Computer Vision

TA-2 -Mini Project

Name : **Prathmesh Rewatkar**

Roll no : **50**

Section : A

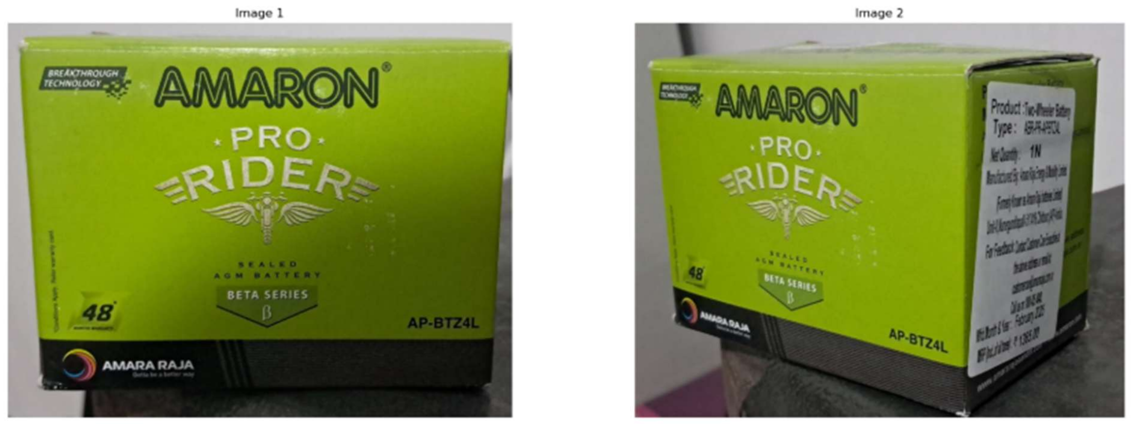
Software Used : Jupyter Notebook

Output Code Screenshot:

SIFT Algorithm:

Aim : Implement the SIFT algorithm to detect and match key points between two images.

Loaded Image:



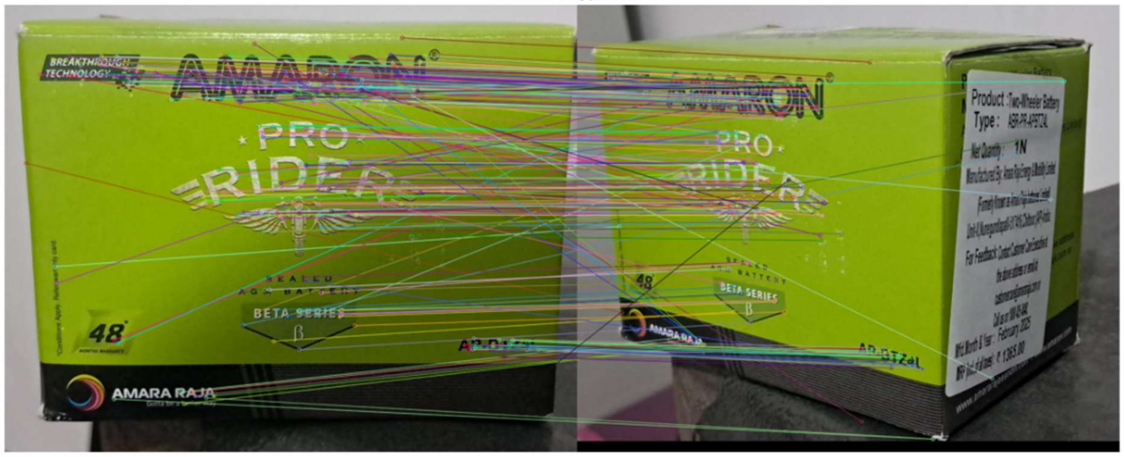
SIFT Feature Matching

```python
def sift(img1_path, img2_path, n_features = 0, contrastThreshold = 0.04, edgeThreshold = 10, sigma = 1.6):

    img1 = cv2.imread(img1_path)
    img2 = cv2.imread(img2_path)

    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)

    sift = cv2.SIFT_create(nfeatures = n_features,
                           contrastThreshold = contrastThreshold,
                           edgeThreshold = edgeThreshold,
                           sigma = sigma)

    # Detect keypoints and compute descriptors
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    print(f"Keypoints in Image 1: {len(kp1)}")
    print(f"Keypoints in Image 2: {len(kp2)}")

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

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

    print(f"Good Matches: {len(good_matches)}")

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

    plt.figure(figsize=(12, 6))
    plt.imshow(match_img)
    plt.title("Matched Keypoints")
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```python
sift(img1_path, img2_path)
```

```
Keypoints in Image 1: 2822
Keypoints in Image 2: 4172
Good Matches: 292
```

