CAMERA BASED 2D FEATURE TRACKING SENSOR FUSION PROJECT #2 (MIDTERM)

Jasmine L. Taketa-Tran

OVERVIEW

The objective of this project was to evaluate and compare a variety of keypoint detectors and descriptors on a constrained set of KITTI video data frames. We also implemented keypoint tracking methods (matching & selection) and assessed quality of matches and processing time across combinations of detectors and descriptors.

MP.1 DATA BUFFER OPTIMIZATION

> *Implement a vector for dataBuffer objects whose size does not exceed a limit (e.g. 2 elements). This can be achieved by pushing in new elements on one end and removing elements on the other end*

For this task I leveraged the boost library. Here is a condensed summary of what I did in the project build – please refer to the actual build for the full implementation:

```
#include <vector>
#include "dataStructures.h"
#include <boost/circular_buffer.hpp>

int main(int argc, const char *argv[])
{
    int imgStartIndex = 0; // first file index to load
    int imgEndIndex = 9;   // last file index to load

    // create ring buffer for video frames
    int dataBufferSize = 2; // number of images that will be held in memory
    boost::circular_buffer<DataFrame> dataBuffer;
    dataBuffer.set_capacity(dataBufferSize);

    for (size_t imgIndex = 0; imgIndex <= imgEndIndex - imgStartIndex; imgIndex++)
    {
        // load image from file and convert to grayscale
        cv::Mat img, imgGray;
        img = cv::imread(imgFullFilename);
        cv::cvtColor(img, imgGray, cv::COLOR_BGR2GRAY);

        // push images into ring buffer of size dataBufferSize = 2
        DataFrame frame;
        frame.cameraImg = imgGray;
        dataBuffer.push_back(frame); // push image into data frame buffer
    }
}
```

MP.2 KEYPOINT DETECTION

> *Implement detectors HARRIS, FAST, BRISK, ORB, AKAZE and SIFT. Make them selectable by setting a string accordingly.*

Detector options were defined in the main() function. The current implementation loops over all detectors for our trade study.

```
int main(int argc, const char *argv[])
{
    const int NUM_DET_TYPES = 7;
    const char *DetectorTypes[NUM_DET_TYPES] =
    {"SIFT", "HARRIS", "FAST", "BRISK", "ORB", "AKAZE", "SHITOMASI"};

    for (int detIdx = 0; detIdx < NUM_DET_TYPES; detIdx++)
    {
        string detectorType = DetectorTypes[detIdx];

        for (size_t imgIndex = 0; imgIndex <= 9; imgIndex++)
        {
            // load image from file and convert to grayscale
            cv::Mat img, imgGray;
            img = cv::imread(imgFullFilename);
            cv::cvtColor(img, imgGray, cv::COLOR_BGR2GRAY);

            // push images into ring buffer
            DataFrame frame; frame.cameraImg = imgGray;
            dataBuffer.push_back(frame);

            // detect image keypoints
            vector<cv::KeyPoint> keypoints;
            detKeypoints(keypoints, imgGray, detectorType);
        }
    }
    return 0;
}
```

For HARRIS and FAST, I used the implementation from the classroom exercises. For BRISK, ORB, AKAZE and SIFT, I used OpenCV library classes with default hyperparameter settings and did not attempt to perform any tuning.

This is the detKeypoints function with Harris Corner Detection option:

```
void detKeypoints(std::vector<cv::KeyPoint> &keypoints, cv::Mat &img, std::string detectorType)
{
    if (detectorType.compare("HARRIS") == 0)
    {
        // Detector parameters
        int blockSize = 2;     // blockSize × blockSize pixel neighborhood
        int apertureSize = 3;  // aperture parameter for Sobel operator (must be odd)
        int minResponse = 100; // min value for a corner in the 8bit scaled response matrix
        double k = 0.04;       // Harris parameter (see equation for details)

        // Detect Harris corners and normalize output
        cv::Mat dst, dst_norm, dst_norm_scaled;
        dst = cv::Mat::zeros(img.size(), CV_32FC1);

        // run the Harris edge detector on the image
        cv::cornerHarris(img, dst, blockSize, apertureSize, k, cv::BORDER_DEFAULT);
        cv::normalize(dst, dst_norm, 0, 255, cv::NORM_MINMAX, CV_32FC1, cv::Mat());
        cv::convertScaleAbs(dst_norm, dst_norm_scaled);

        // Look for prominent corners
        double maxOverlap = 0.0; // max permissible overlap between two features in %
        for (size_t j = 0; j < dst_norm.rows; j++)
        {
            for (size_t i = 0; i < dst_norm.cols; i++)
            {
                int response = (int)dst_norm.at<float>(j, i);
                if (response > minResponse) // only store points above a threshold
                {
                    cv::KeyPoint newKeyPoint;
                    newKeyPoint.pt = cv::Point2f(i, j);
                    newKeyPoint.size = 2 * apertureSize;
                    newKeyPoint.response = response;

                    // perform non-maximum suppression in local neighborhood of kpt
                    bool bOverlap = false;
                    for (auto it = keypoints.begin(); it != keypoints.end(); ++it)
                    {
                        double kptOverlap = cv::KeyPoint::overlap(newKeyPoint, *it);
                        if (kptOverlap > maxOverlap)
                        {
                            bOverlap = true;
                            if (newKeyPoint.response > (*it).response)
                            { // if overlap is >t AND response is higher for new kpt
                                *it = newKeyPoint; // replace old key point with new one
                                break;             // quit loop over keypoints
                            }
                        }
                    }
                    // add new key point if no overlap has been found in previous NMS
                    if (!bOverlap)
                        keypoints.push_back(newKeyPoint);

                } // end if points above threshold
            } // end loop over cols
        } // end loop over rows
    }
```

