Computer Vision CS-GY 6643 - Project 1 Q5

Diego Rosenberg dr3432@nyu.edu, Thiago Viegas tjv235@nyu.edu

1 Question 5

1.1 General Approach

This algorithm identifies constellations in a sky image through a multi-stage geometric pattern matching process. First, it loads and analyzes reference images to build a structural map of each known constellation, representing stars as nodes in a graph. For any given sky image, it detects all potential stars by using image thresholding and contour analysis to isolate bright spots. The core of the algorithm then compares the detected stars to the reference patterns by creating a rotation-invariant "angle signature" for each star based on its neighbors, allowing it to find candidate matches and their likely orientation. Finally, these candidates are validated by projecting the entire constellation pattern onto the sky and calculating a final score based on how well the pattern's stars align with actual detected stars, while also penalizing for extra, unexplained stars in the vicinity to ensure a precise match.

1.2 Classes and Data Structures

patternsHelper.py

This module defines the data structures for representing constellation patterns and includes the logic to extract these patterns from images.

Data Classes

- **`Node`**: This data class represents a single star in a constellation. It stores the star's label, its `(x, y)` position, its size, and a dictionary of links to its neighbors, including the angle and distance to each.
- **`Pattern`**: This data class represents an entire constellation. It contains a dictionary of all its `Node` objects and a list of edges that connect them, effectively defining the constellation's geometric structure.

Functions

- **`extract_pattern_from_image(bgr_or_rgba, ...)`**: This function analyzes a pattern image (which has white dots for stars and green lines for connections). It uses color segmentation to identify the nodes and edges and constructs a `Pattern` object from them. It also returns the binary masks for the nodes (stars) and lines for later use in matching.

imageHelper.py

This file contains the `Image` class, which acts as a primary controller for loading and processing a single sky image.

Class

- **`Image`**: This class encapsulates a sky image and all its associated data and operations. It handles loading the image, performing preprocessing, detecting stars, storing their coordinates, and holding the final constellation prediction and confidence score.

Methods (Functions within the `Image` class)

- **`__init__(self, image_path, ...)`**: The constructor loads an image from a specified file path. It immediately converts the image to grayscale and initializes placeholder variables for storing results.
- **`equalize(self)`**: Applies histogram equalization to the image to improve its contrast. This can help make faint stars more visible before further processing.

- **adaptive_threshold(self, ...)** and **global_threshold(self, ...)**: These methods convert the grayscale image into a binary (black and white) image. They are used to isolate the bright stars from the dark sky background.
- **iterate_through_patches(self)**: This method loops through a directory of smaller "patch" images and uses template matching to find their locations in the main sky image. It is one of the ways the program identifies the key stars of a potential constellation.
- **detect_stars_from_sky(self, ...)**: This is a more direct method for finding stars that does not rely on patches. It uses contour detection on the thresholded image to identify all bright objects that fall within a specific size range, calculating their centroids to get precise coordinates.
- **create_black_image_with_coordinates(self)**: This utility function generates a new, black image and draws lines between all the detected star coordinates. This creates a visual representation of the detected star pattern.

1.3 Helper Functions

helpers.py

This module is a collection of utility functions that perform the core calculations for image processing, geometric analysis, and pattern matching.

Functions

- **template_matching(image, template)**: A standard computer vision function that finds the location of a small template image within a larger source image. It returns the quality of the match (score) and the coordinates of the best match.
- **calculate_all_angle_signatures(coordinates, ...)**: A key function for the matching algorithm. For each star, it calculates a "signature" consisting of the sorted angles to its nearest neighbors, creating a description of its local geometry that is robust to rotation and scale.
- **find_candidate_rotations(pattern_sig, detected_sig, ...)**: This function compares the angle signature of a pattern star to that of a detected star from the sky. It identifies the most likely rotation that would align the two signatures, proposing a hypothesis for how the pattern might be oriented.
- **validate_and_score_match(detected_coords, match_result, ...)**: After a potential match is found based on angle signatures, this function performs a full geometric validation. It projects the entire constellation pattern onto the sky using the hypothesized transformation and calculates a `final_score` based on how many pattern stars correctly align with detected sky stars.
- **calculate_mask_fit_score(...)** and **calculate_sparsity_score(...)**: These functions provide additional scoring metrics to refine the best match. The `mask_fit_score` checks how well the bright pixels of the sky align with the pattern's shape, while the `sparsity_score` penalizes matches that occur in cluttered areas with many extra, unexplained stars.
- **plot_match_in_scene(sky_image, ...)**: A visualization function that overlays the best-matched constellation pattern onto the original sky image. It color-codes the nodes and edges to make it easy to see the quality and accuracy of the final match.

master.py

This module orchestrates the high-level logic of the constellation identification process, bringing together the functionality from all the helper modules.

