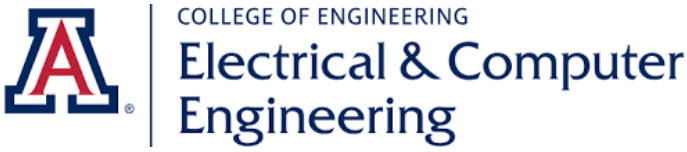
Benchmarking Corner Detection Algorithms

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:

$$M = \sum_{w} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{1}$$

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:

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

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:

$$R = \min(\lambda_1, \lambda_2) \tag{3}$$

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:

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

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:

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

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:

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

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.

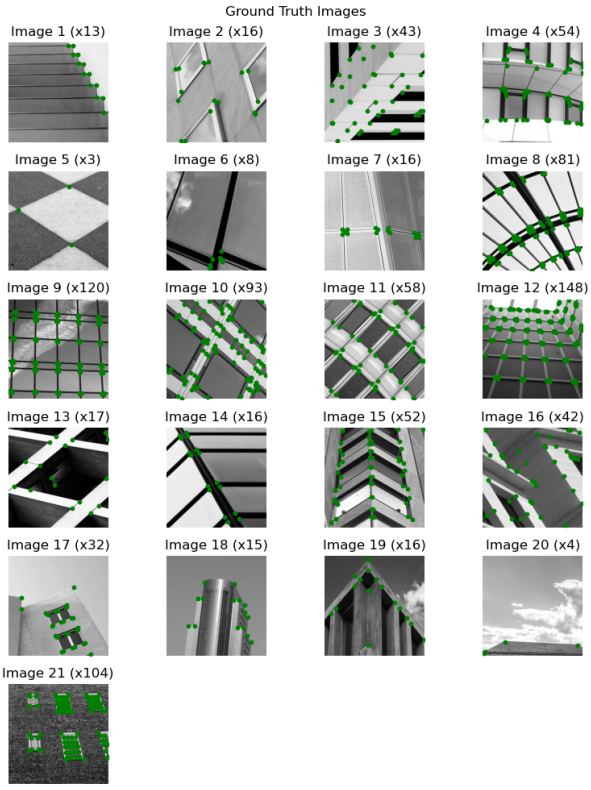


Fig. 1: Urban dataset with ground truth corners applied.

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`.

```
class UniformDist:
def __init__(self, min_val, max_val, is_int=
    False):
    self.min_val = min_val
    self.max_val = max_val
    self.is_int = is_int

def sample(self, n_samples):
    samples = np.round(np.random.uniform(self.
        min_val, self.max_val, n_samples), 4)
    if self.is_int:
        samples = np.round(samples).astype(int)
        samples = np.clip(samples, self.min_val,
             self.max_val)

    return np.unique(samples)[:n_samples]  #
        Ensure unique samples

    def sample_param_grid(param_grid, n_samples)
        :

sampled_grid = {}
for param_name, dist in param_grid.items():
    sampled_grid[param_name] = dist.sample(
        n_samples)
return list(sampled_grid.keys()), list(
    sampled_grid.values())

PARAM_GRIDS = {
    'Harris': {
        'blockSize': UniformDist(min_val=2,
            max_val=11, is_int=True),
        'ksize': CategoricalDist(options=[3, 5,
            7, 9, 11]),  # Only odd integers
        'k': UniformDist(min_val=0.01, max_val
            =0.15),
        'borderType': CategoricalDist(options=[
            cv2.BORDER_DEFAULT, cv2.
                BORDER_CONSTANT, cv2.
                BORDER_REFLECT
        ])
    }
    ... CONTINUED
}
```

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 $\frac{1}{|\text{pred}-\text{gt}|+1}$, which deals with essentially scoring the corner quantity against the

ground truth quantity, normalized between 0 and 1.

```python
def calculate_metrics(pred_corners, gt_corners, threshold=5.0):
    gt = np.array(gt_corners, dtype=np.float32)
    pred = np.array(pred_corners, dtype=np.float32)

    if pred.ndim == 1 and pred.shape[0] == 2:
        pred = pred.reshape(1, 2)
    if pred.ndim == 1 or pred.size == 0:
        pred = np.empty((0, 2), dtype=np.float32)

    if gt.ndim == 1 and gt.shape[0] == 2:
        gt = gt.reshape(1, 2)
    if gt.ndim == 1 or gt.size == 0:
        gt = np.empty((0, 2), dtype=np.float32)

    corner_quantity = len(pred)
    corner_quantity_ratio = 1 / (np.abs(len(pred) - len(gt_corners)) + 1) ## 1/(|gt - pred| + 1)

    if len(gt) == 0 or len(pred) == 0:
        return 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, corner_quantity, corner_quantity_ratio

    dists = cdist(gt, pred)
    matched_gt = np.any(dists <= threshold, axis=1)
    matched_pred = np.any(dists <= threshold, axis=0)

    TP = np.sum(matched_gt)
    FP = len(pred) - np.sum(matched_pred)
    FN = len(gt) - TP

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    repeatability = TP / len(gt) if len(gt) > 0 else 0

    denom = (precision + recall)
    f_score = 2 * (precision * recall) / denom if denom > 0 else 0
    apr = (precision + recall) / 2  # Arithmetic Mean

    localization_error = 0.0
    if TP > 0:
        min_dists = np.min(dists, axis=1)
        matched_dists = min_dists[matched_gt]
        localization_error = np.mean(matched_dists) if len(matched_dists) > 0 else 0.0

    return precision, recall, repeatability, f_score, apr, localization_error, 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.

