Below is a concise overview of each OpenCV/NumPy/Matplotlib function we used, followed by a high-level explanation of the full algorithm pipeline.

Function Introductions

1. **`cv2.imread(path, flag)`**
   - **Purpose**: Load an image from disk into a NumPy array.
   - **Common Flags**:
     - `cv2.IMREAD_GRAYSCALE` : load as single-channel grayscale.
     - `cv2.IMREAD_COLOR` : load as BGR color.
2. **`cv2.threshold(src, thresh, maxval, type)`**
   - **Purpose**: Convert a grayscale image into a binary (black/white) mask.
   - **Parameters**:
     - `thresh` : initial threshold value (often ignored with OTSU).
     - `maxval` : value to assign to "foreground" pixels.
     - `type` : e.g. `cv2.THRESH_BINARY + cv2.THRESH_OTSU` automatically picks the best threshold.
   - **Returns**: `(ret, dst)` where `dst` is the binarized image.
3. **`cv2.ORB_create(n_features)`**
   - **Purpose**: Instantiate an ORB (Oriented FAST and Rotated BRIEF) feature detector + descriptor extractor.
   - **`n_features`** : maximum number of keypoints to retain.
4. **`orb.detectAndCompute(image, mask)`**
   - **Purpose**:
     A. **Detect** keypoints (corners, blobs) in the image.
     B. **Compute** a binary descriptor for each keypoint.
   - **Returns**:
     - `keypoints` : list of keypoint objects (with locations, scale, orientation).
     - `descriptors` : a NumPy array of shape `(len(keypoints), descriptor_length)` .
5. **`cv2.BFMatcher(normType, crossCheck)`**
   - **Purpose**: Create a "brute-force" matcher that compares every descriptor in set A against every descriptor in set B.
   - **`normType`** : e.g. `cv2.NORM_HAMMING` for binary descriptors.
   - **`crossCheck=True`** : only keep matches for which A→B and B→A agree.
6. **`bf.match(des1, des2)`**
   - **Purpose**: Find the best match in `des2` for each descriptor in `des1` .
   - **Returns**: A list of `DMatch` objects containing `.distance` (match score) and index references.
7. **Python built-ins**:
   - **`sorted(iterable, key=…)`** : sort matches by ascending distance.
   - **List slicing**: `matches[:int(len(matches)*0.3)]` takes the top 30%.
8. **`cv2.findHomography(src_pts, dst_pts, method, ransacReprojThreshold)`**
   - **Purpose**: Estimate a 3×3 projective transform (homography) that maps `src_pts` → `dst_pts` .
   - **`method=cv2.RANSAC`** : uses RANSAC to reject outliers.
   - **Returns**: `(H, mask)` where `H` is the homography matrix and `mask` indicates inlier matches.
9. **`np.linalg.svd(A)`**
   - **Purpose**: Singular Value Decomposition of a matrix `A = U·Σ·Vᵀ` .
   - **Use here**: By factoring out any scaling/shear in the top-left 2×2 block of `H` , we isolate the pure rotation `R = U·Vᵀ` .
10. **`math.atan2(y, x)`** & **`math.degrees(rad)`**
    - **Purpose**:
      A. `atan2(b, a)` : compute the signed angle (in radians) of the vector `[a, b]` .
      B. `degrees(...)` : convert radians → degrees.
11. **`cv2.warpPerspective(src, H, dsize, flags, borderValue)`**
    - **Purpose**: Apply a full 3×3 homography `H` to warp the source image into a new perspective.
    - **`dsize`** : output image size `(width, height)` .
    - **`flags`** : interpolation method (e.g., `cv2.INTER_LINEAR`).
    - **`borderValue`** : pixel value to fill outside the source image.
12. **Matplotlib Display**
    - **`plt.subplots(...)`** : create a grid of axes.
    - **`ax.imshow(image, cmap='gray')`** : show a grayscale image.
    - **`ax.set_title(...)`** , **`ax.axis('off')`** : annotate and hide axes.
    - **`plt.tight_layout(); plt.show()`** : render the figure.

Algorithm Overview

1. **Preprocessing & Binarization**
   - Convert both the **original** and **distorted** images into clean binary masks using Otsu's threshold.
   - This strips away noise and leaves behind only the shape silhouette.
2. **Feature Detection & Matching**
   - Use **ORB** to detect a few thousand keypoints on each mask, and extract their binary descriptors.
   - Perform a **Brute-Force match** (Hamming distance) and keep the top 30% of matches by quality.
3. **Robust Homography Estimation**
   - Feed the matched 2D point pairs into `cv2.findHomography(..., method=RANSAC)` to compute a **3×3 homography** `H` that best explains the mapping from the distorted shape back to the original.
   - RANSAC automatically discards mismatches and focuses on the dominant geometric transform.
4. **Rotation Extraction**
   - Extract the **linear part** `A = H[0:2,0:2]` .
   - Perform **SVD** on `A` and reconstruct `R = U·Vᵀ` , which is guaranteed to be a pure rotation (no scale/shear).
   - Compute the angle via `atan2(R[1,0], R[0,0])` and normalize it into **[0, 180°]**.