The Shi-Tomasi option, as provided in the build for us:

```
elseif (detectorType.compare("SHITOMASI") == 0) // Shi-Tomasi detector
{
    // compute detector parameters based on image size
    int blockSize = 4; // for derivative covariance over each pixel neighborhood
    double maxOverlap = 0.0; // max overlap between two features in [%]
    double minDistance = (1.0 - maxOverlap) * blockSize;
    int maxCorners = img.rows * img.cols / max(1.0, minDistance); // max # of keypoints
    double qualityLevel = 0.01; // minimal accepted quality of image corners
    double k = 0.04;
    vector<cv::Point2f> corners;

    // Apply corner detection
    cv::goodFeaturesToTrack(img, corners, maxCorners, qualityLevel, minDistance,
                            cv::Mat(), blockSize, false, k);

    // Add corners to result vector
    for (auto it = corners.begin(); it != corners.end(); ++it)
    {
        cv::KeyPoint newKeyPoint;
        newKeyPoint.pt = cv::Point2f((*it).x, (*it).y);
        newKeyPoint.size = blockSize;
        keypoints.push_back(newKeyPoint);
    }
}
```

And the remaining options: FAST, BRISK, ORB, AKAZE and SIFT:

```
    else // FAST, BRISK, ORB, AKAZE, SIFT --> Use OpenCV Library
    {
        cv::Ptr<cv::FeatureDetector> detector;
        double initTime;
        if (detectorType.compare("FAST") == 0)
        {
            int threshold = 60; // diff between intensity of the central pixel and pixels of a ci
            bool bNMS = true;  // perform non-maxima suppression on keypoints
            cv::FastFeatureDetector::DetectorType type = cv::FastFeatureDetector::TYPE_7_12; // T
            detector = cv::FastFeatureDetector::create(threshold, bNMS, type);
        }
        else if (detectorType.compare("BRISK") == 0)
        {
            detector = cv::BRISK::create();
        }
        else if (detectorType.compare("ORB") == 0)
        {
            detector = cv::ORB::create();
        }
        else if (detectorType.compare("AKAZE") == 0)
        {
            detector = cv::AKAZE::create();
        }
        else if (detectorType.compare("SIFT") == 0)
        {
            detector = cv::xfeatures2d::SIFT::create();
        }
        else
        {
            std::cerr << "ERROR: Detector Type " << detectorType << "Not Recognized!" << std::endl;
            return;
        }

        // detect keypoints
        detector->detect(img, keypoints);
    }
}
```

MP.3 KEYPOINT REMOVAL

> *Remove all keypoints outside of a pre-defined rectangle. Only use the keypoints within the rectangle for further processing.*

I implemented this in 2 ways: (1) generating keypoints on the entire image and then excluding keypoints not contained within the pre-defined rectangle, and (2) actually cropping the image before keypoint detection in order to support easier viewing of the matched keypoints when using drawMatches. The latter approach included a buffer around the clipped image border to account for the various filter sizes applied during description. This produced slight differences in the keypoints that were detected, as well as timing differences.

```
// cropping options
bool cropImage = false; // crop image to preceding vehicle (plus a frame buffer)
bool bFocusOnVehicle = true; // discard any keypoints outside of vehicle bounding box
```

Approach 1: First generate keypoints on the entire image; then exclude keypoints outside of a bounding box on the preceding vehicle. This approach was used to compare detector methods.

```
// extract 2D keypoints from current image
vector<cv::KeyPoint> keypoints;
detKeypoints(keypoints, imgGray, detectorType, writeScoring, scoringFileName);

// discard any keypoints outside of vehicle bounding box
cv::Rect vehicleRect(535, 180, 180, 150);
if (bFocusOnVehicle && !cropImage)
{
    vector<cv::KeyPoint> vehicleKeypoints;
    for (auto kp : keypoints)
    {
        if (vehicleRect.contains(kp.pt))
            vehicleKeypoints.push_back(kp);
    }
    keypoints = vehicleKeypoints;
}
```

Approach 2: First crop the image, then detect, describe and match keypoints on the cropped image. I used this approach primarily to make it easier to visualize keypoint matches across cookie cuts of the preceding vehicle. For consistency with these visualizations, evaluation of descriptor, matcher and selector performance are conducted on the cropped images.

```
// load image from file and convert to grayscale
cv::Mat img, imgGray;
img = cv::imread(imgFullFilename);
cv::cvtColor(img, imgGray, cv::COLOR_BGR2GRAY);

// crop image to preceding vehicle (plus a frame buffer)
if (cropImage)
{
    cv::Rect roi(510, 165, 235, 200); // top left x, top left y, width, height
    imgGray = imgGray(roi);
}

// push images into ring buffer of size dataBufferSize = 2
DataFrame frame;
frame.cameraImg = imgGray;
dataBuffer.push_back(frame); // push image into data frame buffer
```

MP.4 KEYPOINT DESCRIPTORS

> *Implement descriptors BRIEF, ORB, FREAK, AKAZE and SIFT. Make them selectable by setting a string accordingly.*

