

Benchmarking Corner Detection Algorithms

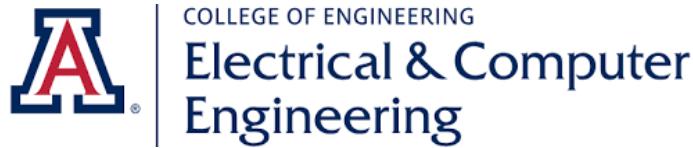
Weston Scott

ECE 533: Digital Image Processing

Spring 2025

University of Arizona

Email: scottwj@arizona.edu



Abstract—This project focuses on benchmarking corner detection algorithms in the OpenCV Python library, including Harris, Shi-Tomasi, FAST (Features from Accelerated Segment Test), ORB (Oriented FAST and Rotated BRIEF), SIFT (Scale-Invariant Feature Transform), BRISK (Binary Robust Invariant Scalable Keypoints), AGAST (Adaptive and Generic Accelerated Segment Test), KAZE, and AKAZE (Accelerated KAZE). The study involves optimizing each algorithm for independent performance, testing their efficiency (speed), accuracy, and robustness to scale invariance through zooming in and out on test images. The MDPI benchmark dataset will be used for evaluation, along with metrics such as execution time, precision, recall, and repeatability.

I. INTRODUCTION

Corner detection is a fundamental task in computer vision, serving as the basis for key applications such as image registration, object recognition, simultaneous localization and mapping (SLAM), and 3D reconstruction. Corners represent points of high two-dimensional intensity variation, making them robust and distinctive features for matching across images. However, reliable corner detection remains challenging due to variability in imaging conditions, including changes in scale, rotation, illumination, noise, and viewpoint. Algorithms must balance competing demands of detection accuracy, computational efficiency, and robustness to these transformations. Over the past decades, several corner detection algorithms have been developed, ranging from early methods like the Harris detector to more recent techniques like AGAST and KAZE. Libraries such as OpenCV have made these algorithms widely accessible, but their relative performance in practical scenarios often depends heavily on hyperparameter choices and specific image conditions.

This project presents a systematic benchmarking of nine widely-used OpenCV corner detection algorithms: Harris, Shi-Tomasi, FAST, ORB, SIFT, BRISK, AGAST, KAZE, and AKAZE. Each algorithm is optimized through parameter tuning on a real-world urban dataset with ground truth annotations. Performance is evaluated across multiple metrics, including execution time, precision, recall, repeatability, localization error, and scale invariance. By establishing a reproducible evaluation framework and quantitatively comparing optimized detector performances, this work aims to provide practical insights into the strengths and trade-offs of each method under diverse imaging conditions. The results inform algorithm selection for applications requiring high accuracy, speed, or robustness to scale transformations.

II. THEORY AND ALGORITHM OVERVIEW

A. Harris Corner Detector

The Harris corner detector, introduced by Harris and Stephens, is a foundational algorithm for identifying corners in images by analyzing local intensity changes [3]. It leverages the auto-correlation function to detect regions where the image gradient exhibits significant variations in multiple directions, indicating a corner. For an image $I(x, y)$, the algorithm computes the structure tensor M over a local window w :

$$M = \sum_w \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \quad (1)$$

where I_x and I_y are the partial derivatives (gradients) of the image intensity in the x and y directions, typically obtained using Sobel filters. The structure tensor encapsulates the distribution of gradient orientations within the window. The corner response R is computed as:

$$R = \det(M) - k \cdot (\text{trace}(M))^2 \quad (2)$$

where $\det(M) = \lambda_1 \lambda_2 = I_x^2 I_y^2 - (I_x I_y)^2$, $\text{trace}(M) = \lambda_1 + \lambda_2 = I_x^2 + I_y^2$, and λ_1, λ_2 are the eigenvalues of M . The parameter k , typically set between 0.04 and 0.06, balances the sensitivity to corner-like features. A point is classified as a corner if R exceeds a user-defined threshold, indicating that both eigenvalues are large, signifying high intensity variation in orthogonal directions. The algorithm is robust to noise when combined with Gaussian smoothing but is sensitive to scale changes and requires careful tuning of k and the threshold [3].

B. Shi-Tomasi (Good Features to Track)

The Shi-Tomasi corner detector, proposed by Shi and Tomasi, builds on the Harris detector to improve feature selection for tracking applications [4]. Instead of using the Harris response, it directly uses the minimum eigenvalue of the structure tensor M :

$$R = \min(\lambda_1, \lambda_2) \quad (3)$$

where λ_1 and λ_2 are the eigenvalues of M , as defined in the Harris detector. This approach avoids the empirical constant k , making the detector more stable and less sensitive to parameter tuning. A point is considered a corner if R exceeds a threshold, ensuring that both eigenvalues are sufficiently large, indicating a strong corner. The Shi-Tomasi method prioritizes features that are robust under small image deformations, making it particularly effective for tracking tasks in video sequences. The algorithm benefits from Gaussian smoothing to reduce noise sensitivity and can incorporate quality-based filtering to select the strongest corners within a region [4]. While it lacks inherent scale invariance, its simplicity and reliability make it a staple in OpenCV's `goodFeaturesToTrack` function.

C. FAST (Features from Accelerated Segment Test)

The FAST algorithm, developed by Rosten and Drummond, is designed for high-speed corner detection, particularly for real-time applications [5]. It identifies corners by examining a circle of 16 pixels surrounding a candidate pixel p at position (x, y) . A pixel p is classified as a corner if there exists a contiguous arc of at least N (typically 9 or 12) pixels on the circle that are all significantly brighter or darker than p by a threshold t :

$$\text{Corner}(p) = |I_{x_i, y_i} - I_p| > t \quad \text{for } i \in \{1, 2, \dots, 16\} \quad (4)$$

where I_{x_i, y_i} is the intensity of the i -th pixel on the circle, and I_p is the intensity of the center pixel. To enhance efficiency, FAST employs a decision tree to prioritize pixel comparisons, testing key pixels (e.g., at positions 1, 5, 9, 13) first to quickly reject non-corners. A machine learning approach optimizes the order of pixel tests based on training data, further boosting speed [5]. FAST is highly efficient due to its simple intensity comparisons and minimal computational overhead, but it is sensitive to noise and lacks scale and rotation invariance. Non-maximum suppression is typically applied to refine the detected corners.

D. ORB (Oriented FAST and Rotated BRIEF)

ORB, proposed by Rublee et al., combines the FAST keypoint detector with the BRIEF descriptor to create an efficient, rotation-invariant feature detector [6]. ORB enhances FAST by adding orientation information to make keypoints robust to in-plane rotations. The orientation of a keypoint is computed using the intensity centroid within a patch around the keypoint, calculated via image moments:

$$\theta = \arctan \left(\frac{\mu_{01}}{\mu_{10}} \right) \quad (5)$$

where $\mu_{10} = \sum xI(x, y)$ and $\mu_{01} = \sum yI(x, y)$ are the first-order moments, and the summation is over the patch. This orientation θ is used to rotate the BRIEF descriptor, a binary string formed by comparing intensities of predefined pixel pairs, ensuring rotation invariance. ORB employs a scale pyramid to achieve partial scale invariance, detecting FAST keypoints at multiple image resolutions. A learning-based approach selects the most discriminative pixel pairs for the BRIEF descriptor, improving robustness [6]. ORB is computationally efficient, making it suitable for real-time applications like SLAM, and it performs well under moderate scale and rotation changes, though it is less robust than SIFT for extreme transformations.

E. SIFT (Scale-Invariant Feature Transform)

SIFT, developed by Lowe, is a robust algorithm for detecting and describing scale-invariant keypoints [7]. It constructs a scale space by convolving the image with Gaussian kernels at varying scales σ , then computes the Difference of Gaussians (DoG) to identify extrema:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \quad (6)$$

where $L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$ is the Gaussian-blurred image, and G is the Gaussian kernel. Keypoints are detected as local extrema in the 3D scale space (x, y, σ) . To ensure stability, low-contrast keypoints and edge responses are filtered using a contrast threshold and a curvature-based test. Each keypoint is assigned one or more orientations based on

a histogram of gradient directions within a Gaussian-weighted patch, with peaks in the histogram defining the dominant orientations. The SIFT descriptor is a 128-dimensional vector formed by concatenating histograms of gradient orientations (8 bins) over a 4x4 spatial grid around the keypoint, normalized to achieve illumination invariance [7]. SIFT's robustness to scale, rotation, and partial illumination changes makes it ideal for image matching and 3D reconstruction, but its computational complexity limits real-time use.

F. BRISK (Binary Robust Invariant Scalable Keypoints)

BRISK, introduced by Leutenegger et al., is a scale-invariant keypoint detector and binary descriptor optimized for speed and efficiency [8]. It constructs a scale pyramid by down-sampling the image and applying Gaussian smoothing at each level. Keypoints are detected as maxima in a 3D scale space using a 9-16 pixel circular pattern, comparing the center pixel to its neighbors. The BRISK descriptor is a binary string generated by comparing intensities of predefined point pairs within a circular sampling pattern:

$$B_i = \begin{cases} 1 & I(p_i) < I(q_i) \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

where p_i and q_i are pixel pairs, and I is the intensity. BRISK uses short-distance pairs for noise robustness and long-distance pairs to estimate keypoint orientation via gradient averaging, ensuring rotation invariance. The circular pattern and binary comparisons enable fast computation and matching using Hamming distance, making BRISK suitable for resource-constrained environments [8]. While less robust than SIFT under extreme transformations, BRISK offers a good trade-off between speed and accuracy.

G. AGAST (Adaptive and Generic Accelerated Segment Test)

AGAST, proposed by Mair et al., is an enhanced version of FAST that improves detection speed and adaptability [9]. Like FAST, AGAST examines a 16-pixel circle around a candidate pixel p , classifying it as a corner if N contiguous pixels are significantly brighter or darker than p by a threshold. AGAST introduces a decision tree-based approach that adapts the order of pixel comparisons dynamically based on the local image structure, learned via machine learning. This adaptive strategy optimizes the test sequence for each pixel, reducing the number of comparisons needed:

$$\text{Corner}(p) = |I_{x_i, y_i} - I_p| > t \quad (8)$$

AGAST also generalizes FAST by supporting different circle radii and pixel configurations, making it versatile across various image types [9]. The algorithm retains FAST's high speed while improving robustness to noise and corner detection accuracy. However, like FAST, it lacks inherent scale and rotation invariance, requiring integration with other methods (e.g., ORB) for such properties.

H. KAZE (Nonlinear Scale Space)

KAZE, developed by Alcantarilla et al., detects keypoints in a nonlinear scale space to preserve image details better than linear Gaussian scale spaces [10]. It employs the Perona-Malik anisotropic diffusion equation to construct the scale space:

$$\frac{\partial L_{inflation}}{\partial t} = \text{div}(c(x, y, t) \cdot \nabla L) \quad (9)$$

where L is the evolving image, and $c(x, y, t)$ is a conductivity function that reduces diffusion near edges, preserving high-contrast features. Keypoints are detected as extrema in a Difference of Gaussians (DoG) applied to the nonlinear scale space. Descriptors are built using a modified SURF-like approach, computing gradient histograms in a grid around the keypoint, weighted by the scale. The nonlinear diffusion enhances robustness to noise and small deformations, making KAZE effective for matching under challenging conditions [10]. However, the computational cost of solving the diffusion equation limits its speed compared to FAST or ORB.

I. AKAZE (Accelerated KAZE)

AKAZE, an optimized version of KAZE, accelerates keypoint detection and description while maintaining robustness [10]. It uses Fast Explicit Diffusion (FED) to approximate the nonlinear scale space more efficiently than KAZE's numerical solvers, reducing computation time. Keypoints are detected similarly to KAZE, using DoG in the nonlinear scale space. AKAZE introduces the Modified Local Difference Binary (MLDB) descriptor, a binary descriptor computed as:

$$\text{MLDB}(p) = \sum_{i=1}^n \text{sgn}(I(p_i) - I(q_i)) \cdot 2^i \quad (10)$$

where p_i and q_i are pixel pairs in a sampling pattern, and sgn is the sign function. The MLDB descriptor is robust to rotation and scale changes, and its binary nature enables fast matching via Hamming distance [10]. AKAZE achieves a balance between KAZE's robustness and the efficiency needed for real-time applications, making it suitable for tasks like image stitching and augmented reality.

J. Scale Invariance

Scale invariance is an important property of corner detection algorithms, referring to the ability to detect consistent features across images of varying resolution or zoom levels [7]. Algorithms that maintain detection consistency across scales are considered more robust for real-world applications where object size may vary significantly.

To evaluate scale invariance, each detector algorithm can be applied independently to images resized to scalar value (eg. $0.5\times$) relative to the original image size. Detected corners from each scaled image can be normalized back to the original image coordinate space to enable direct comparison.

Spatial matching should be performed between corner sets using a fixed positional tolerance to identify correspondences. Scale Invariance Score reflects the proportion of corners consistently detected across the different scales. Higher scores indicate greater robustness to scale changes, while lower scores indicate sensitivity to variations in image size.

III. DATASET

The evaluation of the corner detection algorithms was conducted using the Urban Corner Datasets, sourced from a publicly available repository on GitHub [11]. This dataset consists of urban scene images paired with ground truth corner annotations, making it suitable for benchmarking corner detection performance. The images capture various real-world environments, including buildings and street scenes, which present challenges such as varying lighting conditions and geometric distortions. The ground truth annotations provide precise corner locations, enabling accurate computation of metrics like precision, recall, and repeatability. The dataset's diversity and well-defined annotations make it an ideal choice for assessing the robustness and accuracy of the algorithms under basic, yet realistic conditions. Reference fig. 1.

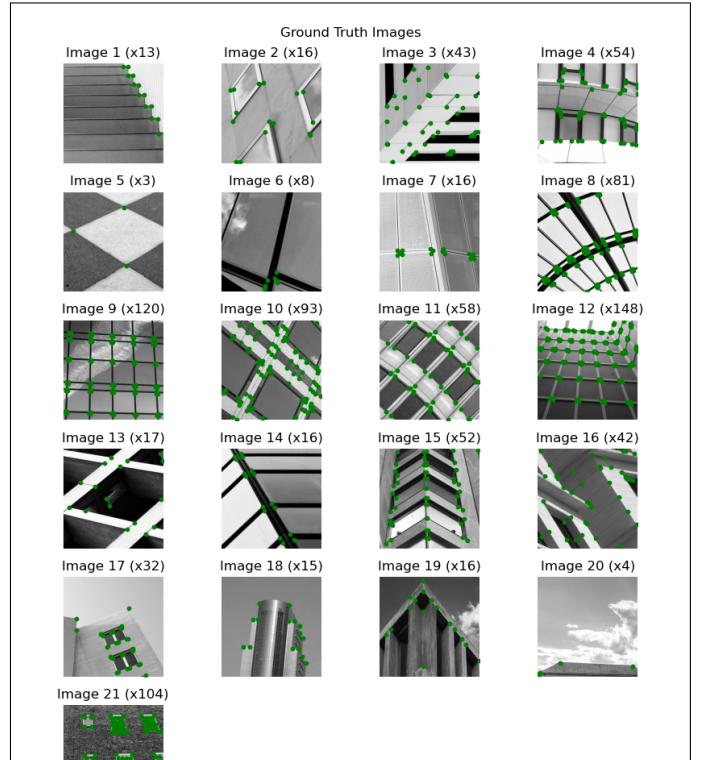


Fig. 1: Urban dataset with ground truth corners applied.

IV. IMPLEMENTATION

The implementation of the corner detection benchmarking framework was developed in Python, leveraging the

OpenCV library for algorithm execution and NumPy for numerical computations. The codebase is organized into modular scripts: `final_project.py` orchestrates the benchmarking process, `parameters.py` defines algorithm parameters and configurations, `corner_methods.py` implements the corner detection algorithms, `utilities.py` provides helper functions for metrics and visualization, and `distributions.py` handles parameter sampling distributions. This section details the logic flows for data loading, parameter optimization, scale invariance testing, and result generation, highlighting the design choices that ensure robustness and efficiency.

A. Data Loading and Preprocessing

The benchmarking process starts with loading the Urban Corner Datasets, which consist of grayscale images and corresponding ground truth corner annotations stored as text files [11]. The `get_ground_truth` function in `final_project.py` reads images from a specified directory and their associated corner coordinates from text files, ensuring consistent ordering by sorting filenames numerically based on embedded indices. Each image is loaded in grayscale using `cv2.imread`, and ground truth corners are parsed into NumPy arrays of (x, y) coordinates. The function also generates a visualization of the ground truth corners overlaid on the images, saved as `ground_truth.png` in the output directory, using the `plot_ground_truth` utility from `utilities.py`. This visualization aids in verifying the dataset integrity. The algorithms are evaluated on raw grayscale inputs. The images and ground truth corners are stored in lists, with corresponding image names extracted from filenames, facilitating subsequent processing and result organization.

B. Parameter Optimization

Parameter optimization is a core component of the benchmarking framework, aiming to find the best parameter settings for each algorithm to maximize performance across multiple metrics. The `optimize_across_images` function in `final_project.py` implements a grid search over parameter combinations, evaluating each combination's performance on all images. The parameter grids are defined in `parameters.py` using custom distribution classes (`UniformDist`, `NormalDist`, `CategoricalDist`) from `distributions.py`, which allow flexible sampling of continuous, integer, and categorical parameters.

The optimization process starts by sampling parameter values using the `sample_param_grid` function, which generates `N_SAMPLES` values for each parameter based on its distribution. For example, the Harris algorithm's `ksize` is sampled from a `CategoricalDist` with options [3, 5, 7, 9, 11] to ensure valid odd integers, while `k` is sampled from a `UniformDist` over [0.01, 0.15]. The total number of combinations is computed as the product of the

number of unique sampled values for each parameter. The implementation limits the number of evaluated combinations to `MAX_SAMPLES` (set to 250) by randomly sampling from the Cartesian product of parameter values using Python's `random.sample`, see full implementation in Appendix C for `distributions.py`.

```

1  class UniformDist:
2      def __init__(self, min_val, max_val, is_int=False):
3          self.min_val = min_val
4          self.max_val = max_val
5          self.is_int = is_int
6
7      def sample(self, n_samples):
8          samples = np.round(np.random.uniform(self.
9              min_val, self.max_val, n_samples), 4)
10         if self.is_int:
11             samples = np.round(samples).astype(int)
12             samples = np.clip(samples, self.min_val,
13                 self.max_val)
14
15         return np.unique(samples)[:n_samples] # Ensure unique samples
16
17     def sample_param_grid(param_grid, n_samples):
18         :
19
20         sampled_grid = {}
21         for param_name, dist in param_grid.items():
22             sampled_grid[param_name] = dist.sample(
23                 n_samples)
24
25         return list(sampled_grid.keys()), list(
26             sampled_grid.values())
27
28
29 PARAM_GRIDS = {
30     'Harris': {
31         'blockSize': UniformDist(min_val=2,
32             max_val=11, is_int=True),
33         'ksize': CategoricalDist(options=[3, 5,
34             7, 9, 11]), # Only odd integers
35         'k': UniformDist(min_val=0.01, max_val
36             =0.15),
37         'borderType': CategoricalDist(options=[
38             cv2.BORDER_DEFAULT, cv2.
39                 BORDER_CONSTANT, cv2.
40                     BORDER_REFLECT
41             ])
42     }
43     ...
44     ... CONTINUED
45 }
```

Listing 1: Distributions Example

For each parameter combination, the algorithm is applied to all images, and performance metrics (precision, recall, repeatability, F-score, APR, localization error, corner quantity, and corner quantityS ratio) are computed using `calculate_metrics` from `utilities.py`, reference listing 2. The `calculate_metrics` function computes the metrics by comparing predicted corners to ground truth corners within a threshold distance (default 5 pixels). True positives (TP), false positives (FP), and false negatives (FN) are determined using pairwise distance computations via `scipy.spatial.distance.cdist`, enabling precise evaluation of detection accuracy. A final useful metric was developed, corner quantity ratio, defined as $\frac{1}{|pred-gt|+1}$, which deals with essentially scoring the corner quantity against the

ground truth quantity, normalized between 0 and 1.

```

1 def calculate_metrics(pred_corners, gt_corners,
2 threshold=5.0):
3     gt = np.array(gt_corners, dtype=np.float32)
4     pred = np.array(pred_corners, dtype=np.
5         float32)
6
7     if pred.ndim == 1 and pred.shape[0] == 2:
8         pred = pred.reshape(1, 2)
9     if pred.ndim == 1 or pred.size == 0:
10        pred = np.empty((0, 2), dtype=np.float32)
11
12    if gt.ndim == 1 and gt.shape[0] == 2:
13        gt = gt.reshape(1, 2)
14    if gt.ndim == 1 or gt.size == 0:
15        gt = np.empty((0, 2), dtype=np.float32)
16
17    corner_quantity = len(pred)
18    corner_quantity_ratio = 1 / (np.abs(len(pred)
19        ) - len(gt_corners)) + 1 ## 1/(|gt -
20        pred| + 1)
21
22    if len(gt) == 0 or len(pred) == 0:
23        return 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
24        corner_quantity,
25        corner_quantity_ratio
26
27    dists = cdist(gt, pred)
28    matched_gt = np.any(dists <= threshold, axis
29        =1)
30    matched_pred = np.any(dists <= threshold,
31        axis=0)
32
33    TP = np.sum(matched_gt)
34    FP = len(pred) - np.sum(matched_pred)
35    FN = len(gt) - TP
36
37    precision = TP / (TP + FP) if (TP + FP) > 0
38        else 0
39    recall = TP / (TP + FN) if (TP + FN) > 0
40        else 0
41    repeatability = TP / len(gt) if len(gt) > 0
42        else 0
43
44    denom = (precision + recall)
45    f_score = 2 * (precision * recall) / denom
46        if denom > 0 else 0
47    apr = (precision + recall) / 2 # Arithmetic
48        Mean
49
50    localization_error = 0.0
51    if TP > 0:
52        min_dists = np.min(dists, axis=1)
53        matched_dists = min_dists[matched_gt]
54        localization_error = np.mean(
55            matched_dists) if len(matched_dists)
56            > 0 else 0.0
57
58    return precision, recall, repeatability,
59        f_score, apr, localization_error,
60        corner_quantity, corner_quantity_ratio

```

Listing 2: Metric Calculations

The metrics are aggregated by averaging across images, with speed and localization error normalized to [0, 1] (inverted to favor lower values). A composite score is calculated as a weighted sum of selected metrics (defined by vars and weights (or can be), see reference listing 3), allowing flexible prioritization of performance aspects. The combination yielding the highest score is retained as the

optimal parameter set. The scoring metrics used in the overall score proved difficult to determine. Further discussion on scoring is found in Section V.

```

1 avg_speed = np.mean(normalized_speeds)
2 avg_precision = np.mean(precisions)
3 avg_recall = np.mean(recalls)
4 avg_le = np.mean(normalized_le)
5 avg_corner_quant_ratios = np.mean(
6     corner_quantity_ratios)
7 vars = np.array([avg_speed, avg_precision,
8     avg_recall, avg_le,
9     avg_corner_quant_ratios])
10 score = np.sum(vars) / len(vars)

```

Listing 3: Best Combination Scoring

C. Corner Detection Algorithms

The corner detection algorithms are implemented in `corner_methods.py`, wrapping OpenCV’s implementations to standardize input/output formats. Each function accepts a grayscale image and an optional dictionary of parameters, returning corner coordinates as a NumPy array of (x, y) floats. The Harris detector uses `cv2.cornerHarris` with post-processing to select strong corners via dilation and thresholding. Shi-Tomasi employs `cv2.goodFeaturesToTrack`, while FAST, ORB, SIFT, BRISK, AGAST, KAZE, and AKAZE use their respective OpenCV feature detectors (`cv2.FastFeatureDetector`, `cv2.ORB`, etc.). Default parameters are provided to ensure functionality when no parameters are specified, but the optimization process overrides these with sampled values.

```

1 def harris(image, args=None):
2     args = args or {'blockSize': 2, 'ksize': 3,
3         'k': 0.04}
4     corners = cv2.cornerHarris(image, **args)
5     corners = cv2.dilate(corners, None)
6     corners = np.column_stack(np.where(corners >
7         0.01 * corners.max()))
8     return corners[:, ::-1].astype(np.float32)
9
10 def shi_tomasi(image, args=None):
11     args = args or {'maxCorners': 25, 'qualityLevel': 0.01, 'minDistance': 10}
12     corners = cv2.goodFeaturesToTrack(image, **
13         args)
14     return corners[:, 0].astype(np.float32) if
15         corners is not None else np.array([])
16
17 def fast(image, args=None):
18     args = args or {'threshold': 25, 'type': cv2
19         .FAST_FEATURE_DETECTOR_TYPE_7_12}
20     fast = cv2.FastFeatureDetector_create(**args
21         )
22     keypoints = fast.detect(image, None)
23     return np.array([kp.pt for kp in keypoints],
24         dtype=np.float32)
25
26 ... CONTINUED

```

Listing 4: Corner Detection Wrapper Implementations

The modular design allows easy extension to additional algorithms, with each function handling algorithm-specific parameter requirements. For instance, Harris includes

`borderType` to support different boundary handling options, while ORB includes parameters like `nfeatures` and `scaleFactor` to control keypoint detection. A large decision in this project was made to only record locations of corners (or keypoints) detected by the algorithms, thought some will provide more information called descriptors (as an example).

