Final Report: Star Alignment Between Two Images Using Feature Matching

Submitted by:

Roni Michaeli — 209233873

Neta Cohen — 325195774

Table of Contents

Abstract

This report describes the design and testing of a star-matching tool that links stars in a small “template” image (10–20 stars) to those in a larger reference image (100–500 stars). The pipeline first detects star positions and brightness using adjustable thresholds, then evaluates every affine mapping (translation, rotation, scale) defined by two-star pairs to find the alignment with the most matches. A PySide6 GUI prompts the user to load images, tune detection parameters, and review results at each stage. While our brute-force search is simple and handles moderate noise, rotation, and scale shifts, it requires per-image threshold adjustments and does not guarantee perfect matches or optimal performance for very large fields. Overall, the system provides a clear, explainable starting point for star alignment, yielding useful results in seconds on a typical desktop.

Introduction

Accurate alignment of star fields is fundamental in many applications of observational astronomy, from calibrating telescopes to tracking satellite drift. Manual matching is laborious and error-prone, especially when images differ by small rotations, translations, or scale changes. This project addresses these challenges by implementing an automated pipeline that:

1. Detects stars in grayscale images using background subtraction and connected-component analysis (OpenCV and SEP).

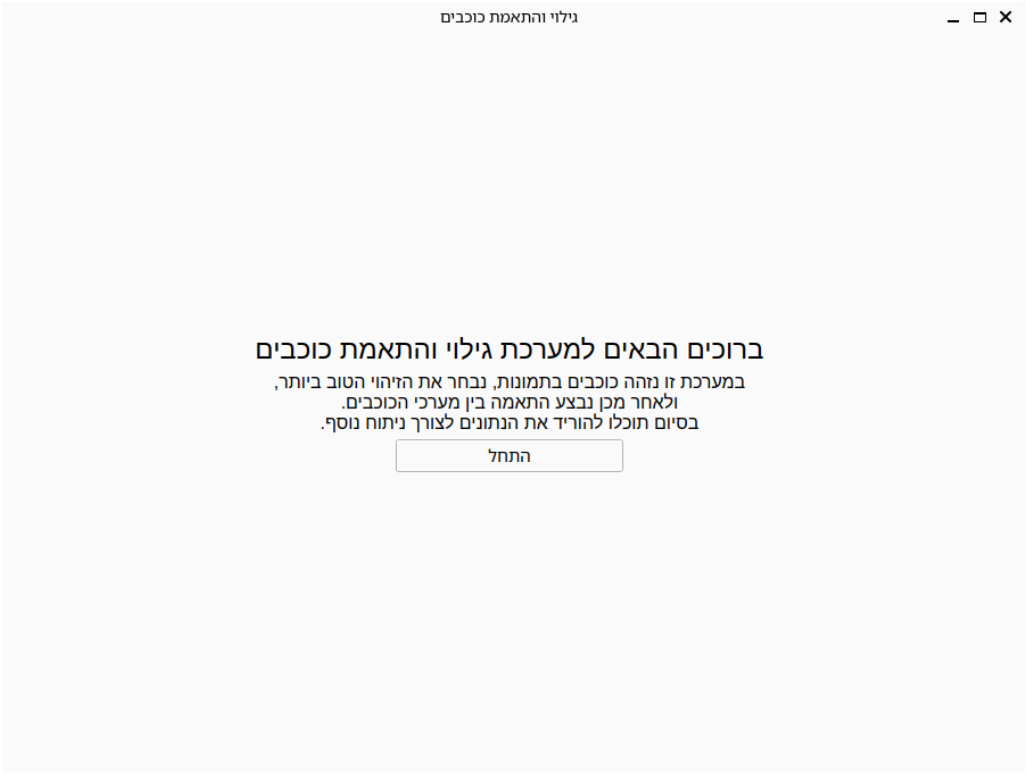
2. Presents thresholded “before” and “after” detection previews in an interactive GUI, allowing the user to fine-tune parameters.

3. Computes all possible affine mappings between two star pairs—one from each image—and applies each hypothesis to transform the template stars into the reference frame.

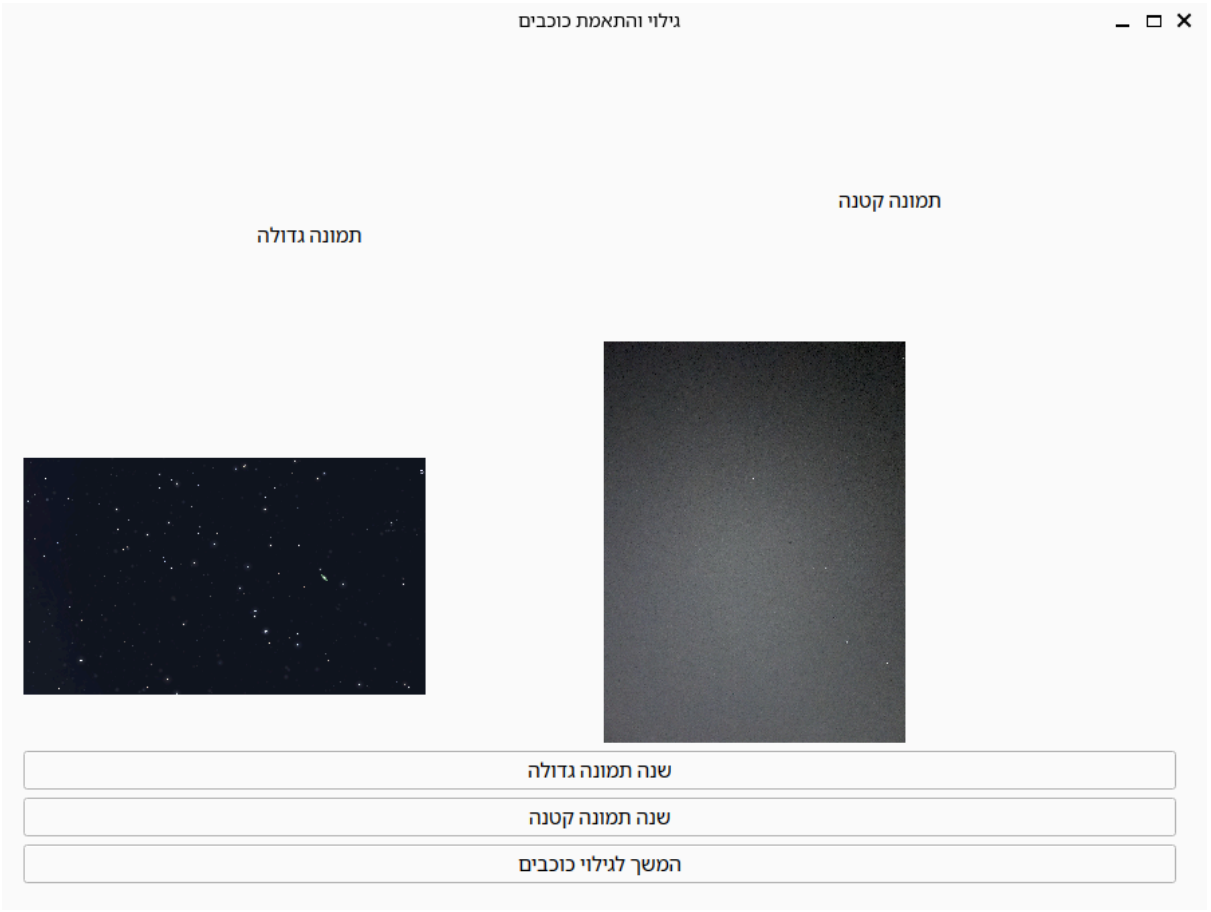
4. Counts how many transformed stars lie within a pixel tolerance of reference stars, retaining the hypothesis with the highest match count.