This is the main() function where descriptors are made selectable by string name, and descKeypoints is called on a frame-by-frame basis (after image loading and keypoint detection):

```
int main(int argc, const char *argv[])
{
    const int NUM_DESC_TYPES = 6;
    const char *DescriptorTypes[NUM_DESC_TYPES] =
        {"SIFT", "AKAZE", "BRIEF", "ORB", "FREAK", "BRISK"};

    for (int descIdx = 0; descIdx < NUM_DESC_TYPES; descIdx++)
    {
        string descriptorType = DescriptorTypes[descIdx];

        for (size_t imgIndex = 0; imgIndex <= 9; imgIndex++)
        {
            // load image from file and convert to grayscale
            cv::Mat img, imgGray; img = cv::imread(imgFullFilename);
            cv::cvtColor(img, imgGray, cv::COLOR_BGR2GRAY);

            // push images into 2-frame ring buffer
            DataFrame frame; frame.cameraImg = imgGray;
            dataBuffer.push_back(frame);

            // extract 2D keypoints from current image
            vector<cv::KeyPoint> keypoints;
            detKeypoints(keypoints, imgGray, detectorType);

            // push keypoints for current frame to end of data buffer
            (dataBuffer.end() - 1)->keypoints = keypoints;

            // extract keypoint descriptors
            cv::Mat descriptors;
            descKeypoints((dataBuffer.end() - 1)->keypoints,
                          (dataBuffer.end() - 1)->cameraImg,
                           descriptors, descriptorType);

            // push descriptors for current frame to end of data buffer
            (dataBuffer.end() - 1)->descriptors = descriptors;
        }
    }
}
```

For the keypoint description portion of the project, I again used the OpenCV library classes and did not attempt to perform any hyperparameter tuning. The current combinatorial test wrapper also creates some constructors twice (e.g., BRISK detection + BRISK description), with the awareness that such repeated operations should be removed after the trade study is completed and a detection-descriptor pair has been selected.

```
void descKeypoints(vector<cv::KeyPoint> &keypoints, cv::Mat &img,
                   cv::Mat &descriptors, string descriptorType)
{
    cv::Ptr<cv::DescriptorExtractor> extractor;
    if (descriptorType.compare("BRISK") == 0)
    {
        int threshold = 30;        // FAST/AGAST detection threshold score
        int octaves = 3;           // detection octaves (use 0 to do single scale)
        float patternScale = 1.0f; // scale used to sample the kpt neighborhood
        extractor = cv::BRISK::create(threshold, octaves, patternScale);
    }
    else if (descriptorType.compare("BRIEF") == 0)
    {
        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();
    }
    else if (descriptorType.compare("ORB") == 0)
    {
        extractor = cv::ORB::create();
    }
    else if (descriptorType.compare("FREAK") == 0)
    {
        extractor = cv::xfeatures2d::FREAK::create();
    }
    else if (descriptorType.compare("AKAZE") == 0)
    {
        extractor = cv::AKAZE::create();
    }
    else if (descriptorType.compare("SIFT") == 0)
    {
        extractor = cv::xfeatures2d::SIFT::create();
    }
    else
    {
        std::cerr << "ERROR: Descriptor Type " << descriptorType
            << "Not Recognized!" << std::endl;
        return;
    }
    // compute keypoint descriptors
    extractor->compute(img, keypoints, descriptors);
}
```

MP.5 DESCRIPTOR MATCHING

> *Implement FLANN matching and k-nearest neighbor selection. Enable method selection using strings in the main function.*

1. Implementation

This is the main() function with string-based method selection of FLANN or Brute Force matching, and KNN or NN selection. For brevity, pseudocode comments of keypoint detection-description and data buffer management illustrate the upfront processes leading to matchDescriptors, which is the focus of this section. Please refer to the build for the full working implementation:

```
int main(int argc, const char *argv[])
{
    // matcher and selectors
    string matcherType  = "FLANN_MATCH"; // BF_MATCH, FLANN_MATCH
    string selectorType = "KNN";         // NN, KNN

    // loop over all video frames ...
    for (size_t imgIndex = 0; imgIndex <= 9; imgIndex++)
    {
        // load image from file & push images into 2-frame ring buffer ...
        // extract 2D keypoints from current image & push keypoints into end of data buffer ...
        // extract keypoint descriptors & push descriptors for into end of data buffer ...

        if (dataBuffer.size() > 1) // if at least two images have been processed ...
        {
            // then match keypoint descriptors
            vector<cv::DMatch> matches;
            matchDescriptors((dataBuffer.end() - 2)->keypoints,(dataBuffer.end() - 1)->keypoints,
                             (dataBuffer.end() - 2)->descriptors,(dataBuffer.end() - 1)->descriptors,
                             matches, descriptorType, matcherType, selectorType);

            // store matches in current data frame
            (dataBuffer.end() - 1)->kptMatches = matches;
        }
    }
}
```

This is the matchDescriptors function, which contains both FLANN and Brute Force matching:

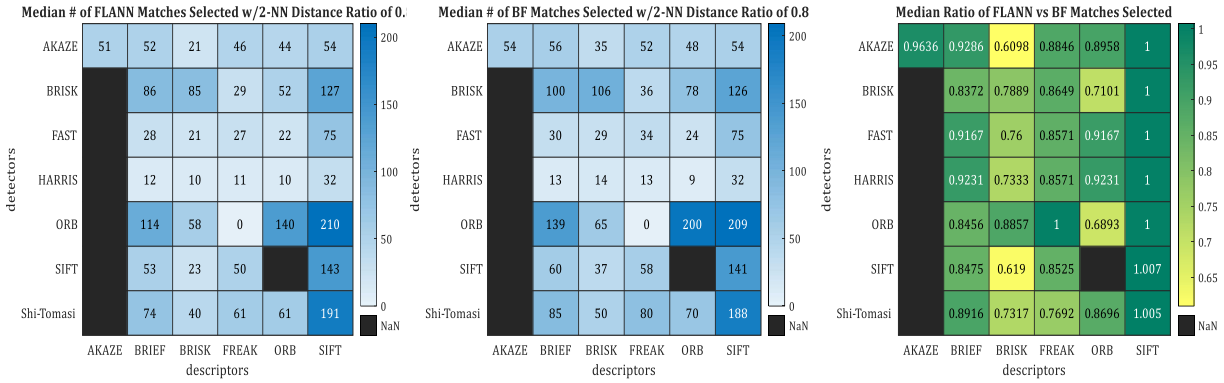
```
/* +--------------------------+ */
/* | KEYPOINT MATCHING METHODS | */
/* +--------------------------+ */
void matchDescriptors(std::vector<cv::KeyPoint> &kPtsSource, std::vector<cv::KeyPoint> &kPtsRef,
                      cv::Mat &descSource, cv::Mat &descRef, std::vector<cv::DMatch> &matches,
                      std::string descriptorType, std::string matcherType, std::string selectorType)
{
    // Configure & Create the Matcher Object
    cv::Ptr<cv::DescriptorMatcher> matcher;

    if (matcherType.compare("FLANN_MATCH") == 0) // Fast Library for Approximate Nearest Neighbors
    {
        if (descRef.type() != CV_32F || descSource.type() != CV_32F)
        { // OpenCV bug: FLANN needs the descriptors to be of type CV_32F
            descRef.convertTo(descRef, CV_32F);
            descSource.convertTo(descSource, CV_32F);
        }
        matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
    }
```

And the k-nearest neighbor implementation with distance ratio test:

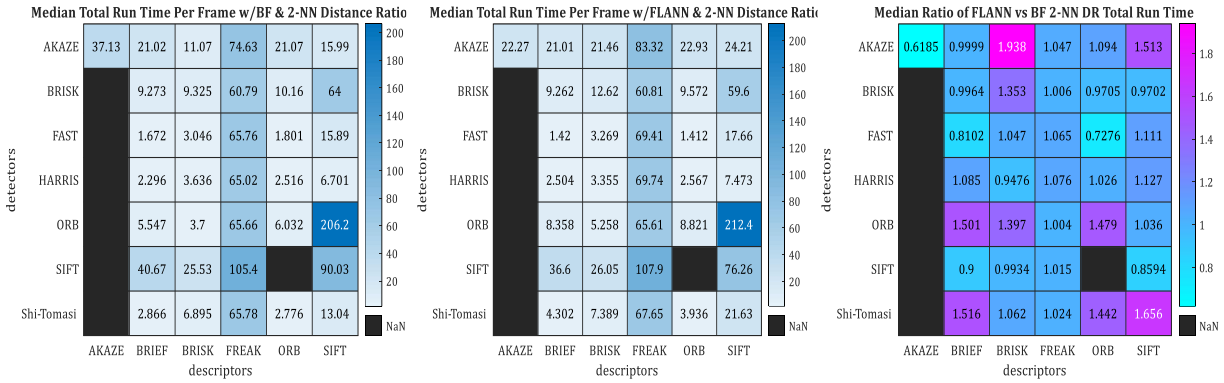
```
// Select Best Matches
if (selectorType.compare("KNN") == 0)
{
    // k nearest neighbors: keep the best k matches
    vector<vector<cv::DMatch>> knn_matches;
    int numNeighbors = 2; // find the 2 best matches
    matcher->knnMatch(descSource, descRef, knn_matches, numNeighbors);

    // Apply the distance ratio test to filter out false matches (alternative to crossCheck)
    //  For a given keypoint:
    //    - take the ratio of the distances between the closest neighbor and the second closest neighbor
    //    - compare the distance ratio to a threshold of minDescDistRatio
    //  If the ratio > threshold, then the match is ambiguous with a lower probability of being correct
    //  If the ratio < threshold, then accept the closest match
    double minDescDistRatio = 0.8;
    for (auto it = knn_matches.begin(); it != knn_matches.end(); ++it)
    {
        //if (((*it)[0].distance / (*it)[1].distance) < minDescDistRatio)
        if ( (*it)[0].distance < minDescDistRatio * (*it)[1].distance ) // use mult to incr clock speed
        {
            // keep the most likely matches
            matches.push_back((*it)[0]);
        }
    }
}
```

2. Performance Comparison: Brute Force vs FLANN Matching

The FLANN Matching implementation showed similar accuracy and timing performance to Brute Force Matching with KNN and distance ratio thresholding. The following figure shows that FLANN usually selected fewer matches than Brute Force – with the exception of SIFT, where they were essentially equal.



Total run time comparisons show statistically similar performance between FLANN and Brute Force Matching. Variations in run-time estimates were shown to vary widely when running in the student workspace.