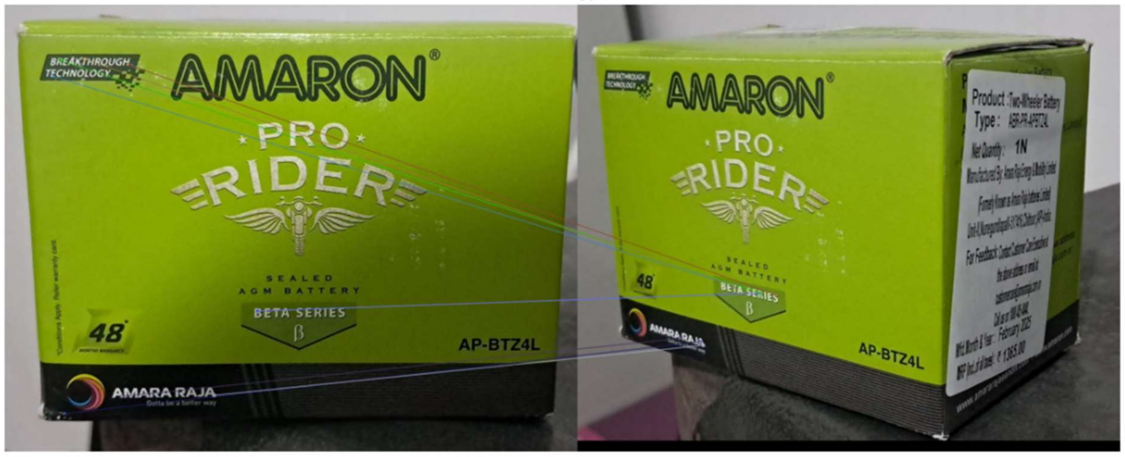
Matched Keypoints



```
# Reduce features and increase contrast threshold
sift(img1_path, img2_path, n_features = 200, contrastThreshold = 0.08)
```

```
Keypoints in Image 1: 200
Keypoints in Image 2: 200
Good Matches: 9
```

Matched Keypoints

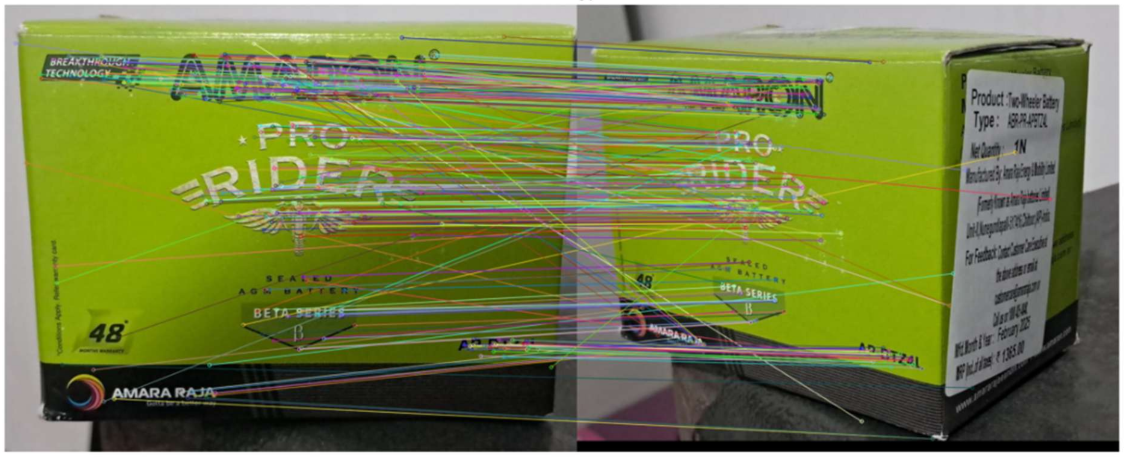


By limiting the number of features to 200 and increasing the contrast threshold, fewer keypoints were detected, especially in low-contrast areas. This resulted in faster computation and more prominent keypoints being prioritized. However, some fine matches were lost due to the aggressive filtering.

```
# Increase edge threshold to ignore fine edges
sift(img1_path, img2_path, edgeThreshold = 20)
```

```
Keypoints in Image 1: 3780
Keypoints in Image 2: 5298
Good Matches: 362
```