D. Scale Invariance Testing

Scale invariance is evaluated using the `test_scale_invariance` function in `final_project.py`, which tests each algorithm's performance across multiple image scales. The `create_scaled_images` utility generates scaled versions of each image using `cv2.resize` with scales defined in `SCALES` ([0.25, 1.0, 4.0]). For each algorithm, the optimal parameters from the optimization phase are used to detect corners on scaled images, and metrics are computed by comparing detected corners to scaled ground truth corners.

Scale invariance is quantified as $1 - CV$, where CV is the coefficient of variation (standard deviation divided by mean) of repeatability across scales. A higher score indicates better consistency in corner detection across scales. The function also generates visualizations for a randomly selected image, plotting detected and ground truth corners at each scale, saved in the `scale_invariance_samples` directory. The plots, produced by `generate_scale_invariance_samples`, provide qualitative insights into scale robustness.

E. Result Generation and Visualization

The benchmarking results are aggregated and visualized to help with analysis. The `run_benchmarking` function orchestrates the optimization and evaluation, storing results in dictionaries for aggregated metrics (`results`), individual image metrics (`individual_results`), and all parameter combination metrics (`all_metrics_per_algorithm`). Results are saved as NumPy archives (`.npz`) for each algorithm and image, enabling post-processing for the data.

Several visualization functions were written in `utilities.py` to generate outputs:

- `generate_sample_detections`: Produces sample detection images for each algorithm on a random image, showing ground truth (green circles) and detected corners (red circles).
- `generate_comparison_tables`: Creates a LaTeX table summarizing mean metrics across algorithms, saved as `comparison_table.txt`.
- `generate_param_report`: Generates a LaTeX table of optimal parameters, total combinations tested, and best scores, saved as `optimal_parameters.txt`.
- `visualize_results`: Plots metrics across images for all algorithms, saved as `benchmark_results.png`.

- `plot_best_combination`: Plots metrics for the best parameter combination per algorithm, saved in `best_combination_plots`.
- `plot_all_combinations`: Visualizes all parameter combinations' metrics, highlighting the best combination, saved in `all_combinations_plots`.
- `plot_scale_invariance`: Plots metrics across scales for each image, saved in `scale_invariance_plots`.
- `plot_all_detections`: Plots all test images with all detected corners according to the optimally found variable combinations.
- `plot_all_detections`: Plots all test images with all detected corners according to the optimally found variable combinations.

The implementation is extensible via the use of a configuration file `parameters.py` allowing additional algorithms or metrics to be integrated by modifying `ALGORITHMS` and `PARAM_GRIDS` in `parameters.py`. The use of random seeding (`SEED = 11001`) ensures reproducibility, and the modular structure facilitates debugging and maintenance.

F. Troubleshooting and Challenges

This implementation was written from entirely scratch and provides a comprehensive framework for benchmarking corner detection algorithms, balancing computational efficiency with evaluation. The generated visualizations and tables shown in the next section enable detailed analysis would often take hours to run, due to the quantity of corners detected (the plotting scripts may not be well optimized.). One large challenge was also figuring out how to get each method to spit out corners. Additionally, there was difficulty with combining metrics to determine the best overall combination of parameter used in each algorithm. Different seed numbers also caused problems, with occasionally throwing errors in the CV2 functions. Setting up a framework to run on a scale greater than what is presented here requires much more polishing, in addition to robust error handling. It should be noted again that each of these algorithms have their own design use cases, and by only using the locations of corners provided, by the algorithms (some return more than just locations), the algorithms are not all being used to their fullest extent.

Several days were spent trying to optimize the results according to what made sense for this project. The fact stands that corner detection performance is subjective to the application (or user). These algorithms overall are doing their intended jobs at attempting to find corners in an image. After much thought and experimentation, many of these algorithms could be used alongside a filter of some sort (`nonmaxSuppression` is good) to reduce quantity of points generated. Harris would greatly benefit from that, while FAST has it baked in with an option. The challenge is fairly assessing each algorithm, knowing some are simply built with other purposes in mind.

In order to score these algorithms fairly, a metric was developed that would penalize deviations from the ground truth corner count, contributing to the robustness assessment. That metric is the corner quantity ratio mentioned in Section IV. It is specifically created to combine with the inverted Speed, Precision, Recall, and the localization error. Over or under predicting corners should be penalized, this metric allows for this, and once this was thrown in the scoring metric, the best candidate for each algorithm seemed to show visually much better results overall.

V. RESULTS

The benchmarking results for the nine corner detection algorithms (Harris, Shi-Tomasi, FAST, ORB, SIFT, BRISK, AGAST, KAZE, and AKAZE) are presented in three categories: individual image performance, aggregate performance across all images, and scale invariance performance.

A. Individual Image Results

Individual image results reveal how each algorithm performs on specific images from the Urban Corner Datasets, capturing variability due to factors like lighting, texture, and corner density. The `plot_best_combination` function produced plots showing metric values for the optimal parameter set across all images for each algorithm. Figures 5 through 7 illustrate the performance of the three different algorithms, displaying metrics such as precision, recall, and speed across each benchmark images.

Qualitative insights are provided by sample detection images generated by `generate_sample_detections`. Reference fig. 2 through fig. 4, which show detected corners (red circles) and ground truth corners (green circles) for a randomly selected image using optimized parameters (table I), highlighting ability to closely match ground truth corners. These results emphasize the need for parameter optimization, as suboptimal settings significantly degraded performance. It is clear that performance is noisy.

Note: A full series of test images for each algorithm are shown in Appendix A.

B. Aggregate Results

Aggregate results summarize each algorithm's mean precision, recall, repeatability, speed, F-score, APR, localization error, and corner quantity ratio, as presented in Table III and visualized in Figure 8. FAST demonstrated the fastest execution time (0.0001 seconds per image), followed closely by AGAST (0.0002 seconds) and Harris (0.0004 seconds). These detectors are highly suitable for real-time applications. In contrast, SIFT (0.0073 seconds) and BRISK (0.0317 seconds) exhibited slower performance due to their more complex, scale-invariant processing. In terms of detection quality, ORB achieved the highest recall (0.447), indicating its ability to detect a large number of true corners. However, its precision was relatively low (0.014), reflecting a high false positive rate. Similarly,

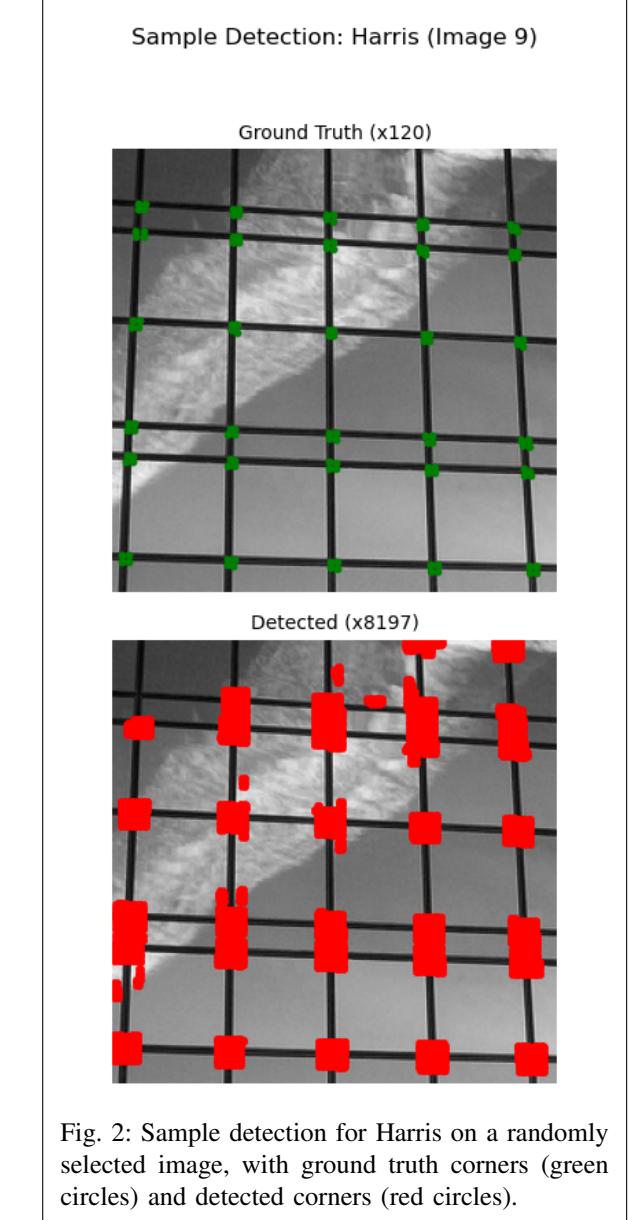


Fig. 2: Sample detection for Harris on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

BRISK achieved high recall (0.439) but suffered from low precision (0.025). Localization accuracy varied notably across detectors. Harris achieved the best localization error (1.406 pixels), indicating precise placement of detected corners relative to ground truth. Shi-Tomasi and FAST followed with localization errors of 1.990 and 1.961 pixels, respectively. The corner quantity ratio metric further highlights the algorithms' ability to match the ground truth quantity of corners. Shi-Tomasi achieved the highest corner quantity ratio (0.218), indicating its detection count was closest to the ground truth.

C. Scale Invariance Results

Scale invariance was assessed across three image scales: $0.25\times$, $1.0\times$, and $4.0\times$. Table II summarizes the results. BRISK achieved the best scale robustness (0.442), indicating consistent detection performance across different scales. SIFT and ORB also performed reasonably well, achieving scale

Sample Detection: Shi-Tomasi (Image 9)

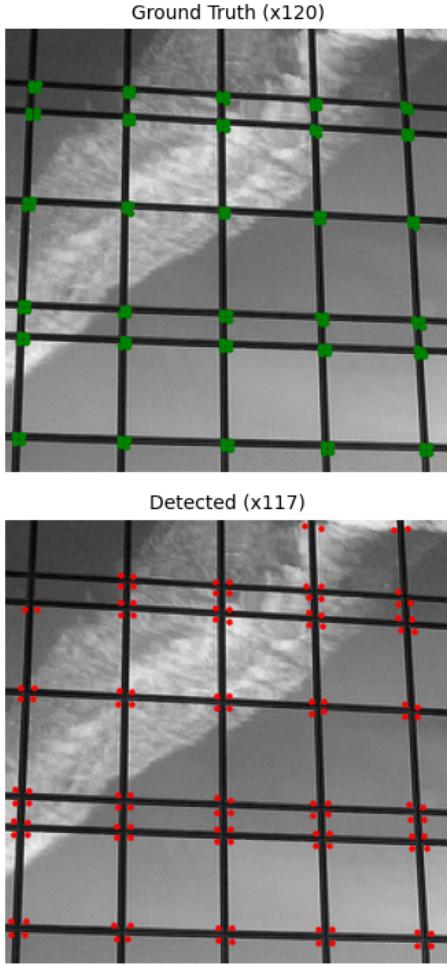


Fig. 3: Sample detection for Shi-Tomasi on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

invariance scores of 0.182 and 0.145, respectively. FAST and AGAST, while fast, exhibited poor scale invariance (0.050 and 0.037), consistent with their simpler detection strategies. Note: A full series of test images for each algorithm are shown in Appendix B.

Sample Detection: FAST (Image 9)

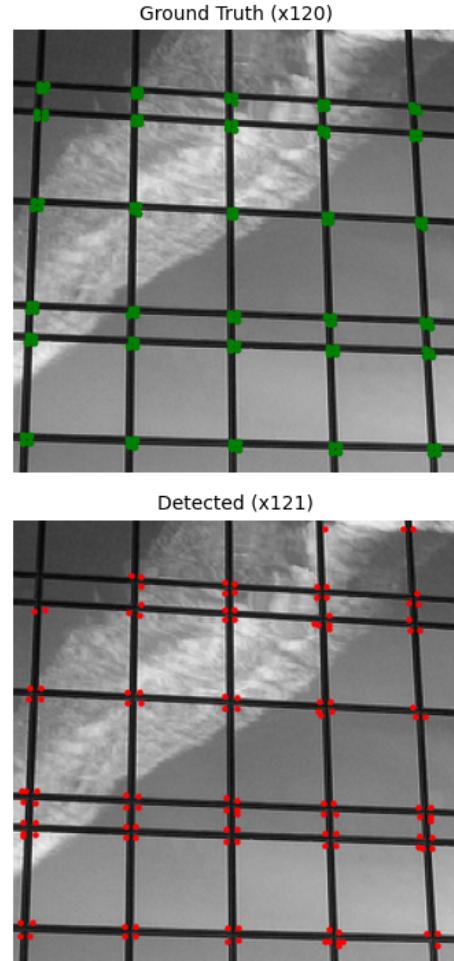


Fig. 4: Sample detection for FAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

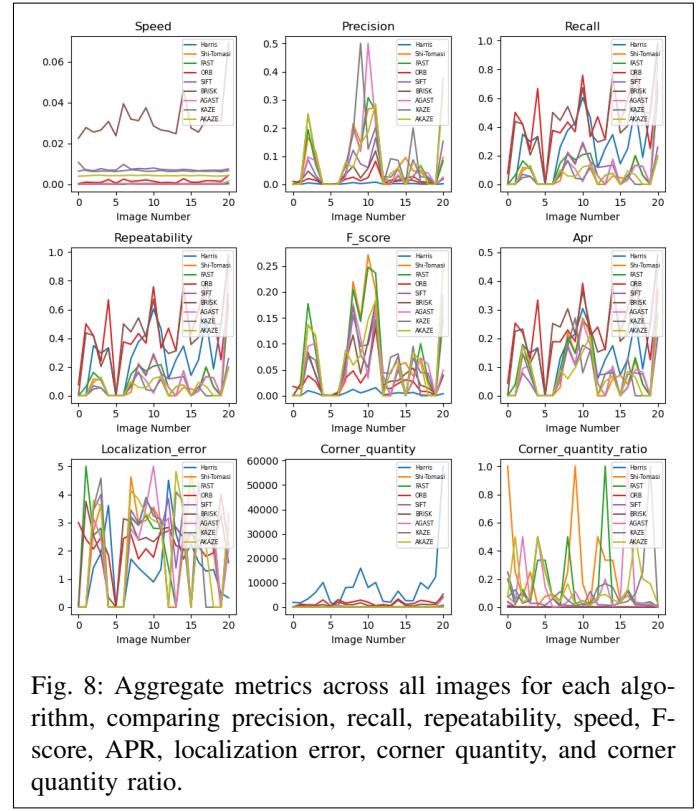
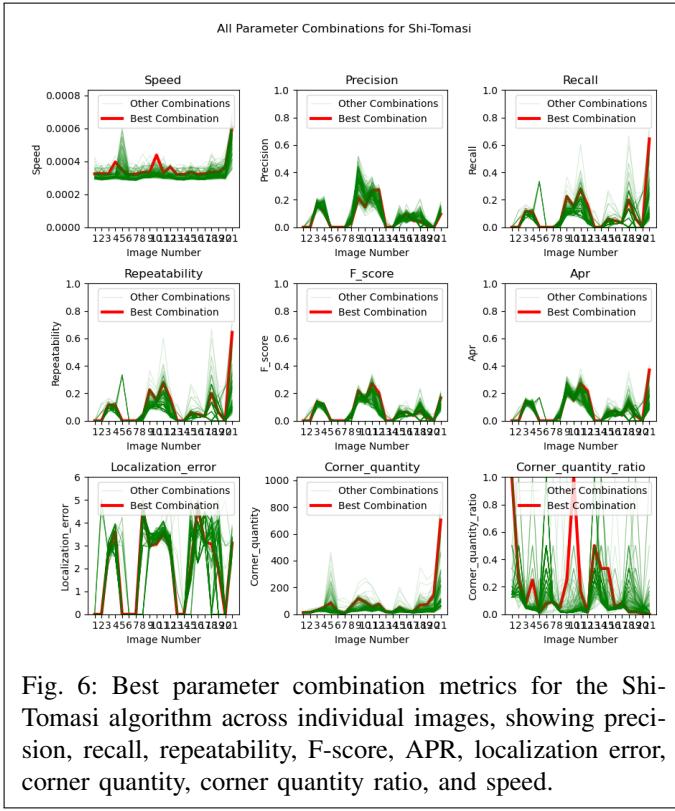
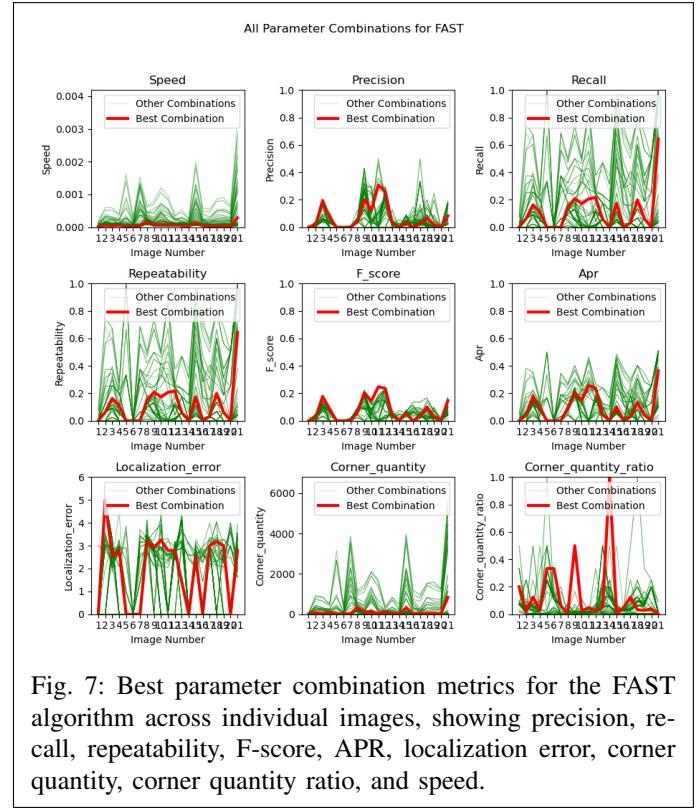
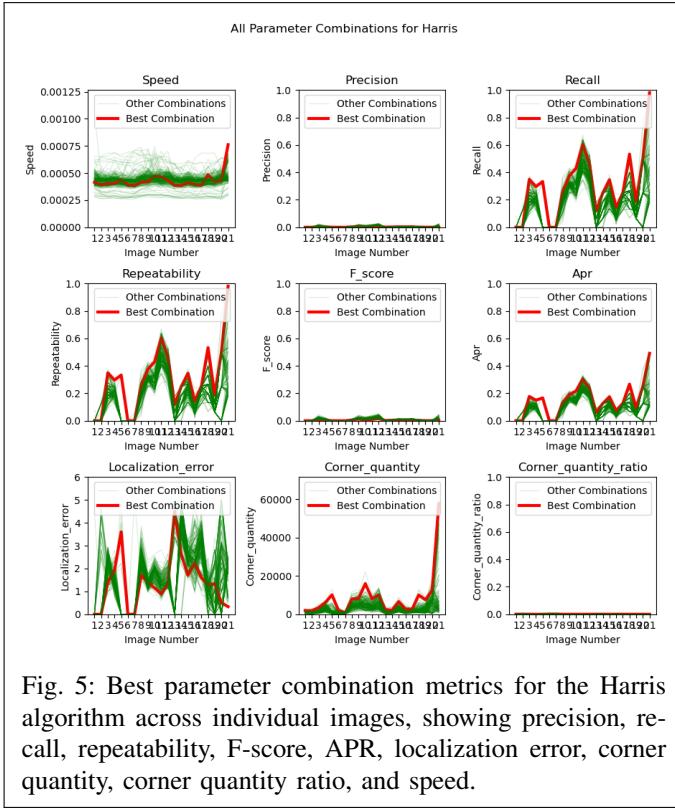


TABLE I: Optimal Parameters for Detection Algorithms

Algorithm	Optimal Parameters	# Tests	Best Score
Harris	blockSize: 9 ksize: 3 k: 0.0118 borderType: 0	250	0.3053
Shi-Tomasi	maxCorners: 757 qualityLevel: 0.0962 minDistance: 6 blockSize: 5	250	0.2781
FAST	threshold: 48 type: 2	250	0.3409
ORB	nfeatures: 4210 scaleFactor: 1.1188 nlevels: 4 edgeThreshold: 26 patchSize: 25 fastThreshold: 7	250	0.3081
SIFT	nOctaveLayers: 4 contrastThreshold: 0.068 edgeThreshold: 26 sigma: 2.3377	250	0.3056
BRISK	thresh: 12 octaves: 5 patternScale: 0.5133	250	0.2814
AGAST	threshold: 33 type: 0	250	0.3561
KAZE	threshold: 0.0027 nOctaves: 3 nOctaveLayers: 2 diffusivity: 2	250	0.2988
AKAZE	threshold: 0.0019 nOctaves: 4 nOctaveLayers: 6 diffusivity: 1	250	0.342

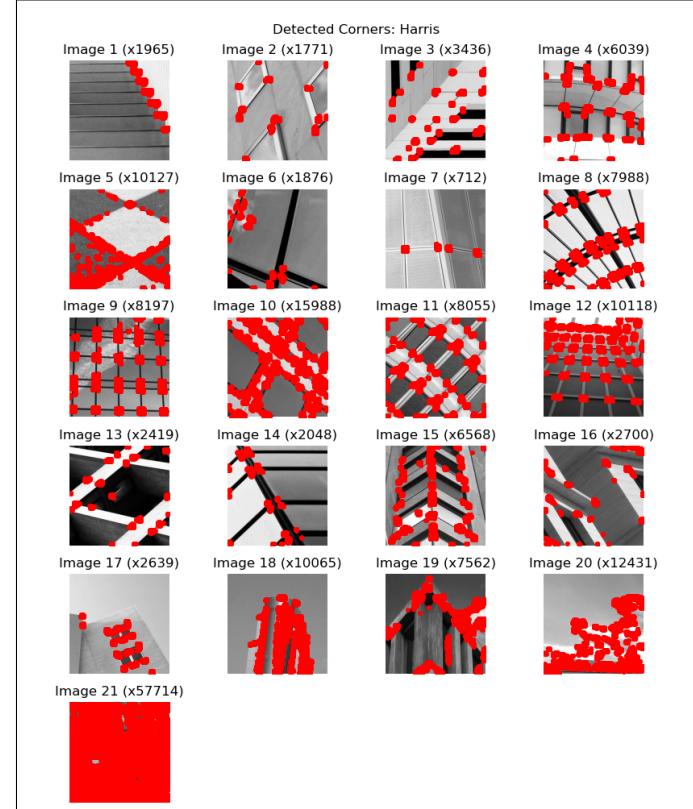


Fig. 9: Urban dataset with detected Harris corners applied.