Functions

- **`get_info_on_patterns(pattern_dir)`**: This function is responsible for loading all the reference constellation patterns. It iterates through the pattern files, calls `extract_pattern_from_image` for each one, and returns a list of structured `Pattern` objects.
- **`find_constelation(sky_file, patch_dir, patterns)`**: This is the core engine of the program. For a single sky image, it detects the stars, then systematically compares them against every known constellation pattern to find the best possible match using the geometric validation and scoring functions from `helpers.py`.
- **`master_function(folder, patterns)`**: This function serves as a convenient wrapper that handles the file management for a single constellation folder. It identifies the main sky image and the patch subfolder, calls `find_constelation` to perform the analysis, and returns the results.

2 AI Prompts and Thought Process

2.1 Initial Approach

We began by identifying star patches in the sky image using the same detection algorithm described in Questions 2, 3, and 4. Once the patches were located, we used AI-assisted filtering to prepare the image for constellation pattern matching.

2.2 Filtering

Our first idea was to draw circles and connecting lines between the detected patches, then attempt to overlay constellation patterns directly. With AI's help, we developed functions to convert images into binary black-and-white masks, highlighting the star patches while suppressing the background and constellation lines. We applied the same preprocessing to both the sky images and the constellation templates.

2.3 Pattern Matching

After generating binary masks for both images, we attempted to use OpenCV's built-in `matchTemplate` function. The initial results were poor, achieving only about 30% accuracy. With further parameter tuning, we improved the score to roughly 58%, but this was still insufficient for reliable constellation recognition.

2.4 Using Image Invariants

Given the limitations of template matching, we shifted focus toward geometric invariants. With support from GitHub Copilot, we reorganized our code into an object-oriented design, allowing each image and constellation template to be represented as objects with helper methods. We then computed the angles formed between all pairs of connected star patches, since these angular relationships are invariant to translation and scaling.

2.5 Angle-Based Score Matching

Once angle extraction was implemented, we developed an algorithm to compare angular patterns between the sky image and constellation templates. Using AI we arrived at a voting system that mimics the Hough Transform method where we vote for the bright spots to select candidate rotations to then fully try to adjust for. Initially, we were only using a subset of all rotational candidates (attempting a greedy approach) which led to poor results. This was due to the fact that smaller constellations with less nodes were receiving a higher rating in the candidate search since they could more easily nearly fit in other constellations (even though they were not the same size). Once we extended the search to consider all possible candidates, this significantly improved accuracy. Still, this approach sometimes failed to identify the correct constellation consistently.

2.6 Adding a Geometric Matching

Given that the angle signature method was not working at the best of its capacity. We leveraged AI to help us now consider not only the best "angle signatures" but also attempt to place them on top of the detected patches to see how well our template fit the available patches. We do this by creating a k-d Tree for easy nearest neighbor detection (between existing patch locations and potential template locations). Since our initial scan only looked at angle signatures here we will look to estimate a full transformation (rotation and scale up/down) to validate the full geometry of the templates over the patches. We iterate through all of the patches and set them as our "anchor" location and check the distances in the original patches to their nearest neighbor. Then we compare the same distances in the template images to determine a potential "scale" factor that we can use to attempt to "layer" our template on top of the image. With the angle signatures and estimated scale value we can now calculate the rotational matrix and translation vector needed to move the candidate template to "best fit" the image. With this, we can now create an expected geometric score score for the candidate template where we check to see how many of the template nodes were correctly overlayed (meaning they have to be on top or in a radius of 25 pixels from the patch coordinate). Our structural score will be:

$$\text{structural\_fit\_score} = (\text{num\_matched\_nodes}/\text{num\_nodes\_pattern})^2$$

Lastly, using a weighted average approach, our final score will be a linear combination of the angle score and calculated structural fit where the final score will be:

$$\text{final\_score} = (0.4 * \text{initial\_angle\_score}) + (0.6 * \text{structural\_fit\_score})$$

2.7 Adding a Complexity Reward

The above algorithm was very efficient at finding the correct answer but had the flaw of sometimes being able to fit more than one template to the patches. This mostly happened with smaller constellations, which had less nodes to fit, and would therefore be able to have one straight line and a near-enough dot that would fit a 3-node constellation. To improve this, we added a complexity reward as a "tie breaker" if two constellations had the same final score. Here we take into consideration the total number of nodes in the template. As an example, if we have two fit constellations, one with 3 nodes and one with 5 nodes, we would select the more complex 5-node template as the "correct" answer in a greedy approach. We believed this was a good selection as it would be unlikely that given the patches we would be able to find a more complex constellation in the same place as a less complex constellation (as that would likely imply that the constellation was on top of the other anyways).