Matched Keypoints

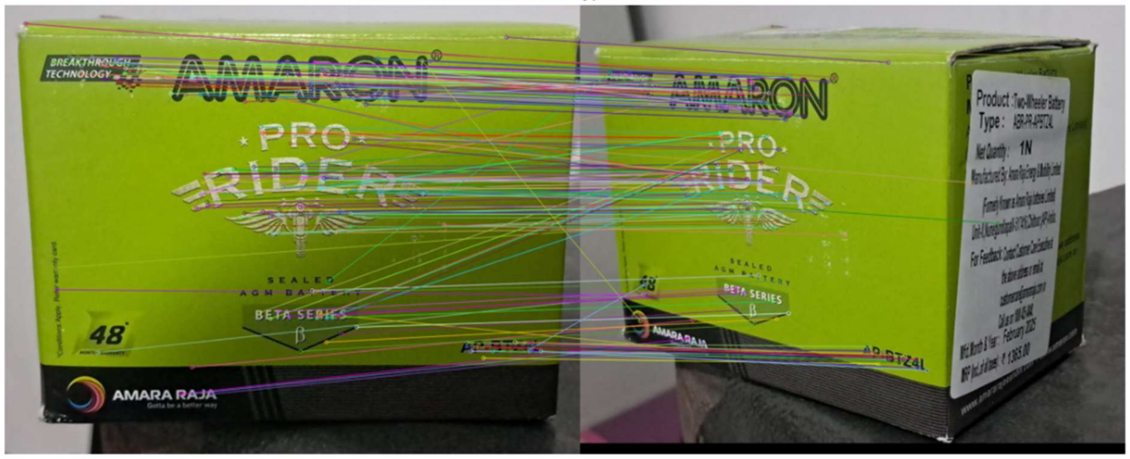


Raising the edge threshold allowed SIFT to ignore edge-like structures, reducing false keypoints along textured borders. This improved the robustness of matches but slightly reduced the total number of detected keypoints. It helped focus on more stable and distinct features.¶

```
# Modify sigma to affect blur applied
sift(img1_path, img2_path, sigma=2.0)
```

```
Keypoints in Image 1: 1624
Keypoints in Image 2: 1912
Good Matches: 204
```

Matched Keypoints



Increasing the sigma value caused more blurring before difference-of-Gaussian computation, making the algorithm capture broader structures instead of fine details. Keypoints were fewer but more scale-invariant. This is useful for detecting larger features but may miss fine texture matches.¶

RANSAC:

Aim : Use RANSAC to remove outlier key point matches and fit a transformation model between two images.

Loaded Image:

Image 1

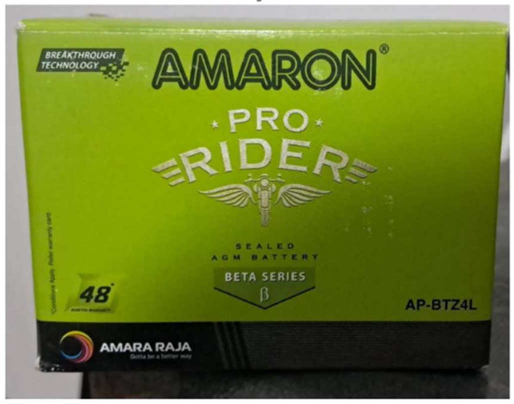
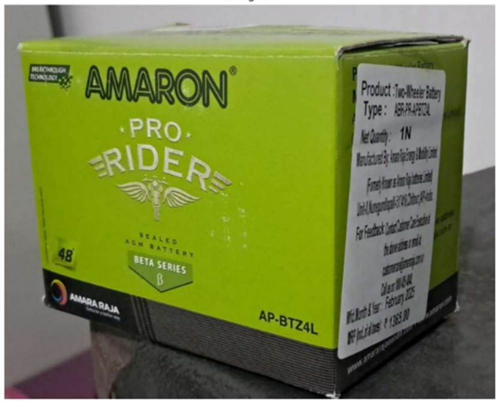