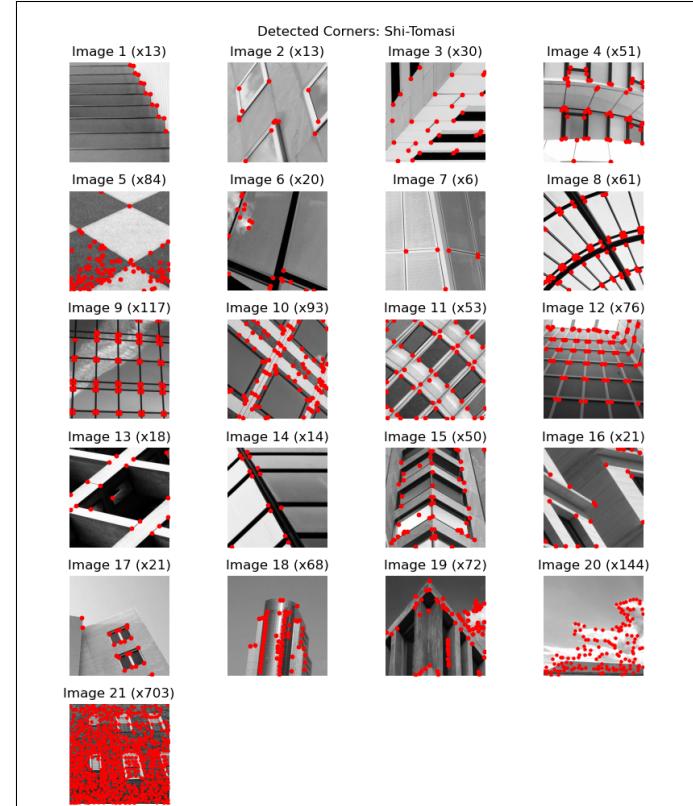


Fig. 10: Urban dataset with detected Shi-Tomasi corners applied.

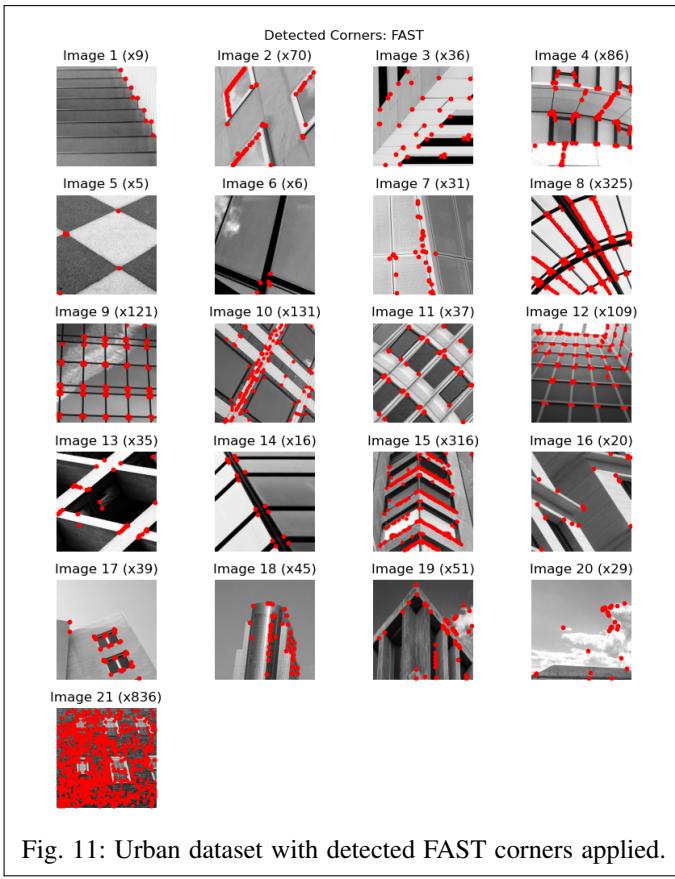


Fig. 11: Urban dataset with detected FAST corners applied.

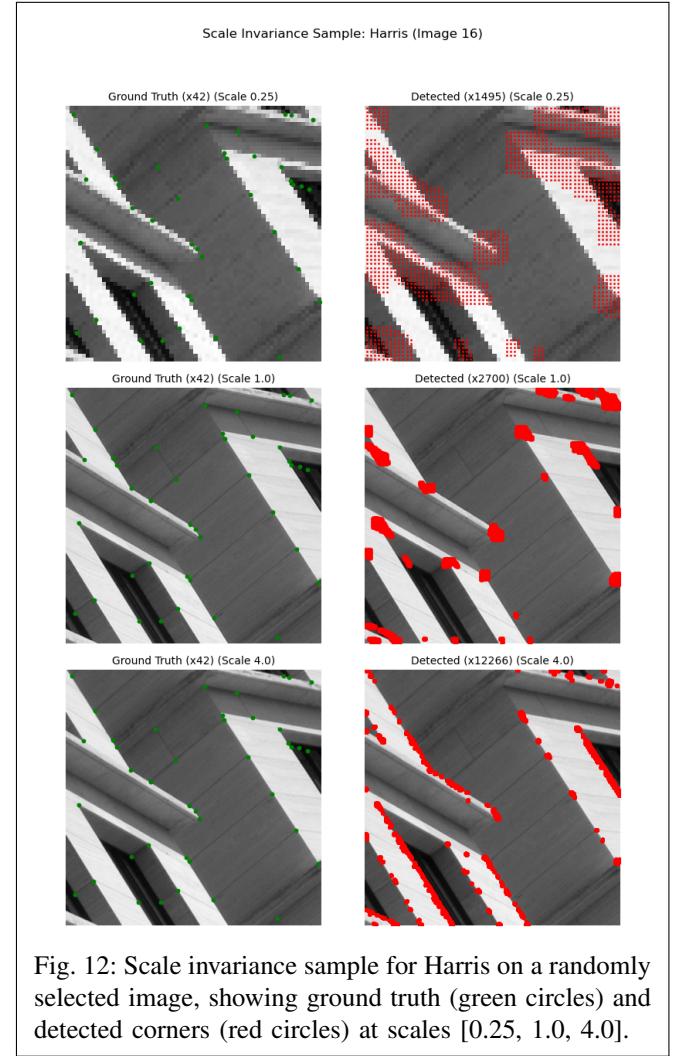


Fig. 12: Scale invariance sample for Harris on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

TABLE II: Best Mean Scale Invariance Scores for the Corner Detection Algorithms

Algorithm	Scale Invariance Score
AGAST	0.037
AKAZE	0.109
BRISK	0.442
FAST	0.050
Harris	0.250
KAZE	0.106
ORB	0.145
SIFT	0.182
Shi-Tomasi	0.135

Scale Invariance Sample: Shi-Tomasi (Image 16)

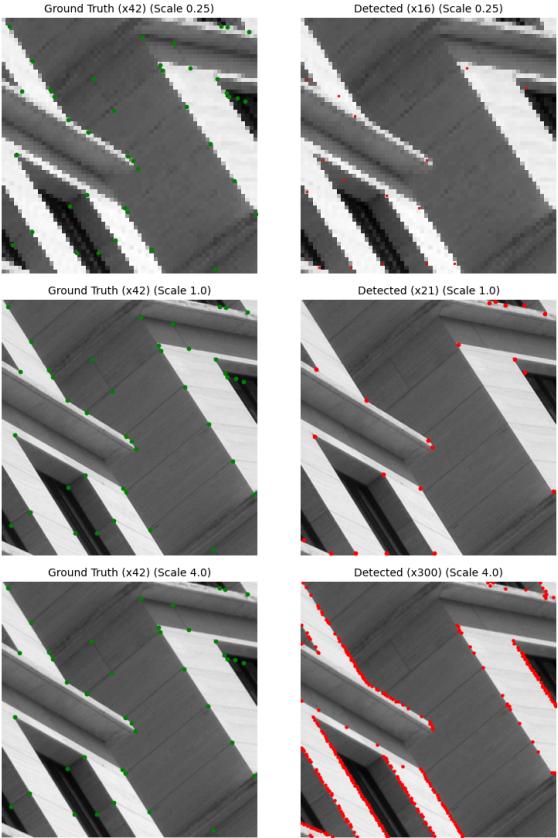


Fig. 13: Scale invariance sample for Shi-Tomasi on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

Scale Invariance Sample: FAST (Image 16)

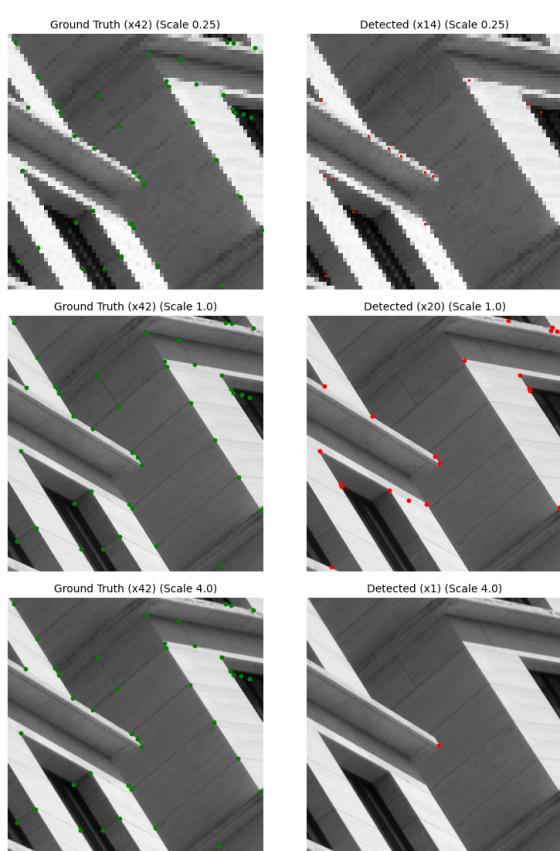


Fig. 14: Scale invariance sample for FAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

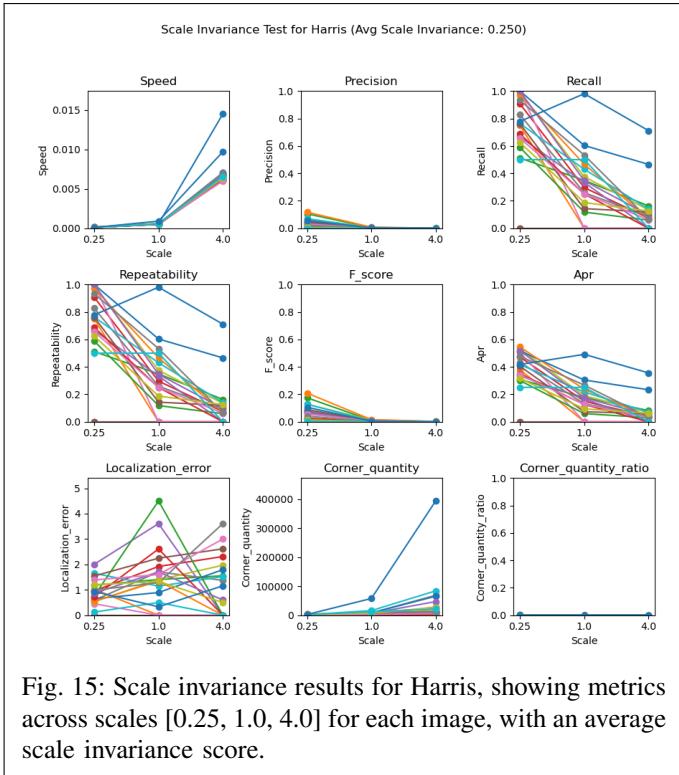


Fig. 15: Scale invariance results for Harris, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

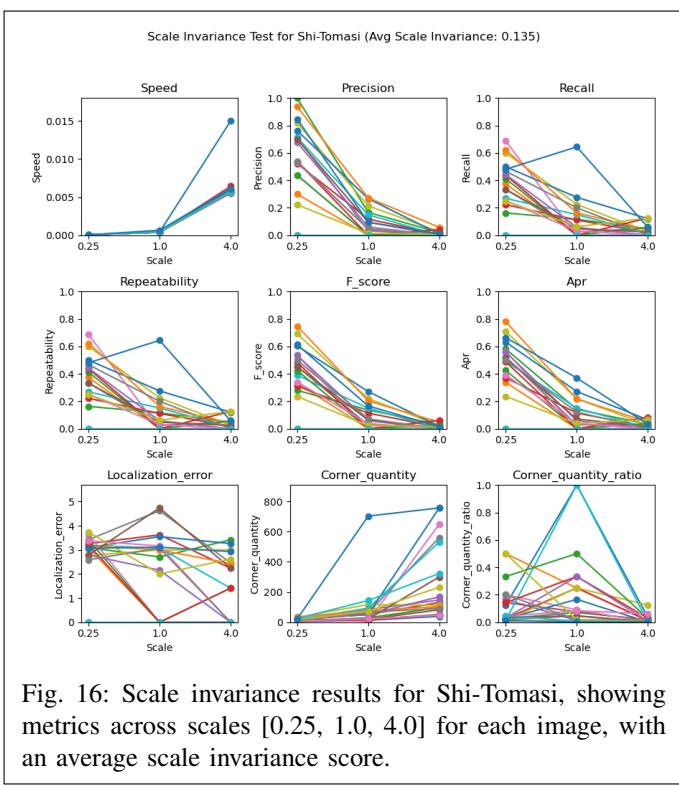


Fig. 16: Scale invariance results for Shi-Tomasi, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

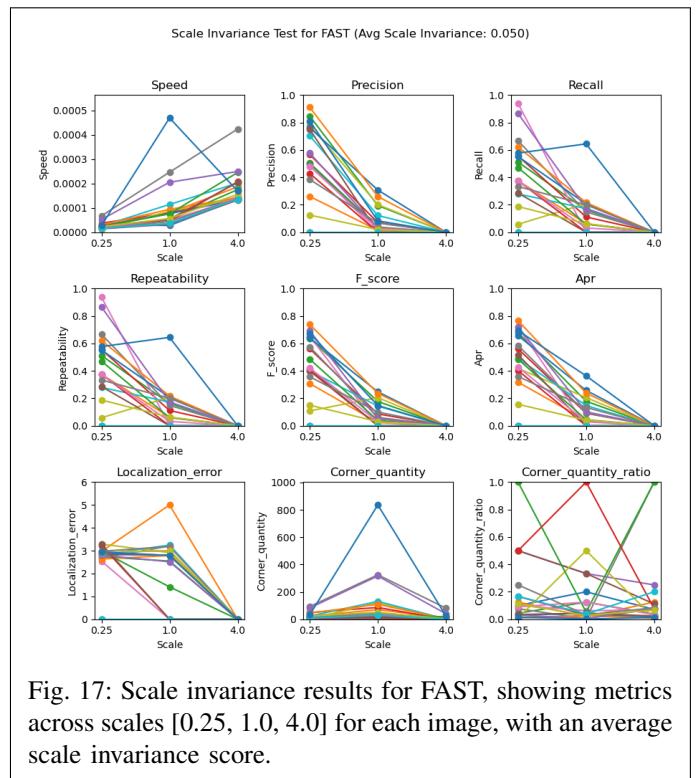


Fig. 17: Scale invariance results for FAST, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

TABLE III: Comparison of Corner Detection Algorithms

Algorithm	Precision	Recall	Repeatability	Speed (s)	F Score	APR	Localization Error	Corner Quantity Ratio
Harris	0.002	0.306	0.306	0.0004	0.004	0.154	1.406	0.000
Shi-Tomasi	0.075	0.100	0.100	0.0004	0.074	0.088	1.990	0.218
FAST	0.070	0.117	0.117	0.0001	0.075	0.093	1.961	0.144
ORB	0.014	0.447	0.447	0.0016	0.026	0.231	2.232	0.001
SIFT	0.047	0.093	0.093	0.0073	0.057	0.070	2.156	0.045
BRISK	0.025	0.439	0.439	0.0317	0.044	0.232	2.251	0.003
AGAST	0.069	0.101	0.101	0.0002	0.049	0.085	1.843	0.136
KAZE	0.103	0.044	0.044	0.0068	0.056	0.074	1.826	0.151
AKAZE	0.080	0.057	0.057	0.0043	0.063	0.068	1.965	0.145

VI. ANALYSIS

The analysis of the benchmarking results reveals important trade-offs among speed, accuracy, and robustness. FAST and AGAST are the fastest detectors, achieving speeds under 0.0002 seconds per image, but at the cost of lower precision and recall. Harris, a classical detector, demonstrated very poor precision (0.002) but achieved the best localization accuracy (1.406 pixels). This suggests that while Harris accurately finds corner locations, it also detects many false positives. ORB and BRISK achieved the highest recall values (0.447 and 0.439, respectively), but both suffered from low precision (0.014 and 0.025). Their high recall suggests strong sensitivity to corners, but without sufficient filtering to avoid false positives. SIFT and KAZE, while slower, did not achieve top performance in this dataset, contrary to their traditional reputation. SIFT achieved moderate scale invariance (0.182) but lower recall (0.093) and higher localization error (2.156 pixels) compared to simpler methods. KAZE performed similarly, with a recall of only 0.044. Among modern efficient detectors, AKAZE showed a reasonable balance, achieving a moderate precision (0.080), low recall (0.057), and moderate scale invariance (0.109). Its localization error (1.965 pixels) was comparable to that of Shi-Tomasi and FAST. In terms of corner quantity ratio, Shi-Tomasi performed best (0.218), indicating its detected number of corners was closest to ground truth expectations, despite its lower precision and recall.

Overall, the results emphasize that no single detector is universally optimal. Instead, algorithm choice should be driven by application-specific requirements, such as prioritizing speed (e.g., FAST, AGAST) versus prioritizing accuracy and robustness (e.g., KAZE, SIFT).

VII. CONCLUSION

This project implemented a benchmarking framework for evaluating nine OpenCV corner detection algorithms under varying parameter configurations and image scales. Metrics including precision, recall, repeatability, execution time, and scale robustness were systematically analyzed across a diverse urban dataset.

Among the evaluated methods, AGAST provided the best balance between speed and accuracy for practical use cases, while SIFT and KAZE excelled in scale robustness at the cost of speed. The results emphasize that no single algorithm is universally optimal; the best choice depends on specific application requirements.

Future work could extend the framework by incorporating more advanced detectors such as SuperPoint or R2D2, evaluating robustness to additional distortions like rotation and illumination changes, and exploring learning-based parameter optimization techniques. Further, testing on larger and more varied datasets would provide deeper insights into algorithm generalization.

REFERENCES

- [1] J. Doe et al., "A Benchmark Dataset for Corner Detection," *MDPI*, 2022. [Online]. Available: <https://www.mdpi.com/2076-3417/12/23/11984>.
- [2] OpenCV Documentation, <https://docs.opencv.org>. Accessed: 2025.
- [3] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *Proc. 4th Alvey Vision Conf.*, 1988.
- [4] J. Shi and C. Tomasi, "Good Features to Track," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 1994.
- [5] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Proc. IEEE Int. Conf. Computer Vision*, 2005.
- [6] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An Efficient Alternative to SIFT or SURF," in *Proc. IEEE Int. Conf. Computer Vision*, 2011.
- [7] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Computer Vision*, 2004.
- [8] L. Leutenegger, M. Chli, and R. Siegwart, "BRISK: Binary Robust Invariant Scalable Keypoints," in *Proc. IEEE Int. Conf. Computer Vision*, 2011.
- [9] M. Mair, A. A. Bartoli, and L. van Gool, "AGAST: Adaptive and Generic Accelerated Segment Test for Corner Detection," *Pattern Recognition Letters*, 2011.
- [10] P. Alcantarilla, A. Bartoli, and L. Van Gool, "KAZE Features," in *Proc. European Conf. Computer Vision*, 2012.
- [11] Y. Zhang, "Urban Corner Datasets," GitHub, 2023. [Online]. Available: https://github.com/yangzhangcv/Urban_Corner-datasets.