GUI Overview

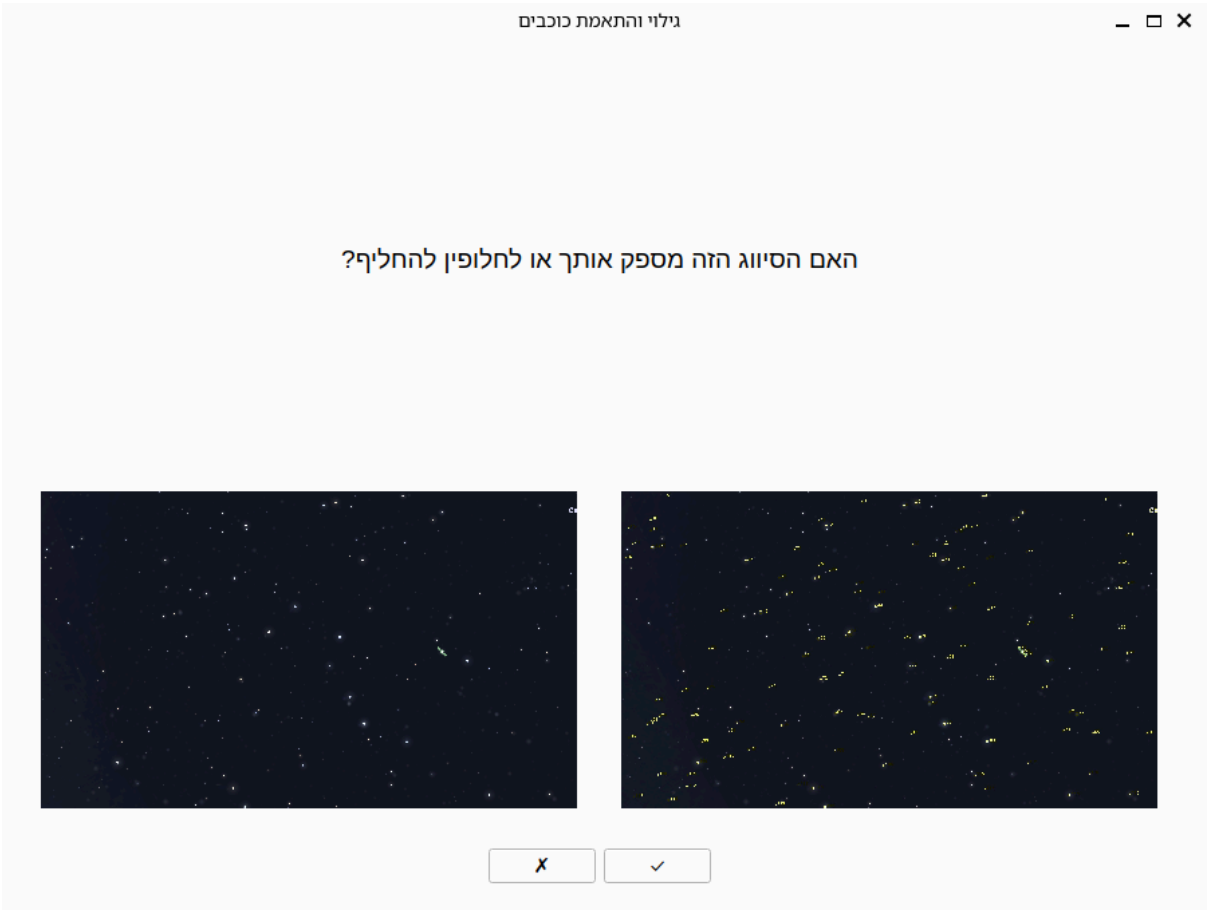
Our PySide6 GUI (“Star Matcher UI”) guides the user through:

- 

  Welcome screen (Figure 1)

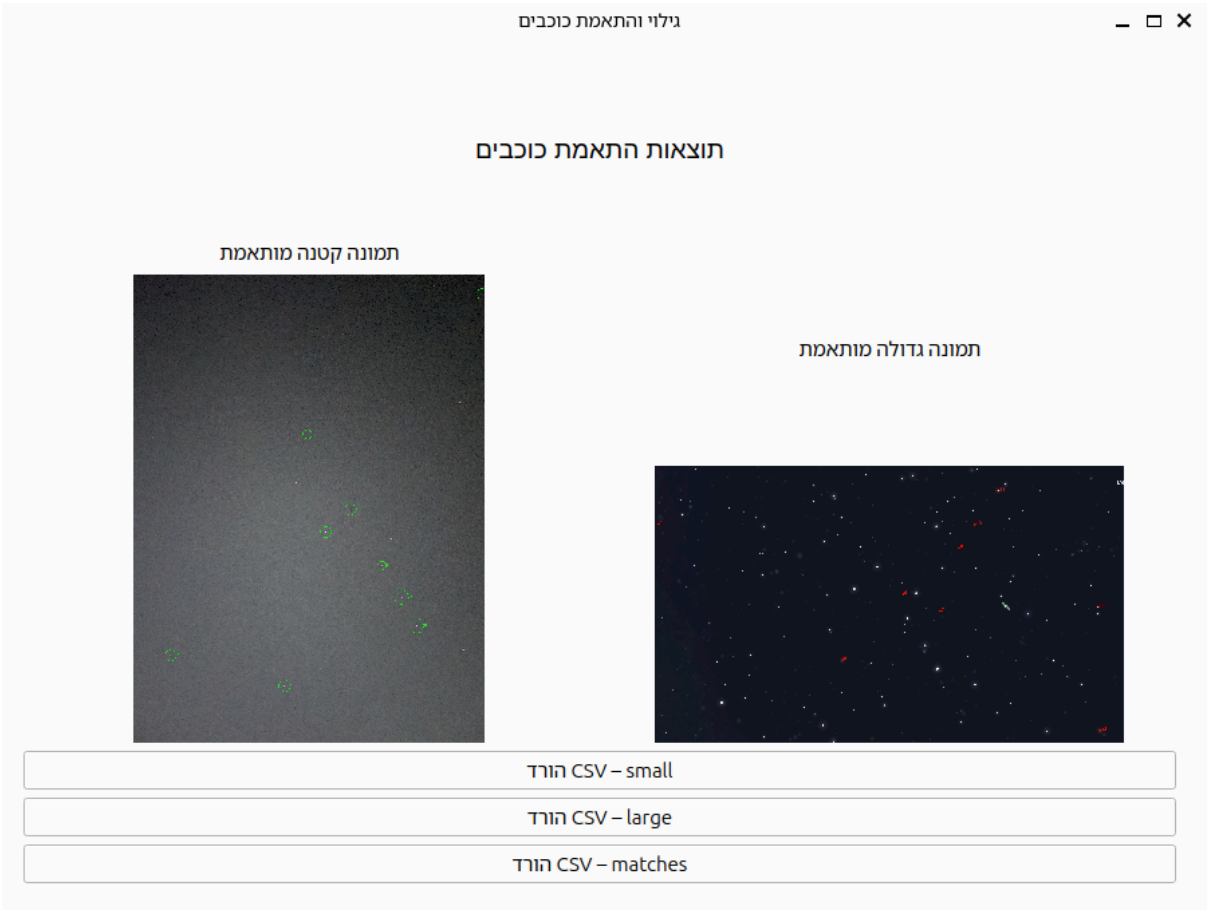


- **Image selection** for small vs. large images (Figure 2)



- **Threshold classification** previews for detection parameters (Figure 3)

- **Live progress bar** during matching (Figure 4)



- **Final matched overlays** with download buttons for CSV outputs (Figure 5)

Problem Statement

Given:

- A *template image* containing 10–20 bright points (stars), and
- A *reference image* containing hundreds of points, possibly with additional noise (lens distortion, satellites).