Image 2



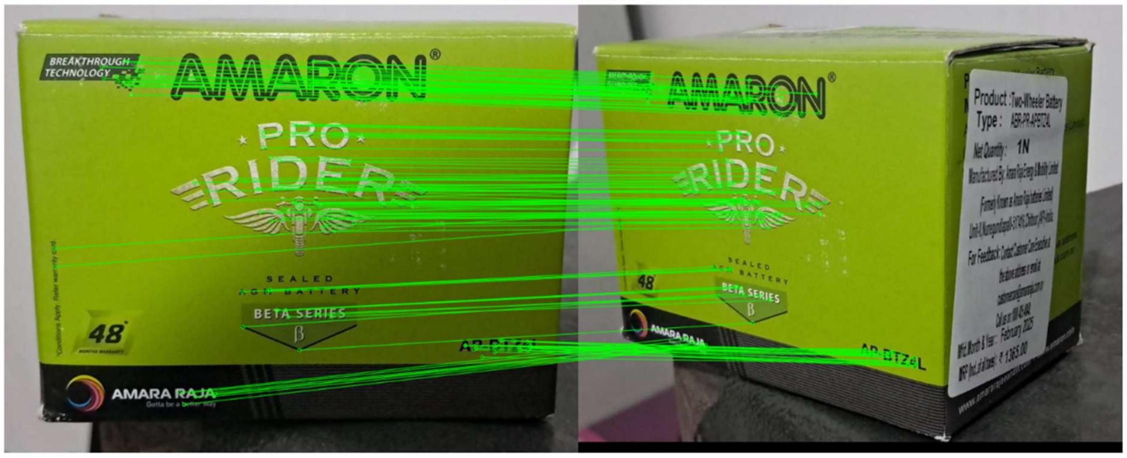
```python
def sift_ransac(img1_path, img2_path, ransac_thresh = 5.0):

    img1_color = cv2.imread(img1_path)
    img2_color = cv2.imread(img2_path)

    img1 = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)

    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    print(f"Total Keypoints in Image 1: {len(kp1)}")
    print(f"Total Keypoints in Image 2: {len(kp2)}")

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

    good_matches = []
    pts1 = []
    pts2 = []

    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)
            pts1.append(kp1[m.queryIdx].pt)
            pts2.append(kp2[m.trainIdx].pt)

    pts1 = np.float32(pts1)
    pts2 = np.float32(pts2)

    print(f"Good Matches before RANSAC: {len(good_matches)}")
```

```python
    # Apply RANSAC to remove outliers
    H, mask = cv2.findHomography(pts1, pts2, cv2.RANSAC, ransac_thresh)
    matches_mask = mask.ravel().tolist()

    inlier_matches = [good_matches[i] for i in range(len(matches_mask)) if matches_mask[i]]

    print(f"Good Matches after RANSAC (Inliers): {len(inlier_matches)}")

    result = cv2.drawMatches(img1_color, kp1, img2_color, kp2, inlier_matches, None,
                             matchColor = (0,255,0), singlePointColor = None,
                             flags = cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    plt.figure(figsize=(16, 8))
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.title("RANSAC Inlier Matches")
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```python
sift_ransac(img1_path, img2_path)
```

```
Total Keypoints in Image 1: 2753
Total Keypoints in Image 2: 4144
Good Matches before RANSAC: 285
Good Matches after RANSAC (Inliers): 156
```