```python
    avg_speed = np.mean(normalized_speeds)
    avg_precision = np.mean(precisions)
    avg_recall = np.mean(recalls)
    avg_le = np.mean(normalized_le)
    avg_corner_quant_ratios = np.mean(corner_quantity_ratios)
    vars = np.array([avg_speed, avg_precision, avg_recall, avg_le, avg_corner_quant_ratios])
    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.

```python
def harris(image, args=None):
    args = args or {'blockSize': 2, 'ksize': 3, 'k': 0.04}
    corners = cv2.cornerHarris(image, **args)
    corners = cv2.dilate(corners, None)
    corners = np.column_stack(np.where(corners > 0.01 * corners.max()))
    return corners[:, ::-1].astype(np.float32)

def shi_tomasi(image, args=None):
    args = args or {'maxCorners': 25, 'qualityLevel': 0.01, 'minDistance': 10}
    corners = cv2.goodFeaturesToTrack(image, **args)
    return corners[:, 0].astype(np.float32) if corners is not None else np.array([])

def fast(image, args=None):
    args = args or {'threshold': 25, 'type': cv2.FAST_FEATURE_DETECTOR_TYPE_7_12}
    fast = cv2.FastFeatureDetector_create(**args)
    keypoints = fast.detect(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

... 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 desciptors (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 $1 - \text{CV}$, 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 optimimally 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 specifcally created to combine with the inverted Speed, Precision, Recall, and the localization error. Over or under prediciting 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.

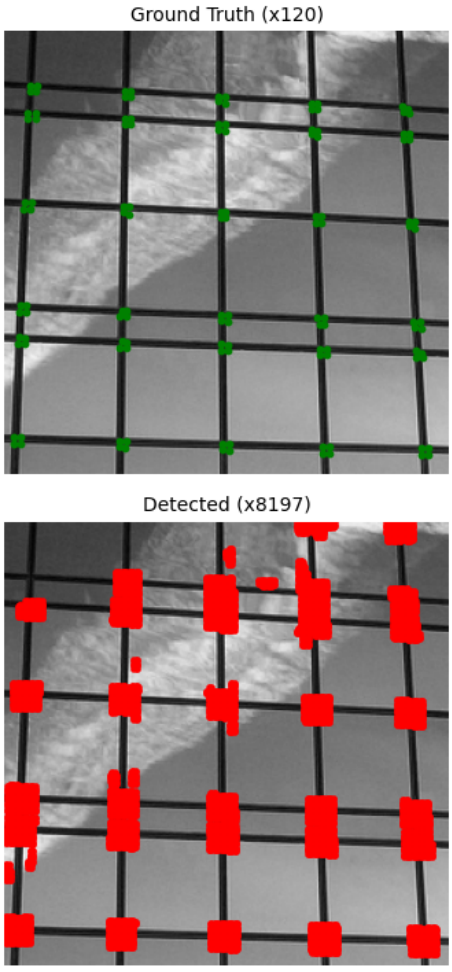


Fig. 2: Sample detection for Harris on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

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, 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×, 1.0×, and 4.0×. 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

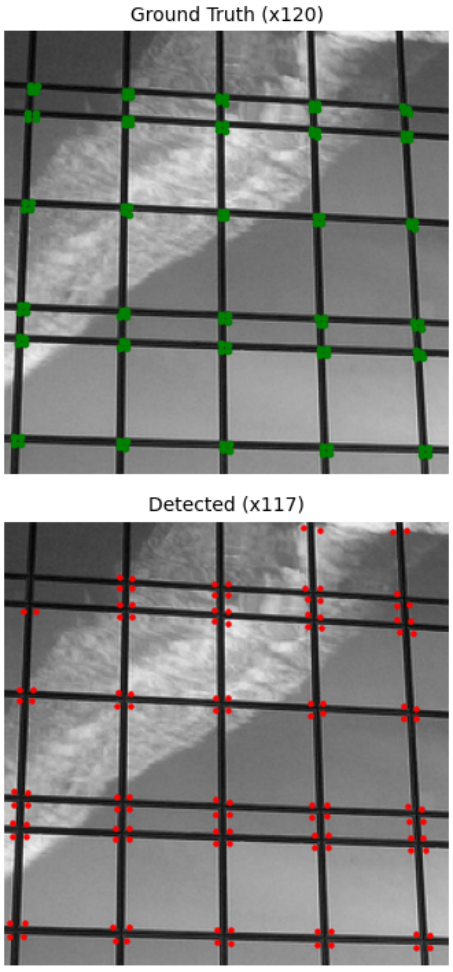


Fig. 3: Sample detection for Shi-Tomasi on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

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.

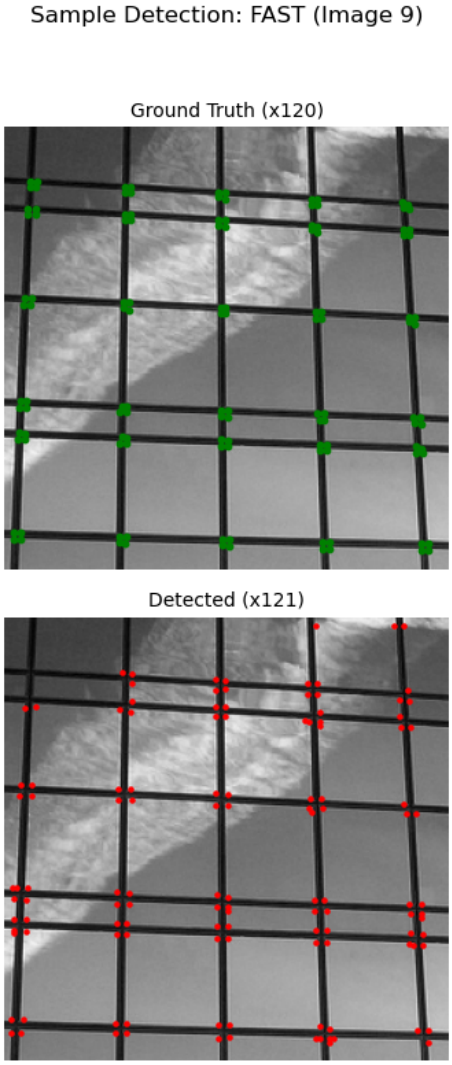


Fig. 4: Sample detection for FAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

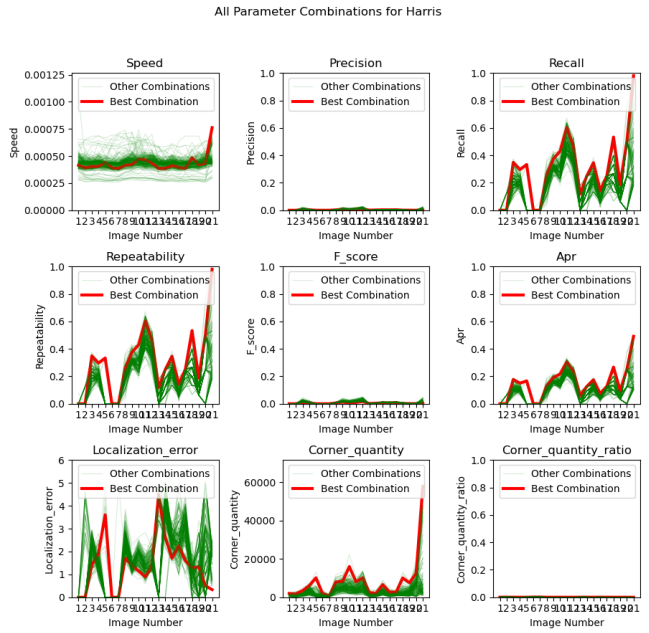


Fig. 5: Best parameter combination metrics for the Harris algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

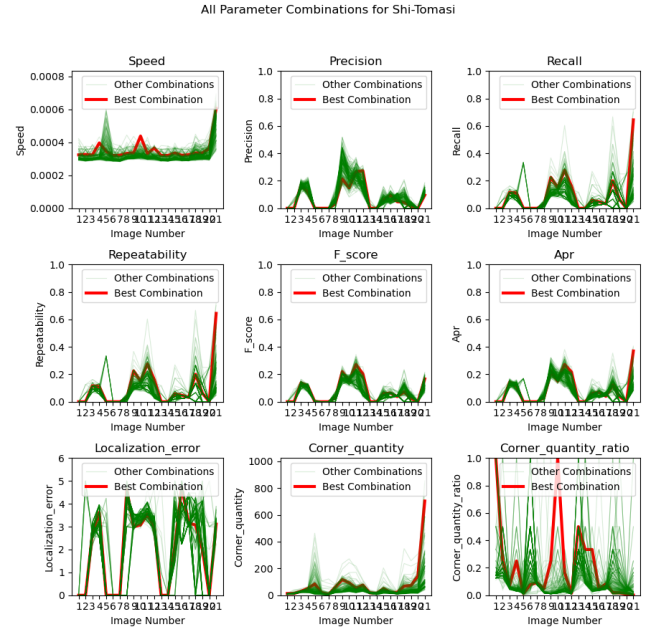