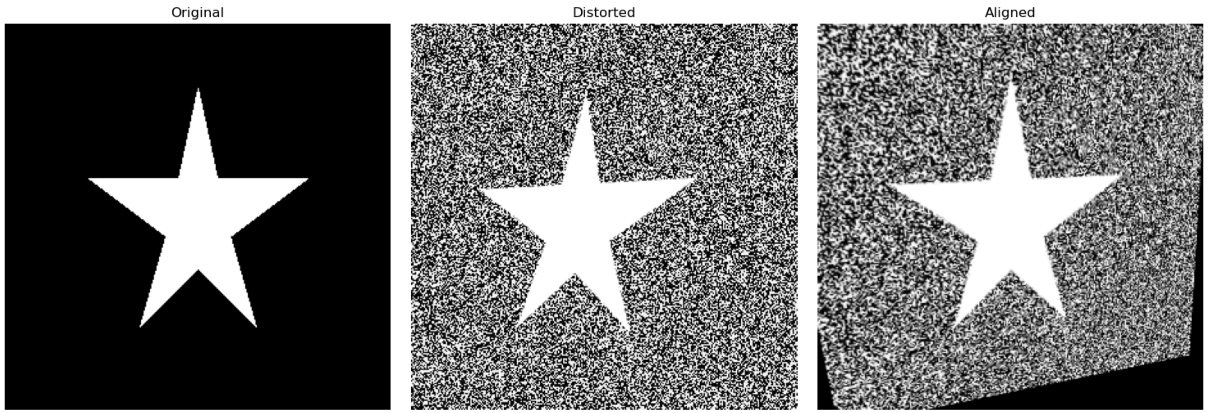
5. **Image Alignment**
    - Finally, apply the **full homography** `H` (not just the rotation) with `cv2.warpPerspective` to the distorted image.
    - This undoes not only the rotation, but also any perspective skew and shear, yielding pixel-accurate alignment.
6. **Visualization**
    - Display the **Original**, **Distorted**, and **Aligned** images side by side to verify the quality of the registration.

This pipeline combines **feature-based matching** with **projective geometry** and a **clean linear decomposition**—making it robust to noise, partial occlusion, and mild perspective distortion, while guaranteeing an accurate rotation estimate in the desired range.

In [1]:

```python
# test 2
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt

# 1. Load grayscale images
img_orig = cv2.imread('demo.images/original_shape_1.png', cv2.IMREAD_GRAYSCALE)
img_dist = cv2.imread('demo.images/distorted_shape_1.png', cv2.IMREAD_GRAYSCALE)

# 2. Binarize (remove noise) so ORB focuses on the shape
_, b_orig = cv2.threshold(img_orig, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
_, b_dist = cv2.threshold(img_dist, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

# 3. Detect ORB features on the clean masks
orb = cv2.ORB_create(5000)
kp1, des1 = orb.detectAndCompute(b_orig, None)
kp2, des2 = orb.detectAndCompute(b_dist, None)

# 4. Match and pick the best 30%
bf = cv2.BFMatcher(cv2.NORM_HAMMING)
matches = sorted(bf.match(des1, des2), key=lambda m: m.distance)
good = matches[: int(len(matches)*0.3)]

# 5. Build point sets
pts_orig = np.float32([kp1[m.queryIdx].pt for m in good])
pts_dist = np.float32([kp2[m.trainIdx].pt for m in good])

# 6. Estimate a homography (handles perspective + affine)
H, mask = cv2.findHomography(pts_dist, pts_orig, cv2.RANSAC, 5.0)

# 7. Decompose the top-left 2×2 of H to get a pure rotation
A = H[0:2, 0:2]
# Use SVD to strip out any scaling/shear, leaving only R
U, S, Vt = np.linalg.svd(A)
R = U @ Vt
# Compute angle in [0, 180)
angle = math.degrees(math.atan2(R[1,0], R[0,0])) % 180
print(f'Estimated rotation angle: {angle:.2f}°')

# 8. Warp the distorted image back with the full homography
h, w = img_orig.shape
img_aligned = cv2.warpPerspective(img_dist, H, (w, h), flags=cv2.INTER_LINEAR, borderValue=0)

# 9. Show the three stages side-by-side
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
for a, im, title in zip(ax,
                       [img_orig, img_dist, img_aligned],
                       ['Original', 'Distorted', 'Aligned']):
    a.imshow(im, cmap='gray')
    a.set_title(title)
    a.axis('off')
plt.tight_layout()
plt.show()
```

```
Estimated rotation angle: 179.54°
```



In [ ]: