretability and precision in assessing detection accuracy. This emphasizes the importance of appropriate visualization parameters to balance visibility and clarity when evaluating feature detection results.