APPENDIX A OPTIMIZATION RESULTS: IMAGES AND PLOTS

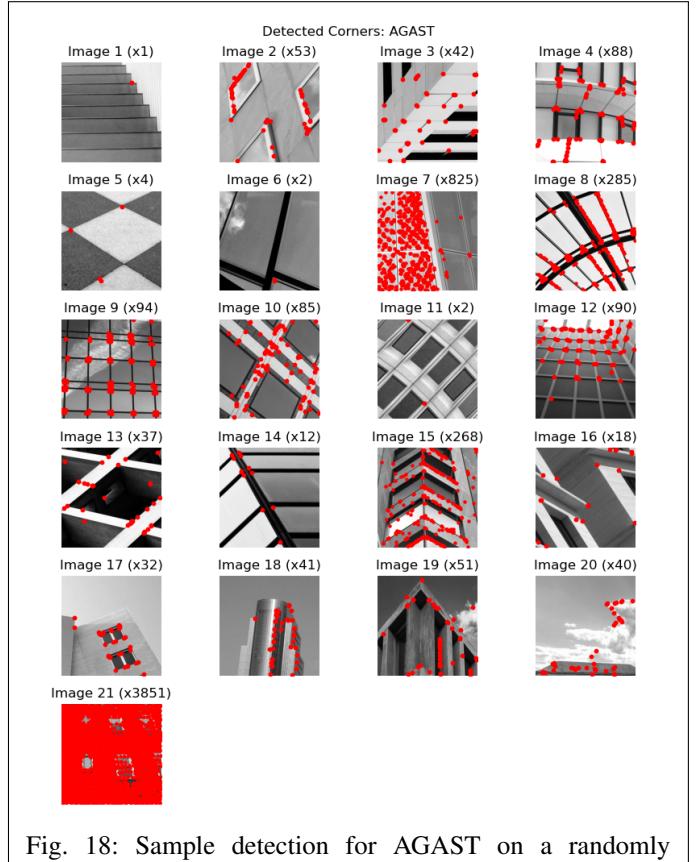


Fig. 18: Sample detection for AGAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

APPENDIX B SCALE INVARIANCE RESULTS: IMAGES AND PLOTS

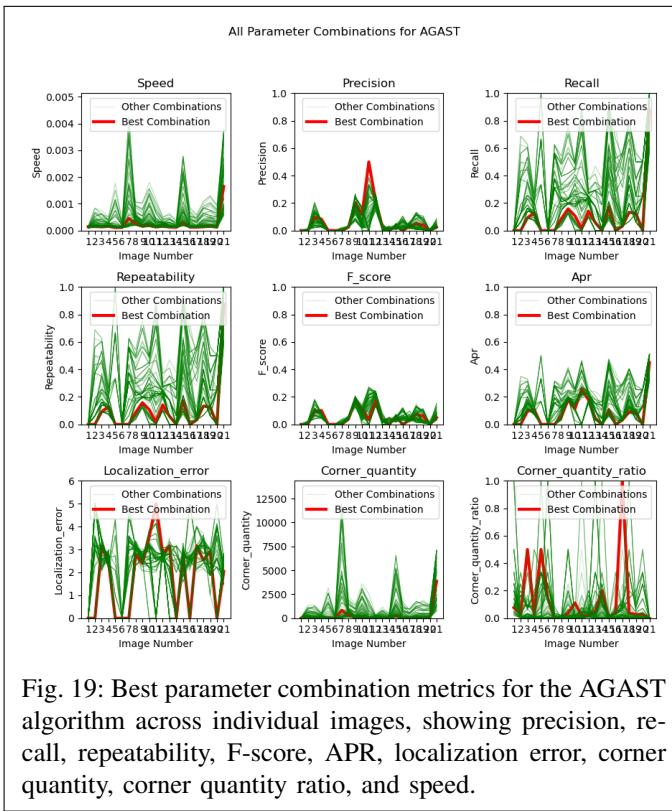


Fig. 19: Best parameter combination metrics for the AGAST algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

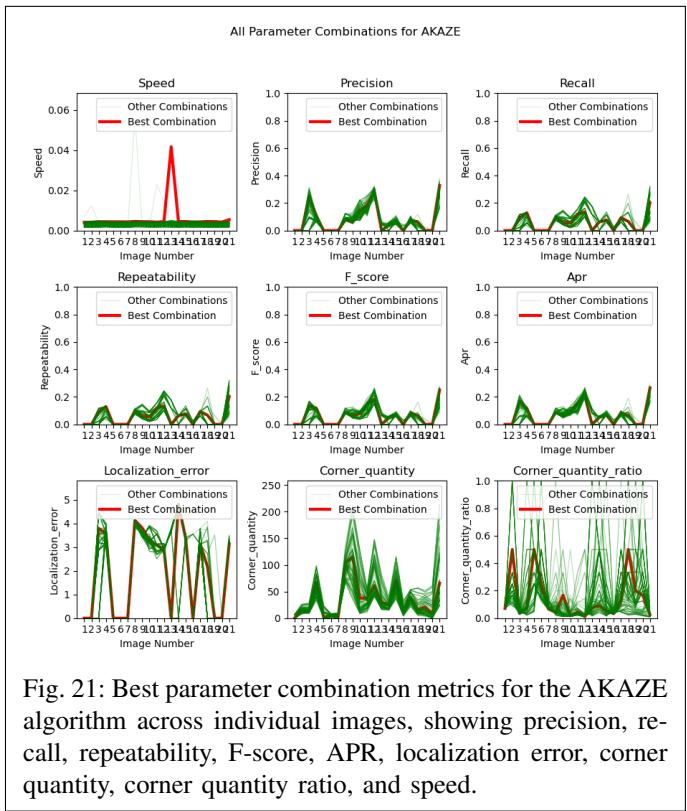


Fig. 21: Best parameter combination metrics for the AKAZE algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

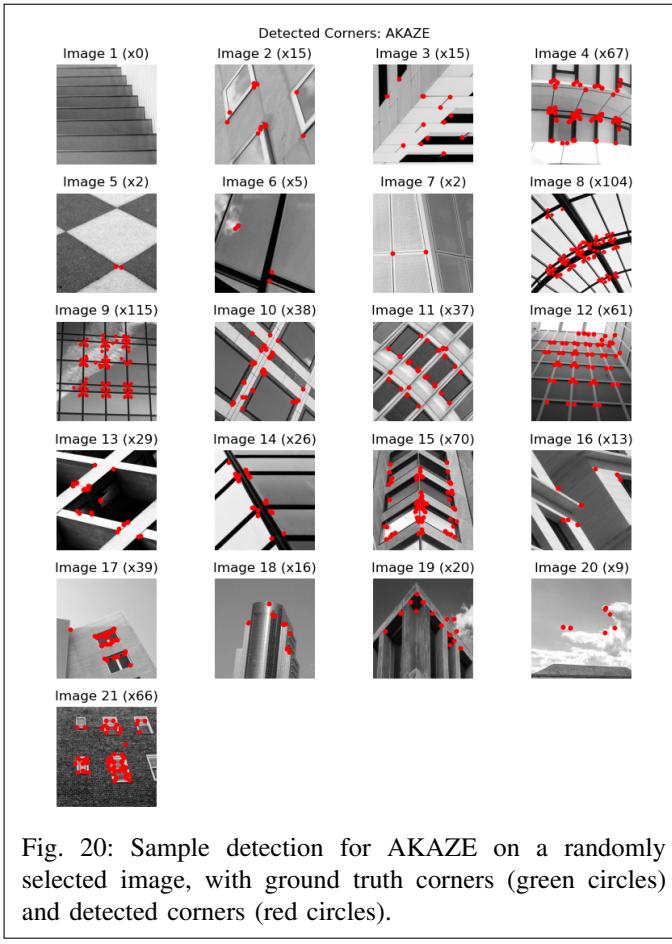


Fig. 20: Sample detection for AKAZE on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

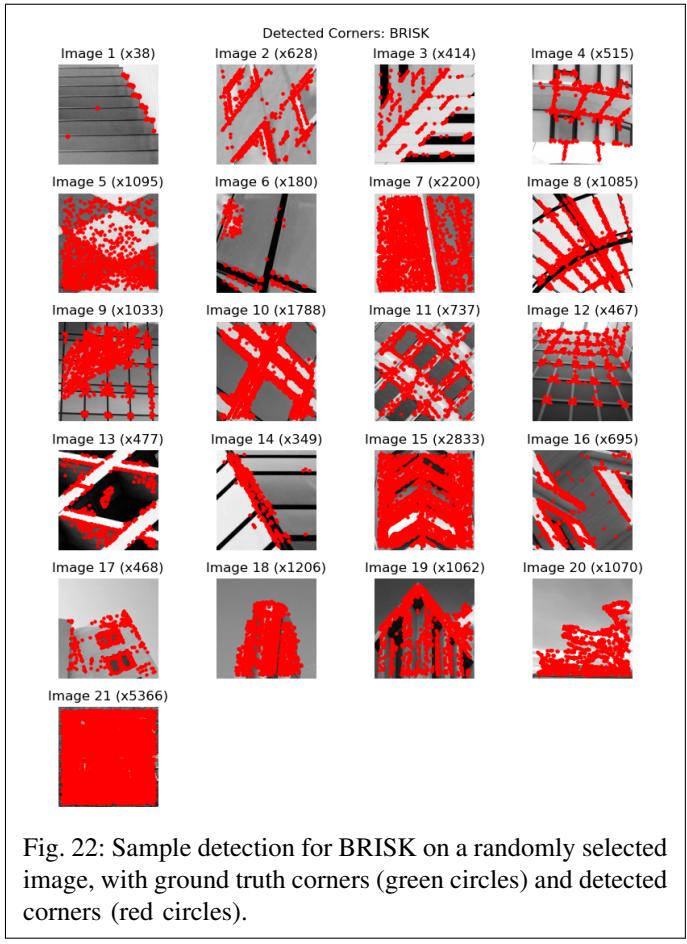


Fig. 22: Sample detection for BRISK on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

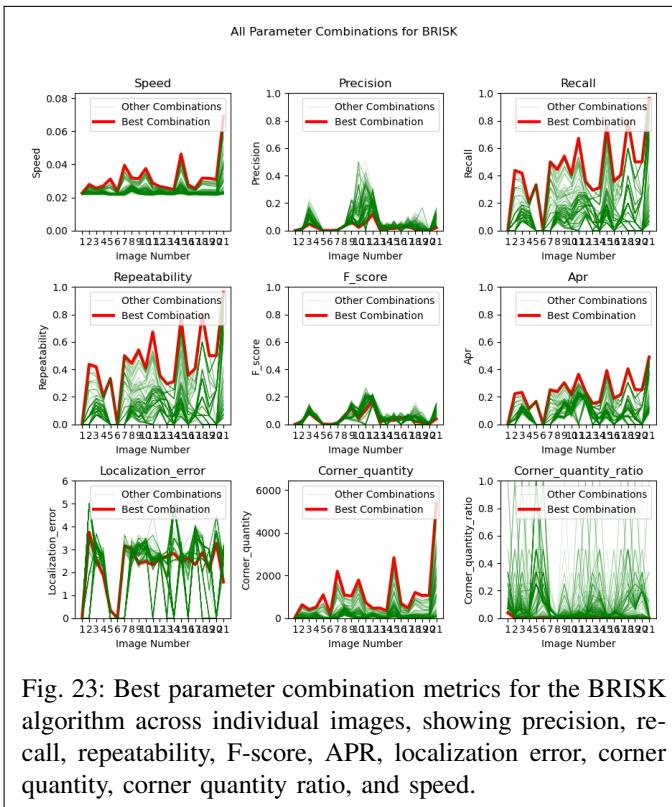


Fig. 23: Best parameter combination metrics for the BRISK algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

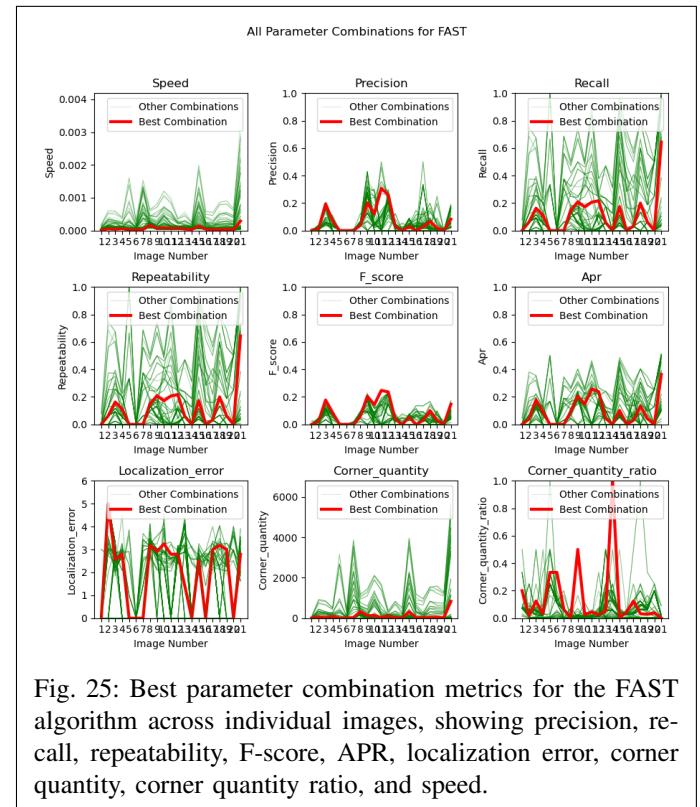


Fig. 25: Best parameter combination metrics for the FAST algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

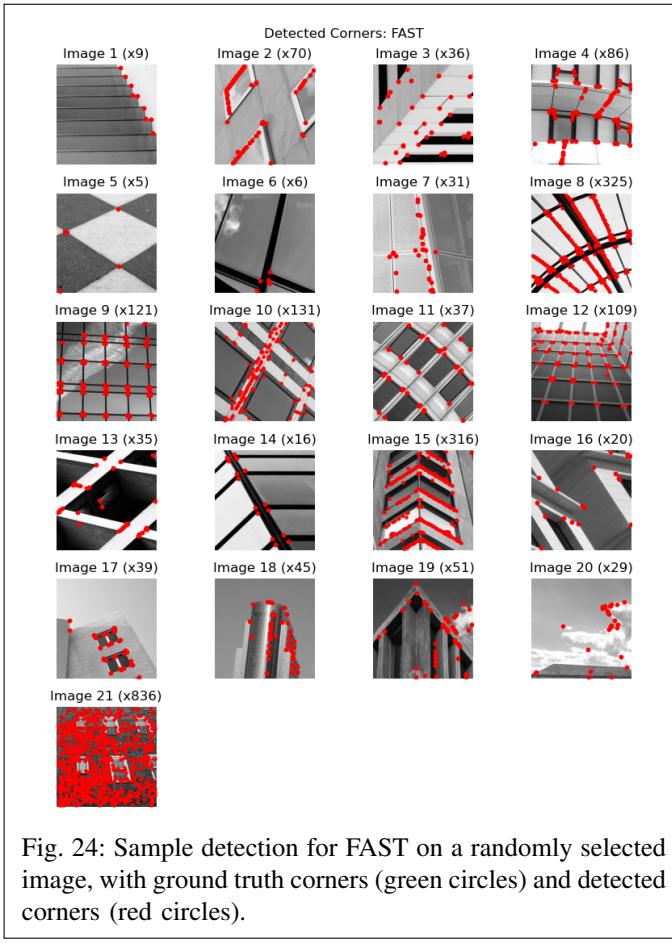


Fig. 24: Sample detection for FAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

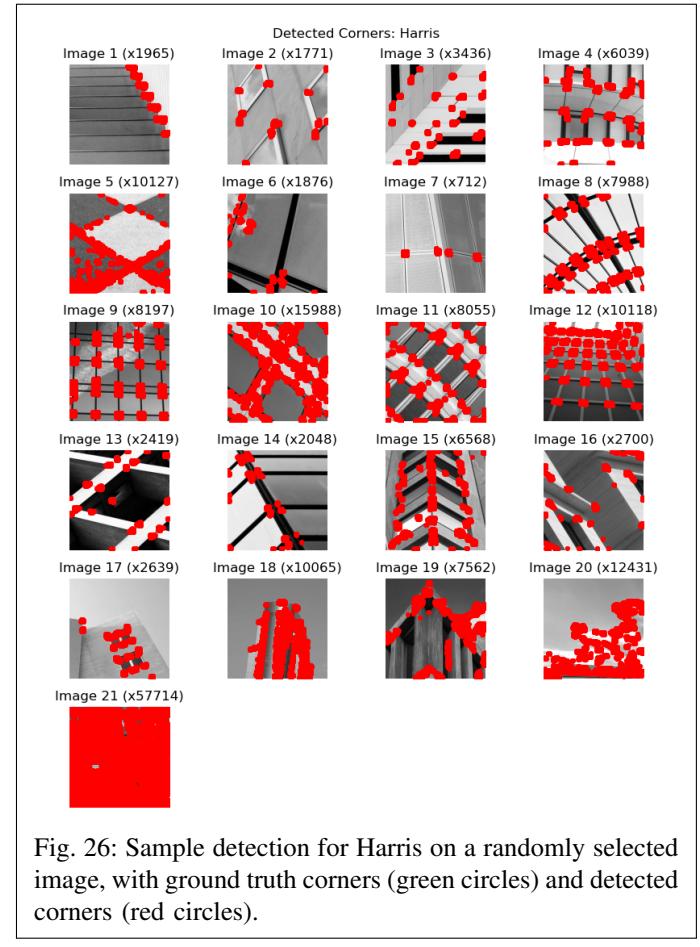


Fig. 26: Sample detection for Harris on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

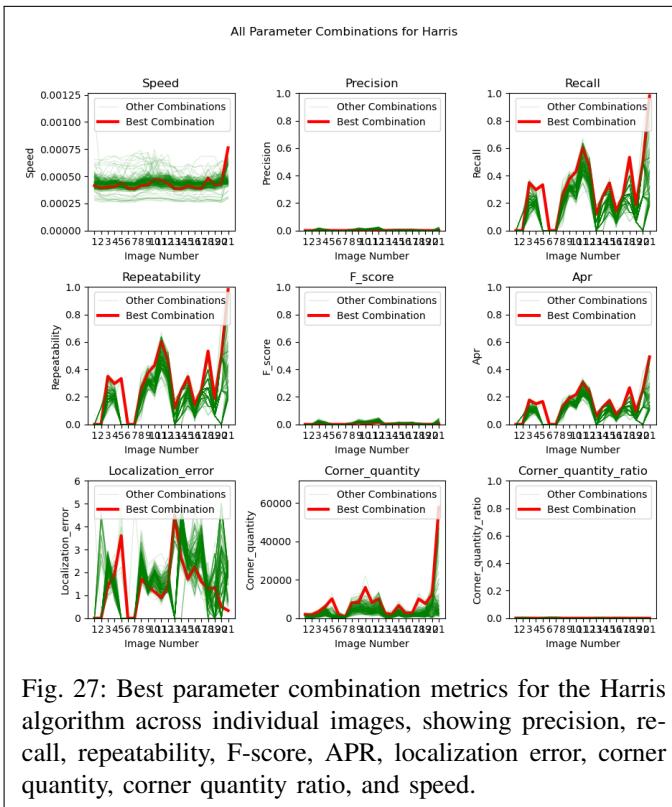


Fig. 27: Best parameter combination metrics for the Harris algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

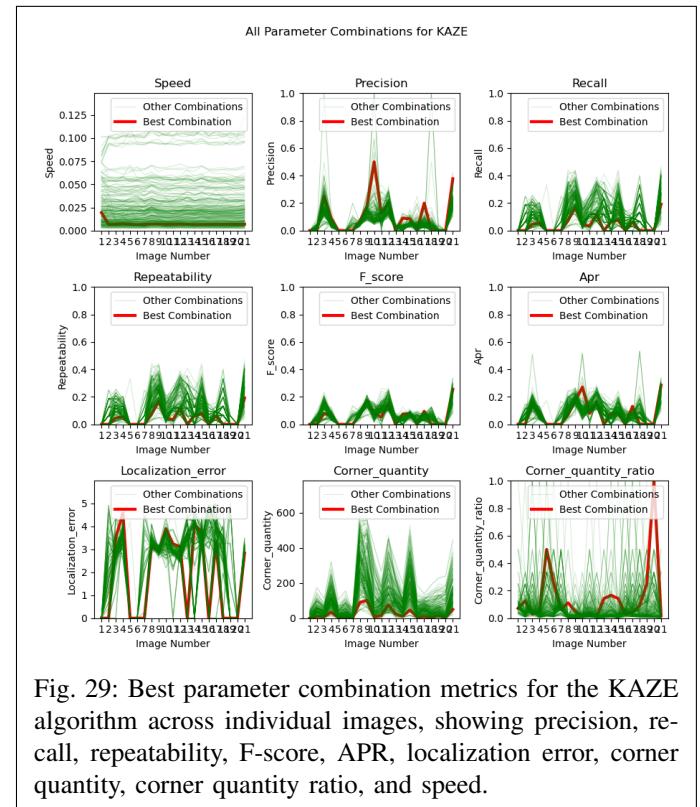


Fig. 29: Best parameter combination metrics for the KAZE algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

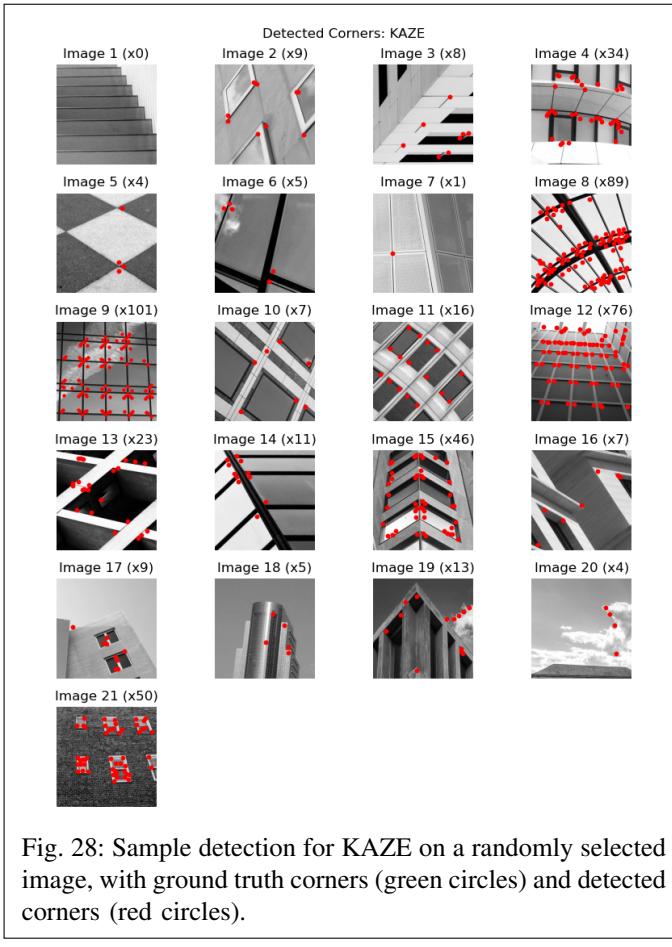


Fig. 28: Sample detection for KAZE on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