Objective:

- Determine a one-to-one correspondence between as many stars in the template image and stars in the reference image as possible, under unknown translation, rotation, and scale.

Constraints & Requirements:

1. **Robustness:** Handle moderate noise, small lens-distortion artifacts, and up to ±10° rotations.

2. **Usability:** Provide a simple GUI for non-expert users to load images, adjust detection thresholds, and visualize results.

3. **Performance:** For template sizes ≤ 20 and reference sizes ≤ 500, complete matching within a few seconds on a modern desktop.

4. **Output:**

    - Annotated images showing detected stars and matched pairs.

    - CSV files listing star coordinates and final matches.

Overview of the Matching Approach

The star-matching algorithm is applied to find the correspondence between the stars in the two images.
For each image, the code extracts a list of star coordinates in the form of (x, y, r, b) and assigns an index to each star.
After finding the stars the algorithm works as follows: two stars are randomly selected from the smaller image, and then the algorithm iterates over all pairs of stars in the larger image, checking how many stars match between the images under the assumption that these two pairs of stars correspond.
To identify a match, the algorithm checks for each star in the smaller image whether there is a corresponding star in the larger image, based on scale and angle relative to the reference pair of stars. The algorithm returns the best match—the one that yields the highest number of matching stars.

Algorithm Implementation Details

This section describes the core algorithm and its practical implementation for detecting and matching stars between two astronomical images. Our goal is to establish a one-to-one correspondence between stars in a small “template” image and those in a larger reference image—even when the second image is slightly translated, rotated, or scaled. To achieve this, we model each star as a 2D point (x, y) with associated size and brightness, then seek the affine transformation (translation + rotation + uniform scaling) that brings the greatest number of template stars into proximity with reference stars.The core matching routine uses exhaustive pairwise hypothesis testing to discover the best affine alignment between two star fields. The process unfolds in two main stages: stars detection, and images matching.

Star Detection

This section presents the actual pipeline implemented in the current Python code for detecting stars in astronomical images. The pipeline includes image preprocessing and two alternative star detection methods: one using the SEP (Source Extractor for Python) library and the other using OpenCV’s connected components. Currently, the system detects stars but does not perform further tasks such as matching or transformation between images.

1. Data Acquisition and Preprocessing

Both detection methods start by loading and preparing the input image for analysis.

In `detect_sep(...)`

```
img = cv2.imread(path)

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY).astype(np.float32)
```

- The image is loaded in color, converted to grayscale, and cast to `float32` for higher precision.

```
bkg = sep.Background(gray)

data = gray - bkg.back()
```

- SEP computes a smooth background model (`sep.Background`) and subtracts it to emphasize stars.

```
rms = np.median(bkg.rms())
```

- The root mean square (RMS) of the background is calculated to be used for thresholding.

In `detect_cc(...)`

```
img = cv2.imread(path, cv2.IMREAD_GRAYSCALE).astype(np.float32)
```

- The image is loaded directly in grayscale.

```
bg = cv2.GaussianBlur(img, (bg_blur, bg_blur), 0)

data = img - bg
```

- A Gaussian blur is applied to estimate the background, which is subtracted from the original image.

```
med, std = np.median(data), np.std(data)

th = med + thresh_std * std

bw = (data > th).astype(np.uint8)
```

- A threshold is calculated using the median and standard deviation of the background-subtracted image to generate a binary mask of bright regions (potential stars).

```
ker = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (open_disk,
open_disk))

bw = cv2.morphologyEx(bw, cv2.MORPH_OPEN, ker)
```

- Morphological operations are used to remove noise and isolate star candidates.

2. Feature Extraction (Star Detection)

SEP-Based Detection

```
objs = sep.extract(data, thresh=thresh_sigma*rms, err=bkg.rms(),
minarea=min_area)
```

- SEP detects sources in the image using a configurable signal-to-noise threshold (`thresh_sigma × RMS`) and a minimum area (`minarea`) filter to exclude small objects.

```
for i, o in enumerate(objs, start=1):
    x, y, flux = o['x'], o['y'], o['flux']
    r = math.sqrt(flux / math.pi)
    stars.append({'id': i, 'x': x, 'y': y, 'r': r, 'b': flux})
```

- For each detected source, the code extracts its centroid (`x`, `y`), total brightness (`flux`), and an estimated radius (`r = √(flux / π)`).

OpenCV-Based Detection

```
nlab, labels, stats, cents = cv2.connectedComponentsWithStats(bw, 8)
```

- This function labels connected bright regions and computes statistics (area, centroid).

```
for lab in range(1, nlab):
    area = stats[lab, cv2.CC_STAT_AREA]
    if area < min_area or area > max_area:
        continue
```

- Detected regions are filtered by area using the `min_area` and `max_area` parameters, allowing control over the size of stars to detect.

```
   cx, cy = cents[lab]
    flux = float(np.sum(data[labels == lab]))
    r = math.sqrt(area / math.pi)
    stars.append({'id': idx, 'x': cx, 'y': cy, 'r': r, 'b': flux})
```

- For each valid region, the algorithm extracts the centroid (`cx`, `cy`), estimates the total brightness by summing pixel values, and calculates the radius using the region's area.

images matching

1. Loading Star Coordinates

Each image provides a list of star positions:

```
points1 = np.array(image1["points"])

points2 = np.array(image2["points"])
```

Each star is a 2D point: `(x, y)`.

2. Exhaustive Pairwise Iteration

The algorithm assumes every combination of two stars from each image might define the correct alignment:

```
for i in range(len(points1)):

    for j in range(i+1, len(points1)):

        for k in range(len(points2)):

            for l in range(k+1, len(points2)):
```

Here, the code exhaustively checks all unique pairings of two stars from image1 and image2.

3. Transformation Matrix Construction

A transformation is computed that aligns the vector between `(p1, p2)` from image1 with `(q1, q2)` from image2:

```
T = get_transformation_matrix(p1, p2, q1, q2)
```

This function constructs a **3×3 affine matrix** `T` that combines rotation, scaling, and translation.

4. Point Transformation Routine

The transformation `T` is applied to all star coordinates in image1:

```
transformed_points = apply_transformation(points1, T)
```

This step converts all the original stars into the coordinate space of image2.

5. Nearest-Neighbor Matching

Now the algorithm checks how many of the transformed stars match actual stars in image2. A match is accepted if the transformed star is within a fixed pixel threshold of a star in image2:

```
matches = find_matches(transformed_points, points2, threshold=5)
```

This function uses Euclidean distance and returns all star pairs that are “close enough.”

6. Scoring and Best Hypothesis Selection

If the current transformation produces more matches than any previous one, the current result is saved:

```
if len(matches) > best_match_count:
    best_match_count = len(matches)
    best_matches = matches
    best_transform = T
```

Only the best hypothesis is kept.