MP.6 DESCRIPTOR DISTANCE RATIO

> *Implement the descriptor distance ratio test within the k-nearest neighbor matching method.*

As shown in the code snippet in the previous section:

```
// Apply the distance ratio test to filter out false matches (alternative to crossCheck)
//  For a given keypoint:
//    - take the ratio of the distances between the closest neighbor and the second closest neighbor
//    - compare the distance ratio to a threshold of minDescDistRatio
//  If the ratio > threshold, then the match is ambiguous with a lower probability of being correct
//  If the ratio < threshold, then accept the closest match
double minDescDistRatio = 0.8;
for (auto it = knn_matches.begin(); it != knn_matches.end(); ++it)
{
    //if (((*it)[0].distance / (*it)[1].distance) < minDescDistRatio)
    if ( (*it)[0].distance < minDescDistRatio * (*it)[1].distance ) // use mult to incr clock speed
    {
        // keep the most likely matches
        matches.push_back((*it)[0]);
    }
}
```

MP.7 PERFORMANCE EVALUATION, PART I

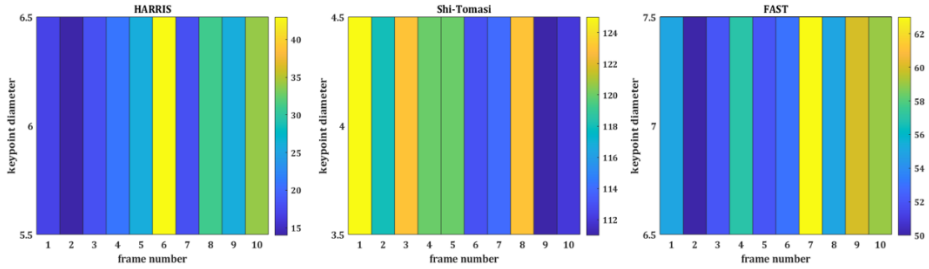
For all detector implementations, count the number of keypoints on the preceding vehicle for all 10 images and take note of the distribution of their neighborhood size.

Metrics for assessing neighborhood detection quality include accuracy, completeness and repeatability of keypoint positions detected across video frames. Depending on the problem space, this may require that the algorithms possess any or all of the following characteristics: robustness to variations in orientation, size, scale, noise, illumination, contrast and perspective distortion.

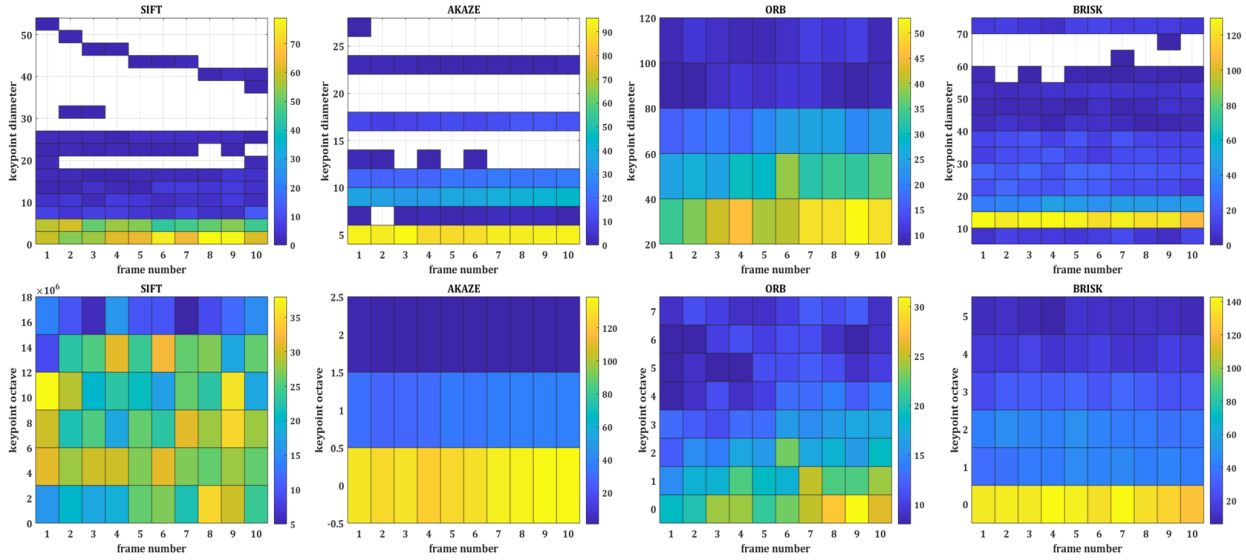
The OpenCV Keypoint class included public attributes that are measured by some of the keypoint detector methods. Where applicable, that information is shown here. For the other methods, we relied on a qualitative assessment based on the video imagery with keypoints overlaid.

1. Keypoint Size & Scale of Meaningful Neighborhoods

Shi-Tomasi, Harris, and Features from Accelerated Segment Test (FAST) are very efficient corner detectors that used fixed-diameter image patches of size 4, 6, and 7 pixels, respectively. They are each tuned to a specific size, scale and image resolution; keypoint detection performance may suffer when variations in size, scale and resolution are present. The 2D histograms below represent the number of keypoints of fixed diameter on each frame, where the colorbar represents number of keypoints in each bin.

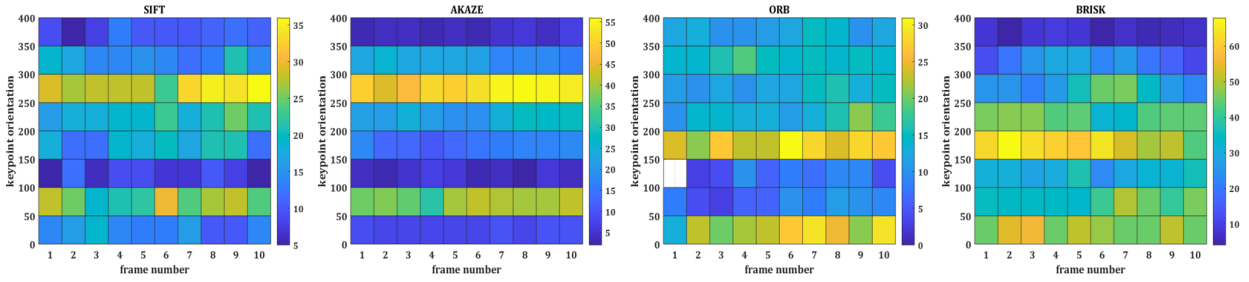


SIFT, Akaze, ORB and BRISK use multi-scale methods that enable keypoints to adapt to variations in size, scale and image resolution. The range and granularity of their collection regions are captured in the 2D histograms below. The colorbar represents the number of counts for each bin.



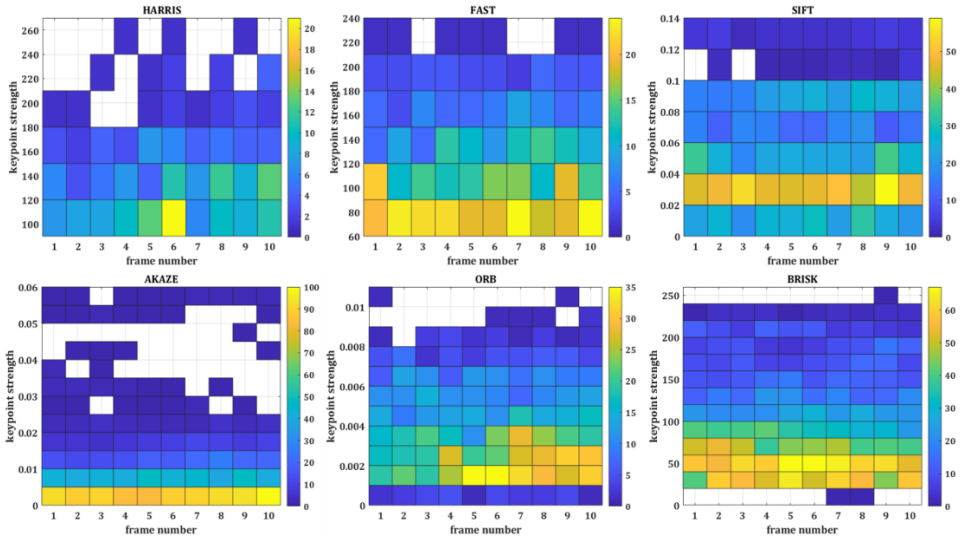
2. Keypoint Orientation

The following 2D histograms show the estimates of keypoint orientation computed by SIFT, AKAZE, ORB and BRISK. Harris, Shi-Tomasi and FAST are designed to be partially invariant to rotation, but they do not compute the orientation component.



3. Keypoint Strengths

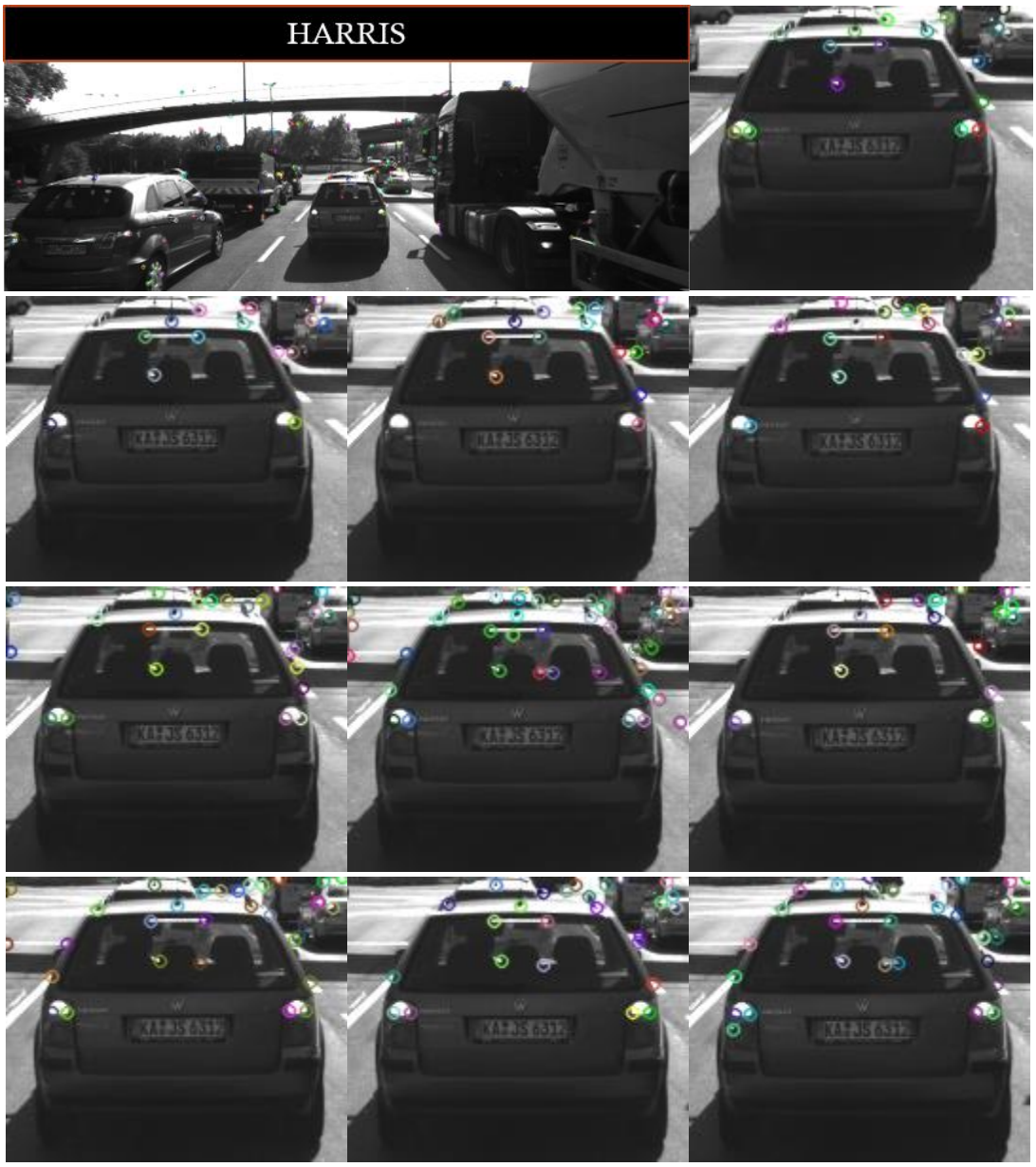
The following 2D histograms show the keypoint response or strength, which can be used to select the strongest keypoints and/or be used for further sorting or subsampling.