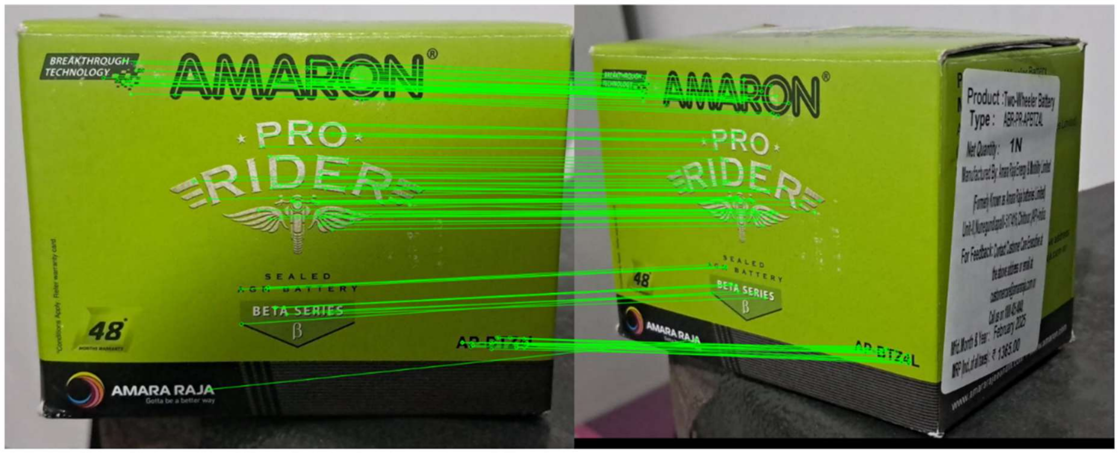
RANSAC Inlier Matches



```
# Tight RANSAC threshold (only very precise inliers kept)
sift_ransac(img1_path, img2_path, ransac_thresh=2.0)
```

```
Total Keypoints in Image 1: 2753
Total Keypoints in Image 2: 4144
Good Matches before RANSAC: 285
Good Matches after RANSAC (Inliers): 123
```

RANSAC Inlier Matches

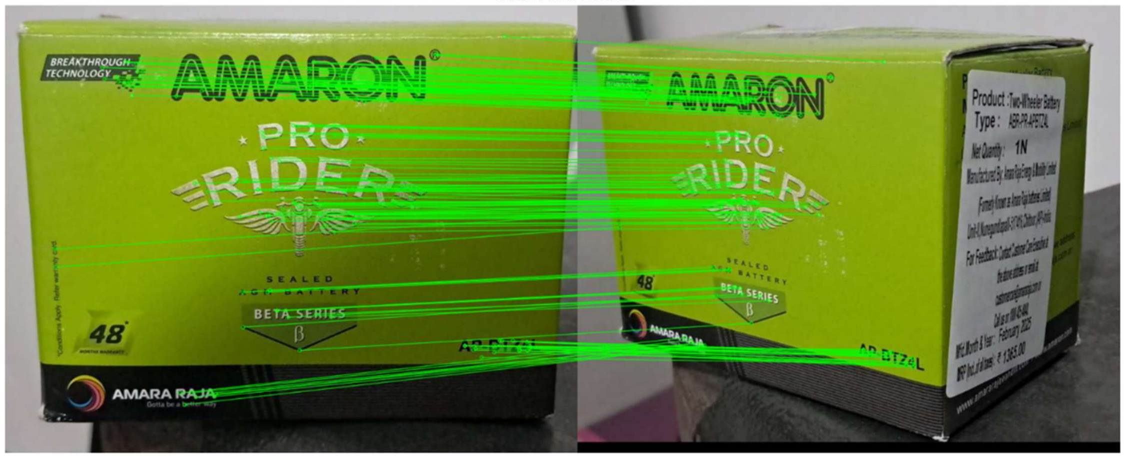


A lower RANSAC threshold of 2.0 made the inlier selection stricter, resulting in fewer matches. It removed more potential outliers but risked discarding some correct matches that were slightly misaligned.¶

```
# Looser RANSAC threshold (more leniency, risk of outliers)
sift_ransac(img1_path, img2_path, ransac_thresh=10.0)
```

```
Total Keypoints in Image 1: 2753
Total Keypoints in Image 2: 4144
Good Matches before RANSAC: 285
Good Matches after RANSAC (Inliers): 169
```