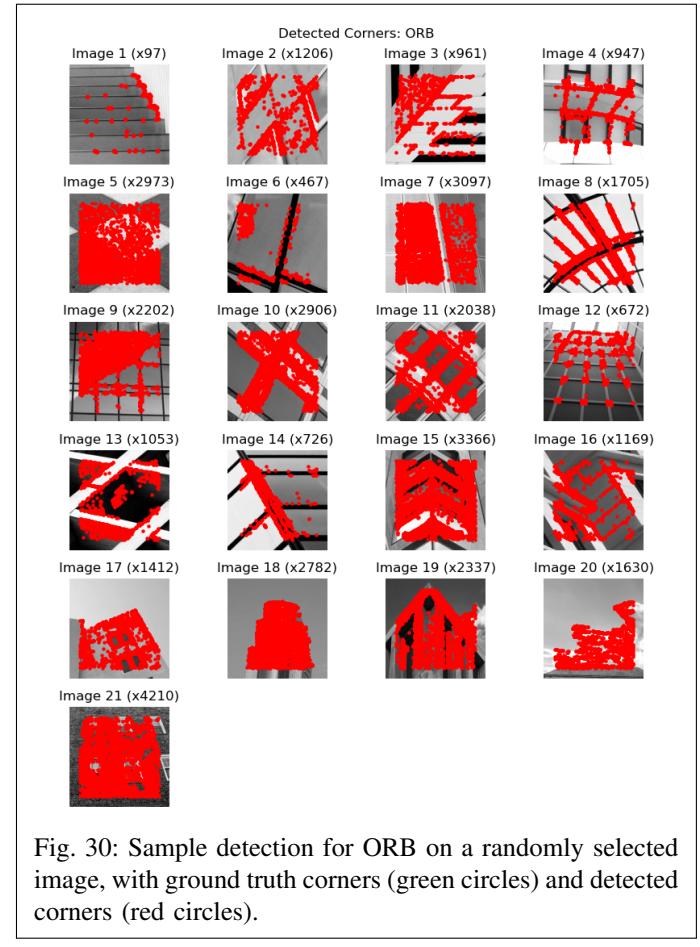


Fig. 30: Sample detection for ORB on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

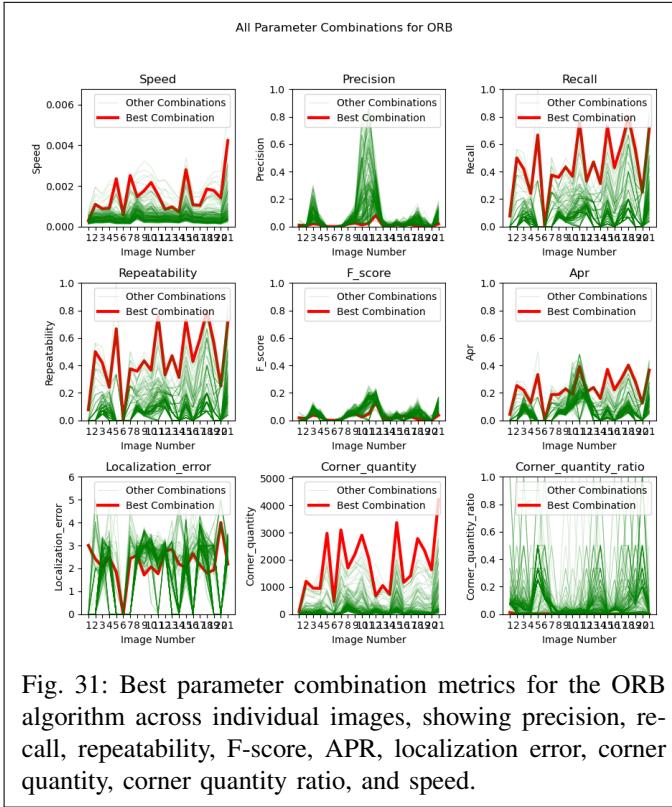


Fig. 31: Best parameter combination metrics for the ORB algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

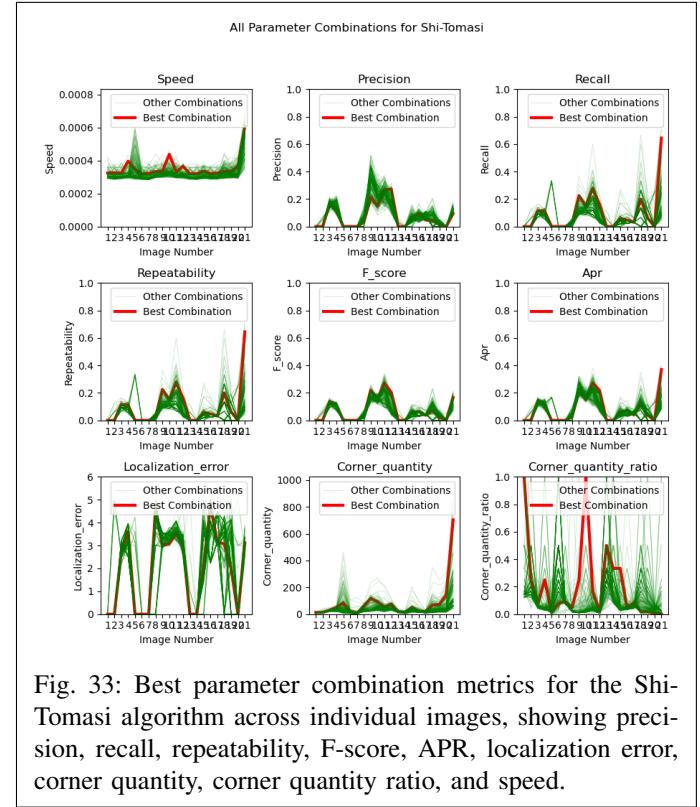


Fig. 33: Best parameter combination metrics for the Shi-Tomasi algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

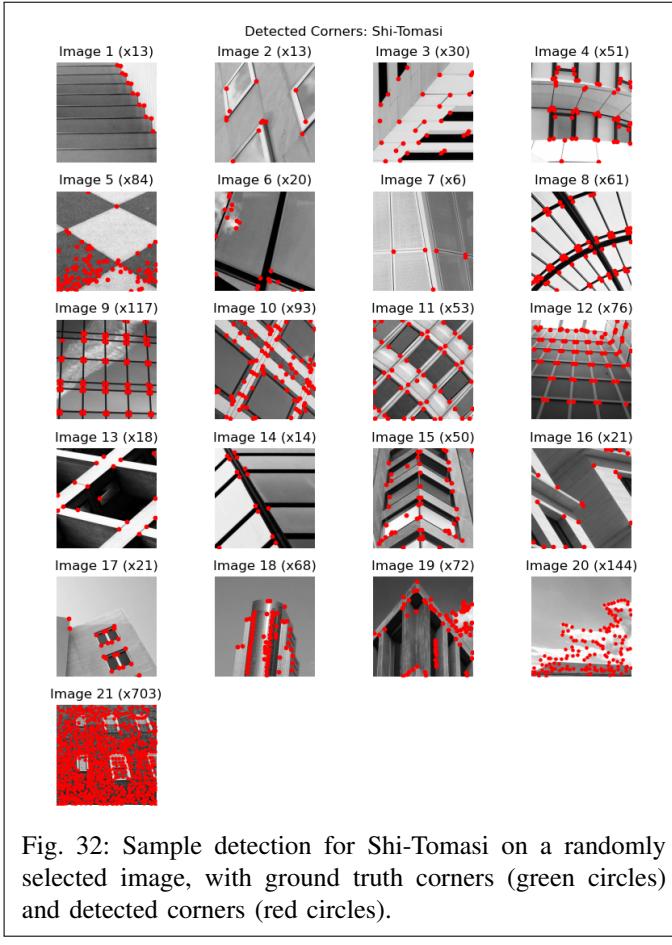


Fig. 32: Sample detection for Shi-Tomasi on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

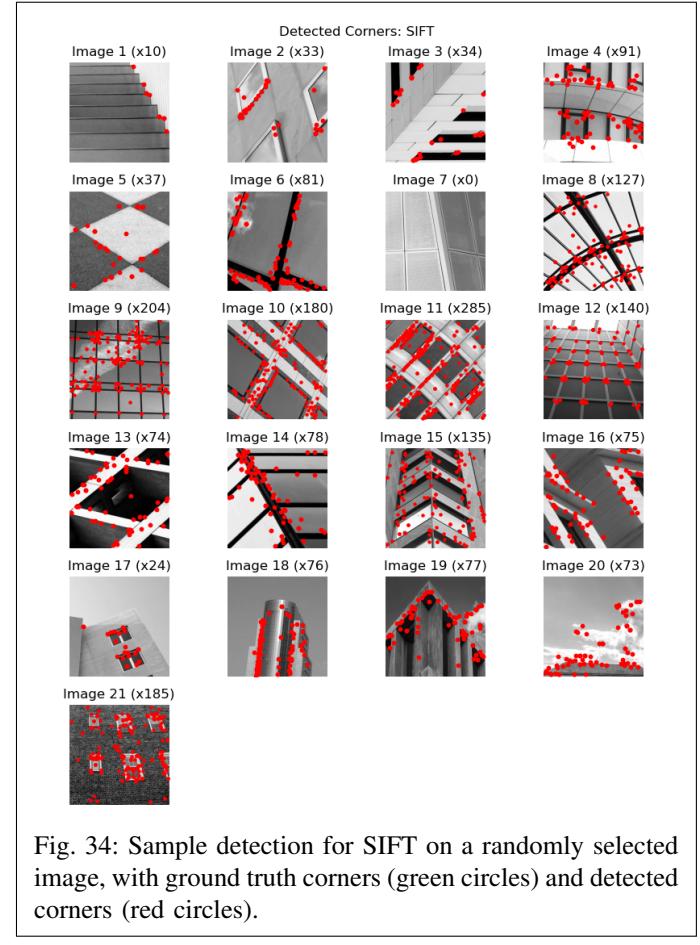


Fig. 34: Sample detection for SIFT on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

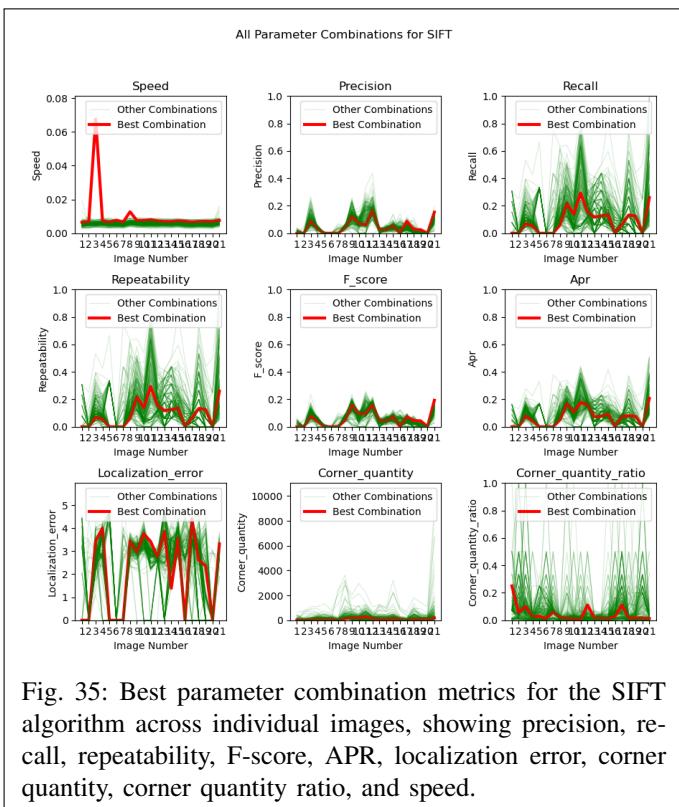


Fig. 35: Best parameter combination metrics for the SIFT algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

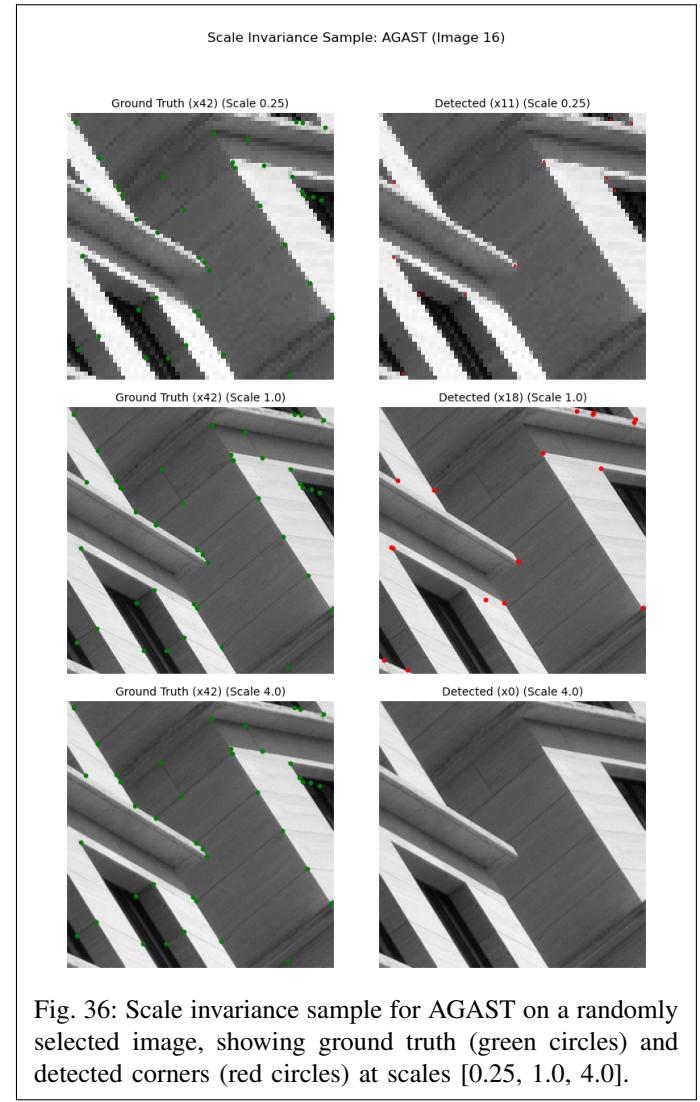
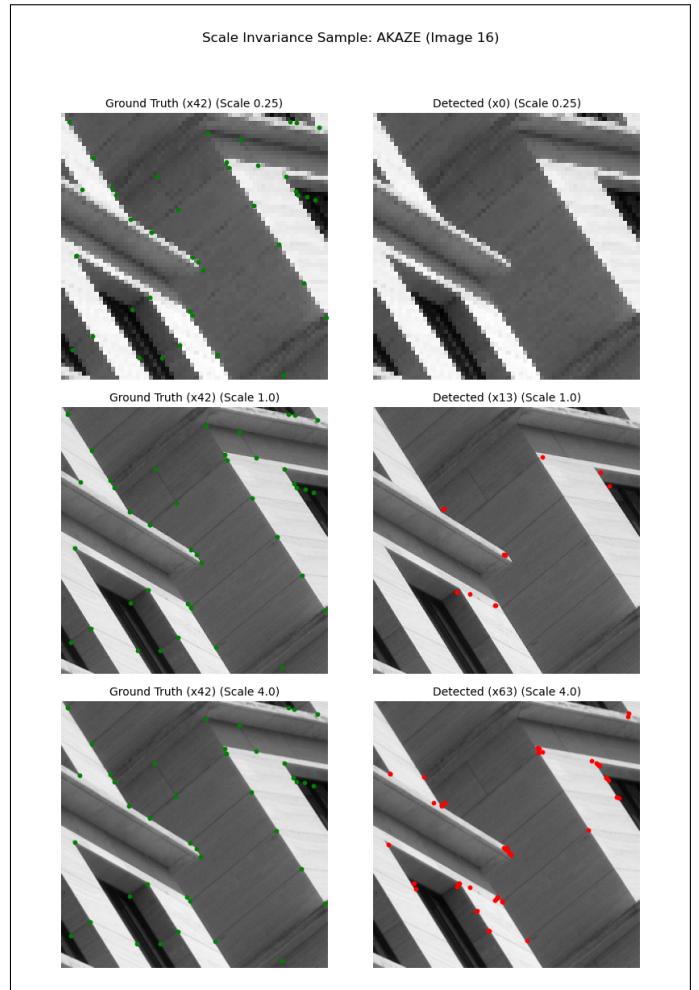
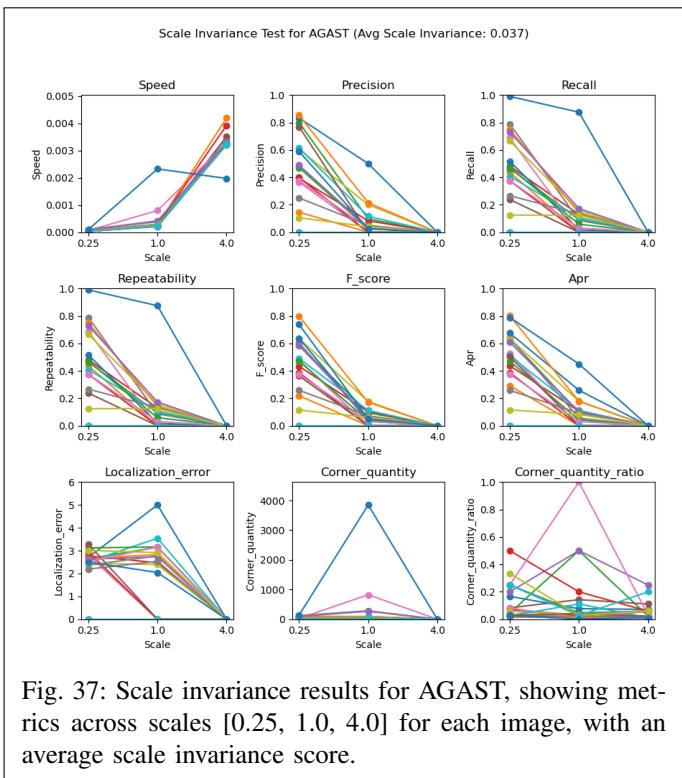
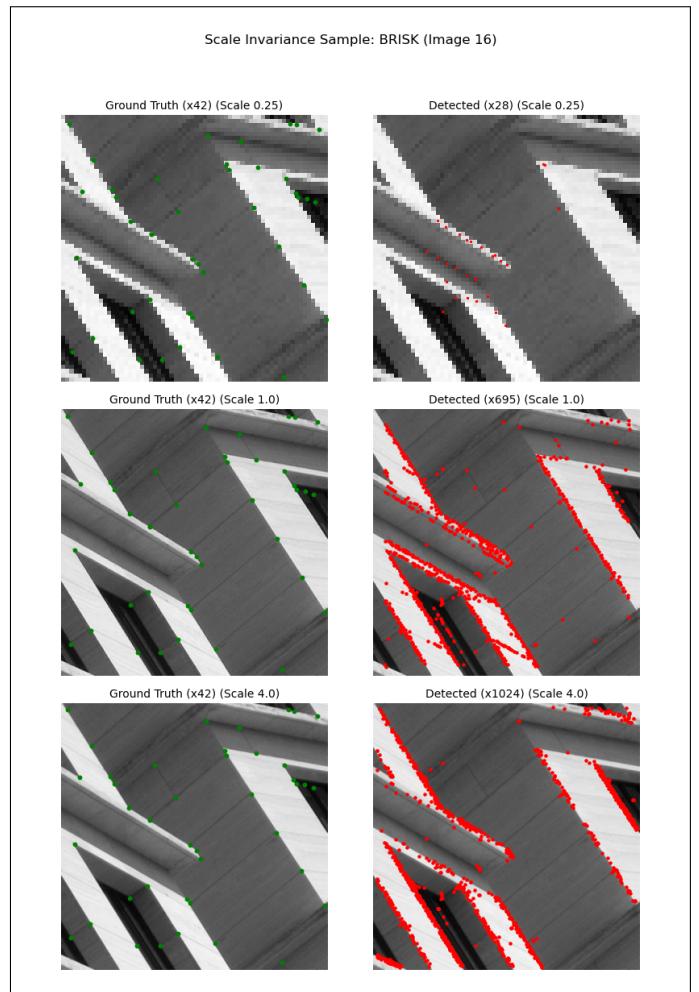
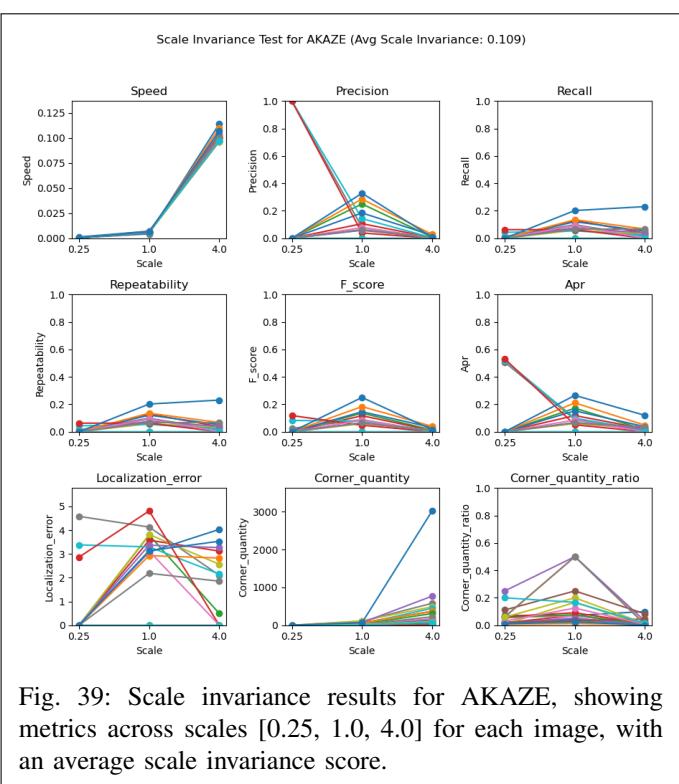


Fig. 36: Scale invariance sample for AGAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].