4. Drawing Rich Keypoints

Harris Corner Detector

The Harris Corner Detector has a fixed keypoint diameter of 6 pixels. It is limited to detecting corners at that scale, which is smaller than the size of each tail light, but large enough to consistently capture the narrow white horizontal bar just below the roof antenna. This size is slightly sub-optimal relative to the image resolution, resulting in sparse corner detection that still manages captures most of the important details most of the time (e.g., tail lights, base of roof antenna, horizonal bar ends, roof corner points, rear view mirrors, and sharper corner(s) on the silhouette of the seat backs within the vehicle). However, it occasionally misses some of these important points, which may be an issue for computing things like Time-to-Collision depending on the tracker's robustness to dropouts at the given rates of motion. The algorithm also demonstrates its robustness to rotation by maintaining corner detection of sharper, high contrast points across the slight relative motion of objects in the scene across the frames of video.



Shi-Tomasi Corner Detector

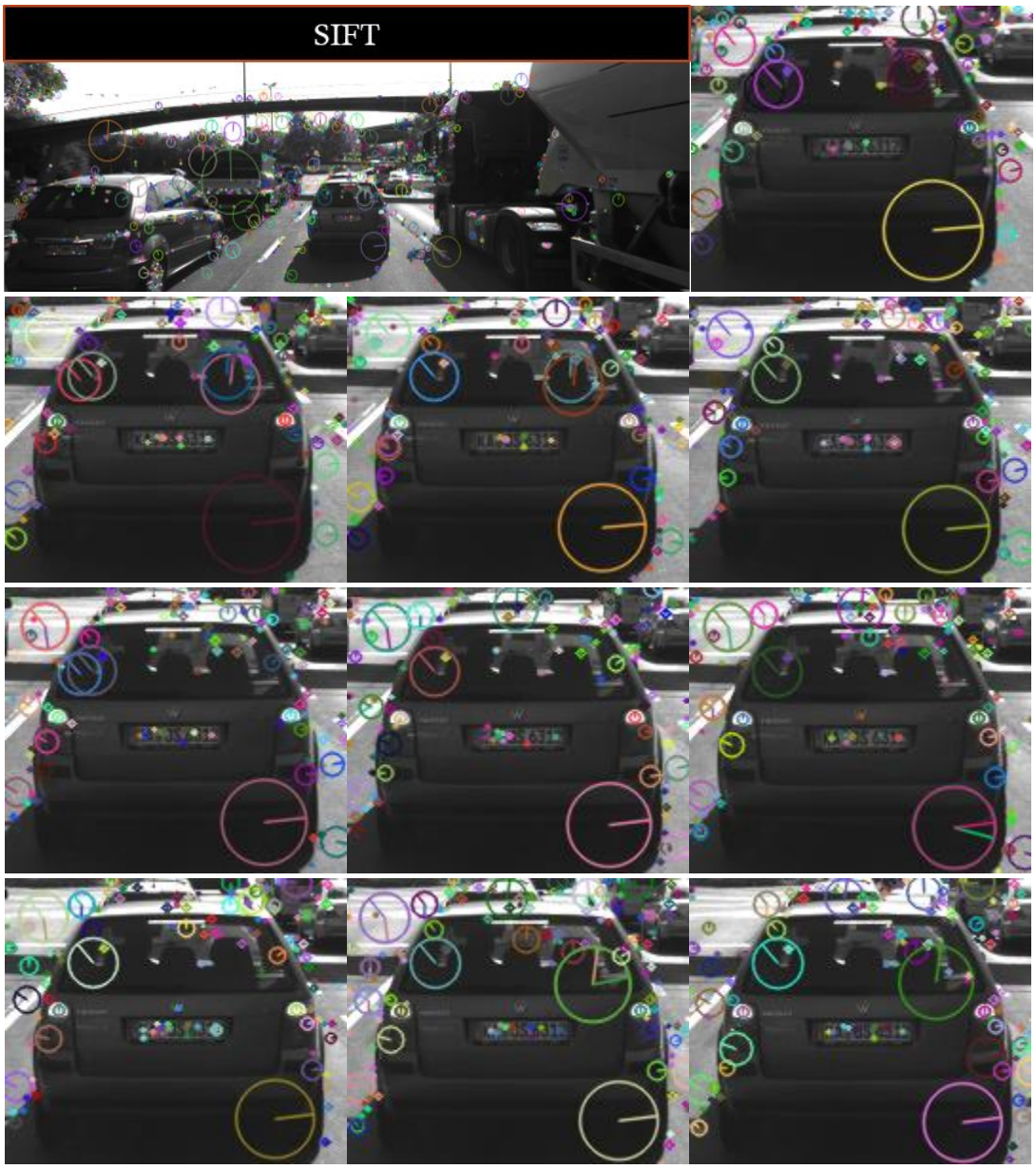
The Shi-Tomasi corner detector had a fixed collection region of 4 pixels, which tuned it to pick up very small scale details such as letter corners on the license plate and the car logo above it. It also seems to pick up a lot of high frequency noise and edges, which might be due to pixel quantization error. The small tweak in the implementation of corner detection relative to the Harris approach seems to pick up a lot more details. Double counting of salient features (such as the tail lights) with closely-spaced keypoints tends to occur frequently because of the small image patch focus, but this may provide more opportunities for keypoint matching. In this sense, the lack of variation in scale is accounted for spatially, and in some cases it might be possible to cluster keypoints and combine their respective collection regions to produce similar size patches to the multi-scale approaches.