RANSAC Inlier Matches

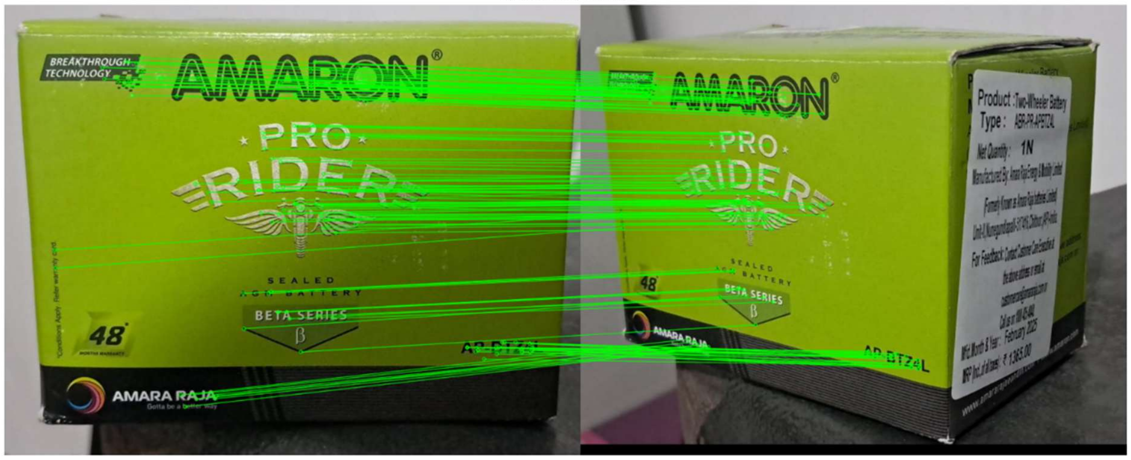


A higher RANSAC threshold of 10.0 accepted more matches, even those with larger geometric error. This retained more inliers but also slightly increased the risk of including false matches.¶

```
# Medium threshold (balanced)
sift_ransac(img1_path, img2_path, ransac_thresh=5.0)
```

```
Total Keypoints in Image 1: 2753
Total Keypoints in Image 2: 4144
Good Matches before RANSAC: 285
Good Matches after RANSAC (Inliers): 156
```

RANSAC Inlier Matches



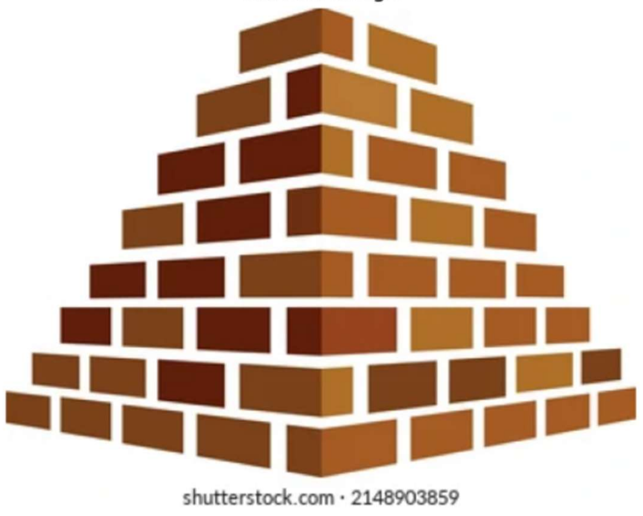
A balanced threshold of 5.0 provided a good trade-off between match quantity and geometric accuracy. It preserved valid matches while eliminating clear outliers, resulting in stable transformation fitting.¶

Shi-Tomasi:

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

Loaded Image:

Loaded Image

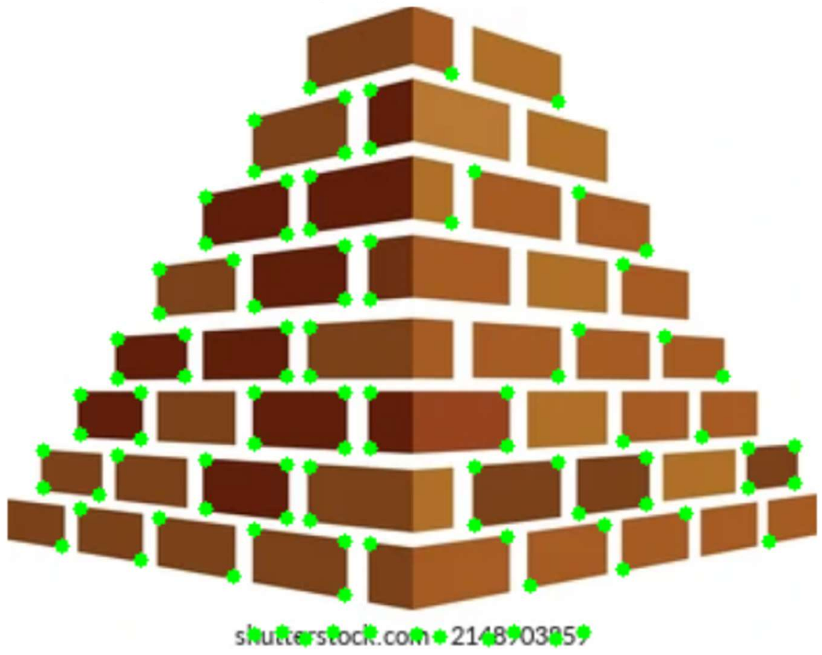


```python
def shi_tomasi(img_path, max_corners = 100, quality_level = 0.01, min_distance = 10):

    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    corners = cv2.goodFeaturesToTrack(gray,
                                      maxCorners = max_corners,
                                      qualityLevel = quality_level,
                                      minDistance = min_distance)

    if corners is not None:
        corners = np.int0(corners)
        print(f"Detected Corners: {len(corners)}")
    else:
        corners = []
        print("No corners detected!")

    img_marked = img.copy()

    for corner in corners:
        x, y = corner.ravel()
        cv2.circle(img_marked, (x, y), 3, (0, 255, 0), -1)

    img_rgb = cv2.cvtColor(img_marked, cv2.COLOR_BGR2RGB)

    plt.figure(figsize=(6, 6))
    plt.imshow(img_rgb)
    plt.title(f"Shi-Tomasi\nmaxCorners={max_corners}, qualityLevel={quality_level}, minDistance={min_distance}")
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```python
shi_tomasi(img_path)
```