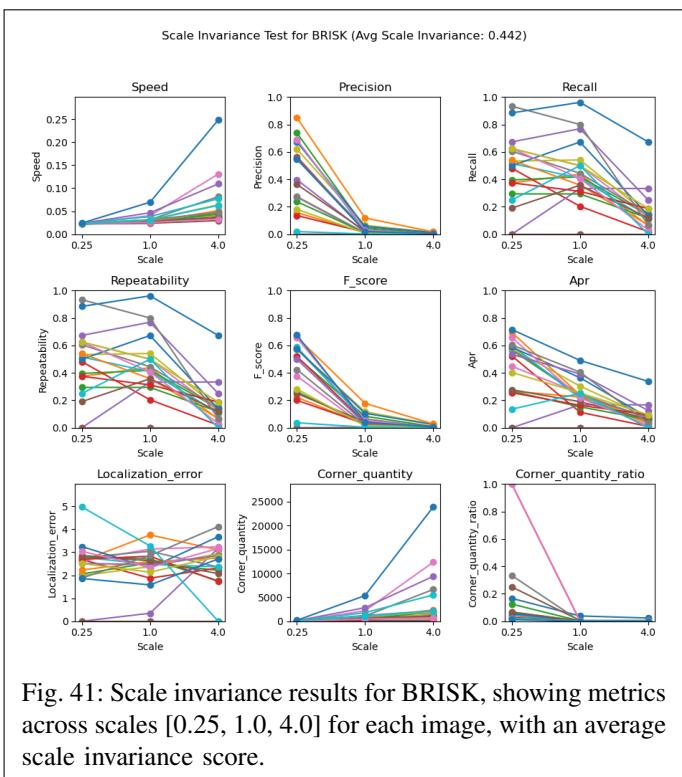


Fig. 41: Scale invariance results for BRISK, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

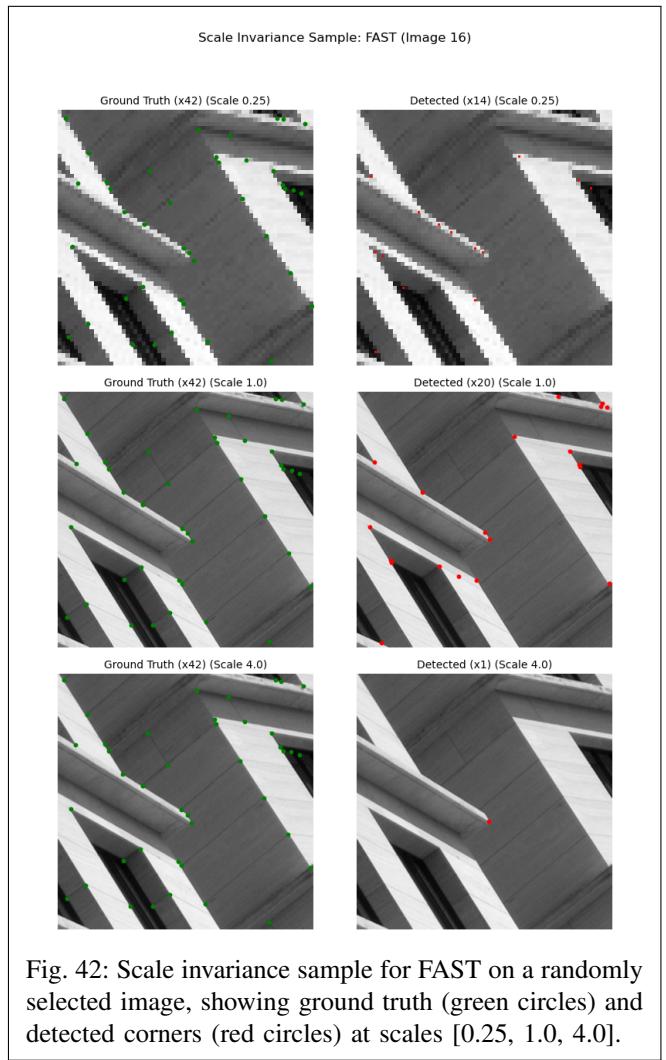
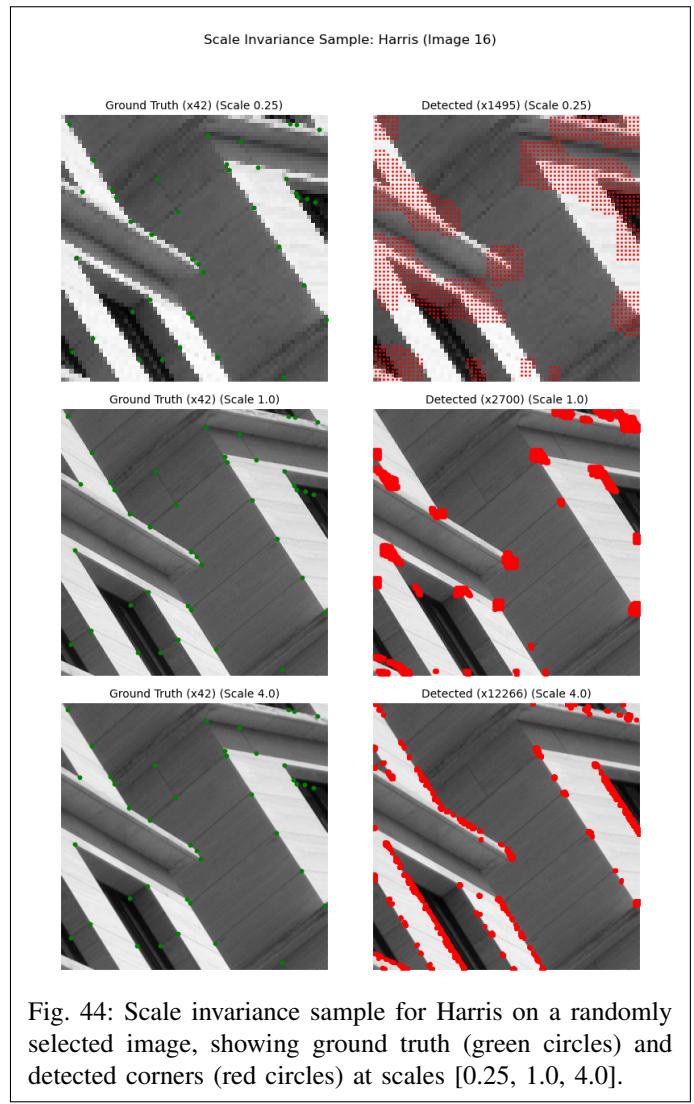
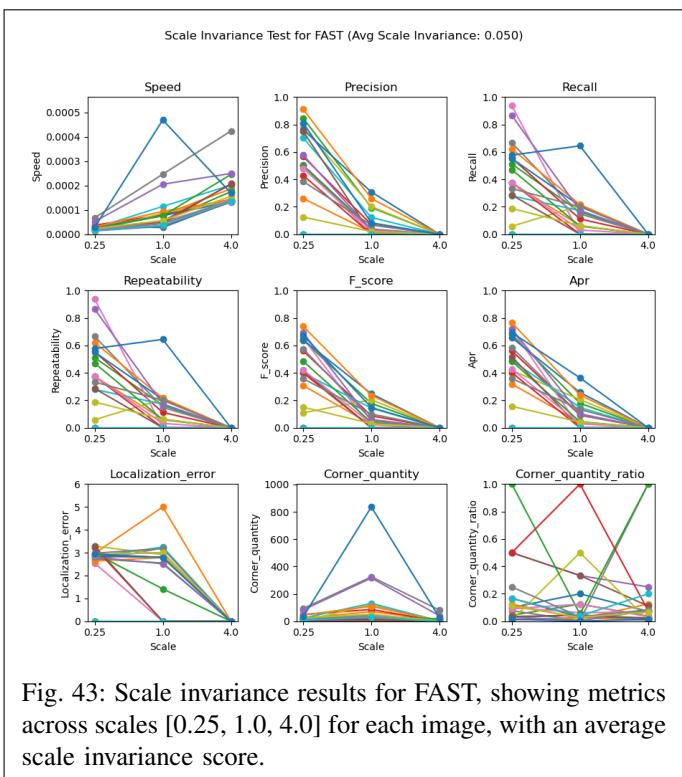
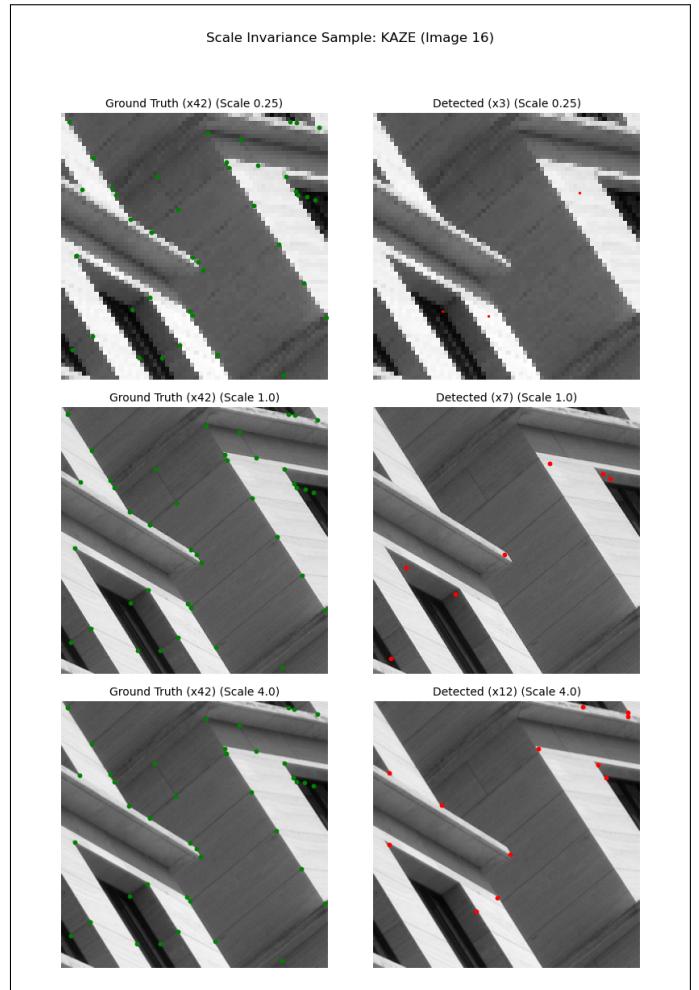
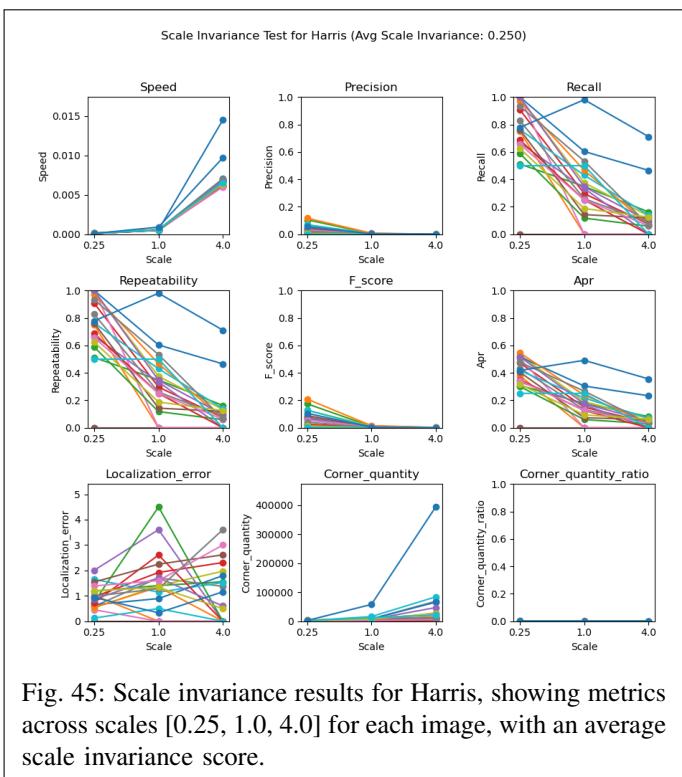
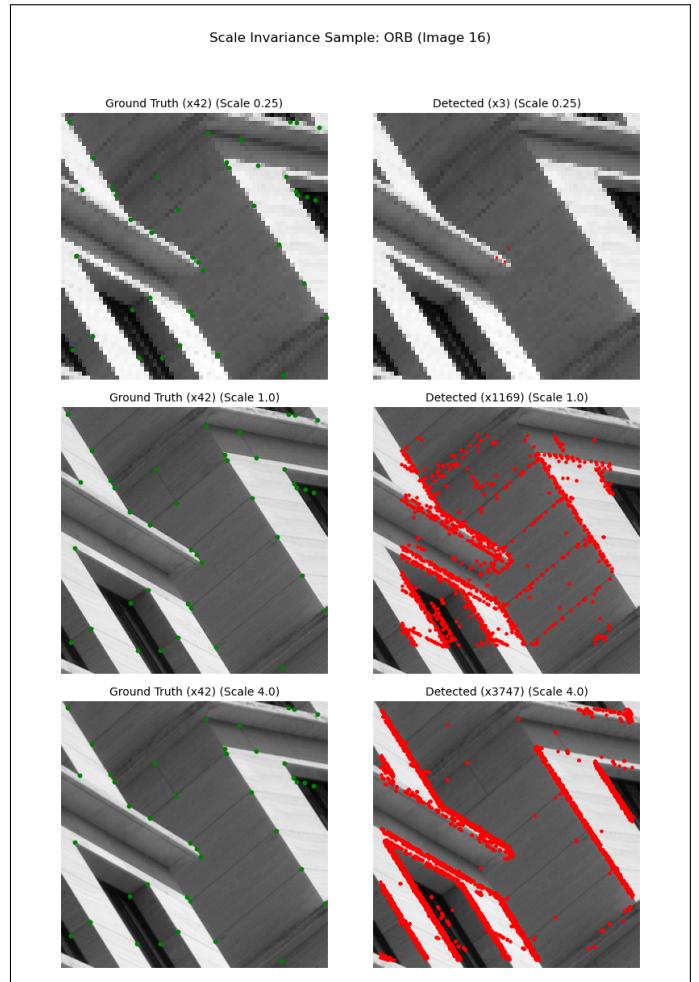
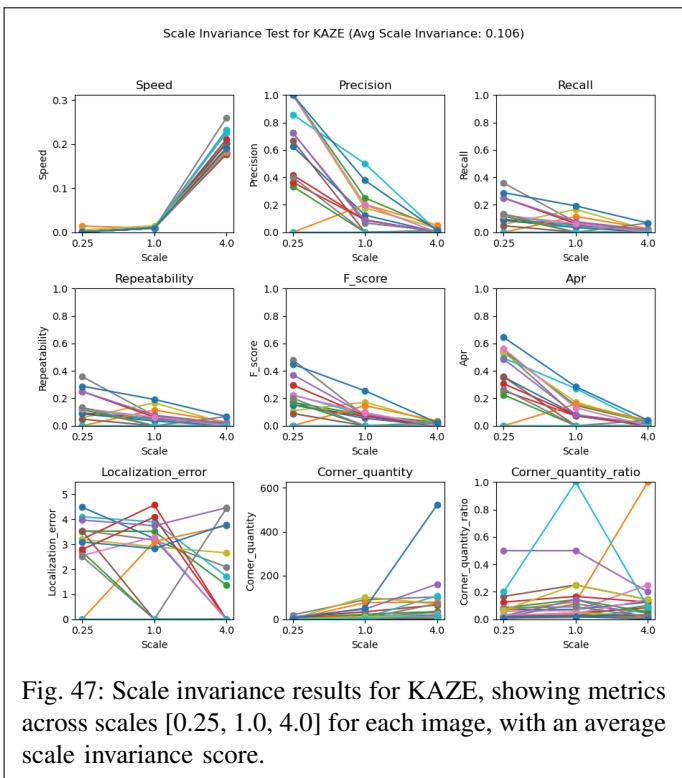


Fig. 42: Scale invariance sample for FAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].







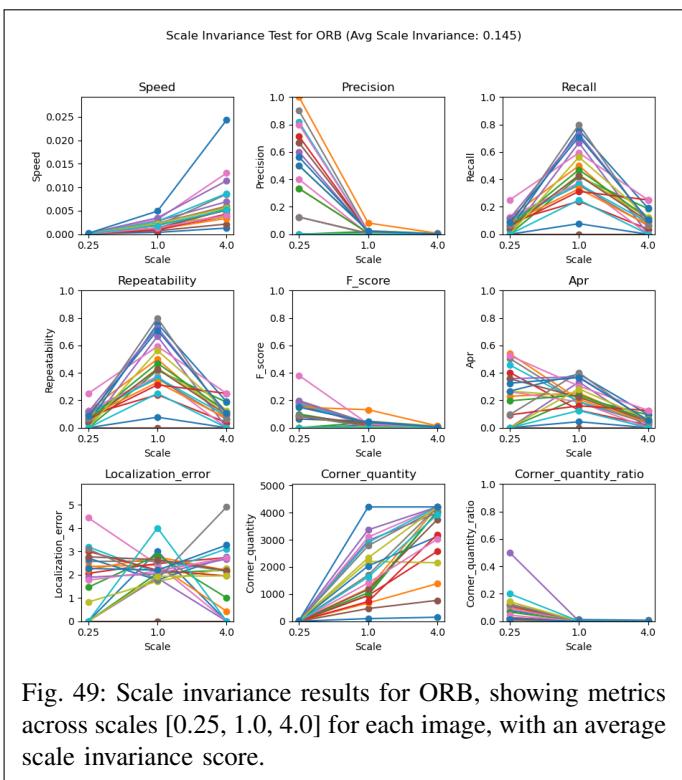


Fig. 49: Scale invariance results for ORB, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

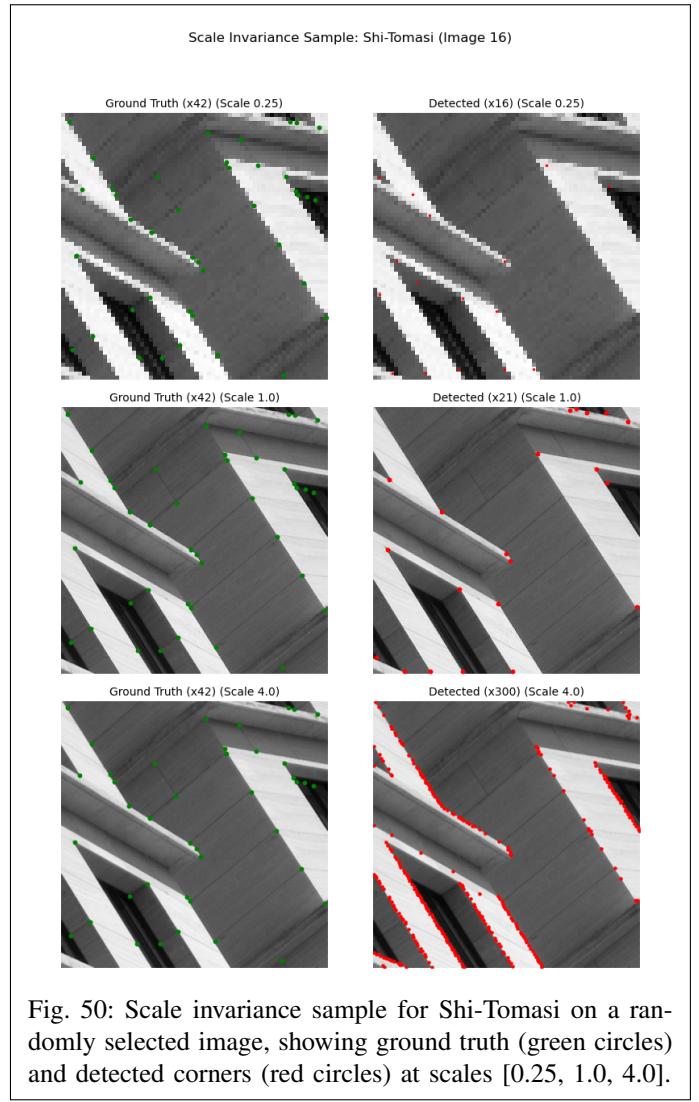


Fig. 50: Scale invariance sample for Shi-Tomasi on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

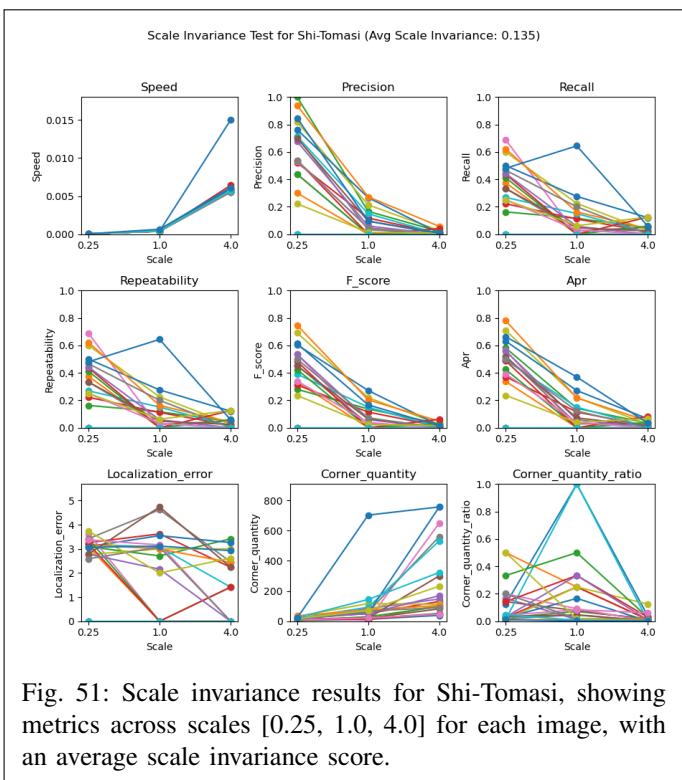


Fig. 51: Scale invariance results for Shi-Tomasi, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

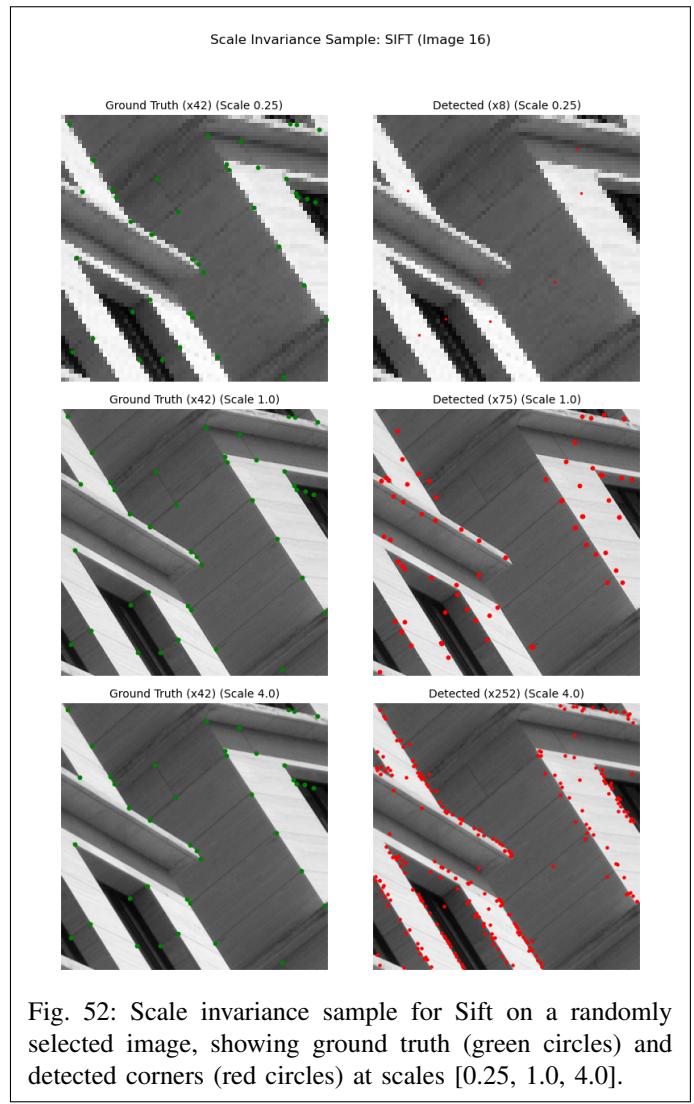


Fig. 52: Scale invariance sample for Sift on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

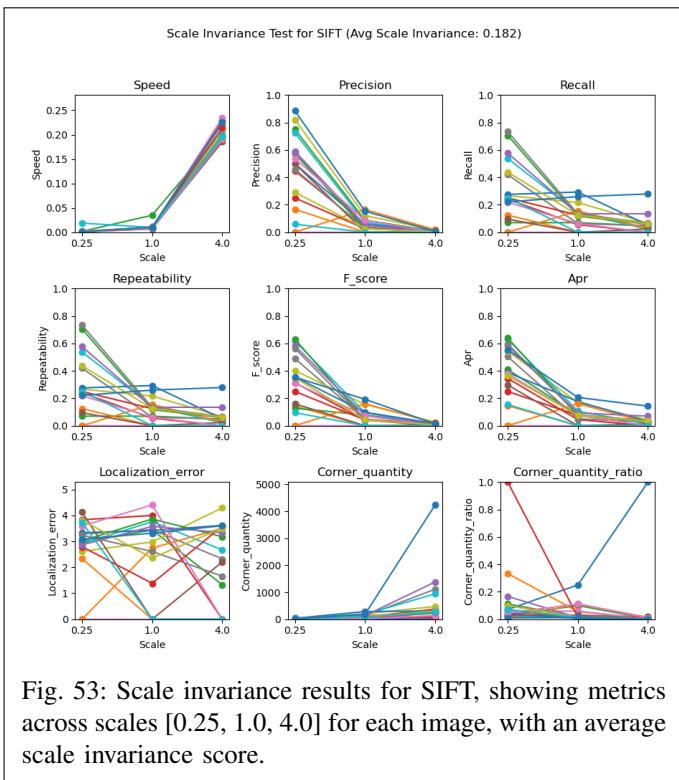


Fig. 53: Scale invariance results for SIFT, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

APPENDIX C

ALL DEVELOPED PYTHON CODE

```

1 ## import standard libraries
2 import numpy as np
3 import cv2
4 import os
5 import time
6 import itertools
7 from tqdm import tqdm
8 import re
9 import random
10
11 ## import custom scripts
12 import corner_methods as cm
13 from parameters import ALGORITHMS, PARAM_GRIDS, image_path, ground_truth_path, output_path, SCALES,
14     N_SAMPLES, MAX_SAMPLES, SEED
15 import utilities as utils
16 from distributions import sample_param_grid
17
18 def get_ground_truth(image_path, ground_truth_path, output_path, images=None, image_names=None,
19     ground_truth_corners=None):
20
21     images = images or []
22     image_names = image_names or []
23     ground_truth_corners = ground_truth_corners or []
24
25     sorted_files = sorted(os.listdir(image_path), key=lambda x: int(re.search(r'\d+', x).group()))
26
27     ## read in images and truth points
28     for image in sorted_files:
29         corners = f"{image.split('.')[0]}.txt"
30         image_names.append(image.split('.')[0])
31         images.append(cv2.imread(os.path.join(image_path, image), cv2.IMREAD_GRAYSCALE))
32
33         with open(os.path.join(ground_truth_path, corners), 'r') as file:
34             coordinate_pairs = np.array([tuple(map(int, line.split())) for line in file])
35             ground_truth_corners.append(coordinate_pairs)
36
37     if not os.path.isdir(output_path):
38         os.makedirs(output_path)
39     utils.plot_ground_truth(images, image_names, ground_truth_corners, output_path)
40
41     return images, image_names, ground_truth_corners
42
43 def optimize_across_images(algorithm, alg_name, images, gt_corners_list, param_grid):
44
45     best_params = {}
46     best_score = -1
47
48     param_names, param_values = sample_param_grid(param_grid, N_SAMPLES)
49     total_combinations = np.prod([len(values) for values in param_values])
50
51     # Limit to a maximum number of combinations
52     param_combinations = list(itertools.product(*param_values))
53     if MAX_SAMPLES:
54         if len(param_combinations) > MAX_SAMPLES:
55             param_combinations = random.sample(param_combinations, MAX_SAMPLES)
56             total_combinations = MAX_SAMPLES
57
58     # Store metrics for each parameter combination
59     all_metrics = []
60
61     # Use tqdm to show progress with total combinations
62     for combo in tqdm(param_combinations, total=total_combinations, desc=f"Optimizing_{alg_name}"):
63         total_combinationsRuns: {alg_name}:
64         params = dict(zip(param_names, combo))
65         speeds = []
66         precisions = []
67         recalls = []
68         repeatabilities = []
69         f_scores = []
70         aprs = []
71         localization_errors = []
72         corner_quantities = []

```

```

71 corner_quantity_ratios = []
72
73 for image, gt_corners in zip(images, gt_corners_list):
74     start_time = time.time()
75     corners = algorithm(image, args=params)
76     exec_time = time.time() - start_time
77     precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
78         corner_quantity_ratio = utils.calculate_metrics(corners, gt_corners)
79
80     speeds.append(exec_time)
81     precisions.append(precision)
82     recalls.append(recall)
83     repeatabilities.append(repeatability)
84     f_scores.append(f_score)
85     aprs.append(apr)
86     localization_errors.append(localization_error)
87     corner_quantities.append(corner_quantity)
88     corner_quantity_ratios.append(corner_quantity_ratio)
89
90 # Normalize speed to [0, 1] for scoring (lower speed is better, so invert it)
91 speed_norm = np.max(speeds) if speeds else 1.0
92 if speed_norm > np.mean(speeds) + 4 * np.std(speeds):
93     speed_norm = np.mean(speeds) + 4 * np.std(speeds)
94 normalized_speeds = [1 - (speed / speed_norm) if speed_norm > 0 else 1.0 for speed in speeds]
95
96 # Normalize localization error to [0, 1] for scoring (lower is better, so invert it)
97 max_le = np.max(localization_errors) if localization_errors and np.max(localization_errors) > 0
98     else 1.0
99 normalized_le = [1 - (le / max_le) if max_le > 0 else 1.0 for le in localization_errors]
100
101 # Compute average metrics
102 avg_speed = np.mean(normalized_speeds)
103 avg_precision = np.mean(precisions)
104 avg_recall = np.mean(recalls)
105 avg_repeatability = np.mean(repeatabilities)
106 avg_f_score = np.mean(f_scores)
107 avg_apr = np.mean(aprs)
108 avg_le = np.mean(normalized_le)
109 avg_corner_quant = np.mean(corner_quantity)
110 avg_corner_quant_ratios = np.mean(corner_quantity_ratios)
111
112 # Score is the average of the normalized metrics (excluding corner_quantity since it's not part
113     of optimization)
114 # weights = np.array([2/12, 3/12, 3/12, 1/12, 2/12, 1/12])
115 vars = np.array([avg_speed, avg_precision, avg_recall, avg_le, avg_corner_quant_ratios]) #
116     avg_repeatability
117 # score = np.sum(vars * weights)
118 score = np.sum(vars) / len(vars)
119
120 # Store metrics for this combination
121 all_metrics.append({
122     'params': params,
123     'speed': speeds,
124     'precision': precisions,
125     'recall': recalls,
126     'repeatability': repeatabilities,
127     'f_score': f_scores,
128     'apr': aprs,
129     'localization_error': localization_errors,
130     'corner_quantity': corner_quantities,
131     'corner_quantity_ratio': corner_quantity_ratios,
132     'score': score
133 })
134
135 if score > best_score:
136     best_score = score
137     best_params = params
138
139 return best_params, best_score, total_combinations, all_metrics
140
141 def test_scale_invariance(images, image_names, ground_truth_corners, optimized_params):
142
143     scale_results = {name: {img_name: {
144         'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
145         'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': [], 'scale_invariance':
146             0.0
147     } for img_name in image_names} for name in ALGORITHMS}

```

```

143
144     # Select a random image for sample visualization
145     sample_idx = np.random.randint(0, len(images))
146     sample_image = images[sample_idx]
147     sample_name = image_names[sample_idx]
148     sample_gt_corners = ground_truth_corners[sample_idx]
149
150     for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Testing_Scale_Invariance"):
151         best_params = optimized_params[alg_name]
152         scaled_images = utils.create_scaled_images(sample_image)
153         sample_corners = []
154
155         for img_idx, (image, gt_corners, name) in enumerate(zip(images, ground_truth_corners,
156             image_names)):
157             scaled_images_current = utils.create_scaled_images(image)
158             repeatabilities = [] # To compute scale invariance
159
160             for scale_idx, scaled_img in enumerate(scaled_images_current):
161                 start_time = time.time()
162                 corners = alg_func(scaled_img, args=best_params)
163                 exec_time = time.time() - start_time
164
165                 # Adjust ground truth for scale
166                 scaled_gt = gt_corners * SCALES[scale_idx]
167
168                 precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
169                 corner_quantity_ratio = utils.calculate_metrics(corners, scaled_gt)
170
171                 # Store results for this scale
172                 scale_results[alg_name][name]['speed'].append(exec_time)
173                 scale_results[alg_name][name]['precision'].append(precision)
174                 scale_results[alg_name][name]['recall'].append(recall)
175                 scale_results[alg_name][name]['repeatability'].append(repeatability)
176                 scale_results[alg_name][name]['f_score'].append(f_score)
177                 scale_results[alg_name][name]['apr'].append(apr)
178                 scale_results[alg_name][name]['localization_error'].append(localization_error)
179                 scale_results[alg_name][name]['corner_quantity'].append(corner_quantity)
180                 scale_results[alg_name][name]['corner_quantity_ratio'].append(corner_quantity_ratio)
181
182                 repeatabilities.append(repeatability)
183
184                 # Store corners for the sample image
185                 if img_idx == sample_idx:
186                     sample_corners.append(corners)
187
188                 # Compute scale invariance as 1 - coefficient of variation of repeatability
189                 repeatabilities = np.array(repeatabilities)
190                 if repeatabilities.mean() > 0:
191                     cv = repeatabilities.std() / repeatabilities.mean()
192                     scale_invariance = max(0, min(1, 1 - cv)) # Higher value = better scale invariance
193                 else:
194                     scale_invariance = 0.0
195
196                 scale_results[alg_name][name]['scale_invariance'] = scale_invariance
197
198                 # Generate sample visualization for this algorithm
199                 utils.generate_scale_invariance_samples(
200                     scaled_images, sample_corners, sample_gt_corners,
201                     alg_name, sample_name, output_path
202                 )
203
204             return scale_results
205
206     def run_benchmarking(images, image_names, ground_truth_corners):
207
208         results = {name: {
209             'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
210             'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': []
211         } for name in ALGORITHMS}
212
213         individual_results = {name: {img_name: {
214             'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
215             'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': []
216         } for img_name in image_names} for name in ALGORITHMS}
217
218         optimized_params = {}
219         all_metrics_per_algorithm = {} # Store all metrics for each parameter combination per algorithm
220
221         # Optimize parameters and benchmark for each algorithm

```

```

218     params = []
219     combinations = []
220     scores = []
221
222     for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Generating_Optimization_Data"):
223         # Optimize parameters across all images and store all metrics
224         best_params, best_score, total_combinations, all_metrics = optimize_across_images(alg_func,
225             alg_name, images, ground_truth_corners, PARAM_GRIDS[alg_name])
226         params.append(best_params)
227         combinations.append(total_combinations)
228         scores.append(best_score)
229         optimized_params[alg_name] = best_params
230         all_metrics_per_algorithm[alg_name] = all_metrics
231
232         # Benchmark with optimized parameters for this algorithm
233         for img_idx, (image, gt_corners, name) in enumerate(zip(images, ground_truth_corners,
234             image_names)):
235             start_time = time.time()
236             corners = alg_func(image, args=best_params)
237             exec_time = time.time() - start_time
238
239             precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
240             corner_quantity_ratio = utils.calculate_metrics(corners, gt_corners)
241
242             # Store aggregated results
243             results[alg_name]['speed'].append(exec_time)
244             results[alg_name]['precision'].append(precision)
245             results[alg_name]['recall'].append(recall)
246             results[alg_name]['repeatability'].append(repeatability)
247             results[alg_name]['f_score'].append(f_score)
248             results[alg_name]['apr'].append(apr)
249             results[alg_name]['localization_error'].append(localization_error)
250             results[alg_name]['corner_quantity'].append(corner_quantity)
251             results[alg_name]['corner_quantity_ratio'].append(corner_quantity_ratio)
252
253             # Store individual results
254             individual_results[alg_name][name]['speed'].append(exec_time)
255             individual_results[alg_name][name]['precision'].append(precision)
256             individual_results[alg_name][name]['recall'].append(recall)
257             individual_results[alg_name][name]['repeatability'].append(repeatability)
258             individual_results[alg_name][name]['f_score'].append(f_score)
259             individual_results[alg_name][name]['apr'].append(apr)
260             individual_results[alg_name][name]['localization_error'].append(localization_error)
261             individual_results[alg_name][name]['corner_quantity'].append(corner_quantity)
262             individual_results[alg_name][name]['corner_quantity_ratio'].append(corner_quantity_ratio)
263
264     return results, optimized_params, individual_results, params, combinations, scores,
265         all_metrics_per_algorithm
266
267 if __name__ == "__main__":
268
269     np.random.seed(SEED)
270
271     print("Loading_Ground_Truth...")
272     images, image_names, ground_truth_corners = get_ground_truth(image_path, ground_truth_path,
273         output_path)
274     results, optimized_params, individual_results, params, combinations, scores,
275     all_metrics_per_algorithm = run_benchmarking(images, image_names, ground_truth_corners)
276
277     # Generate sample detection images with optimized parameters
278     print("Creating_Sample_Imagery_for_Optimized_Algorithms...")
279     utils.generate_sample_detections(images, image_names, ground_truth_corners, optimized_params,
280         output_path)
281
282     # Save results
283     print("Writing_Results_Files...")
284     os.makedirs(output_path, exist_ok=True)
285     os.makedirs(os.path.join(output_path, "data"), exist_ok=True)
286     for alg_name in results:
287         np.savez(os.path.join(output_path, f'data/{alg_name}_results.npz'), **results[alg_name])
288         for img_name in image_names:
289             np.savez(os.path.join(output_path, f'data/{alg_name}_{img_name}_individual_results.npz'),
290                 **individual_results[alg_name][img_name])
291
292     # Generate comparison tables and reports
293     utils.generate_param_report(params, combinations, scores, output_path)
294     utils.generate_comparison_tables(results, output_path)

```

```

287 # Generate individual result plots and pairwise metric plots
288 print("Plotting Result Images...")
289 utils.visualize_results(output_path, image_names)
290 # utils.plot_pairwise_metrics(results, output_path)
291 utils.plot_best_combination(individual_results, image_names, output_path)
292 utils.plot_all_combinations(all_metrics_per_algorithm, image_names, output_path)
293
294 # Run scale invariance test
295 scale_results = test_scale_invariance(images, image_names, ground_truth_corners, optimized_params)
296 utils.plot_scale_invariance(scale_results, image_names, output_path)
297 utils.generate_scale_invariance_table(scale_results, output_path)
298
299 utils.plot_all_detections(images, image_names, optimized_params, output_path)

```

Listing 5: Driver Script for Project

```

1 from corner_methods import *
2 import os
3 from distributions import UniformDist, NormalDist, CategoricalDist
4
5 image_path = os.path.join(os.getcwd(), "data/Urban_Corner_datasets/Images")
6 ground_truth_path = os.path.join(os.getcwd(), "data/Urban_Corner_datasets/Ground_Truth")
7 output_path = os.path.join(os.getcwd(), "results")
8
9 SCALES = [0.25, 1.0, 4.0] # Scales to test
10
11 N_SAMPLES = 20
12 MAX_SAMPLES = 250
13 SEED = 11003
14
15 ALGORITHMS = {
16     'Harris' : harris,
17     'Shi-Tomasi' : shi_tomasi,
18     'FAST' : fast,
19     'ORB' : orb,
20     'SIFT' : sift,
21     'BRISK' : brisk,
22     'AGAST' : agast,
23     'KAZE' : kaze,
24     'AKAZE' : akaze
25 }
26
27 ## Using custom distribution generation code
28 PARAM_GRIDS = {
29     'Harris': {
30         'blockSize': UniformDist(min_val=2, max_val=11, is_int=True),
31         'ksize': CategoricalDist(options=[3, 5, 7, 9, 11]), # Only odd integers
32         'k': UniformDist(min_val=0.01, max_val=0.15),
33         'borderType': CategoricalDist(options=[
34             cv2.BORDER_DEFAULT, cv2.BORDER_CONSTANT, cv2.BORDER_REFLECT
35         ])
36     },
37     # ... (rest of PARAM_GRIDS unchanged)
38     'Shi-Tomasi': {
39         'maxCorners': UniformDist(min_val=25, max_val=1000, is_int=True),
40         'qualityLevel': UniformDist(min_val=0.005, max_val=0.2),
41         'minDistance': UniformDist(min_val=3, max_val=30, is_int=True),
42         'blockSize': UniformDist(min_val=3, max_val=9, is_int=True)
43     },
44     'FAST': {
45         'threshold': UniformDist(min_val=5, max_val=100, is_int=True),
46         'type': CategoricalDist(options=[
47             cv2.FAST_FEATURE_DETECTOR_TYPE_5_8,
48             cv2.FAST_FEATURE_DETECTOR_TYPE_7_12,
49             cv2.FAST_FEATURE_DETECTOR_TYPE_9_16
50         ])
51     },
52     'ORB': {
53         'nfeatures': UniformDist(min_val=100, max_val=5000, is_int=True),
54         ' scaleFactor': UniformDist(min_val=1.05, max_val=1.5),
55         'nlevels': UniformDist(min_val=4, max_val=16, is_int=True),
56         'edgeThreshold': UniformDist(min_val=15, max_val=70, is_int=True),
57         'patchSize': UniformDist(min_val=15, max_val=50, is_int=True),

```

```

58     'fastThreshold': UniformDist(min_val=5, max_val=100, is_int=True)
59 },
60 'SIFT': {
61     'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
62     'contrastThreshold': UniformDist(min_val=0.02, max_val=0.16),
63     'edgeThreshold': UniformDist(min_val=5, max_val=30, is_int=True),
64     'sigma': NormalDist(mean=1.6, std=0.4, min_val=1.0, max_val=2.4)
65 },
66 'BRISK': {
67     'thresh': UniformDist(min_val=10, max_val=100, is_int=True),
68     'octaves': UniformDist(min_val=2, max_val=6, is_int=True),
69     'patternScale': UniformDist(min_val=0.5, max_val=2.0)
70 },
71 'AGAST': {
72     'threshold': UniformDist(min_val=5, max_val=50, is_int=True),
73     'type': CategoricalDist(options=[
74         cv2.AgastFeatureDetector_AGAST_5_8,
75         cv2.AgastFeatureDetector_AGAST_7_12d,
76         cv2.AgastFeatureDetector_OAST_9_16
77     ])
78 },
79 'KAZE': {
80     'threshold': UniformDist(min_val=0.0005, max_val=0.004),
81     'nOctaves': UniformDist(min_val=2, max_val=6, is_int=True),
82     'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
83     'diffusivity': CategoricalDist(options=[
84         cv2.KAZE_DIFF_PM_G1, cv2.KAZE_DIFF_PM_G2, cv2.KAZE_DIFF_WEICKERT
85     ])
86 },
87 'AKAZE': {
88     'threshold': UniformDist(min_val=0.0005, max_val=0.004),
89     'nOctaves': UniformDist(min_val=3, max_val=6, is_int=True),
90     'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
91     'diffusivity': CategoricalDist(options=[
92         cv2.KAZE_DIFF_PM_G1, cv2.KAZE_DIFF_PM_G2
93     ])
94 }
95 }
96
97 PARAM_REPORT = [
98     "\\begin{table}[h]",
99     "\\centering",
100     "\\small",
101     "\\caption{Optimal Parameters for Corner Detection Algorithms}",
102     "\\label{tab:optimal_parameters}",
103     "\\begin{tabular}{lp{3.5cm}c}",
104     "\\toprule",
105     "\\textbf{Algorithm} & \\textbf{Optimal Parameters} & \\textbf{$\\#$_Tests} & \\textbf{Best Score} \\\\"\\midrule"
106 ]

```

Listing 6: Setup Parameters

```

1 import numpy as np
2
3 class UniformDist:
4     def __init__(self, min_val, max_val, is_int=False):
5         self.min_val = min_val
6         self.max_val = max_val
7         self.is_int = is_int
8
9     def sample(self, n_samples):
10        samples = np.round(np.random.uniform(self.min_val, self.max_val, n_samples), 4)
11        if self.is_int:
12            samples = np.round(samples).astype(int)
13            samples = np.clip(samples, self.min_val, self.max_val)
14
15        return np.unique(samples)[:n_samples] # Ensure unique samples
16
17 class NormalDist:
18     def __init__(self, mean, std, min_val=None, max_val=None, is_int=False):
19         self.mean = mean
20         self.std = std

```

```

21     self.min_val = min_val
22     self.max_val = max_val
23     self.is_int = is_int
24
25     def sample(self, n_samples):
26         samples = np.random.normal(self.mean, self.std, n_samples)
27         if self.min_val is not None or self.max_val is not None:
28             samples = np.round(np.clip(samples, self.min_val or -np.inf, self.max_val or np.inf), 4)
29         if self.is_int:
30             samples = np.round(samples).astype(int)
31             samples = np.clip(samples, self.min_val or -np.inf, self.max_val or np.inf)
32         return np.unique(samples)[:n_samples]
33
34 class CategoricalDist:
35     def __init__(self, options):
36         self.options = options
37
38     def sample(self, n_samples):
39         if n_samples <= len(self.options):
40             return np.random.choice(self.options, n_samples, replace=False)
41         else:
42             return np.random.choice(self.options, n_samples, replace=True)
43
44
45     def sample_param_grid(param_grid, n_samples):
46
47         sampled_grid = {}
48         for param_name, dist in param_grid.items():
49             sampled_grid[param_name] = dist.sample(n_samples)
50         return list(sampled_grid.keys()), list(sampled_grid.values())