Fig. 6: Best parameter combination metrics for the Shi-Tomasi algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

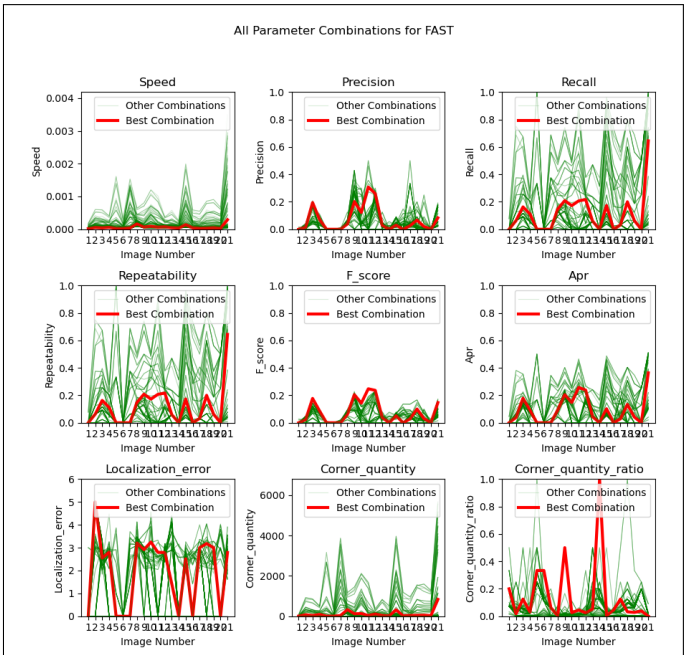


Fig. 7: Best parameter combination metrics for the FAST algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

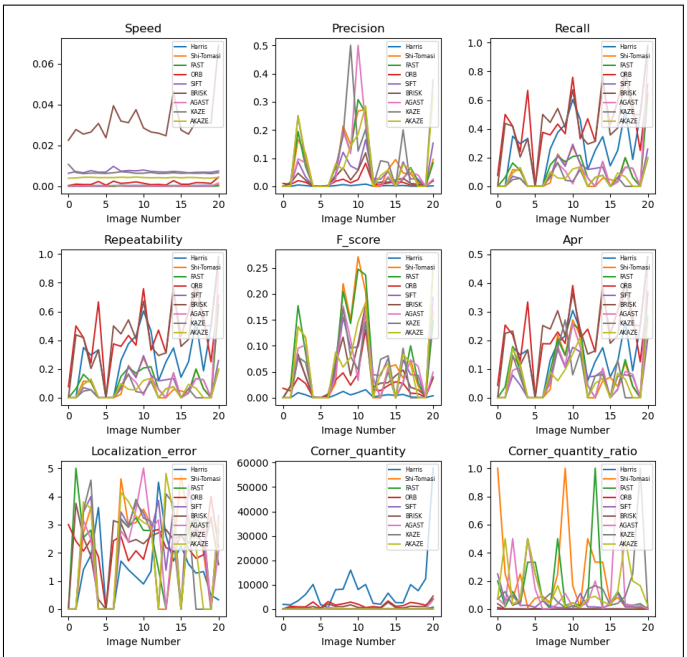


Fig. 8: Aggregate metrics across all images for each algorithm, comparing precision, recall, repeatability, speed, F-score, APR, localization error, corner quantity, and corner quantity ratio.

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

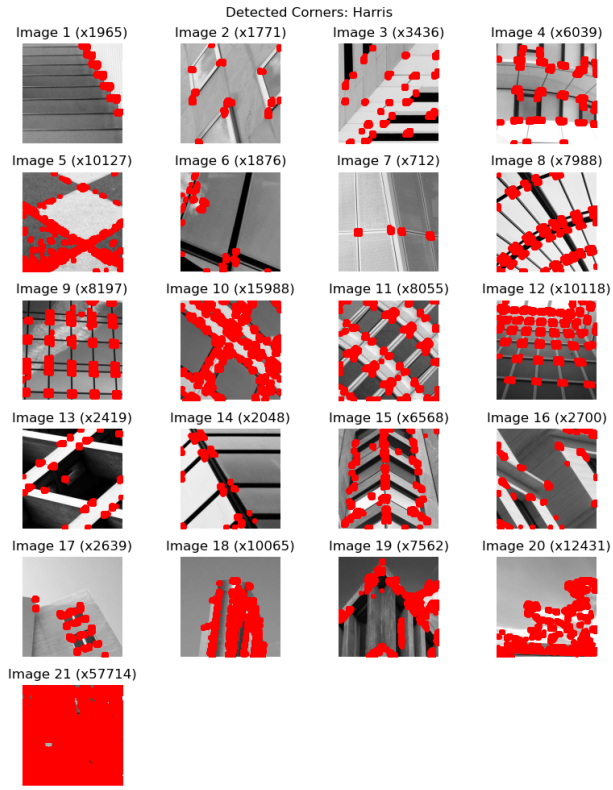


Fig. 9: Urban dataset with detected Harris corners applied.

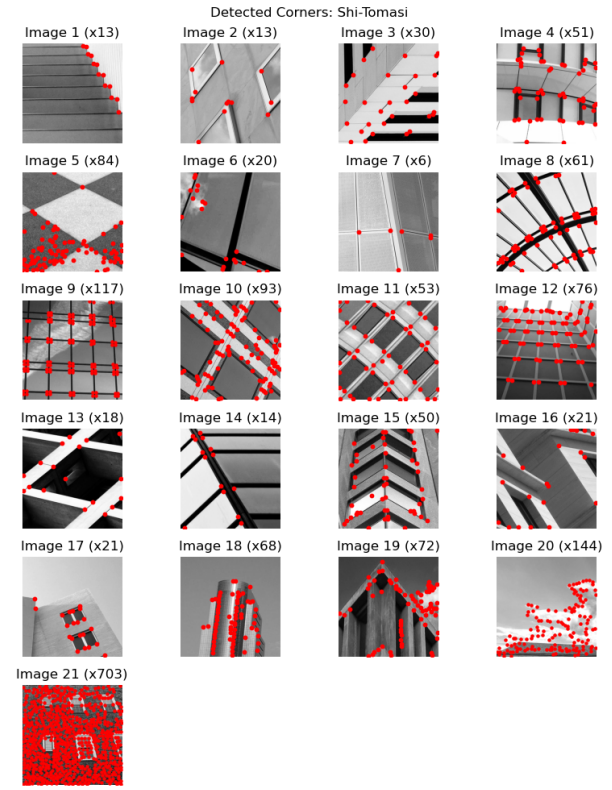


Fig. 10: Urban dataset with detected Shi-Tomasi corners applied.

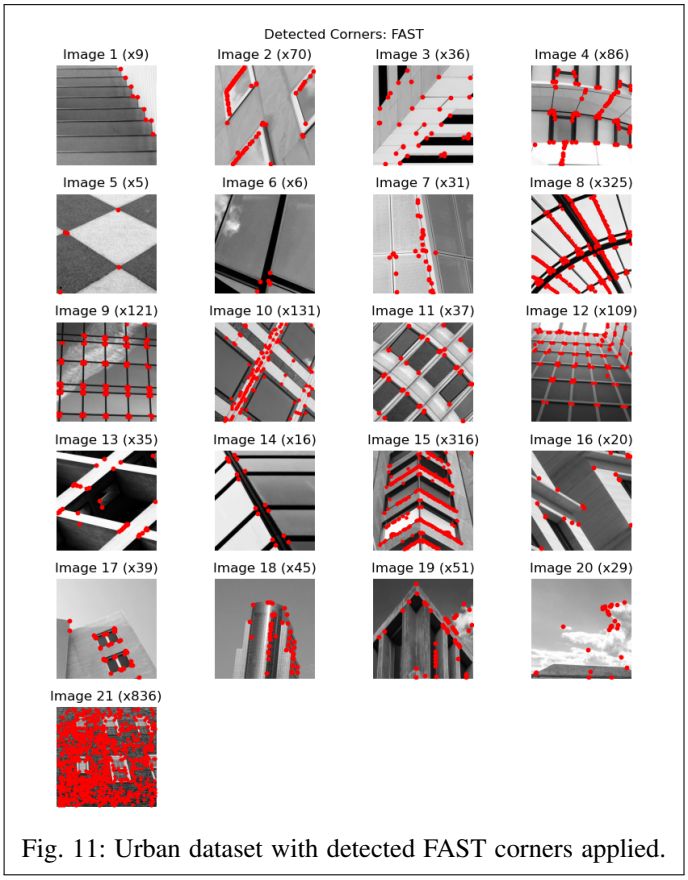


Fig. 11: Urban dataset with detected FAST corners applied.

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

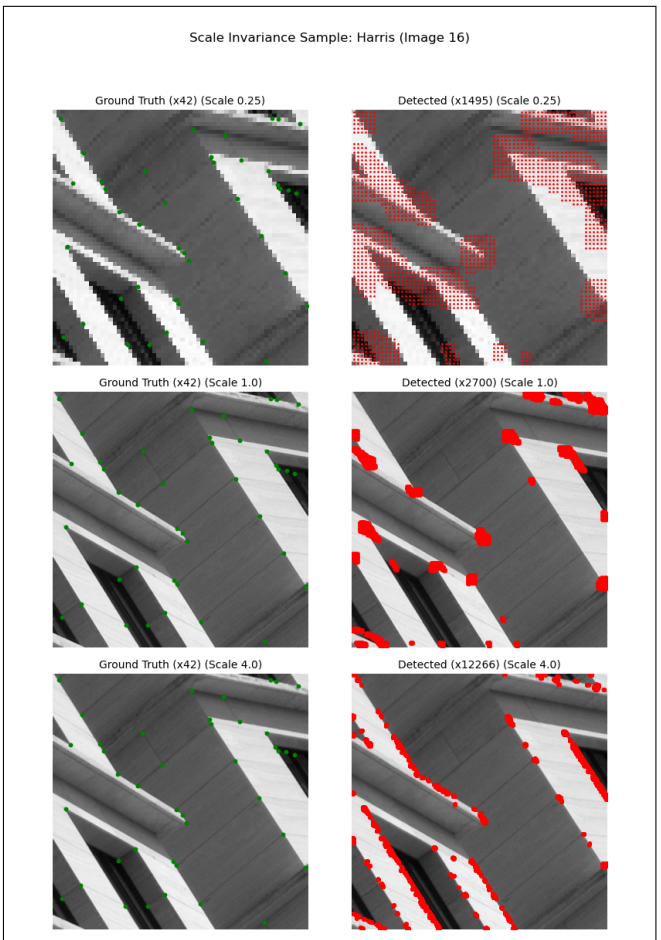


Fig. 12: Scale invariance sample for Harris on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

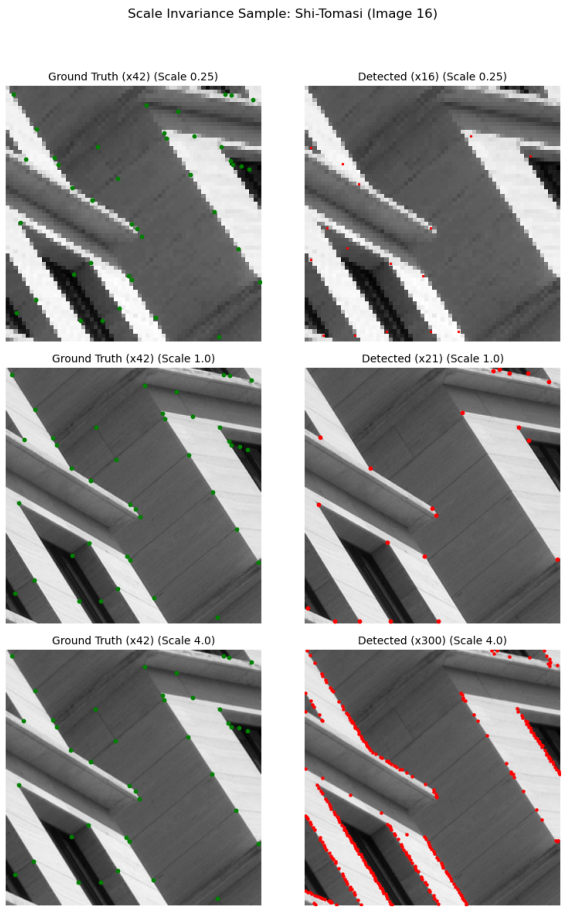


Fig. 13: Scale invariance sample for Shi-Tomasi on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

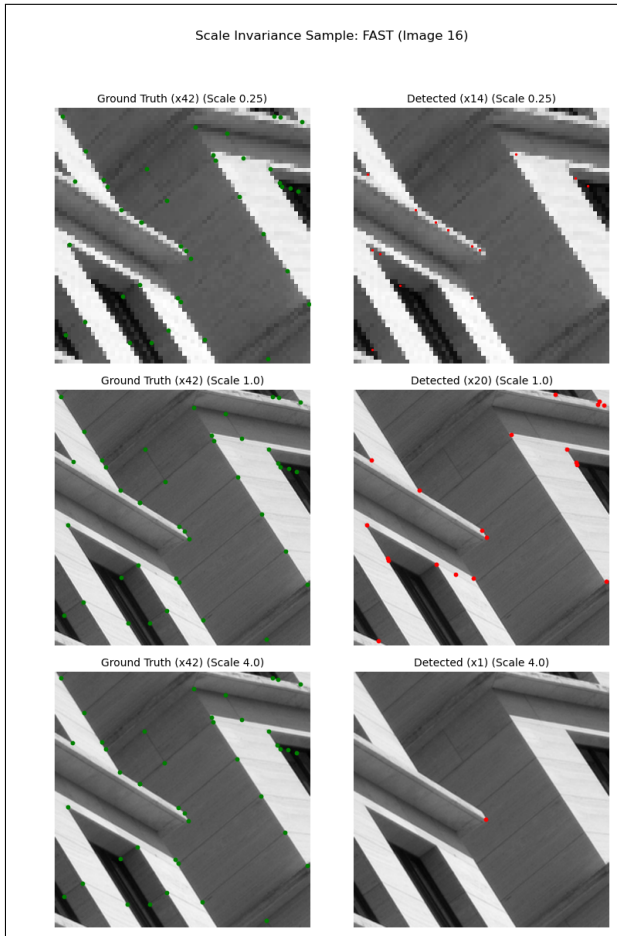


Fig. 14: Scale invariance sample for FAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

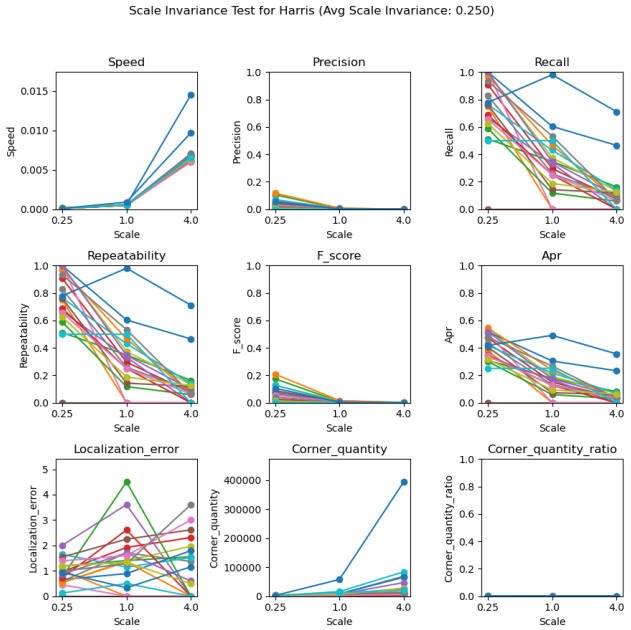


Fig. 15: Scale invariance results for Harris, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

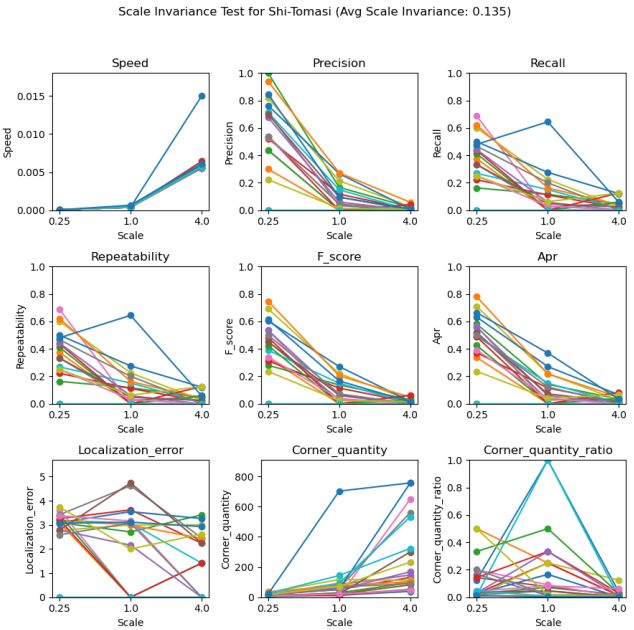


Fig. 16: Scale invariance results for Shi-Tomasi, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

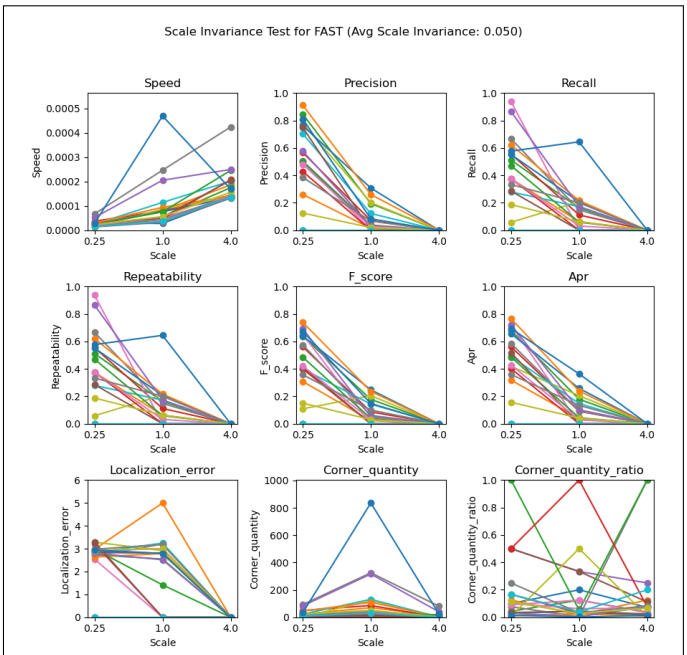


Fig. 17: Scale invariance results for FAST, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

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

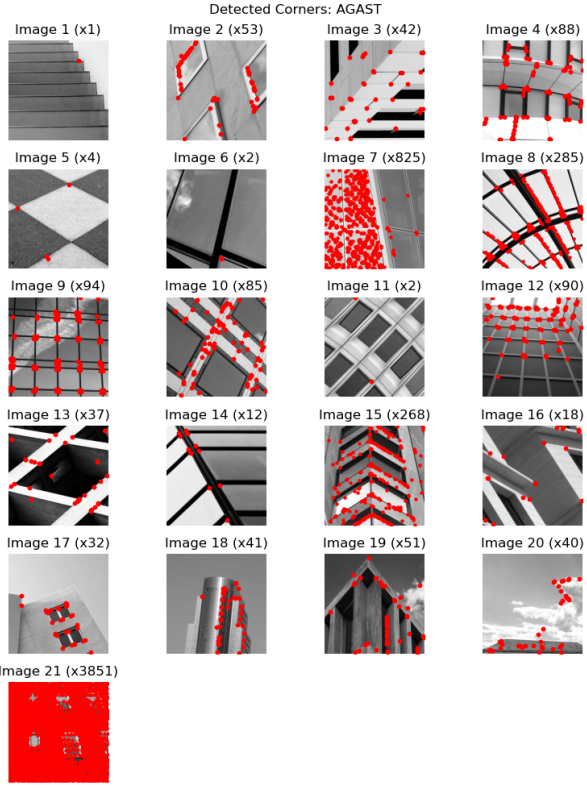


Fig. 18: Sample detection for AGAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

Appendix B
Scale Invariance Results: Images and Plots

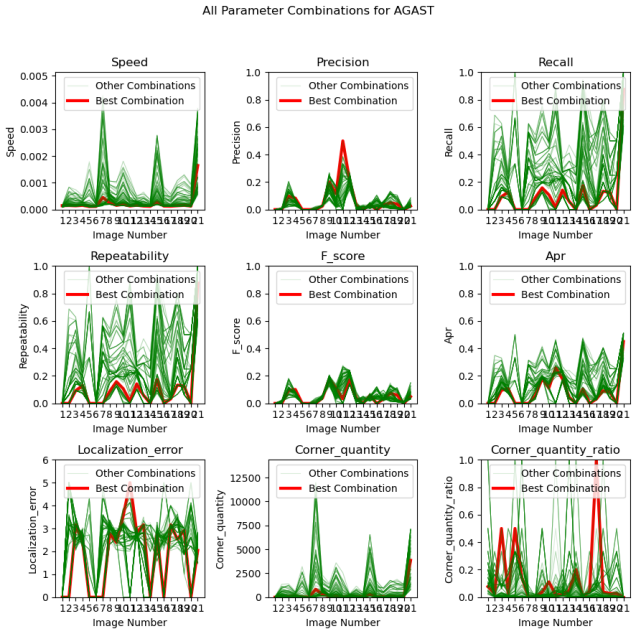


Fig. 19: Best parameter combination metrics for the AGAST algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

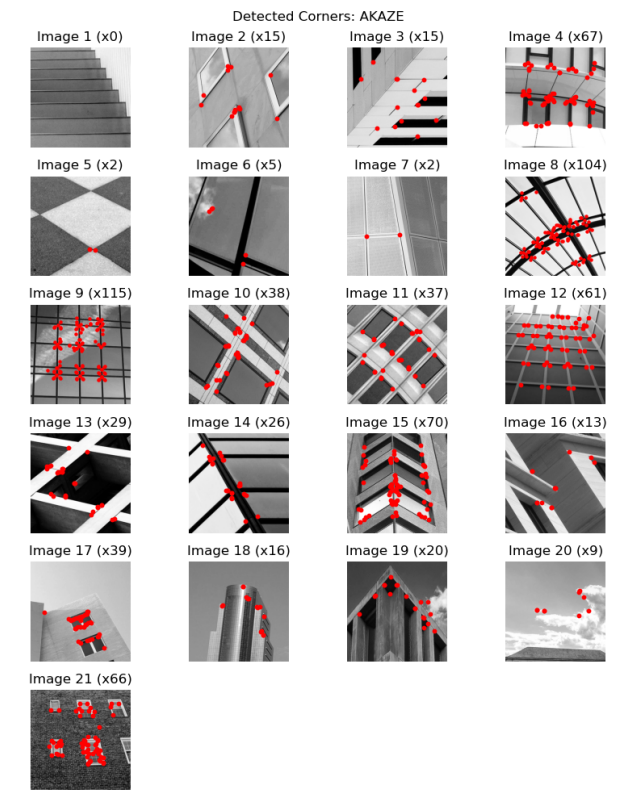


Fig. 20: Sample detection for AKAZE on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

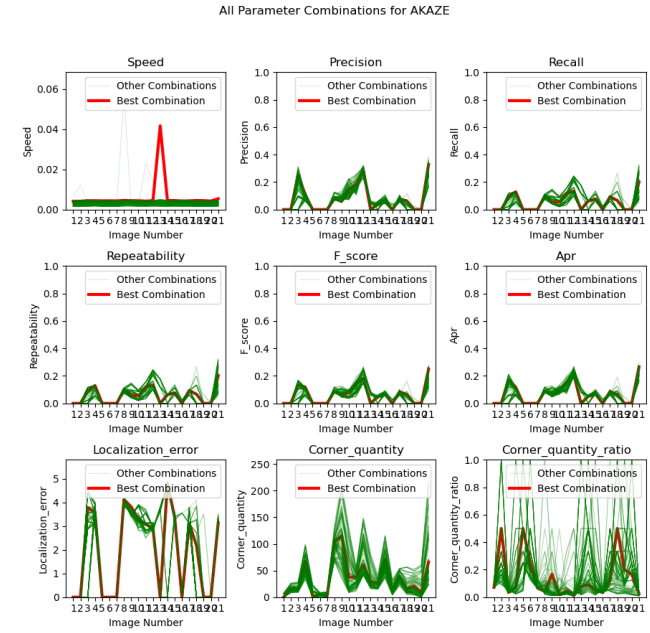


Fig. 21: Best parameter combination metrics for the AKAZE algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