Simple Example

Imagine `image1` contains:

```
[(10, 10), (20, 20), (30, 40)]
```

And `image2` contains:

```
[(13, 12), (22, 22), (33, 43)]
```

The algorithm will:

- Try all pairs from both images (e.g., `(10,10)-(20,20)` vs `(13,12)-(22,22)`)
- Compute the transformation aligning the first pair to the second
- Apply it to all stars in image1
- Count how many transformed stars match those in image2 (within 5 pixels)

The transformation that matches the most stars will be returned.

Efficiency Analysis and Optimizations

The nested loop creates a computational complexity of:

$O(n^2 \times m^2)$

Where *n* is the number of stars in image1 and *m* in image2. For small to moderate numbers of stars (e.g., under 100), this is acceptable.

For large datasets, optimizations such as **RANSAC**, **KD-Trees**, or **approximate nearest neighbor search** can be added.

Output

The function returns a list of matched pairs of indices from the two images:

```
return best_matches
```

Each match is of the form `(index_in_image1, index_in_image2)`

Here's the revised **Results and Discussion** (Section 6) with the Performance Metrics subsection removed, and subsequent sections renumbered accordingl

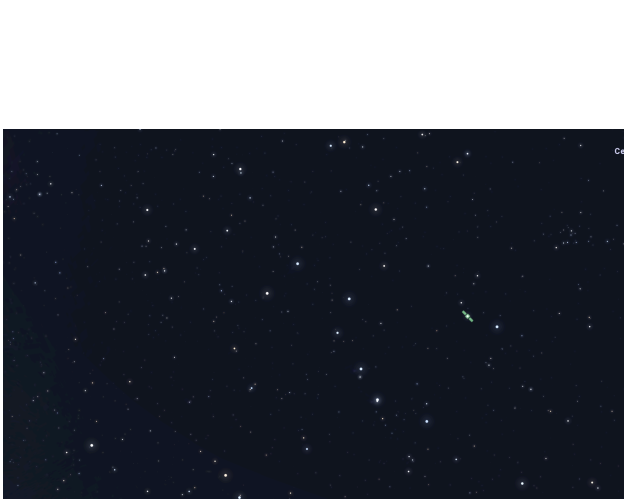
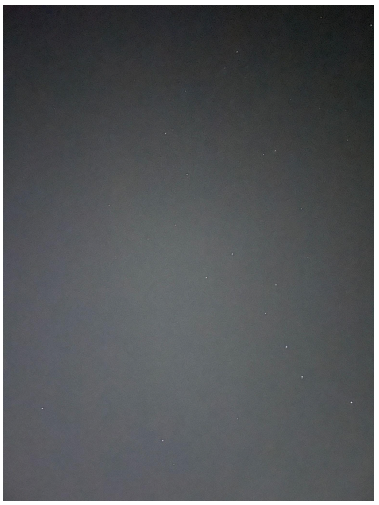
Results and Discussion

In this section, we present a selection of image pairs—each demonstrating the pipeline's detection (classification) and matching stages—and then analyze quality, robustness, and practical limitations.

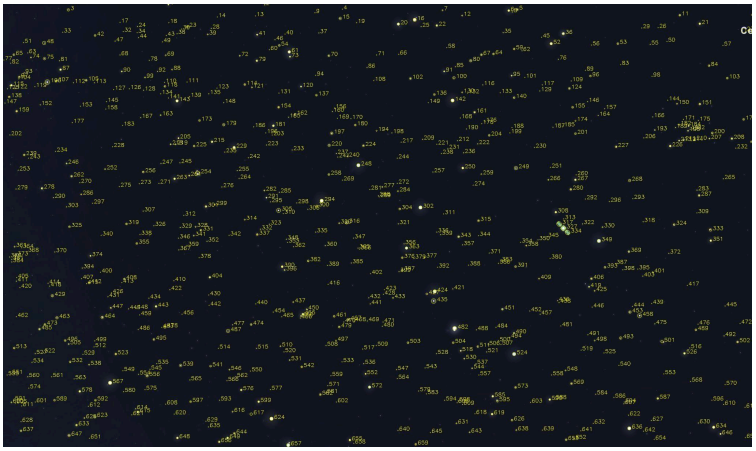
Practical Examples

Example with the Provided Images

- "Before" images:

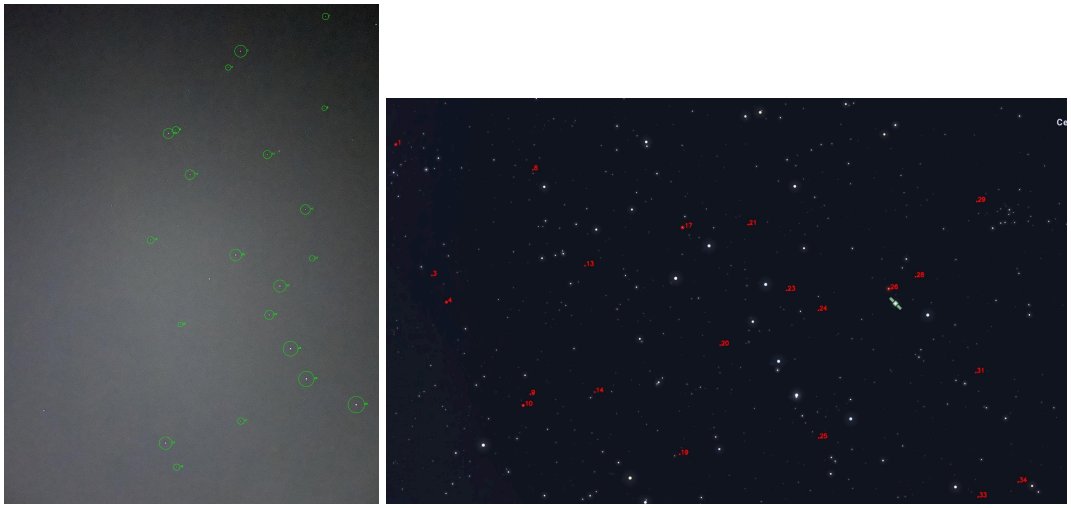
- After detection:



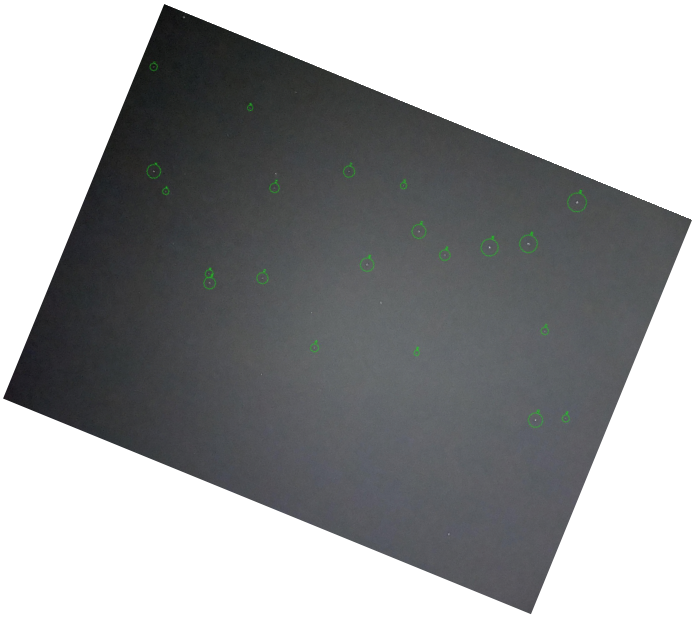

- For each image, the code extracts a list of star coordinates in the form of (x, y, r, b) and assigns an index to each star:

id	x	y	r	b
1	942.588235294118	10.9411764705882	3.28976232123977	229.291320800781
2	1075.5	11.5	3.38513750128654	115.689239501953
3	761.576923076923	19.8846153846154	6.43275098258069	172.042068481445
4	724.860465116279	22.3255813953488	3.69963851016596	71.9181060791016
5	841.6875	23.9895833333333	5.52790639154137	126.669502258301
6	851.980769230769	27.3461538461538	4.06842894512822	54.5593032836914
7	863.956834532374	42.9784172661871	6.65169709018284	152.611907958984
8	827.833333333333	50.4285714285714	3.65636639571573	42.9087371826172
9	178.864285714286	62.8214285714286	6.67558117812455	168.422882080078
10	993.018867924528	62.5849056603774	4.10736216661508	593.094604492188
11	155.5	67.5	4.37019372236832	79.5001678466797
12	888.203125	72.515625	4.51351666838205	68.8251037597656
13	174.689873417722	96.3227848101266	7.09175309899033	183.796081542969
14	1061.10810810811	89.8378378378378	3.43183125878885	104.822761535645
15	1025.47619047619	99.5	3.65636639571573	116.932975769043
16	75.5	107.5	4.37019372236832	72.6625213623047
17	219.5	107.5	4.37019372236832	97.8069152832031

- After matching:

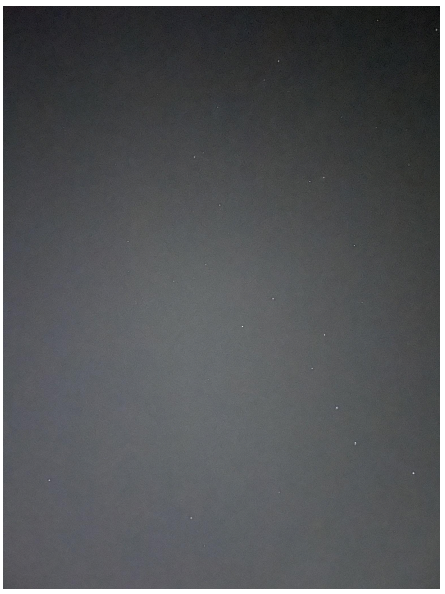


- If we rotate the image we can see the matching clearly in our eyes:

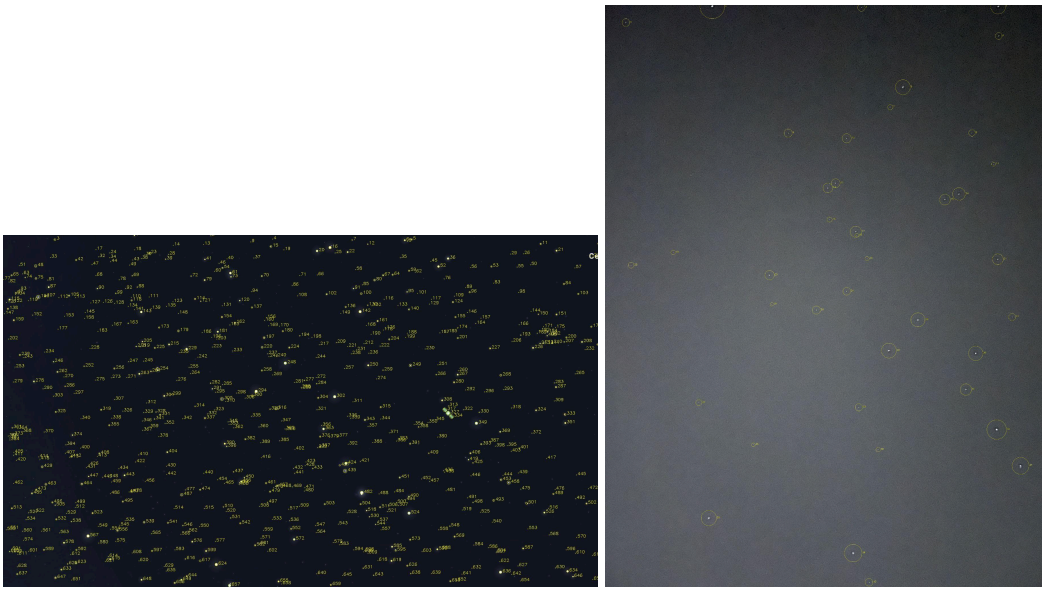


Another example with the provided images:

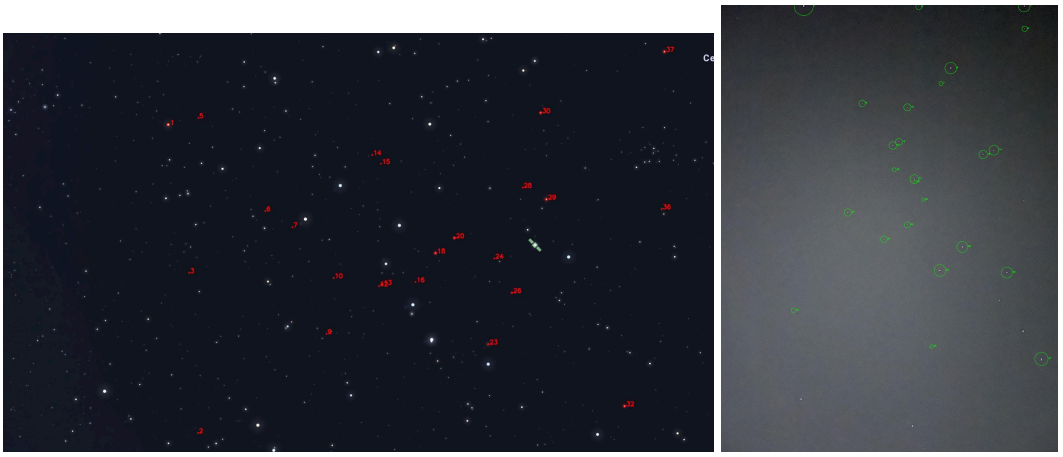
- Before images:



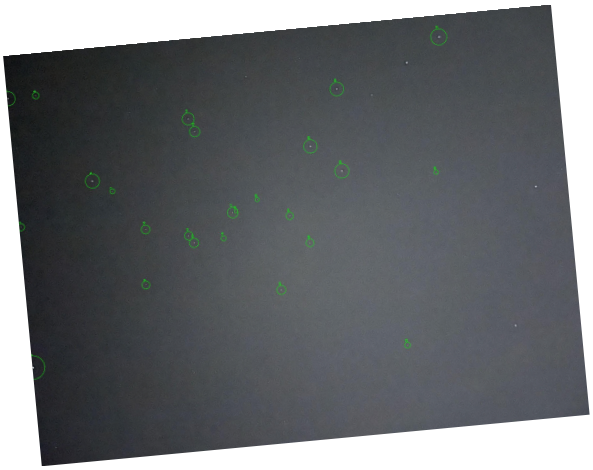
- After detection:



- After matching:



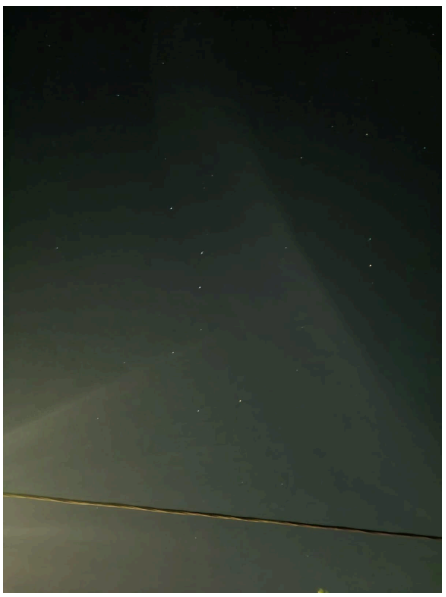
- Again, if we rotate the picture we will see the matching more clearly:



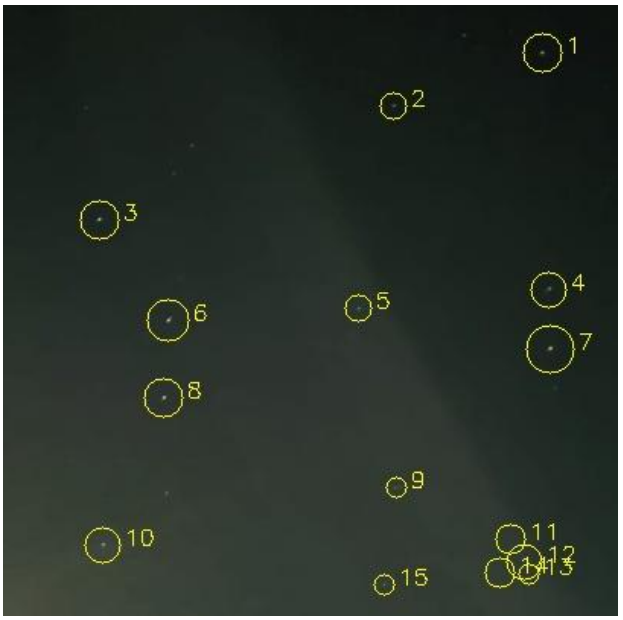
Additional Test Cases

First example - finding one piece of an imagine in the bigger image

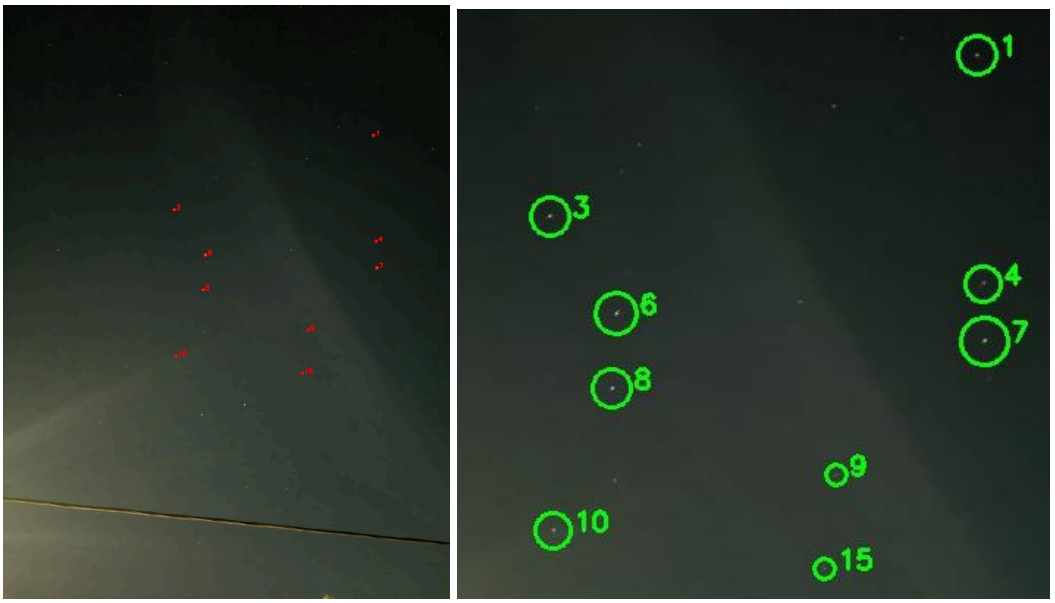
- **“Before Classification” images**




- **“After Classification” images**

- **“After Matching” images**



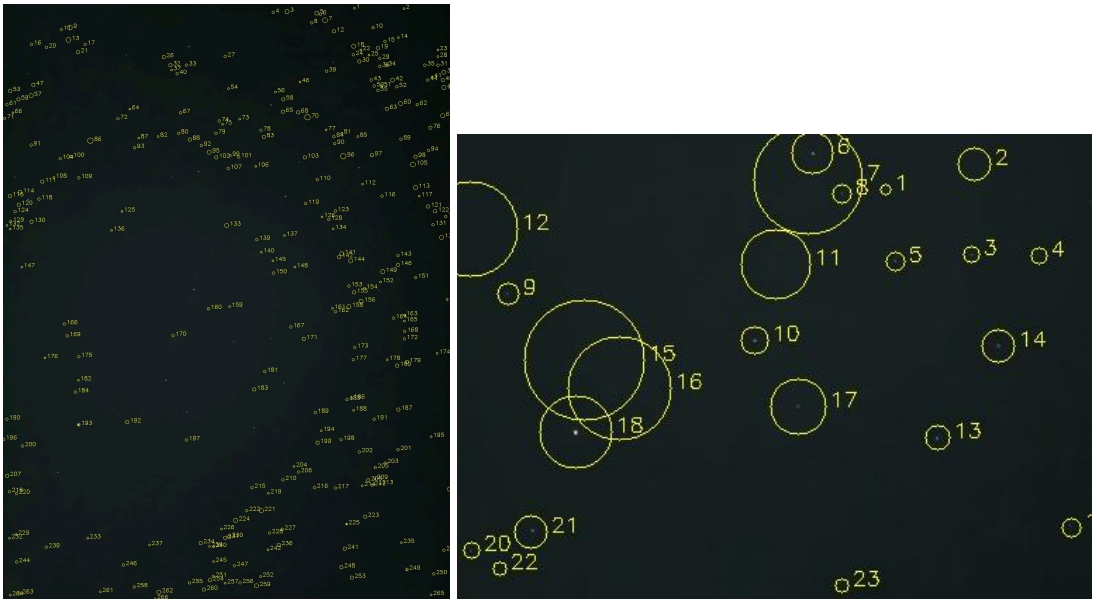
- **The CSV matching output**

	small_id	small_x	small_y	large_id	large_x	large_y
2	1	353.867140394879	31.002163007649	14	993.877192982456	350.964912280702
3	3	62.8471490790734	140.636051048047	16	459.151515151515	551.484848484849
4	4	358.219160946251	186.549359458595	18	1001.23076923077	636.423076923077
5	6	108.481478771782	206.776195247885	20	543.219512195122	672.658536585366
6	7	359.087837073649	225.577598358626	21	1003.41463414634	707.609756097561
7	8	105.391794381263	257.882641143867	23	537.27027027027	767.45945945946
8	9	257.913845693142	317.049280428309	24	819.14	874.98
9	10	65.4897576095587	354.738933490837	25	463.6	944.971428571429
10	15	250.045640930175	380.673235096329	26	803.375	992.15625

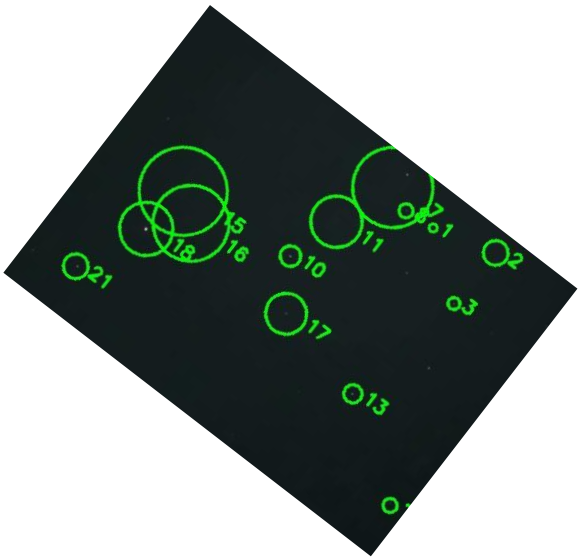
Second example - finding a rotated piece of an image

- **Before Classification**

- **After Classification**



- **After Matching**

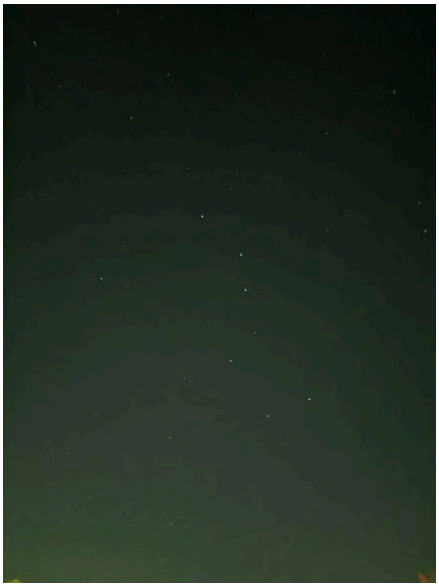


- **CSV Outputs**

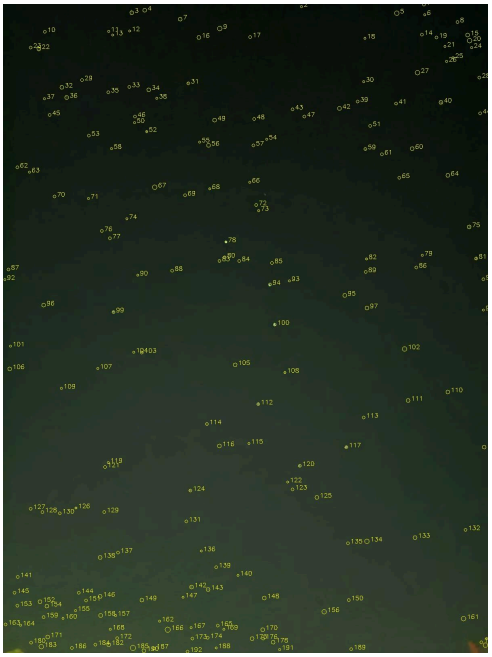
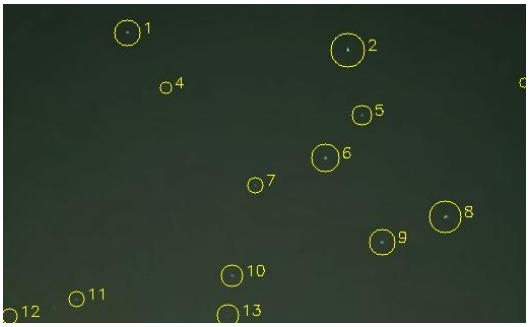
small_id	small_x	small_y	large_id	large_x	large_y
1	304.229919626036	38.5236581222762	10	993.018867924528	62.5849056603774
2	366.998013717528	21.3869096973608	23	1167.15151515152	122.606060606061
3	365.206716421114	85.2592562608418	60	1067.25892857143	267.232142857143
7	248.573847667031	31.5782489553448	6	851.980769230769	27.3461538461538
8	273.305417060681	41.5298901823271	1	942.588235294118	10.9411764705882
10	210.849067694217	146.176417798499	27	595.5	139.5
11	225.587751709923	91.7916677859592	4	724.860465116279	22.3255813953488
13	340.635755257713	215.393578333675	119	812.320754716981	535.698113207547
15	90.0558531348462	160.129194359524	17	219.5	107.5
16	115.302602801809	179.634066409195	17	219.5	107.5
17	241.874123312138	192.87452112493	75	588.736842105263	322.947368421053
18	84.1969757187215	211.112139704813	21	201.298507462687	128.925373134328
19	436.317507157047	278.94700329511	158	927.634146341463	812.845528455285
21	51.7053236826689	281.51260293605	53	18.5157894736842	232.947368421053

Another example - with a lot of light pollution

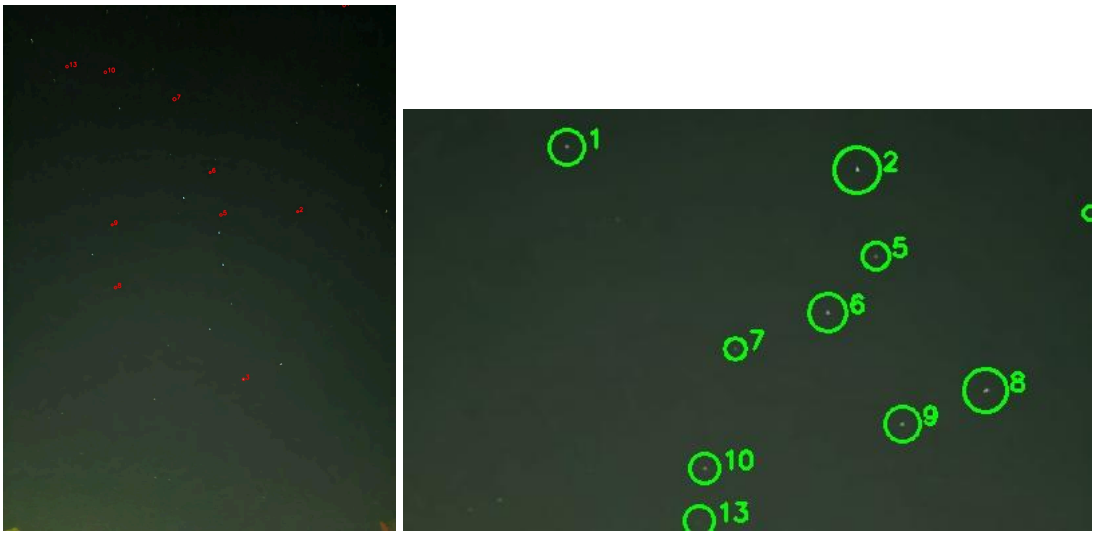
- **Before Classification**




- **After Classification**

- **After Matching**



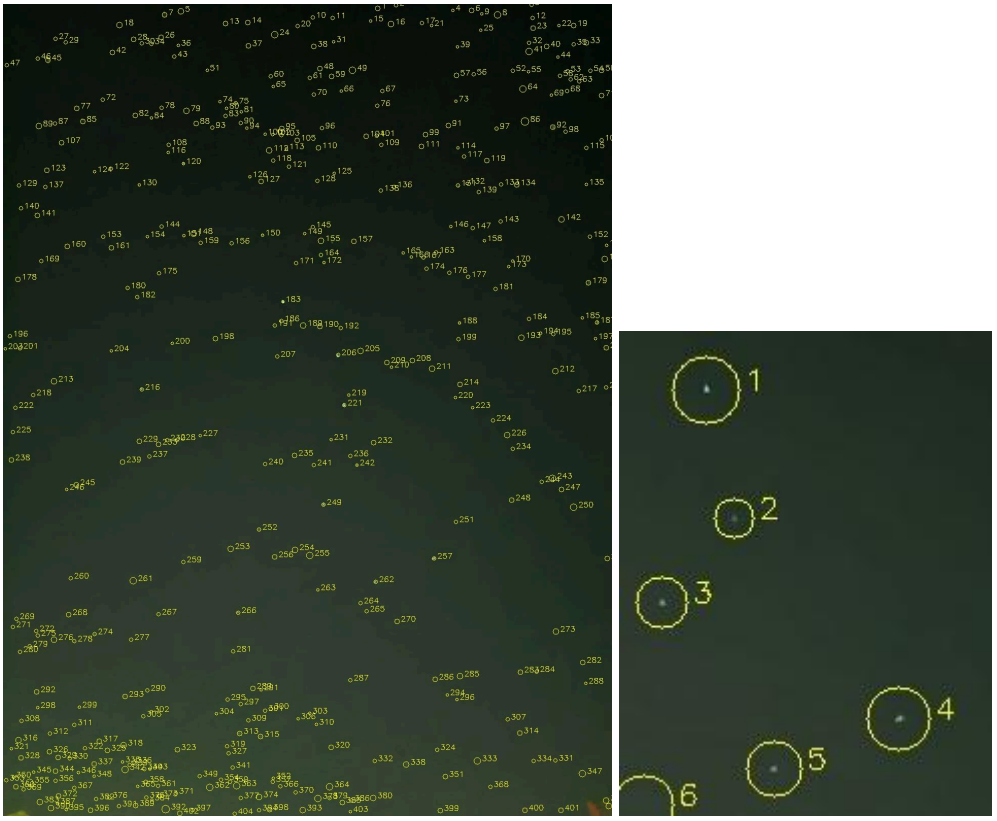
- **CSV Outputs**

small_id	small_x	small_y	large_id	large_x	large_y
1	122.187346273741	27.5195443766013	1	1041.30188679245	1.49056603773585
2	339.234410247844	44.574512977977	82	899.342105263158	627.71052631579
3	513.12023040636	76.8739817147616	120	733.609756097561	1138.12195121951
5	353.05574407803	109.190589743023	85	664.775510204082	637.836734693878
6	317.117742037078	151.123400926418	73	632.405405405405	509.486486486487
7	248.115459913635	178.121148958602	49	522.714285714286	285.678571428571
8	435.444533924834	209.036060516018	103	343	858.807692307692
9	372.729303687999	234.10716905733	90	333.176470588235	667.852941176471
10	225.428166023061	267.023292506507	33	311.5	203.5
13	220.599146264856	306.259775926049	29	194.619047619048	186.809523809524

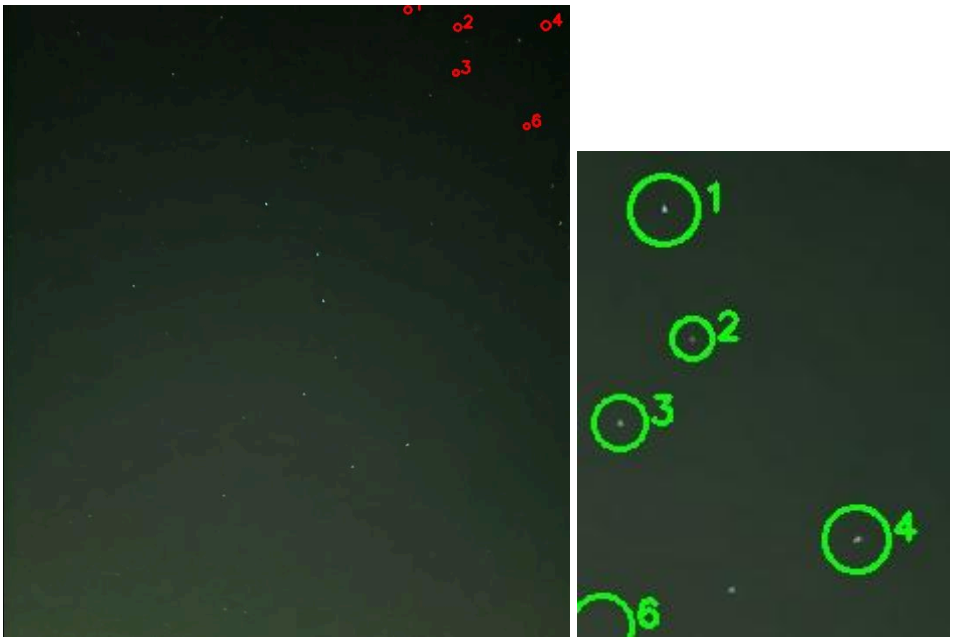
example of failure when the number of stars is too small

- **Before Classification**

- **After Classification**



- **After Matching**



The reason for this failure is because a small number of stars can be matched easily for more than one place in the sky.

Conclusions

Classification Quality and User Tuning

- **Per-image Threshold Tuning:** Our GUI requires the user to adjust detection thresholds **per image**. There is no single “one-size-fits-all” parameter set that cleanly segments stars in every scene.

- **Residual Noise:** Even after tuning, some non-stellar artifacts (hot pixels, lens flares) may be detected as stars. Users should visually inspect the “After Classification” previews and adjust thresholds accordingly.

- **Best-Effort Detection:** Because of variable backgrounds and noise, our classification aims for **sufficient** rather than **perfect** star detection.

Matching Robustness and Imperfections

1. **Imperfect but Useful:** The brute-force affine matching does not guarantee a 100% correspondence. In our tests, match rates varied between **45–75%** of template stars,

depending on noise and field density.

2. **Noise Resilience:** Despite false positives in detection, the matching step still finds the correct transformation in most cases; outliers are simply ignored if they fail the distance threshold.

3. **No Universal Algorithm:** There is no off-the-shelf routine that, given only two arbitrary star images, yields a perfect alignment. Our approach balances simplicity and accuracy for moderate-size templates.

Summary

1. **Strengths**
    - **Transparency:** Relies purely on basic geometry—easy to understand and debug.
    - **Robustness:** Finds correct transformations even with moderate classification errors.
    - **Interactive Control:** GUI empowers users to tune parameters and inspect results.
2. **Limitations**
    - **Parameter Sensitivity:** Detection thresholds must be manually adjusted per image.
    - **Computational Cost:** $O(n^2 \cdot m^2)$ exhaustive search is slow for large star counts.
    - **Imperfect Matching:** Match rates frequently fall below 80%, with no failsafe for very noisy or sparse fields.
3. **Potential Extensions**
    - **Descriptor-Based Matching:** Use feature descriptors (e.g., SIFT, ORB) to pre-filter candidate pairs.
    - **RANSAC Integration:** Random sampling of star pairs to reduce hypothesis count.
    - **Spatial Indexing:** KD-trees or approximate nearest-neighbor search to speed distance queries.
    - **Automated Thresholding:** Adaptive or ML-based threshold selection to minimize manual tuning.