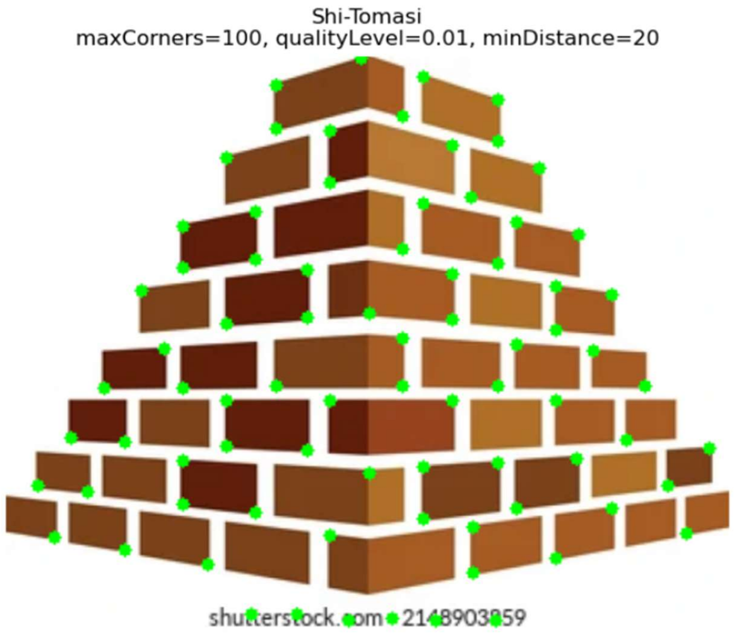
```
Detected Corners: 100
```

Shi-Tomasi
maxCorners=100, qualityLevel=0.01, minDistance=10



```
shi_tomasi(img_path, min_distance = 20)
```

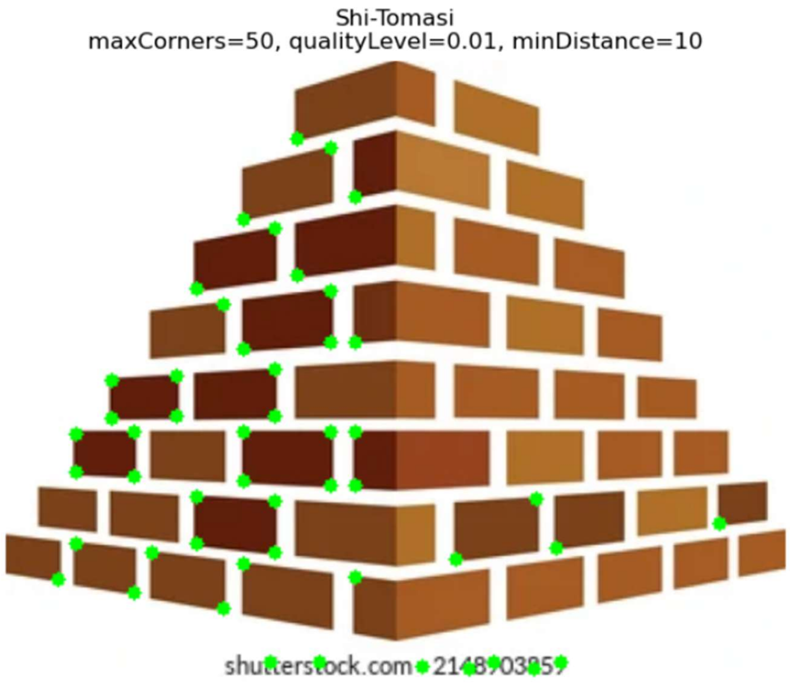
```
Detected Corners: 79
```


Shi-Tomasi
maxCorners=100, qualityLevel=0.01, minDistance=20
shutterstock.com • 2148903859

Ensures corners are spread out. Prevents clustering in textured areas.

```
shi_tomasi(img_path, max_corners=50, quality_level=0.01, min_distance=10)
```

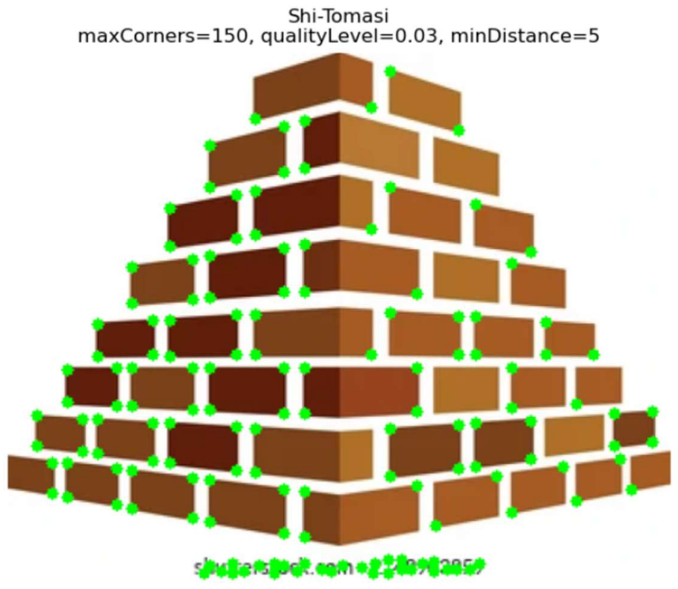
```
Detected Corners: 50
```


Shi-Tomasi
maxCorners=50, qualityLevel=0.01, minDistance=10
shutterstock.com • 2148903859

1. Fewer corners are detected compared to the default.
2. Focuses only on the most prominent corners.
3. Useful when visual clarity is needed with reduced clutter

```
shi_tomasi(img_path, max_corners=150, quality_level=0.03, min_distance=5)
```

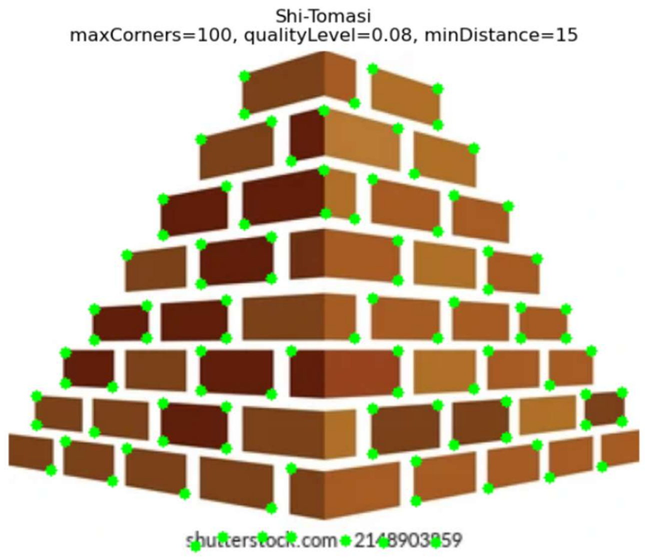
```
Detected Corners: 150
```



1. More corners detected due to lower distance and higher cap.
2. Detects finer details in clustered regions.
3. May introduce some unstable keypoints due to closeness.

```
shi_tomasi(img_path, max_corners=100, quality_level=0.08, min_distance=15)
```

```
Detected Corners: 98
```


Shi-Tomasi
maxCorners=100, qualityLevel=0.08, minDistance=15
shutterstock.com • 2148903859

1. Only very strong corners are kept due to higher qualityLevel.
2. Widely spaced keypoints with minimal overlap.
3.Best suited for tracking robust and distinctive features.