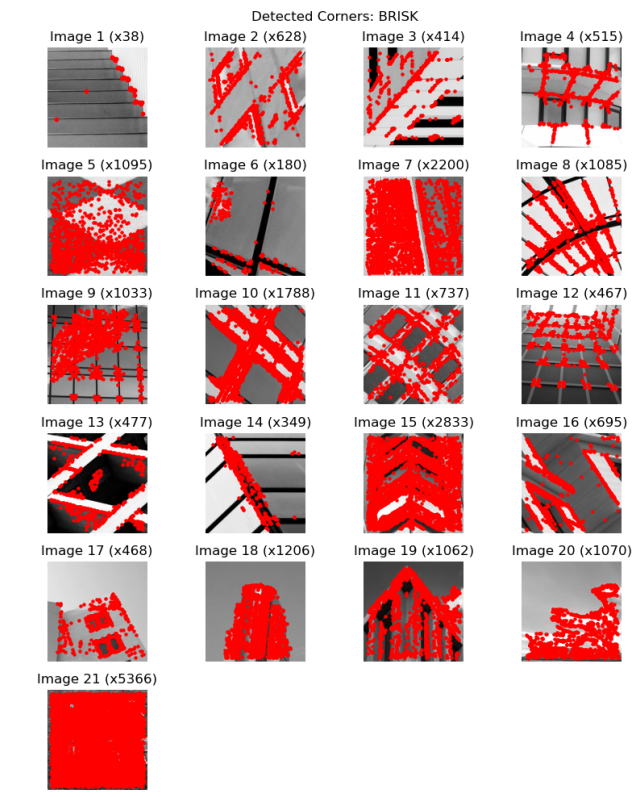


Fig. 22: Sample detection for BRISK on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

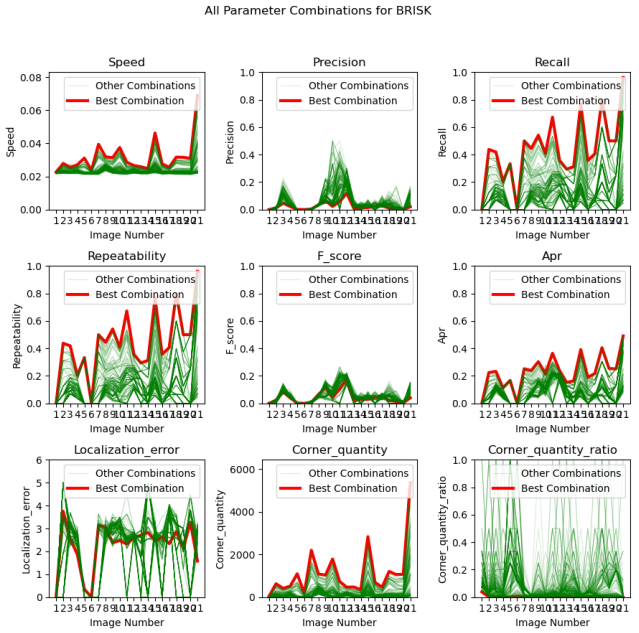


Fig. 23: Best parameter combination metrics for the BRISK algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

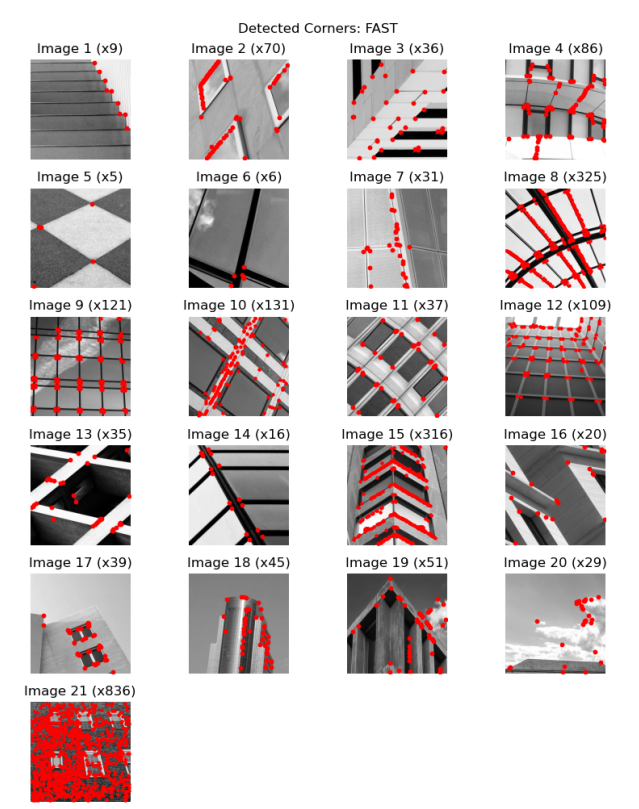


Fig. 24: Sample detection for FAST on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

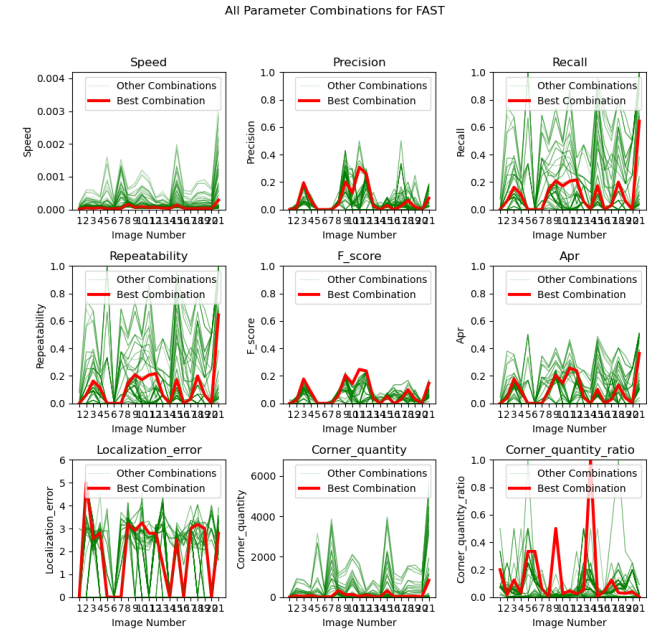


Fig. 25: Best parameter combination metrics for the FAST algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

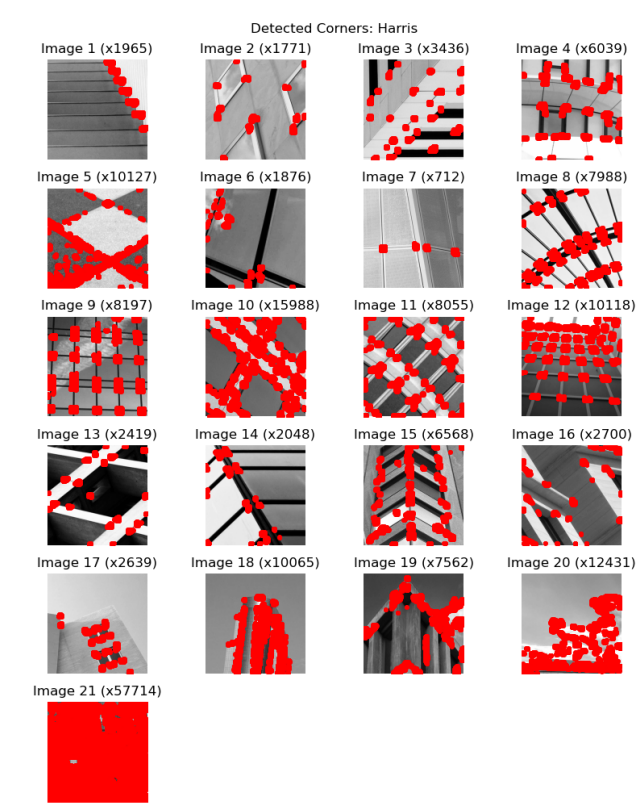


Fig. 26: Sample detection for Harris on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

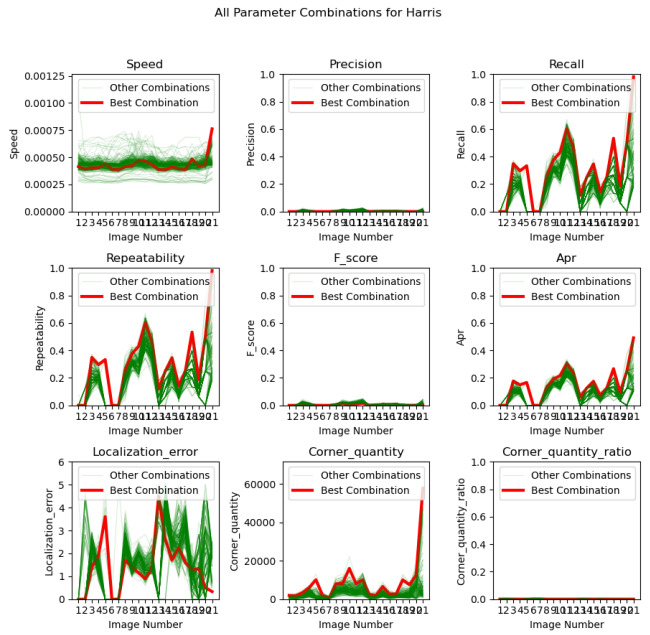


Fig. 27: Best parameter combination metrics for the Harris algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

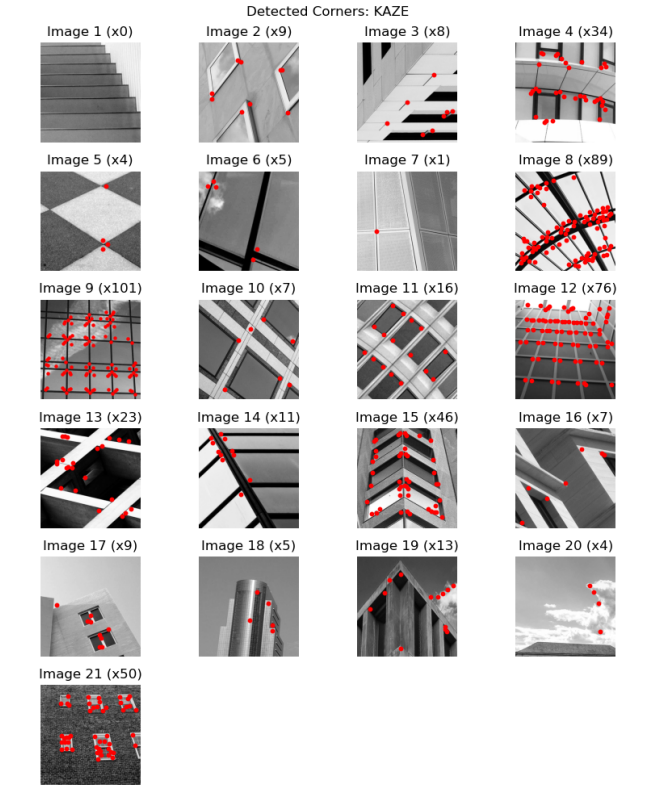


Fig. 28: Sample detection for KAZE on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

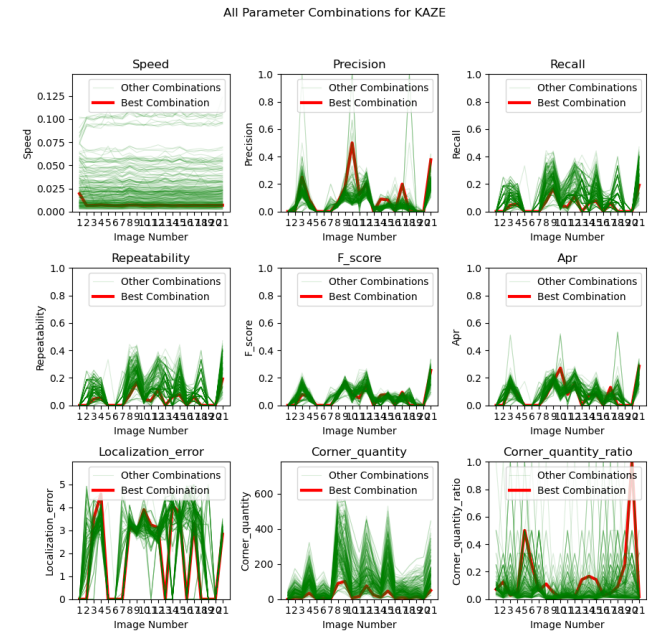


Fig. 29: Best parameter combination metrics for the KAZE algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

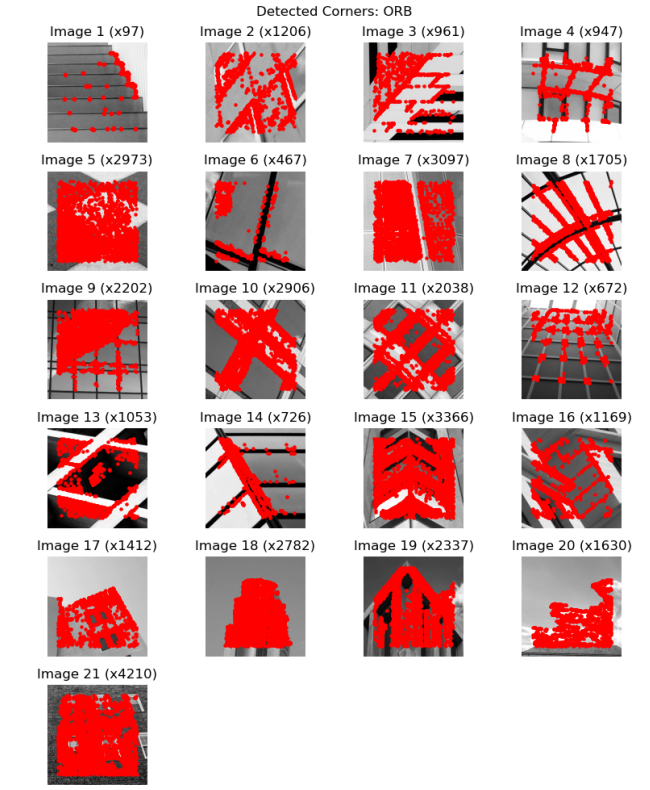


Fig. 30: Sample detection for ORB on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

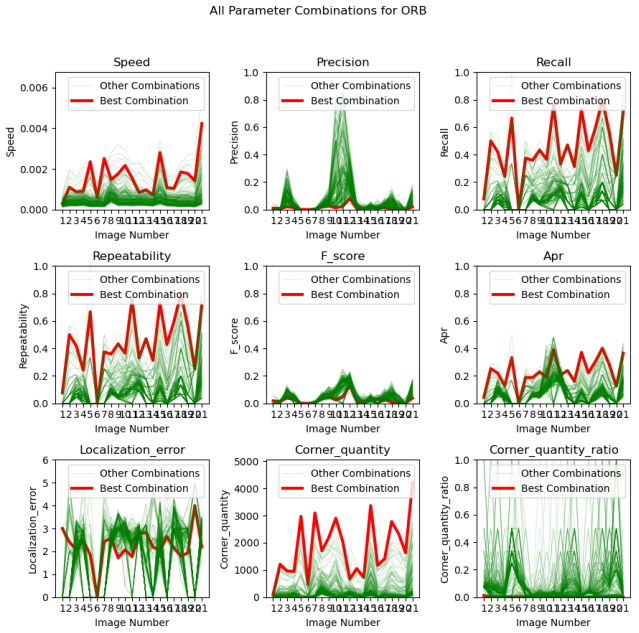


Fig. 31: Best parameter combination metrics for the ORB algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

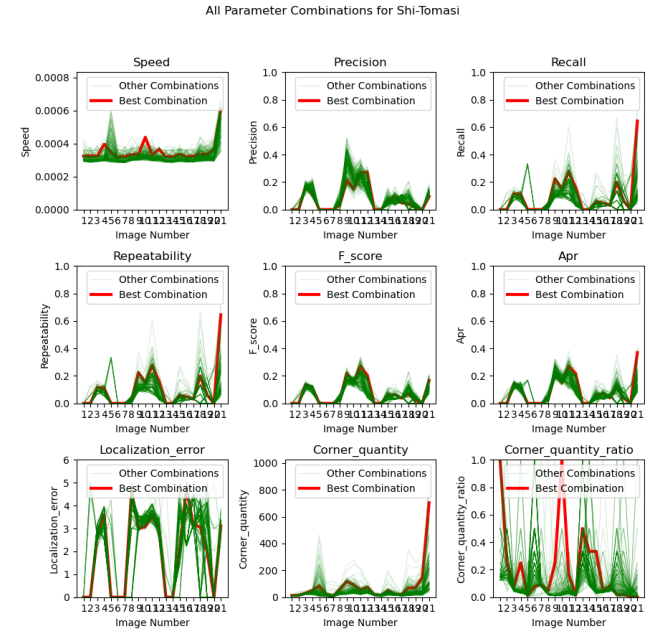


Fig. 33: Best parameter combination metrics for the Shi-Tomasi algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

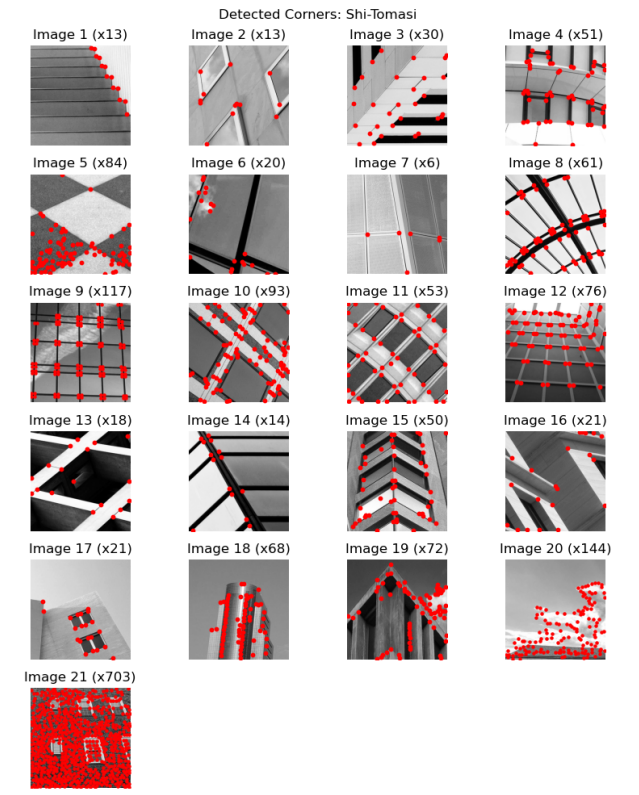


Fig. 32: Sample detection for Shi-Tomasi on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

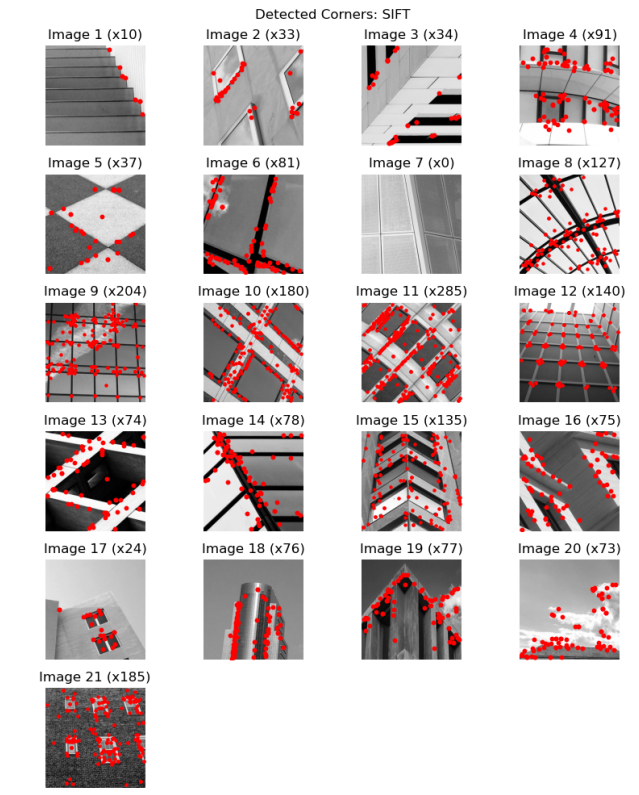


Fig. 34: Sample detection for SIFT on a randomly selected image, with ground truth corners (green circles) and detected corners (red circles).

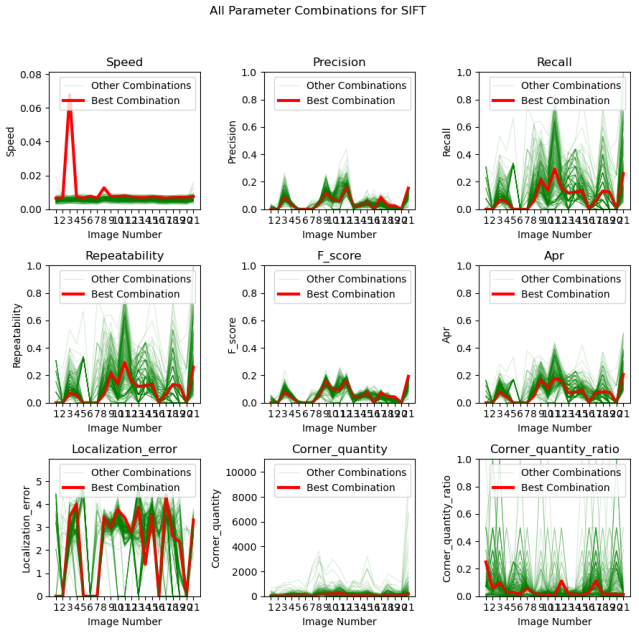


Fig. 35: Best parameter combination metrics for the SIFT algorithm across individual images, showing precision, recall, repeatability, F-score, APR, localization error, corner quantity, corner quantity ratio, and speed.

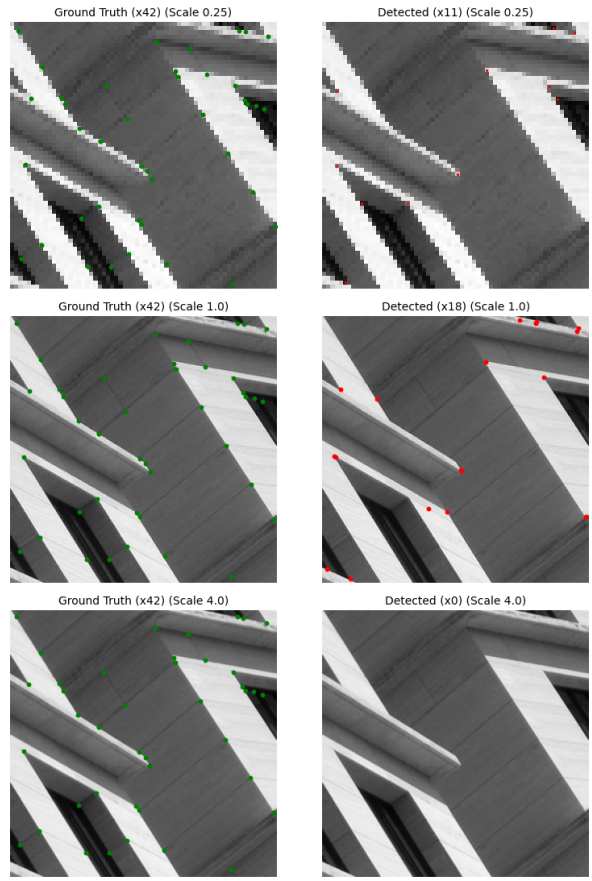


Fig. 36: Scale invariance sample for AGAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

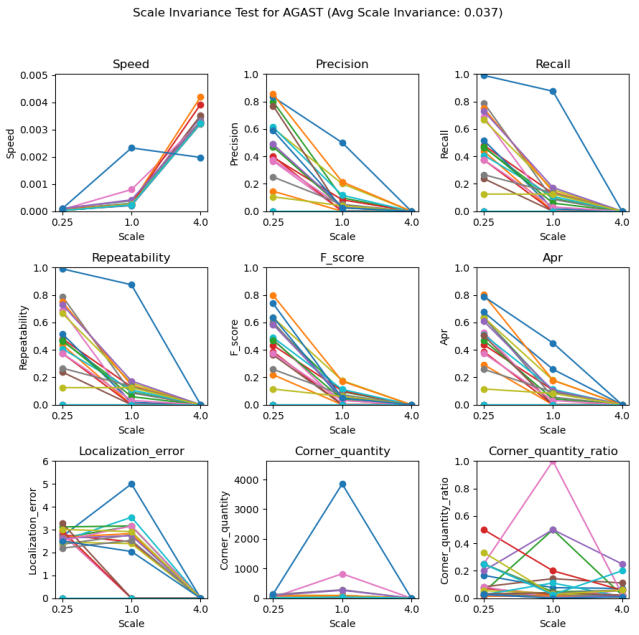


Fig. 37: Scale invariance results for AGAST, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