```

Listing 7: Random Variable Distribution Setup

```

1 import cv2
2 import numpy as np
3
4 def harris(image, args=None):
5
6     args = args or {'blockSize': 2, 'ksize': 3, 'k': 0.04}
7     corners = cv2.cornerHarris(image, **args)
8     corners = cv2.dilate(corners, None)
9     corners = np.column_stack(np.where(corners > 0.01 * corners.max()))
10    return corners[:, ::-1].astype(np.float32)
11
12 def shi_tomasi(image, args=None):
13
14     args = args or {'maxCorners': 25, 'qualityLevel': 0.01, 'minDistance': 10}
15     corners = cv2.goodFeaturesToTrack(image, **args)
16     return corners[:, 0].astype(np.float32) if corners is not None else np.array([])
17
18 def fast(image, args=None):
19
20     args = args or {'threshold': 25, 'type': cv2.FAST_FEATURE_DETECTOR_TYPE_7_12}
21     fast = cv2.FastFeatureDetector_create(nonmaxSuppression=True, **args)
22     keypoints = fast.detect(image, None)
23     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
24
25 def orb(image, args=None):
26
27     args = args or {
28         'nfeatures': 500, 'scaleFactor': 1.1, 'nlevels': 10,
29         'edgeThreshold': 31, 'firstLevel': 0, 'WTA_K': 2,
30         'scoreType': cv2.ORB_HARRIS_SCORE, 'patchSize': 31, 'fastThreshold': 20
31     }
32     orb = cv2.ORB_create(**args)
33     keypoints = orb.detect(image, None)
34     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
35
36 def sift(image, args=None):
37
38     args = args or {
39         'nOctaveLayers': 3, 'contrastThreshold': 0.04, 'edgeThreshold': 10, 'sigma': 1.6
40     }
41     sift = cv2.SIFT_create(**args)

```

```

42     keypoints, _ = sift.detectAndCompute(image, None)
43     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
44
45 def brisk(image, args=None):
46
47     args = args or {'thresh': 30, 'octaves': 3, 'patternScale': 1.0}
48     brisk = cv2.BRISK_create(**args)
49     keypoints, _ = brisk.detectAndCompute(image, None)
50     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
51
52 def agast(image, args=None):
53
54     args = args or {'threshold': 10, 'type': 1}
55     agast = cv2.AgastFeatureDetector_create(nonmaxSuppression=True, **args)
56     keypoints = agast.detect(image, None)
57     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
58
59 def kaze(image, args=None):
60
61     args = args or {'threshold': 0.001, 'nOctaves': 3, 'nOctaveLayers': 3, 'diffusivity': 1}
62     kaze = cv2.KAZE_create(**args)
63     keypoints, _ = kaze.detectAndCompute(image, None)
64     return np.array([kp.pt for kp in keypoints], dtype=np.float32)
65
66 def akaze(image, args=None):
67
68     args = args or {'threshold': 0.001, 'nOctaves': 4, 'nOctaveLayers': 4, 'diffusivity': 2}
69     akaze = cv2.AKAZE_create(**args)
70     keypoints, _ = akaze.detectAndCompute(image, None)
71     return np.array([kp.pt for kp in keypoints], dtype=np.float32)

```

Listing 8: All Corner Detection Wrappers

```

1 import numpy as np
2 import cv2
3 from itertools import product
4 import matplotlib.pyplot as plt
5 import os
6 from scipy.spatial.distance import cdist
7 from tqdm import tqdm
8 from parameters import PARAM_REPORT, ALGORITHMS, SCALES
9
10
11 def plot_ground_truth(images, image_names, ground_truth_corners, output_path):
12
13     plt.figure(figsize=(8,10))
14     plt.suptitle("Ground_Truth_Images")
15     for i in range(len(images)):
16         plt.subplot(6, 4, i+1)
17         plt.imshow(images[i], cmap="gray")
18         plt.axis("off")
19         plt.title(f"Image_{image_names[i]}_{(x{len(ground_truth_corners[i])})}")
20         for corner in ground_truth_corners[i]:
21             plt.scatter(corner[1], corner[0], color="green", marker='o', s=10)
22
23     plt.tight_layout()
24     plt.savefig(os.path.join(output_path, "ground_truth.png"))
25     plt.close()
26
27
28 def create_scaled_images(image):
29     scaled_images = []
30     for scale in SCALES:
31         scaled = cv2.resize(image, None, fx=scale, fy=scale, interpolation=cv2.INTER_LINEAR)
32         scaled_images.append(scaled)
33     return scaled_images
34
35
36 def calculate_metrics(pred_corners, gt_corners, threshold=5.0):
37     gt = np.array(gt_corners, dtype=np.float32)
38     pred = np.array(pred_corners, dtype=np.float32)
39
40     # Reshape if necessary
41     if pred.ndim == 1 and pred.shape[0] == 2:

```

```

42     pred = pred.reshape(1, 2)
43 if pred.ndim == 1 or pred.size == 0:
44     pred = np.empty((0, 2), dtype=np.float32)
45
46 if gt.ndim == 1 and gt.shape[0] == 2:
47     gt = gt.reshape(1, 2)
48 if gt.ndim == 1 or gt.size == 0:
49     gt = np.empty((0, 2), dtype=np.float32)
50
51 # Overall corner quantity (total number of detected corners)
52 corner_quantity = len(pred)
53 corner_quantity_ratio = 1 / (np.abs(len(pred) - len(gt_corners)) + 1) ## 1/(|gt - pred| + 1)
54 # Early return if either side has no corners
55 if len(gt) == 0 or len(pred) == 0:
56     return 0.0, 0.0, 0.0, 0.0, 0.0, corner_quantity, corner_quantity_ratio
57
58 # Compute distances between ground truth and predicted corners
59 dists = cdist(gt, pred) # shape: (num_gt, num_pred)
60 matched_gt = np.any(dists <= threshold, axis=1)
61 matched_pred = np.any(dists <= threshold, axis=0)
62
63 TP = np.sum(matched_gt)
64 FP = len(pred) - np.sum(matched_pred)
65 FN = len(gt) - TP
66
67 precision = TP / (TP + FP) if (TP + FP) > 0 else 0
68 recall = TP / (TP + FN) if (TP + FN) > 0 else 0
69 repeatability = TP / len(gt) if len(gt) > 0 else 0
70
71 denom = (precision + recall)
72 f_score = 2 * (precision * recall) / denom if denom > 0 else 0
73 apr = (precision + recall) / 2 # Arithmetic Mean of Precision and Recall
74
75 # Compute localization error for matched corners
76 localization_error = 0.0
77 if TP > 0:
78     # Find the closest predicted corner for each ground truth corner
79     min_dists = np.min(dists, axis=1) # Minimum distance for each ground truth corner
80     matched_dists = min_dists[matched_gt] # Distances for matched ground truth corners
81     localization_error = np.mean(matched_dists) if len(matched_dists) > 0 else 0.0
82
83 return precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
corner_quantity_ratio
84
85
86 def generate_sample_detections(images, image_names, ground_truth_corners, optimized_params, output_path):
87
88     sample_output_path = os.path.join(output_path, "sample_detections")
89     os.makedirs(sample_output_path, exist_ok=True)
90
91     idx = np.random.randint(0, len(images))
92     image = images[idx]
93     name = image_names[idx]
94
95     for alg_name, alg_func in ALGORITHMS.items():
96         params = optimized_params[alg_name]
97         corners = alg_func(image, args=params)
98         gt_corners = ground_truth_corners[idx]
99
100        plt.figure(figsize=(4, 8)) # Adjusted for 1x2 grid
101        plt.suptitle(f"Sample_Detection:{alg_name}_{Image_{name}}", fontsize=12)
102
103        # Ground truth subplot (left column)
104        plt.subplot(2, 1, 1)
105        plt.imshow(image, cmap="gray")
106        plt.title(f"Ground_Truth_{x{len(gt_corners)}}", fontsize=10)
107        plt.axis("off")
108
109        # Plot ground truth corners
110        for corner in gt_corners:
111            plt.scatter(corner[1], corner[0], c='green', marker='o', s=10, label='Ground_Truth')
112
113        # Detected corners subplot (right column)
114        plt.subplot(2, 1, 2)
115        plt.imshow(image, cmap="gray")
116        plt.title(f"Detected_{x{len(corners)}}", fontsize=10)

```

```

117     plt.axis("off")
118
119     # Plot detected corners
120     size = 10
121     if len(corners) > 100:
122         size = 5
123     for corner in corners:
124         plt.scatter(corner[0], corner[1], c='red', marker='o', s=size, label='Detected')
125
126     plt.tight_layout(rect=[0, 0, 1, 0.95])
127     plt.savefig(os.path.join(sample_output_path, f"{alg_name}_detected.png"))
128     plt.close()
129
130
131 def plot_all_detections(images, image_names, optimized_params, output_path):
132     sample_output_path = os.path.join(output_path, "sample_detections")
133     os.makedirs(sample_output_path, exist_ok=True)
134
135     for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Plotting All Optimized Detections"):
136         params = optimized_params.get(alg_name, {})
137         plt.figure(figsize=(8, 10))
138         plt.suptitle(f"Detected Corners: {alg_name}", fontsize=12)
139
140         for i in range(len(images)):
141             try:
142                 corners = alg_func(images[i], args=params)
143                 if corners is None or len(corners) == 0:
144                     corners = np.array([])
145             except Exception as e:
146                 print(f"Error processing {alg_name} for image {image_names[i]}: {e}")
147                 corners = np.array([])
148
149                 plt.subplot(6, 4, i + 1)
150                 plt.imshow(images[i], cmap="gray")
151                 plt.axis("off")
152                 plt.title(f"Image {image_names[i]} ({len(corners)})")
153                 size = 10
154                 if len(corners) > 100:
155                     size = 5
156                 for corner in corners:
157                     plt.scatter(corner[0], corner[1], color="red", marker='o', s=size)
158
159             plt.tight_layout()
160             plt.savefig(os.path.join(sample_output_path, f"{alg_name}_all_detections.png"))
161             plt.close()
162
163
164 def generate_scale_invariance_samples(scaled_images, detected_corners, gt_corners, alg_name, image_name,
165                                       output_path):
166
167     sample_output_path = os.path.join(output_path, "scale_invariance_samples")
168     os.makedirs(sample_output_path, exist_ok=True)
169
170     plt.figure(figsize=(8, 12)) # Adjusted for 3x2 grid
171     plt.suptitle(f"Scale Invariance Sample: {alg_name} (Image {image_name})", fontsize=12)
172
173     for i, (scale, image, corners) in enumerate(zip(SCALES, scaled_images, detected_corners)):
174         # Ground truth subplot (left column, i.e., subplot 1, 3, 5)
175         plt.subplot(3, 2, 2 * i + 1)
176         plt.imshow(image, cmap="gray")
177         plt.title(f"Ground Truth ({len(gt_corners)}) (Scale {scale})", fontsize=10)
178         plt.axis("off")
179
180         # Plot ground truth corners (adjusted for scale)
181         scaled_gt = gt_corners * scale
182         for corner in scaled_gt:
183             plt.scatter(corner[1], corner[0], c='green', marker='o', s=10, label='Ground Truth' if i == 0 else "")
184
185         # Detected corners subplot (right column, i.e., subplot 2, 4, 6)
186         plt.subplot(3, 2, 2 * i + 2)
187         plt.imshow(image, cmap="gray")
188         plt.title(f"Detected ({len(corners)}) (Scale {scale})", fontsize=10)
189         plt.axis("off")
190
191         size = 10
192         if len(corners) > 100:

```

```

192         size = 5
193
194     if scale < 1:
195         size = int(size * scale)
196     # Plot detected corners
197     for corner in corners:
198         plt.scatter(corner[0], corner[1], c='red', marker='o', s=size, label='Detected' if i == 0
199             else "")
200
201     plt.tight_layout(rect=[0, 0, 1, 0.95])
202     plt.savefig(os.path.join(sample_output_path, f"{alg_name}_scale_invariance_sample.png"))
203     plt.close()
204
205
206 def generate_comparison_tables(results, output_path):
207     metrics = ['precision', 'recall', 'repeatability', 'speed', 'f_score', 'apr', 'localization_error',
208         'corner_quantity_ratio']
209     table_content = [
210         "\\begin{table}[h]",
211         "\\centering",
212         "\\caption{Comparison\_of\_Corner\_Detection\_Algorithms}",
213         "\\label{tab:comparison_table}",
214         "\\begin{tabular}{lccccccc}",
215         "\\toprule",
216         "\\textbf{Algorithm} & \\textbf{Precision} & \\textbf{Recall} & \\textbf{Repeatability} & \\textbf{Speed} & \\textbf{F\_Score} & \\textbf{APR} & \\textbf{Localization\_Error} & \\textbf{Corner\_Quantity\_Ratio} \\\\"\\midrule",
217     ]
218
219     for alg_name in results:
220         row = f'{alg_name}'
221         for metric in metrics:
222             data = results[alg_name][metric]
223             mean_value = np.mean(data) if data else 0.0
224             if metric == 'speed':
225                 row += f'& {mean_value:.4f}'"
226             elif metric == 'corner_quantity':
227                 row += f'& {int(mean_value)}'
228             else:
229                 row += f'& {mean_value:.3f}'"
230         row += "\\\\""
231         table_content.append(row)
232
233     table_content.extend([
234         "\\bottomrule",
235         "\\end{tabular}",
236         "\\end{table}"
237     ])
238
239     table_path = os.path.join(output_path, "comparison_table.txt")
240     with open(table_path, 'w') as f:
241         f.write("\n".join(table_content))
242
243 def visualize_results(output_path, image_names):
244     plt.figure(figsize=(9, 9))
245
246     x_ticks = []
247     for idx, metric in enumerate(['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr',
248         'localization_error', 'corner_quantity', 'corner_quantity_ratio']):
249         plt.subplot(3, 3, idx + 1)
250         for alg_name in ALGORITHMS:
251             data = np.load(os.path.join(output_path, f'data/{alg_name}_results.npz'))[metric]
252             plt.plot(data, label=alg_name)
253             plt.title(metric.capitalize())
254             plt.legend(loc="upper_right", fontsize="xx-small")
255             plt.xlabel('Image Number')
256
257     plt.tight_layout()
258     plt.savefig(os.path.join(output_path, 'benchmark_results.png'))
259     plt.close()
260
261 def plot_best_combination(individual_results, image_names, output_path):
262     """Plot the metrics of the best parameter combination for each algorithm across images."""
263     plot_output_path = os.path.join(output_path, "best_combination_plots")
264     os.makedirs(plot_output_path, exist_ok=True)

```

```

264
265 metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
266   'corner_quantity', 'corner_quantity_ratio']
267
268 for alg_name in individual_results:
269     plt.figure(figsize=(9, 9))
270     plt.suptitle(f"Best Combination Metrics for {alg_name}")
271
272     for idx, metric in enumerate(metrics):
273         plt.subplot(3, 3, idx + 1)
274         # Gather metric data across all images
275         metric_data = [individual_results[alg_name][img_name][metric][0] for img_name in
276                         image_names]
277         plt.plot(image_names, metric_data, marker='o', color='red', label=alg_name)
278         plt.title(metric.capitalize())
279         plt.xlabel('Image Number')
280         plt.ylabel(metric.capitalize())
281
282         if metric not in ['speed', 'localization_error', 'corner_quantity']:
283             plt.ylim(0, 1)
284         else:
285             plt.ylim(0, np.max(metric_data) * 1.2 if np.max(metric_data) > 0 else 1)
286             plt.legend(loc="upper right")
287
288     plt.tight_layout(rect=[0, 0, 1, 0.95])
289     plt.savefig(os.path.join(plot_output_path, f"{alg_name}_best_combination.png"))
290     plt.close()
291
292 def plot_all_combinations(all_metrics_per_algorithm, image_names, output_path):
293     plot_output_path = os.path.join(output_path, "all_combinations_plots")
294     os.makedirs(plot_output_path, exist_ok=True)
295
296     metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
297       'corner_quantity', 'corner_quantity_ratio']
298
299     for alg_name in all_metrics_per_algorithm:
300         all_metrics = all_metrics_per_algorithm[alg_name]
301         # Find the best combination (highest score)
302         best_idx = np.argmax([m['score'] for m in all_metrics])
303
304         plt.figure(figsize=(9, 9))
305         plt.suptitle(f"All Parameter Combinations for {alg_name}")
306
307         for idx, metric in enumerate(metrics):
308             plt.subplot(3, 3, idx + 1)
309
310             # Plot all combinations with transparency
311             for combo_idx, metrics_dict in enumerate(all_metrics):
312                 metric_data = metrics_dict[metric]
313                 if combo_idx == best_idx:
314                     # Highlight the best combination with a bold line
315                     plt.plot(image_names, metric_data, color='red', linewidth=3, label='Best Combination',
316                             markersize=3)
317                 else:
318                     # Plot other combinations with transparency
319                     plt.plot(image_names, metric_data, alpha=0.15, color='green', markersize=1,
320                             linewidth=0.75, label='Other Combinations' if combo_idx == 0 else "")
321
322             plt.title(metric.capitalize())
323             plt.xlabel('Image Number')
324             plt.ylabel(metric.capitalize())
325
326             if metric not in ['speed', 'localization_error', 'corner_quantity']:
327                 plt.ylim(0, 1)
328             else:
329                 all_values = [np.max(metrics_dict[metric]) for metrics_dict in all_metrics]
330                 plt.ylim(0, np.max(all_values) * 1.2 if np.max(all_values) > 0 else 1)
331             plt.legend(loc="upper right")
332
333             plt.tight_layout(rect=[0, 0, 1, 0.95])
334             plt.savefig(os.path.join(plot_output_path, f"{alg_name}_all_combinations.png"))
335             plt.close()
336
337 def plot_scale_invariance(scale_results, image_names, output_path):
338     plot_output_path = os.path.join(output_path, "scale_invariance_plots")

```

```

336     os.makedirs(plot_output_path, exist_ok=True)
337
338     metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
339                 'corner_quantity', 'corner_quantity_ratio']
340     scale_labels = [str(val) for val in SCALES]
341
342     for alg_name in scale_results:
343         plt.figure(figsize=(9, 9))
344         # Compute average scale invariance score across all images
345         avg_scale_invariance = np.mean([scale_results[alg_name][img_name]['scale_invariance'] for
346                                         img_name in image_names])
347         plt.suptitle(f"Scale_Invariance_Test_for_{alg_name}_(Avg_Scale_Invariance:_{avg_scale_invariance:.3f})")
348
349     for idx, metric in enumerate(metrics):
350         plt.subplot(3, 3, idx + 1)
351
352         # Plot a line for each image across scales
353         for img_idx, img_name in enumerate(image_names):
354             metric_data = scale_results[alg_name][img_name][metric]
355             plt.plot(scale_labels, metric_data, marker='o', label=f"Image_{img_name}")
356
357         plt.title(metric.capitalize())
358         plt.xlabel('Scale')
359         plt.ylabel(metric.capitalize())
360         if metric not in ['speed', 'localization_error', 'corner_quantity']:
361             plt.ylim(0, 1)
362         else:
363             all_values = [np.max(scale_results[alg_name][img_name][metric]) for img_name in
364                           image_names]
365             plt.ylim(0, np.max(all_values) * 1.2 if np.max(all_values) > 0 else 1)
366         # plt.legend(loc="upper right")
367
368         plt.tight_layout(rect=[0, 0, 1, 0.95])
369         plt.savefig(os.path.join(plot_output_path, f"{alg_name}_scale_invariance.png"))
370         plt.close()
371
372 def generate_param_report(params, combinations, scores, output_path):
373     param_report = PARAM_REPORT
374     for i, (alg_name, alg_func) in enumerate(ALGORITHMS.items()):
375         best_params = params[i]
376         total_combinations = combinations[i]
377         best_score = scores[i]
378
379         param_str = "\\\\".join([f"{{key.replace('_', '_\\_')}}: {{value}}" for key, value in best_params.
380                               items()])
381         param_str2 = f"\\\multirow{{\" + f\"{len(best_params.items())} + f\"}}{{*}}{{{{alg_name}}}}&\\\
382                     multirow{{\" + f\"{len(best_params.items())} + f\"}}{{*}}{{{{\\parbox{{3.25cm}}}{\\raggedright_\
383 {param_str}}}}}}&\\\
384                     multirow{{\" + f\"{len(best_params.items())} + f\"}}{{*}}{{{{total_combinations}}}}&\\\
385                     multirow{{\" + f\"{len(best_params.items())} + f\"}}{{*}}{{{{np.\
386 round(best_score, 4)}}}}}}\\\\\\"
387         param_report.append(param_str2)
388         param_report.append("\\&\\\\\\")
389         param_report.append("\\&\\\\\\")
390         param_report.append("\\&\\\\\\")
391
392         if len(best_params.items()) > 2:
393             for i in range(len(best_params.items()) - 2):
394                 param_report.append("\\&\\\\\\")
395
396         param_report.extend([
397             "\\bottomrule",
398             "\\end{tabular}",
399             "\\end{table}"])
400
401     with open(os.path.join(output_path, "optimal_parameters.txt"), 'w') as f:
402         f.write("\n".join(param_report))
403
404 def generate_scale_invariance_table(scale_results, output_path):
405     table_output_path = os.path.join(output_path, "scale_invariance_table.txt")
406     os.makedirs(output_path, exist_ok=True)
407
408     # Initialize table content
409     table_content = [
410         r"\begin{table}[t]",
411         r"\centering",

```

```

404     r"\small",
405     r"\caption{Mean\_Scale\_Invariance\_Scores\_for\_Corner\_Detection\_Algorithms}",
406     r"\label{tab:scale_invariance_scores}",
407     r"\begin{tabular}{lc}",
408     r"\toprule",
409     r"Algorithm & Scale Invariance Score \\",
410     r"\midrule",
411 ]
412
413 # Compute mean scale invariance score for each algorithm
414 for alg_name in sorted(scale_results.keys()):
415     scores = [scale_results[alg_name][img_name]['scale_invariance']
416               for img_name in scale_results[alg_name]]
417     mean_score = np.mean(scores)
418     # Format algorithm name (e.g., convert 'ShiTomas' to 'Shi-Tomas')
419     formatted_name = alg_name.replace('ShiTomas', 'Shi-Tomas')
420     table_content.append(f"{formatted_name} & {mean_score:.3f} \\\\"")
421
422 # Close table
423 table_content.extend([
424     r"\bottomrule",
425     r"\end{tabular}",
426     r"\end{table}"
427 ])
428
429 # Write table to file
430 with open(table_output_path, 'w') as f:
431     f.write('\n'.join(table_content))

```

Listing 9: Utility Functions for Data Processing