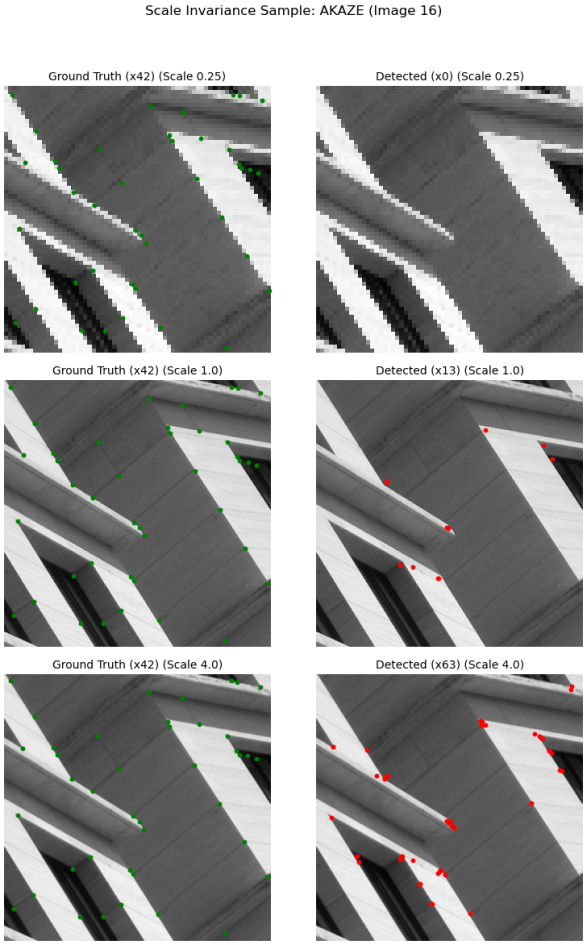


Fig. 38: Scale invariance sample for AKAZE on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

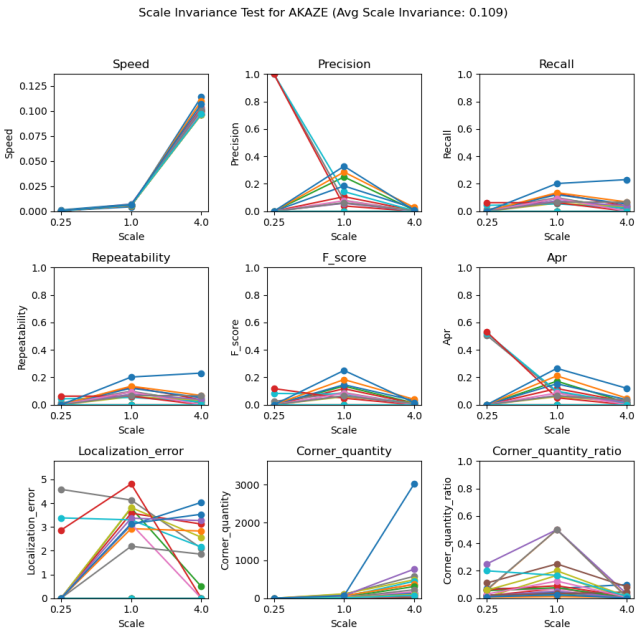


Fig. 39: Scale invariance results for AKAZE, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

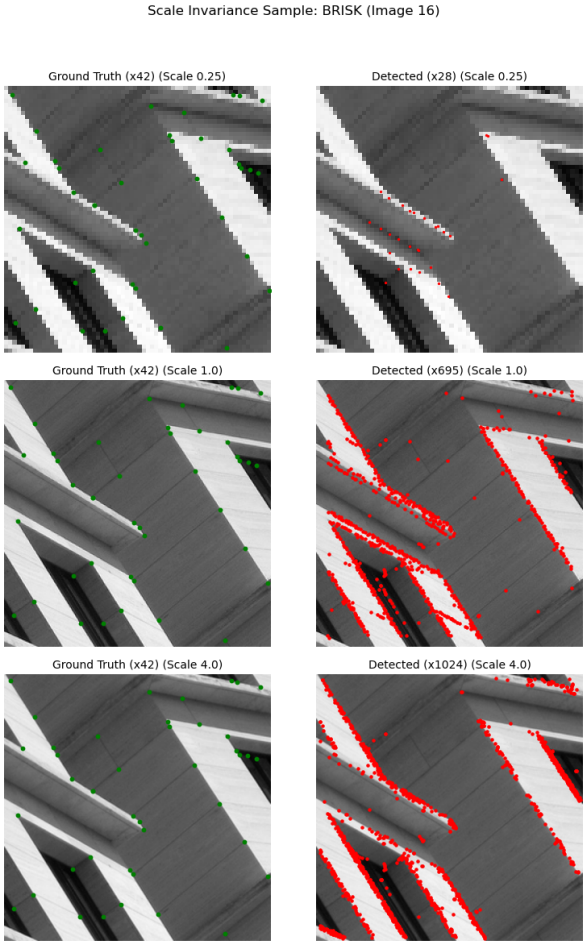


Fig. 40: Scale invariance sample for BRISK on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

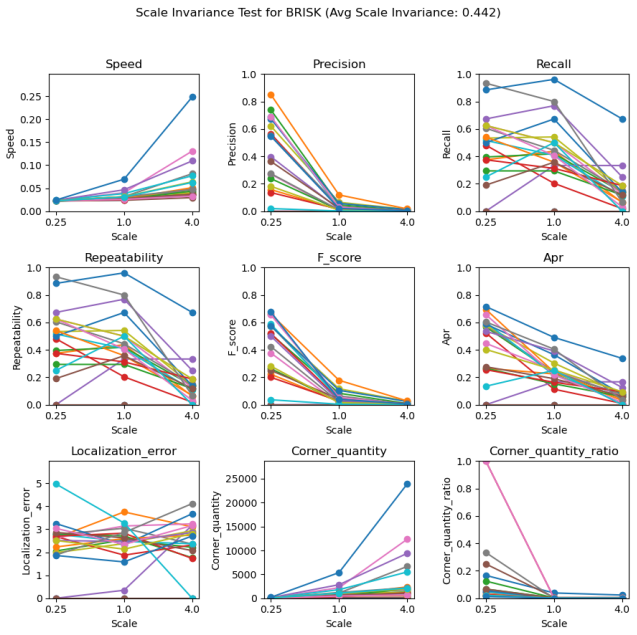


Fig. 41: Scale invariance results for BRISK, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

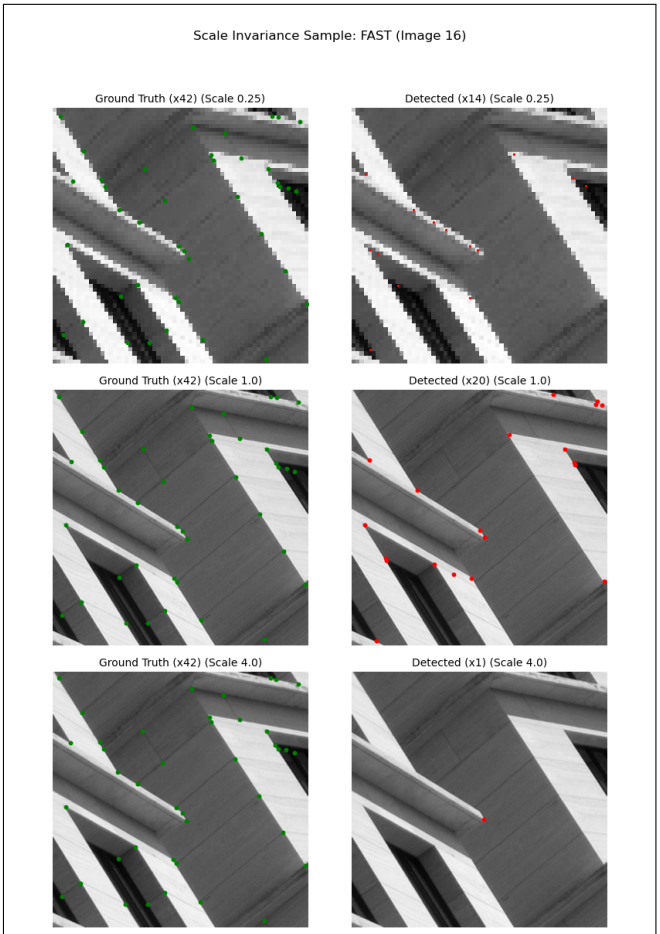


Fig. 42: Scale invariance sample for FAST on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

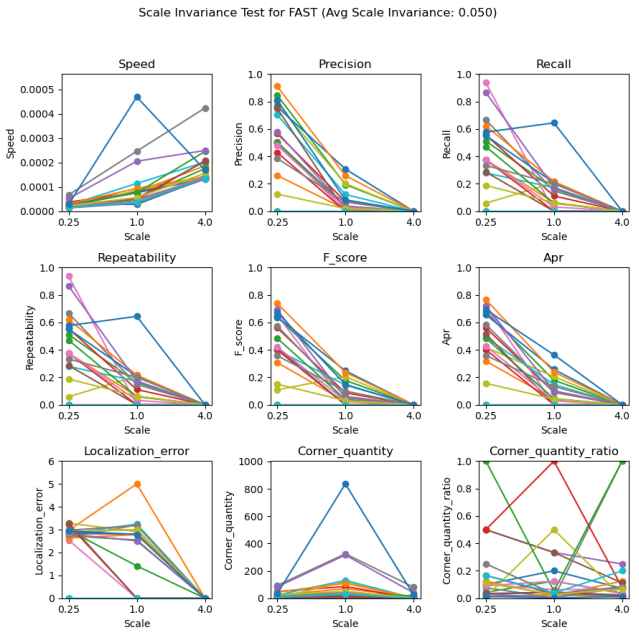


Fig. 43: Scale invariance results for FAST, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

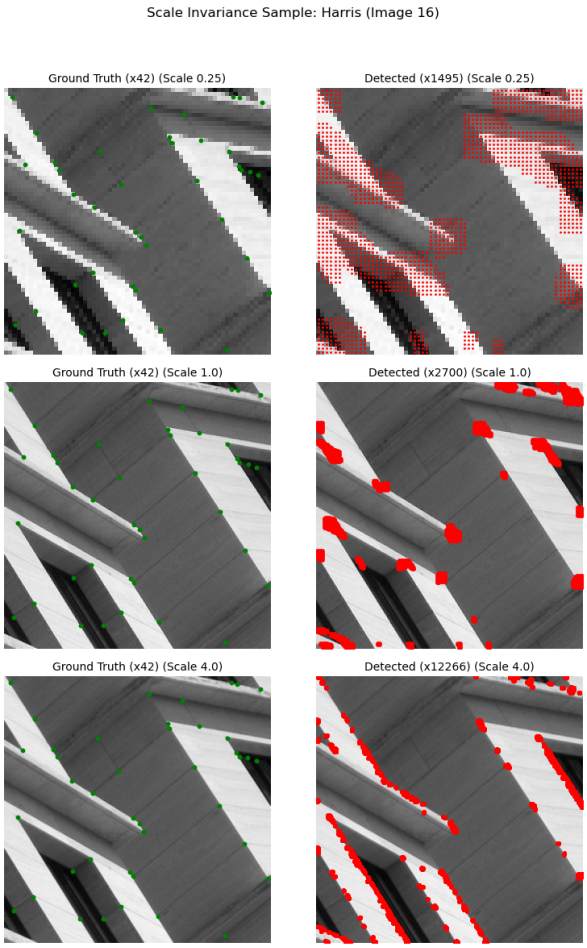


Fig. 44: Scale invariance sample for Harris on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

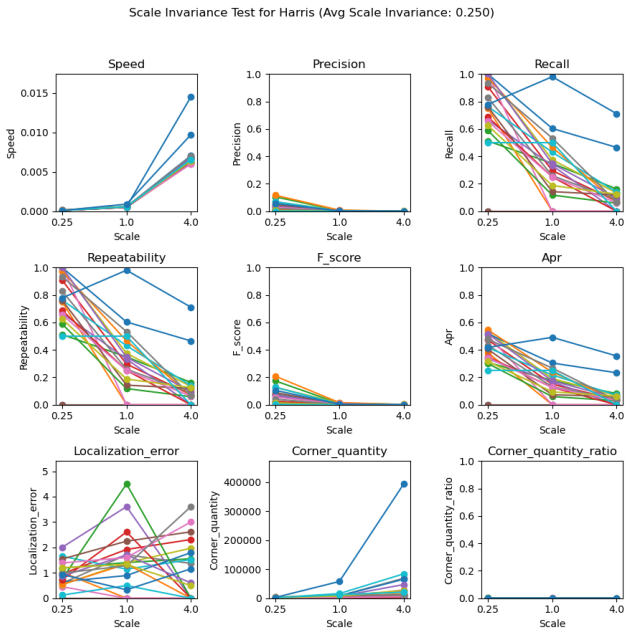


Fig. 45: Scale invariance results for Harris, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

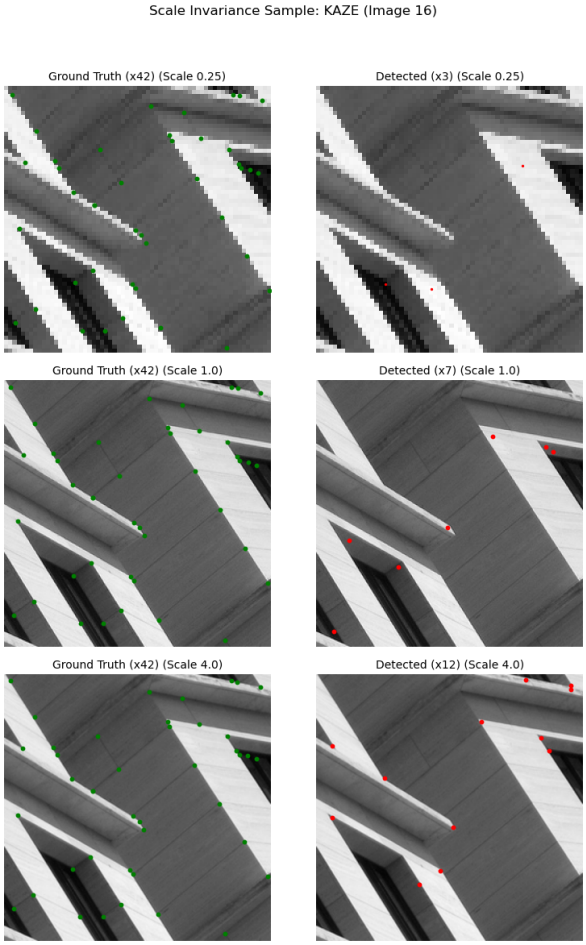


Fig. 46: Scale invariance sample for KAZE on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

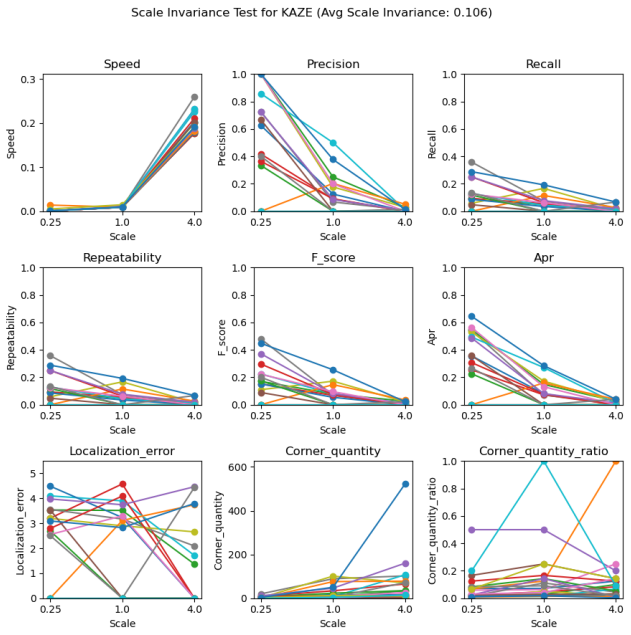


Fig. 47: Scale invariance results for KAZE, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

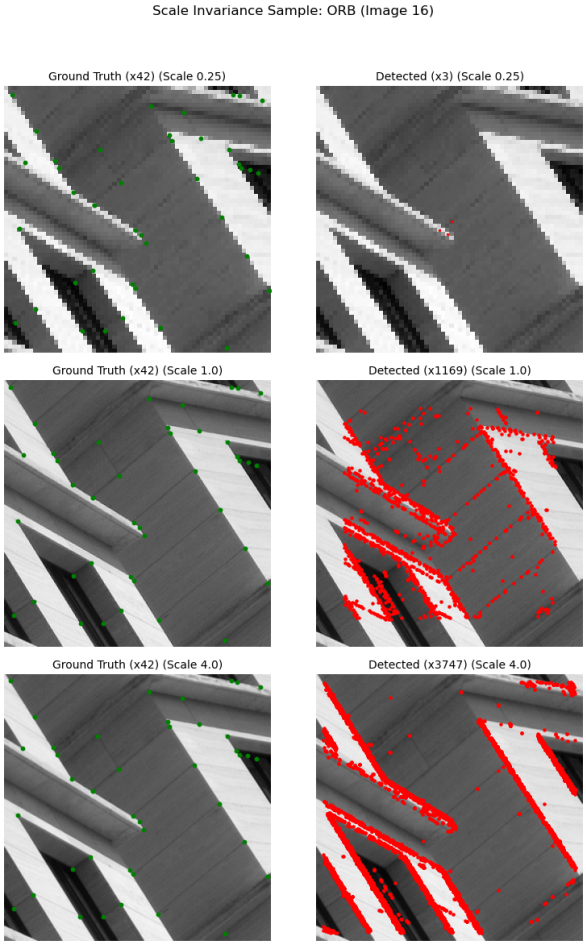


Fig. 48: Scale invariance sample for ORB on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

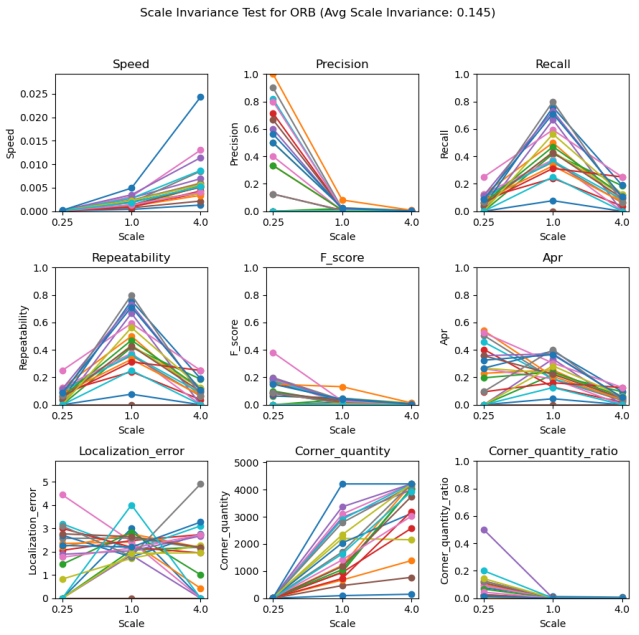


Fig. 49: Scale invariance results for ORB, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

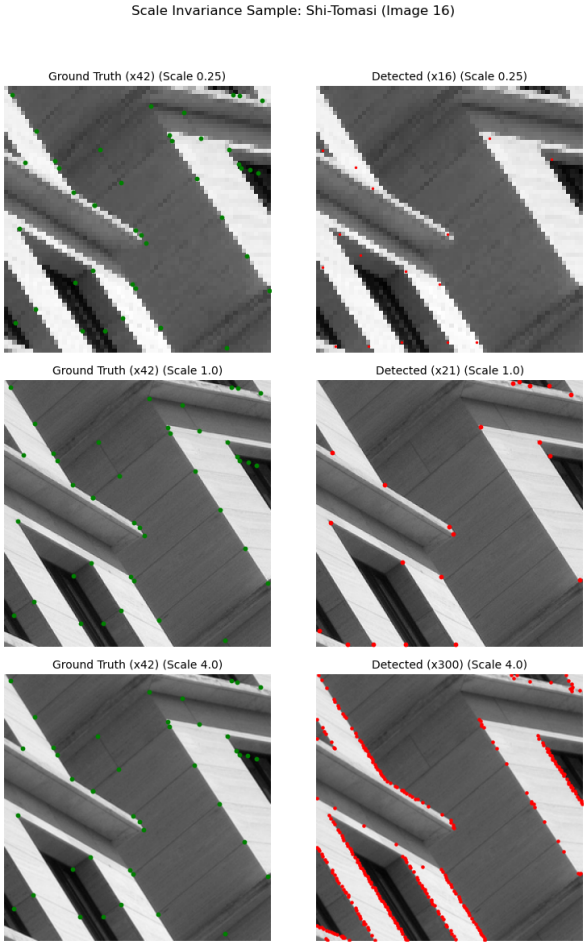


Fig. 50: Scale invariance sample for Shi-Tomasi on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

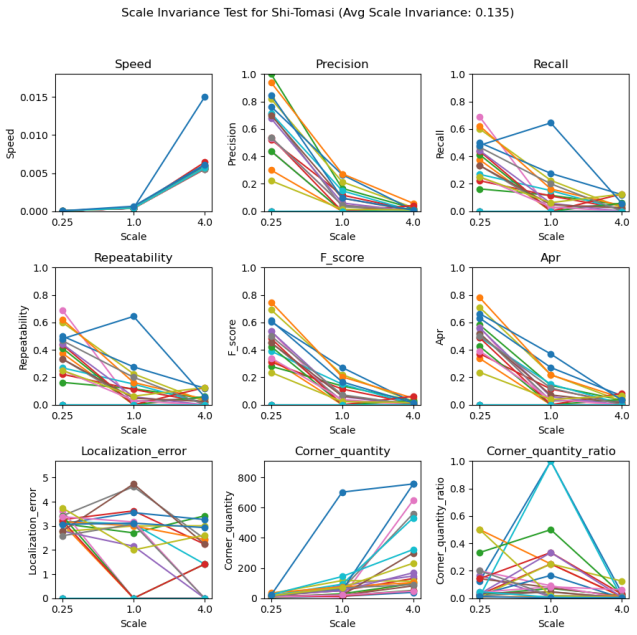


Fig. 51: Scale invariance results for Shi-Tomasi, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

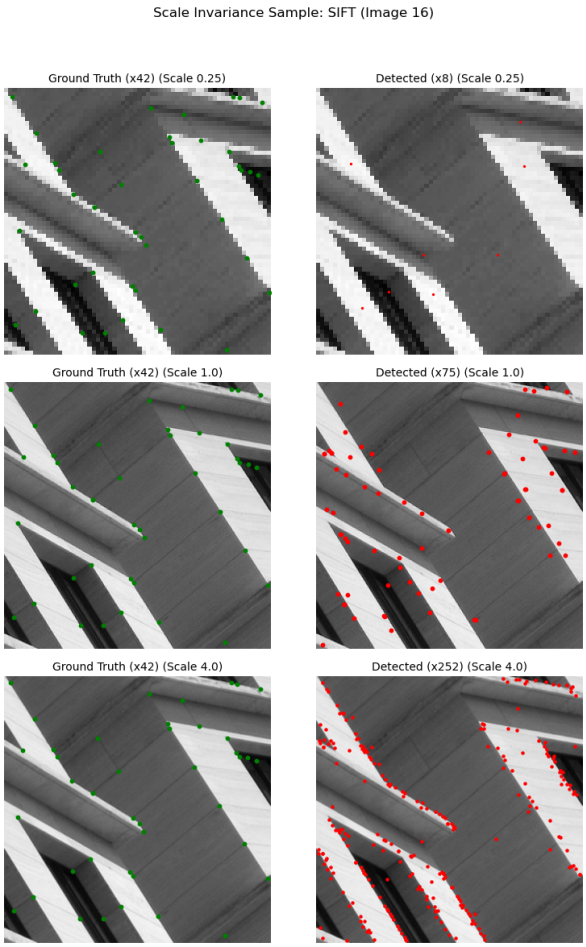


Fig. 52: Scale invariance sample for Sift on a randomly selected image, showing ground truth (green circles) and detected corners (red circles) at scales [0.25, 1.0, 4.0].

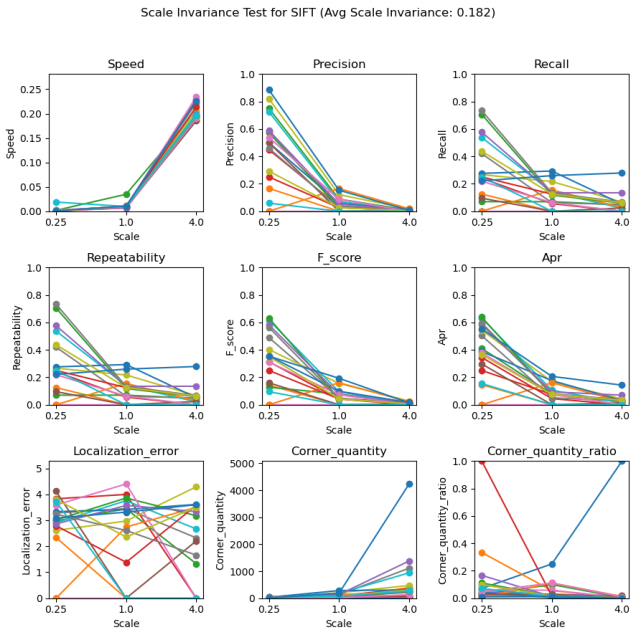


Fig. 53: Scale invariance results for SIFT, showing metrics across scales [0.25, 1.0, 4.0] for each image, with an average scale invariance score.

APPENDIX C
ALL DEVELOPED PYTHON CODE

```python
## import standard libraries
import numpy as np
import cv2
import os
import time
import itertools
from tqdm import tqdm
import re
import random

## import custom scripts
import corner_methods as cm
from parameters import ALGORITHMS, PARAM_GRIDS, image_path, ground_truth_path, output_path, SCALES, N_SAMPLES, MAX_SAMPLES, SEED
import utilities as utils
from distributions import sample_param_grid

def get_ground_truth(image_path, ground_truth_path, output_path, images=None, image_names=None, ground_truth_corners=None):

    images = images or []
    image_names = image_names or []
    ground_truth_corners = ground_truth_corners or []

    sorted_files = sorted(os.listdir(image_path), key=lambda x: int(re.search(r'\d+', x).group()))

    ## read in images and truth points
    for image in sorted_files:
        corners = f"{image.split('.')[0]}.txt"
        image_names.append(image.split('.')[0])
        images.append(cv2.imread(os.path.join(image_path, image), cv2.IMREAD_GRAYSCALE))

        with open(os.path.join(ground_truth_path, corners), 'r') as file:
            coordinate_pairs = np.array([tuple(map(int, line.split())) for line in file])
        ground_truth_corners.append(coordinate_pairs)


    if not os.path.isdir(output_path):
        os.makedirs(output_path)
    utils.plot_ground_truth(images, image_names, ground_truth_corners, output_path)

    return images, image_names, ground_truth_corners

def optimize_across_images(algorithm, alg_name, images, gt_corners_list, param_grid):

    best_params = {}
    best_score = -1

    param_names, param_values = sample_param_grid(param_grid, N_SAMPLES)
    total_combinations = np.prod([len(values) for values in param_values])

    # Limit to a maximum number of combinations
    param_combinations = list(itertools.product(*param_values))
    if MAX_SAMPLES:
        if len(param_combinations) > MAX_SAMPLES:
            param_combinations = random.sample(param_combinations, MAX_SAMPLES)
            total_combinations = MAX_SAMPLES

    # Store metrics for each parameter combination
    all_metrics = []

    # Use tqdm to show progress with total combinations
    for combo in tqdm(param_combinations, total=total_combinations, desc=f"Optimizing {total_combinations} Runs: {alg_name}"):
        params = dict(zip(param_names, combo))
        speeds = []
        precisions = []
        recalls = []
        repeatabilities = []
        f_scores = []
        aprs = []
        localization_errors = []
        corner_quantities = []
```

```
        corner_quantity_ratios = []

        for image, gt_corners in zip(images, gt_corners_list):
            start_time = time.time()
            corners = algorithm(image, args=params)
            exec_time = time.time() - start_time
            precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
                corner_quantity_ratio = utils.calculate_metrics(corners, gt_corners)

            speeds.append(exec_time)
            precisions.append(precision)
            recalls.append(recall)
            repeatabilities.append(repeatability)
            f_scores.append(f_score)
            aprs.append(apr)
            localization_errors.append(localization_error)
            corner_quantities.append(corner_quantity)
            corner_quantity_ratios.append(corner_quantity_ratio)

        # Normalize speed to [0, 1] for scoring (lower speed is better, so invert it)
        speed_norm = np.max(speeds) if speeds else 1.0
        if speed_norm > np.mean(speeds) + 4 * np.std(speeds):
            speed_norm = np.mean(speeds) + 4 * np.std(speeds)
        normalized_speeds = [1 - (speed / speed_norm) if speed_norm > 0 else 1.0 for speed in speeds]

        # Normalize localization error to [0, 1] for scoring (lower is better, so invert it)
        max_le = np.max(localization_errors) if localization_errors and np.max(localization_errors) > 0
            else 1.0
        normalized_le = [1 - (le / max_le) if max_le > 0 else 1.0 for le in localization_errors]

        # Compute average metrics
        avg_speed = np.mean(normalized_speeds)
        avg_precision = np.mean(precisions)
        avg_recall = np.mean(recalls)
        avg_repeatability = np.mean(repeatabilities)
        avg_f_score = np.mean(f_scores)
        avg_apr = np.mean(aprs)
        avg_le = np.mean(normalized_le)
        avg_corner_quant = np.mean(corner_quantity)
        avg_corner_quant_ratios = np.mean(corner_quantity_ratios)

        # Score is the average of the normalized metrics (excluding corner_quantity since it's not part
            of optimization)
        # weights = np.array([2/12, 3/12, 3/12, 1/12, 2/12, 1/12])
        vars = np.array([avg_speed, avg_precision, avg_recall, avg_le, avg_corner_quant_ratios]) #
            avg_repeatability
        # score = np.sum(vars * weights)
        score = np.sum(vars) / len(vars)

        # Store metrics for this combination
        all_metrics.append({
            'params': params,
            'speed': speeds,
            'precision': precisions,
            'recall': recalls,
            'repeatability': repeatabilities,
            'f_score': f_scores,
            'apr': aprs,
            'localization_error': localization_errors,
            'corner_quantity': corner_quantities,
            'corner_quantity_ratio': corner_quantity_ratios,
            'score': score
        })

        if score > best_score:
            best_score = score
            best_params = params

    return best_params, best_score, total_combinations, all_metrics

def test_scale_invariance(images, image_names, ground_truth_corners, optimized_params):

    scale_results = {name: {img_name: {
        'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
        'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': [], 'scale_invariance
            ': 0.0
    } for img_name in image_names} for name in ALGORITHMS}
```

```
    # Select a random image for sample visualization
    sample_idx = np.random.randint(0, len(images))
    sample_image = images[sample_idx]
    sample_name = image_names[sample_idx]
    sample_gt_corners = ground_truth_corners[sample_idx]

    for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Testing Scale Invariance"):
        best_params = optimized_params[alg_name]
        scaled_images = utils.create_scaled_images(sample_image)
        sample_corners = []

        for img_idx, (image, gt_corners, name) in enumerate(zip(images, ground_truth_corners,
            image_names)):
            scaled_images_current = utils.create_scaled_images(image)
            repeatabilities = []  # To compute scale invariance

            for scale_idx, scaled_img in enumerate(scaled_images_current):
                start_time = time.time()
                corners = alg_func(scaled_img, args=best_params)
                exec_time = time.time() - start_time

                # Adjust ground truth for scale
                scaled_gt = gt_corners * SCALES[scale_idx]

                precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
                    corner_quantity_ratio = utils.calculate_metrics(corners, scaled_gt)

                # Store results for this scale
                scale_results[alg_name][name]['speed'].append(exec_time)
                scale_results[alg_name][name]['precision'].append(precision)
                scale_results[alg_name][name]['recall'].append(recall)
                scale_results[alg_name][name]['repeatability'].append(repeatability)
                scale_results[alg_name][name]['f_score'].append(f_score)
                scale_results[alg_name][name]['apr'].append(apr)
                scale_results[alg_name][name]['localization_error'].append(localization_error)
                scale_results[alg_name][name]['corner_quantity'].append(corner_quantity)
                scale_results[alg_name][name]['corner_quantity_ratio'].append(corner_quantity_ratio)

                repeatabilities.append(repeatability)

                # Store corners for the sample image
                if img_idx == sample_idx:
                    sample_corners.append(corners)

            # Compute scale invariance as 1 - coefficient of variation of repeatability
            repeatabilities = np.array(repeatabilities)
            if repeatabilities.mean() > 0:
                cv = repeatabilities.std() / repeatabilities.mean()
                scale_invariance = max(0, min(1, 1 - cv))  # Higher value = better scale invariance
            else:
                scale_invariance = 0.0

            scale_results[alg_name][name]['scale_invariance'] = scale_invariance

        # Generate sample visualization for this algorithm
        utils.generate_scale_invariance_samples(
            scaled_images, sample_corners, sample_gt_corners,
            alg_name, sample_name, output_path
        )

    return scale_results

def run_benchmarking(images, image_names, ground_truth_corners):

    results = {name: {
        'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
        'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': []
    } for name in ALGORITHMS}
    individual_results = {name: {img_name: {
        'speed': [], 'precision': [], 'recall': [], 'repeatability': [], 'f_score': [], 'apr': [],
        'localization_error': [], 'corner_quantity': [], 'corner_quantity_ratio': []
    } for img_name in image_names} for name in ALGORITHMS}
    optimized_params = {}
    all_metrics_per_algorithm = {}  # Store all metrics for each parameter combination per algorithm

    # Optimize parameters and benchmark for each algorithm
```

```
    params = []
    combinations = []
    scores = []

    for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Generating Optimization Data"):
        # Optimize parameters across all images and store all metrics
        best_params, best_score, total_combinations, all_metrics = optimize_across_images(alg_func,
            alg_name, images, ground_truth_corners, PARAM_GRIDS[alg_name])
        params.append(best_params)
        combinations.append(total_combinations)
        scores.append(best_score)
        optimized_params[alg_name] = best_params
        all_metrics_per_algorithm[alg_name] = all_metrics

        # Benchmark with optimized parameters for this algorithm
        for img_idx, (image, gt_corners, name) in enumerate(zip(images, ground_truth_corners,
            image_names)):
            start_time = time.time()
            corners = alg_func(image, args=best_params)
            exec_time = time.time() - start_time

            precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
                corner_quantity_ratio = utils.calculate_metrics(corners, gt_corners)

            # Store aggregated results
            results[alg_name]['speed'].append(exec_time)
            results[alg_name]['precision'].append(precision)
            results[alg_name]['recall'].append(recall)
            results[alg_name]['repeatability'].append(repeatability)
            results[alg_name]['f_score'].append(f_score)
            results[alg_name]['apr'].append(apr)
            results[alg_name]['localization_error'].append(localization_error)
            results[alg_name]['corner_quantity'].append(corner_quantity)
            results[alg_name]['corner_quantity_ratio'].append(corner_quantity_ratio)

            # Store individual results
            individual_results[alg_name][name]['speed'].append(exec_time)
            individual_results[alg_name][name]['precision'].append(precision)
            individual_results[alg_name][name]['recall'].append(recall)
            individual_results[alg_name][name]['repeatability'].append(repeatability)
            individual_results[alg_name][name]['f_score'].append(f_score)
            individual_results[alg_name][name]['apr'].append(apr)
            individual_results[alg_name][name]['localization_error'].append(localization_error)
            individual_results[alg_name][name]['corner_quantity'].append(corner_quantity)
            individual_results[alg_name][name]['corner_quantity_ratio'].append(corner_quantity_ratio)

    return results, optimized_params, individual_results, params, combinations, scores,
        all_metrics_per_algorithm

if __name__ == "__main__":

    np.random.seed(SEED)

    print("Loading Ground Truth ...")
    images, image_names, ground_truth_corners = get_ground_truth(image_path, ground_truth_path,
        output_path)
    results, optimized_params, individual_results, params, combinations, scores,
        all_metrics_per_algorithm = run_benchmarking(images, image_names, ground_truth_corners)

    # Generate sample detection images with optimized parameters
    print("Creating Sample Imagery for Optimized Algorithms ...")
    utils.generate_sample_detections(images, image_names, ground_truth_corners, optimized_params,
        output_path)

    # Save results
    print("Writing Results Files ...")
    os.makedirs(output_path, exist_ok=True)
    os.makedirs(os.path.join(output_path, "data"), exist_ok=True)
    for alg_name in results:
        np.savez(os.path.join(output_path, f'data/{alg_name}_results.npz'), **results[alg_name])
        for img_name in image_names:
            np.savez(os.path.join(output_path, f'data/{alg_name}_{img_name}_individual_results.npz'),
                **individual_results[alg_name][img_name])

    # Generate comparison tables and reports
    utils.generate_param_report(params, combinations, scores, output_path)
    utils.generate_comparison_tables(results, output_path)
```

```
287
288     # Generate individual result plots and pairwise metric plots
289     print("Plotting Result Images ...")
290     utils.visualize_results(output_path, image_names)
291     # utils.plot_pairwise_metrics(results, output_path)
292     utils.plot_best_combination(individual_results, image_names, output_path)
293     utils.plot_all_combinations(all_metrics_per_algorithm, image_names, output_path)
294
295     # Run scale invariance test
296     scale_results = test_scale_invariance(images, image_names, ground_truth_corners, optimized_params)
297     utils.plot_scale_invariance(scale_results, image_names, output_path)
298     utils.generate_scale_invariance_table(scale_results, output_path)
299
300     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),
```

```
        'fastThreshold': UniformDist(min_val=5, max_val=100, is_int=True)
    },
    'SIFT': {
        'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
        'contrastThreshold': UniformDist(min_val=0.02, max_val=0.16),
        'edgeThreshold': UniformDist(min_val=5, max_val=30, is_int=True),
        'sigma': NormalDist(mean=1.6, std=0.4, min_val=1.0, max_val=2.4)
    },
    'BRISK': {
        'thresh': UniformDist(min_val=10, max_val=100, is_int=True),
        'octaves': UniformDist(min_val=2, max_val=6, is_int=True),
        'patternScale': UniformDist(min_val=0.5, max_val=2.0)
    },
    'AGAST': {
        'threshold': UniformDist(min_val=5, max_val=50, is_int=True),
        'type': CategoricalDist(options=[
            cv2.AgastFeatureDetector_AGAST_5_8,
            cv2.AgastFeatureDetector_AGAST_7_12d,
            cv2.AgastFeatureDetector_OAST_9_16
        ])
    },
    'KAZE': {
        'threshold': UniformDist(min_val=0.0005, max_val=0.004),
        'nOctaves': UniformDist(min_val=2, max_val=6, is_int=True),
        'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
        'diffusivity': CategoricalDist(options=[
            cv2.KAZE_DIFF_PM_G1, cv2.KAZE_DIFF_PM_G2, cv2.KAZE_DIFF_WEICKERT
        ])
    },
    'AKAZE': {
        'threshold': UniformDist(min_val=0.0005, max_val=0.004),
        'nOctaves': UniformDist(min_val=3, max_val=6, is_int=True),
        'nOctaveLayers': UniformDist(min_val=2, max_val=6, is_int=True),
        'diffusivity': CategoricalDist(options=[
            cv2.KAZE_DIFF_PM_G1, cv2.KAZE_DIFF_PM_G2
        ])
    }
}

PARAM_REPORT = [
    "\\begin{table}[h]",
    "\\centering",
    "\\small",
    "\\caption{Optimal␣Parameters␣for␣Corner␣Detection␣Algorithms}",
    "\\label{tab:optimal_parameters}",
    "\\begin{tabular}{lp{3.5cm}cc}",
    "\\toprule",
    "\\textbf{Algorithm}␣&␣\\textbf{Optimal␣Parameters}␣&␣\\textbf{$\\#$␣Tests}␣&␣\\textbf{Best␣Score}␣
        \\\\",
    "\\midrule"
]
```

Listing 6: Setup Parameters

```
import numpy as np

class UniformDist:
    def __init__(self, min_val, max_val, is_int=False):
        self.min_val = min_val
        self.max_val = max_val
        self.is_int = is_int

    def sample(self, n_samples):
        samples = np.round(np.random.uniform(self.min_val, self.max_val, n_samples), 4)
        if self.is_int:
            samples = np.round(samples).astype(int)
            samples = np.clip(samples, self.min_val, self.max_val)

        return np.unique(samples)[:n_samples]  # Ensure unique samples

class NormalDist:
    def __init__(self, mean, std, min_val=None, max_val=None, is_int=False):
        self.mean = mean
        self.std = std
```

```
        self.min_val = min_val
        self.max_val = max_val
        self.is_int = is_int

    def sample(self, n_samples):
        samples = np.random.normal(self.mean, self.std, n_samples)
        if self.min_val is not None or self.max_val is not None:
            samples = np.round(np.clip(samples, self.min_val or -np.inf, self.max_val or np.inf), 4)
        if self.is_int:
            samples = np.round(samples).astype(int)
            samples = np.clip(samples, self.min_val or -np.inf, self.max_val or np.inf)
        return np.unique(samples)[:n_samples]

class CategoricalDist:
    def __init__(self, options):
        self.options = options

    def sample(self, n_samples):
        if n_samples <= len(self.options):
            return np.random.choice(self.options, n_samples, replace=False)
        else:
            return np.random.choice(self.options, n_samples, replace=True)


def sample_param_grid(param_grid, n_samples):

    sampled_grid = {}
    for param_name, dist in param_grid.items():
        sampled_grid[param_name] = dist.sample(n_samples)
    return list(sampled_grid.keys()), list(sampled_grid.values())
```

Listing 7: Random Variable Distribution Setup

```
import cv2
import numpy as np

def harris(image, args=None):

    args = args or {'blockSize': 2, 'ksize': 3, 'k': 0.04}
    corners = cv2.cornerHarris(image, **args)
    corners = cv2.dilate(corners, None)
    corners = np.column_stack(np.where(corners > 0.01 * corners.max()))
    return corners[:, ::-1].astype(np.float32)

def shi_tomasi(image, args=None):

    args = args or {'maxCorners': 25, 'qualityLevel': 0.01, 'minDistance': 10}
    corners = cv2.goodFeaturesToTrack(image, **args)
    return corners[:, 0].astype(np.float32) if corners is not None else np.array([])

def fast(image, args=None):

    args = args or {'threshold': 25, 'type': cv2.FAST_FEATURE_DETECTOR_TYPE_7_12}
    fast = cv2.FastFeatureDetector_create(nonmaxSuppression=True, **args)
    keypoints = fast.detect(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def orb(image, args=None):

    args = args or {
        'nfeatures': 500, 'scaleFactor': 1.1, 'nlevels': 10,
        'edgeThreshold': 31, 'firstLevel': 0, 'WTA_K': 2,
        'scoreType': cv2.ORB_HARRIS_SCORE, 'patchSize': 31, 'fastThreshold': 20
    }
    orb = cv2.ORB_create(**args)
    keypoints = orb.detect(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def sift(image, args=None):

    args = args or {
        'nOctaveLayers': 3, 'contrastThreshold': 0.04, 'edgeThreshold': 10, 'sigma': 1.6
    }
    sift = cv2.SIFT_create(**args)
```

```
    keypoints, _ = sift.detectAndCompute(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def brisk(image, args=None):

    args = args or {'thresh': 30, 'octaves': 3, 'patternScale': 1.0}
    brisk = cv2.BRISK_create(**args)
    keypoints, _ = brisk.detectAndCompute(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def agast(image, args=None):

    args = args or {'threshold': 10, 'type': 1}
    agast = cv2.AgastFeatureDetector_create(nonmaxSuppression=True, **args)
    keypoints = agast.detect(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def kaze(image, args=None):

    args = args or {'threshold': 0.001, 'nOctaves': 3, 'nOctaveLayers': 3, 'diffusivity': 1}
    kaze = cv2.KAZE_create(**args)
    keypoints, _ = kaze.detectAndCompute(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)

def akaze(image, args=None):

    args = args or {'threshold': 0.001, 'nOctaves': 4, 'nOctaveLayers': 4, 'diffusivity': 2}
    akaze = cv2.AKAZE_create(**args)
    keypoints, _ = akaze.detectAndCompute(image, None)
    return np.array([kp.pt for kp in keypoints], dtype=np.float32)
```

Listing 8: All Corner Detection Wrappers

```
import numpy as np
import cv2
from itertools import product
import matplotlib.pyplot as plt
import os
from scipy.spatial.distance import cdist
from tqdm import tqdm
from parameters import PARAM_REPORT, ALGORITHMS, SCALES


def plot_ground_truth(images, image_names, ground_truth_corners, output_path):

    plt.figure(figsize=(8,10))
    plt.suptitle("Ground Truth Images")
    for i in range(len(images)):
        plt.subplot(6, 4, i+1)
        plt.imshow(images[i], cmap="gray")
        plt.axis("off")
        plt.title(f"Image {image_names[i]} (x{len(ground_truth_corners[i])})")
        for corner in ground_truth_corners[i]:
            plt.scatter(corner[1], corner[0], color="green", marker='o', s=10)

    plt.tight_layout()
    plt.savefig(os.path.join(output_path, "ground_truth.png"))
    plt.close()


def create_scaled_images(image):
    scaled_images = []
    for scale in SCALES:
        scaled = cv2.resize(image, None, fx=scale, fy=scale, interpolation=cv2.INTER_LINEAR)
        scaled_images.append(scaled)
    return scaled_images


def calculate_metrics(pred_corners, gt_corners, threshold=5.0):
    gt = np.array(gt_corners, dtype=np.float32)
    pred = np.array(pred_corners, dtype=np.float32)

    # Reshape if necessary
    if pred.ndim == 1 and pred.shape[0] == 2:
```

```
        pred = pred.reshape(1, 2)
    if pred.ndim == 1 or pred.size == 0:
        pred = np.empty((0, 2), dtype=np.float32)

    if gt.ndim == 1 and gt.shape[0] == 2:
        gt = gt.reshape(1, 2)
    if gt.ndim == 1 or gt.size == 0:
        gt = np.empty((0, 2), dtype=np.float32)

    # Overall corner quantity (total number of detected corners)
    corner_quantity = len(pred)
    corner_quantity_ratio = 1 / (np.abs(len(pred) - len(gt_corners)) + 1) ## 1/(|gt - pred| + 1)
    # Early return if either side has no corners
    if len(gt) == 0 or len(pred) == 0:
        return 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, corner_quantity, corner_quantity_ratio

    # Compute distances between ground truth and predicted corners
    dists = cdist(gt, pred)  # shape: (num_gt, num_pred)
    matched_gt = np.any(dists <= threshold, axis=1)
    matched_pred = np.any(dists <= threshold, axis=0)

    TP = np.sum(matched_gt)
    FP = len(pred) - np.sum(matched_pred)
    FN = len(gt) - TP

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    repeatability = TP / len(gt) if len(gt) > 0 else 0

    denom = (precision + recall)
    f_score = 2 * (precision * recall) / denom if denom > 0 else 0
    apr = (precision + recall) / 2  # Arithmetic Mean of Precision and Recall

    # Compute localization error for matched corners
    localization_error = 0.0
    if TP > 0:
        # Find the closest predicted corner for each ground truth corner
        min_dists = np.min(dists, axis=1)  # Minimum distance for each ground truth corner
        matched_dists = min_dists[matched_gt]  # Distances for matched ground truth corners
        localization_error = np.mean(matched_dists) if len(matched_dists) > 0 else 0.0

    return precision, recall, repeatability, f_score, apr, localization_error, corner_quantity,
        corner_quantity_ratio


def generate_sample_detections(images, image_names, ground_truth_corners, optimized_params, output_path
    ):

    sample_output_path = os.path.join(output_path, "sample_detections")
    os.makedirs(sample_output_path, exist_ok=True)

    idx = np.random.randint(0, len(images))
    image = images[idx]
    name = image_names[idx]

    for alg_name, alg_func in ALGORITHMS.items():
        params = optimized_params[alg_name]
        corners = alg_func(image, args=params)
        gt_corners = ground_truth_corners[idx]

        plt.figure(figsize=(4, 8))  # Adjusted for 1x2 grid
        plt.suptitle(f"Sample␣Detection:␣{alg_name}␣(Image␣{name})", fontsize=12)

        # Ground truth subplot (left column)
        plt.subplot(2, 1, 1)
        plt.imshow(image, cmap="gray")
        plt.title(f"Ground␣Truth␣(x{len(gt_corners)})", fontsize=10)
        plt.axis("off")

        # Plot ground truth corners
        for corner in gt_corners:
            plt.scatter(corner[1], corner[0], c='green', marker='o', s=10, label='Ground␣Truth')

        # Detected corners subplot (right column)
        plt.subplot(2, 1, 2)
        plt.imshow(image, cmap="gray")
        plt.title(f"Detected␣(x{len(corners)})", fontsize=10)
```

```
        plt.axis("off")

        # Plot detected corners
        size = 10
        if len(corners) > 100:
            size = 5
        for corner in corners:
            plt.scatter(corner[0], corner[1], c='red', marker='o', s=size, label='Detected')

        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.savefig(os.path.join(sample_output_path, f"{alg_name}_detected.png"))
        plt.close()


def plot_all_detections(images, image_names, optimized_params, output_path):
    sample_output_path = os.path.join(output_path, "sample_detections")
    os.makedirs(sample_output_path, exist_ok=True)

    for alg_name, alg_func in tqdm(ALGORITHMS.items(), desc="Plotting All Optimized Detections"):
        params = optimized_params.get(alg_name, {})
        plt.figure(figsize=(8, 10))
        plt.suptitle(f"Detected Corners: {alg_name}", fontsize=12)

        for i in range(len(images)):
            try:
                corners = alg_func(images[i], args=params)
                if corners is None or len(corners) == 0:
                    corners = np.array([])
            except Exception as e:
                print(f"Error processing {alg_name} for image {image_names[i]}: {e}")
                corners = np.array([])

            plt.subplot(6, 4, i + 1)
            plt.imshow(images[i], cmap="gray")
            plt.axis("off")
            plt.title(f"Image {image_names[i]} (x{len(corners)})")
            size = 10
            if len(corners) > 100:
                size = 5
            for corner in corners:
                plt.scatter(corner[0], corner[1], color="red", marker='o', s=size)

        plt.tight_layout()
        plt.savefig(os.path.join(sample_output_path, f"{alg_name}_all_detections.png"))
        plt.close()


def generate_scale_invariance_samples(scaled_images, detected_corners, gt_corners, alg_name, image_name
    , output_path):

    sample_output_path = os.path.join(output_path, "scale_invariance_samples")
    os.makedirs(sample_output_path, exist_ok=True)

    plt.figure(figsize=(8, 12))  # Adjusted for 3x2 grid
    plt.suptitle(f"Scale Invariance Sample: {alg_name} (Image {image_name})", fontsize=12)

    for i, (scale, image, corners) in enumerate(zip(SCALES, scaled_images, detected_corners)):
        # Ground truth subplot (left column, i.e., subplot 1, 3, 5)
        plt.subplot(3, 2, 2 * i + 1)
        plt.imshow(image, cmap="gray")
        plt.title(f"Ground Truth (x{len(gt_corners)}) (Scale {scale})", fontsize=10)
        plt.axis("off")

        # Plot ground truth corners (adjusted for scale)
        scaled_gt = gt_corners * scale
        for corner in scaled_gt:
            plt.scatter(corner[1], corner[0], c='green', marker='o', s=10, label='Ground Truth' if i ==
                0 else "")

        # Detected corners subplot (right column, i.e., subplot 2, 4, 6)
        plt.subplot(3, 2, 2 * i + 2)
        plt.imshow(image, cmap="gray")
        plt.title(f"Detected (x{len(corners)}) (Scale {scale})", fontsize=10)
        plt.axis("off")

        size = 10
        if len(corners) > 100:
```

```
            size = 5

        if scale < 1:
            size = int(size * scale)
        # Plot detected corners
        for corner in corners:
            plt.scatter(corner[0], corner[1], c='red', marker='o', s=size, label='Detected' if i == 0
                else "")


    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.savefig(os.path.join(sample_output_path, f"{alg_name}_scale_invariance_sample.png"))
    plt.close()


def generate_comparison_tables(results, output_path):
    metrics = ['precision', 'recall', 'repeatability', 'speed', 'f_score', 'apr', 'localization_error',
         'corner_quantity_ratio']
    table_content = [
        "\\begin{table}[h]",
        "\\centering",
        "\\caption{Comparison of Corner Detection Algorithms}",
        "\\label{tab:comparison_table}",
        "\\begin{tabular}{lccccccc}",
        "\\toprule",
        "\\textbf{Algorithm} & \\textbf{Precision} & \\textbf{Recall} & \\textbf{Repeatability} & \\
            textbf{Speed (s)} & \\textbf{F Score} & \\textbf{APR} & \\textbf{Localization Error} & \\
            textbf{Corner Quantity Ratio} \\\\",
        "\\midrule"
    ]

    for alg_name in results:
        row = f"{alg_name} "
        for metric in metrics:
            data = results[alg_name][metric]
            mean_value = np.mean(data) if data else 0.0
            if metric == 'speed':
                row += f"& {mean_value:.4f} "
            elif metric == 'corner_quantity':
                row += f"& {int(mean_value)} "
            else:
                row += f"& {mean_value:.3f} "
        row += "\\\\"
        table_content.append(row)

    table_content.extend([
        "\\bottomrule",
        "\\end{tabular}",
        "\\end{table}"
    ])

    table_path = os.path.join(output_path, "comparison_table.txt")
    with open(table_path, 'w') as f:
        f.write("\n".join(table_content))

def visualize_results(output_path, image_names):
    plt.figure(figsize=(9, 9))

    x_ticks = []
    for idx, metric in enumerate(['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', '
        localization_error', 'corner_quantity', 'corner_quantity_ratio']):
        plt.subplot(3, 3, idx + 1)
        for alg_name in ALGORITHMS:
            data = np.load(os.path.join(output_path, f'data/{alg_name}_results.npz'))[metric]
            plt.plot(data, label=alg_name)
        plt.title(metric.capitalize())
        plt.legend(loc="upper right", fontsize="xx-small")
        plt.xlabel('Image Number')

    plt.tight_layout()
    plt.savefig(os.path.join(output_path, 'benchmark_results.png'))
    plt.close()

def plot_best_combination(individual_results, image_names, output_path):
    """Plot the metrics of the best parameter combination for each algorithm across images."""
    plot_output_path = os.path.join(output_path, "best_combination_plots")
    os.makedirs(plot_output_path, exist_ok=True)
```

```
    metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
         'corner_quantity', 'corner_quantity_ratio']

    for alg_name in individual_results:
        plt.figure(figsize=(9, 9))
        plt.suptitle(f"Best Combination Metrics for {alg_name}")

        for idx, metric in enumerate(metrics):
            plt.subplot(3, 3, idx + 1)
            # Gather metric data across all images
            metric_data = [individual_results[alg_name][img_name][metric][0] for img_name in
                image_names]
            plt.plot(image_names, metric_data, marker='o', color='red', label=alg_name)
            plt.title(metric.capitalize())
            plt.xlabel('Image Number')
            plt.ylabel(metric.capitalize())

            if metric not in ['speed', 'localization_error', 'corner_quantity']:
                plt.ylim(0, 1)
            else:
                plt.ylim(0, np.max(metric_data) * 1.2 if np.max(metric_data) > 0 else 1)
            plt.legend(loc="upper right")

        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.savefig(os.path.join(plot_output_path, f"{alg_name}_best_combination.png"))
        plt.close()


def plot_all_combinations(all_metrics_per_algorithm, image_names, output_path):
    plot_output_path = os.path.join(output_path, "all_combinations_plots")
    os.makedirs(plot_output_path, exist_ok=True)

    metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
         'corner_quantity', 'corner_quantity_ratio']

    for alg_name in all_metrics_per_algorithm:
        all_metrics = all_metrics_per_algorithm[alg_name]
        # Find the best combination (highest score)
        best_idx = np.argmax([m['score'] for m in all_metrics])

        plt.figure(figsize=(9, 9))
        plt.suptitle(f"All Parameter Combinations for {alg_name}")

        for idx, metric in enumerate(metrics):
            plt.subplot(3, 3, idx + 1)

            # Plot all combinations with transparency
            for combo_idx, metrics_dict in enumerate(all_metrics):
                metric_data = metrics_dict[metric]
                if combo_idx == best_idx:
                    # Highlight the best combination with a bold line
                    plt.plot(image_names, metric_data, color='red', linewidth=3, label='Best
                        Combination', markersize=3)
                else:
                    # Plot other combinations with transparency
                    plt.plot(image_names, metric_data, alpha=0.15, color='green', markersize=1,
                        linewidth=0.75, label='Other Combinations' if combo_idx == 0 else "")

            plt.title(metric.capitalize())
            plt.xlabel('Image Number')
            plt.ylabel(metric.capitalize())

            if metric not in ['speed', 'localization_error', 'corner_quantity']:
                plt.ylim(0, 1)
            else:
                all_values = [np.max(metrics_dict[metric]) for metrics_dict in all_metrics]
                plt.ylim(0, np.max(all_values) * 1.2 if np.max(all_values) > 0 else 1)
            plt.legend(loc="upper right")

        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.savefig(os.path.join(plot_output_path, f"{alg_name}_all_combinations.png"))
        plt.close()


def plot_scale_invariance(scale_results, image_names, output_path):
    plot_output_path = os.path.join(output_path, "scale_invariance_plots")
```

```
    os.makedirs(plot_output_path, exist_ok=True)

    metrics = ['speed', 'precision', 'recall', 'repeatability', 'f_score', 'apr', 'localization_error',
         'corner_quantity', 'corner_quantity_ratio']
    scale_labels = [str(val) for val in SCALES]

    for alg_name in scale_results:
        plt.figure(figsize=(9, 9))
        # Compute average scale invariance score across all images
        avg_scale_invariance = np.mean([scale_results[alg_name][img_name]['scale_invariance'] for
            img_name in image_names])
        plt.suptitle(f"Scale Invariance Test for {alg_name} (Avg Scale Invariance: {
            avg_scale_invariance:.3f})")

        for idx, metric in enumerate(metrics):
            plt.subplot(3, 3, idx + 1)

            # Plot a line for each image across scales
            for img_idx, img_name in enumerate(image_names):
                metric_data = scale_results[alg_name][img_name][metric]
                plt.plot(scale_labels, metric_data, marker='o', label=f"Image {img_name}")

            plt.title(metric.capitalize())
            plt.xlabel('Scale')
            plt.ylabel(metric.capitalize())
            if metric not in ['speed', 'localization_error', 'corner_quantity']:
                plt.ylim(0, 1)
            else:
                all_values = [np.max(scale_results[alg_name][img_name][metric]) for img_name in
                    image_names]
                plt.ylim(0, np.max(all_values) * 1.2 if np.max(all_values) > 0 else 1)
            # plt.legend(loc="upper right")

        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.savefig(os.path.join(plot_output_path, f"{alg_name}_scale_invariance.png"))
        plt.close()


def generate_param_report(params, combinations, scores, output_path):
    param_report = PARAM_REPORT
    for i, (alg_name, alg_func) in enumerate(ALGORITHMS.items()):
        best_params = params[i]
        total_combinations = combinations[i]
        best_score = scores[i]

        param_str = "\\\\".join([f"{key.replace('_','\\_')}: {value}" for key, value in best_params.
            items()])
        param_str2 = f"\\multirow{{" + f"{len(best_params.items())}" + f"}}{{*}}{{{alg_name}}} &\\
            multirow{{" + f"{len(best_params.items())}" + f"}}{{*}}{{\\parbox{{3.25cm}}{{\\raggedright
            {param_str}}}}} &\\multirow{{" + f"{len(best_params.items())}" + f"}}{{*}}{{{
            total_combinations}}} &\\multirow{{" + f"{len(best_params.items())}" + f"}}{{*}}{{{np.
            round(best_score, 4)}}} \\\\"
        param_report.append(param_str2)
        param_report.append("& & \\\\")
        param_report.append("& & \\\\")
        param_report.append("& & \\\\")

        if len(best_params.items()) > 2:
            for i in range(len(best_params.items())-2):
                param_report.append("& & \\\\")

    param_report.extend([
        "\\bottomrule",
        "\\end{tabular}",
        "\\end{table}"
    ])
    with open(os.path.join(output_path, "optimal_parameters.txt"), 'w') as f:
        f.write("\n".join(param_report))

def generate_scale_invariance_table(scale_results, output_path):
    table_output_path = os.path.join(output_path, "scale_invariance_table.txt")
    os.makedirs(output_path, exist_ok=True)

    # Initialize table content
    table_content = [
        r"\begin{table}[t]",
        r"\centering",
```

```
        r"\small",
        r"\caption{Mean␣Scale␣Invariance␣Scores␣for␣Corner␣Detection␣Algorithms}",
        r"\label{tab:scale_invariance_scores}",
        r"\begin{tabular}{lc}",
        r"\toprule",
        r"Algorithm␣&␣Scale␣Invariance␣Score␣\\",
        r"\midrule",
    ]

    # Compute mean scale invariance score for each algorithm
    for alg_name in sorted(scale_results.keys()):
        scores = [scale_results[alg_name][img_name]['scale_invariance']
                  for img_name in scale_results[alg_name]]
        mean_score = np.mean(scores)
        # Format algorithm name (e.g., convert 'ShiTomasi' to 'Shi-Tomasi')
        formatted_name = alg_name.replace('ShiTomasi', 'Shi-Tomasi')
        table_content.append(f"{formatted_name}␣&␣{mean_score:.3f}␣\\\\")

    # Close table
    table_content.extend([
        r"\bottomrule",
        r"\end{tabular}",
        r"\end{table}"
    ])

    # Write table to file
    with open(table_output_path, 'w') as f:
        f.write('\n'.join(table_content))
```

Listing 9: Utility Functions for Data Processing