Features from Accelerated Segment Test (FAST)

The FAST detector had a fixed collection region of 7 pixels. It seems to pick up more information than the Harris Corner Detection approach, without being as sensitive to noise and pixel quantization errors as Shi-Tomasi. In some cases the keypoints appear to be overlapped, despite performing non-maximum suppression. The intensity threshold was set to 60; lowering this value might result in the algorithm picking up some of the details on the license plate.

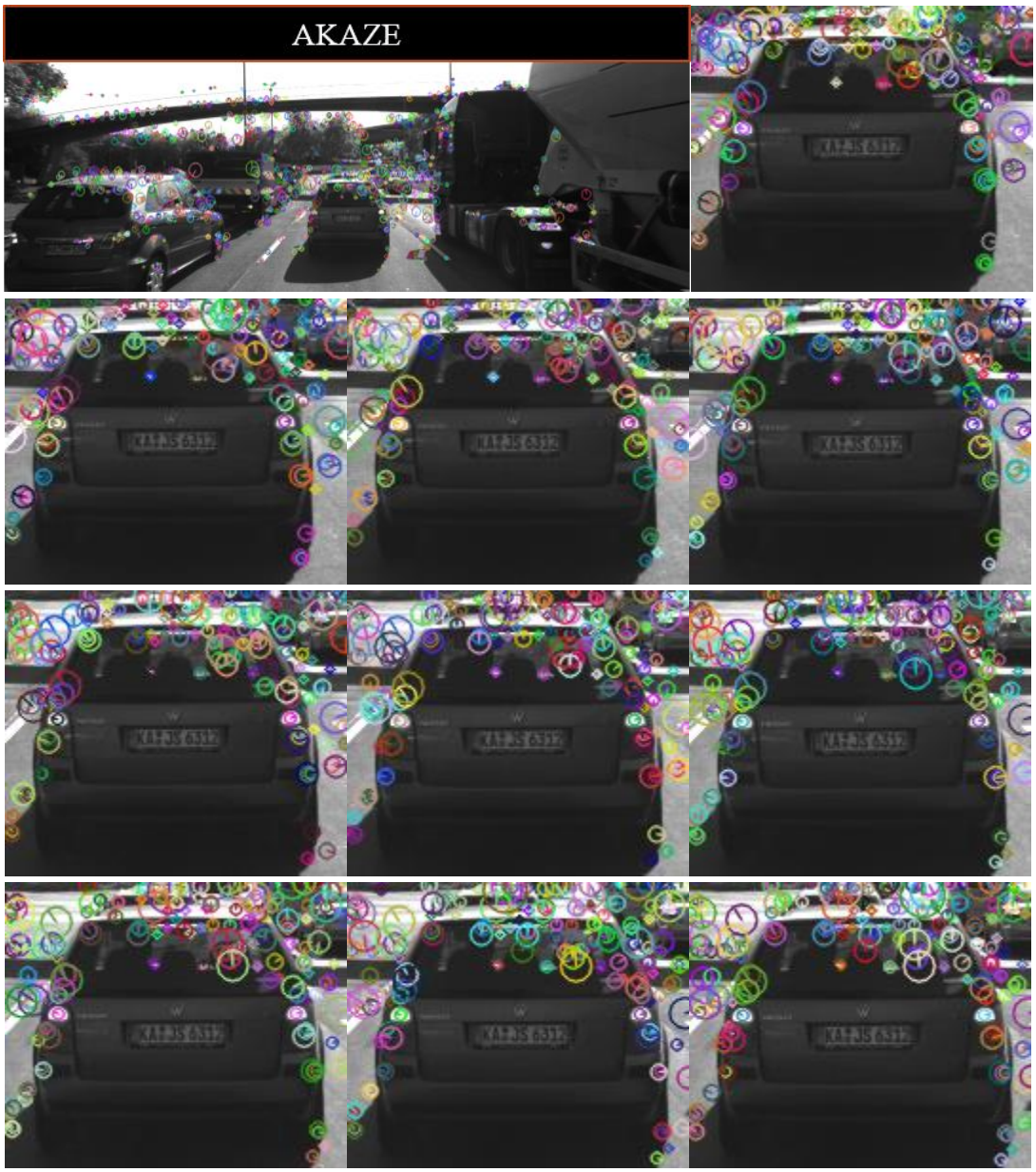
Scale Invariant Feature Transform (SIFT)

SIFT is a multi-scale method, able to pick up details much smaller than Shi-Tomasi up to regions as large as the seat-backs and shadow of the preceding car, with a lot of granularity in size variations in between. It also demonstrates precise orientation estimation across variations in illumination with high repeatability across frames. In terms of accuracy, this is the algorithm to beat – but it comes at a significant cost to computational complexity and speed.



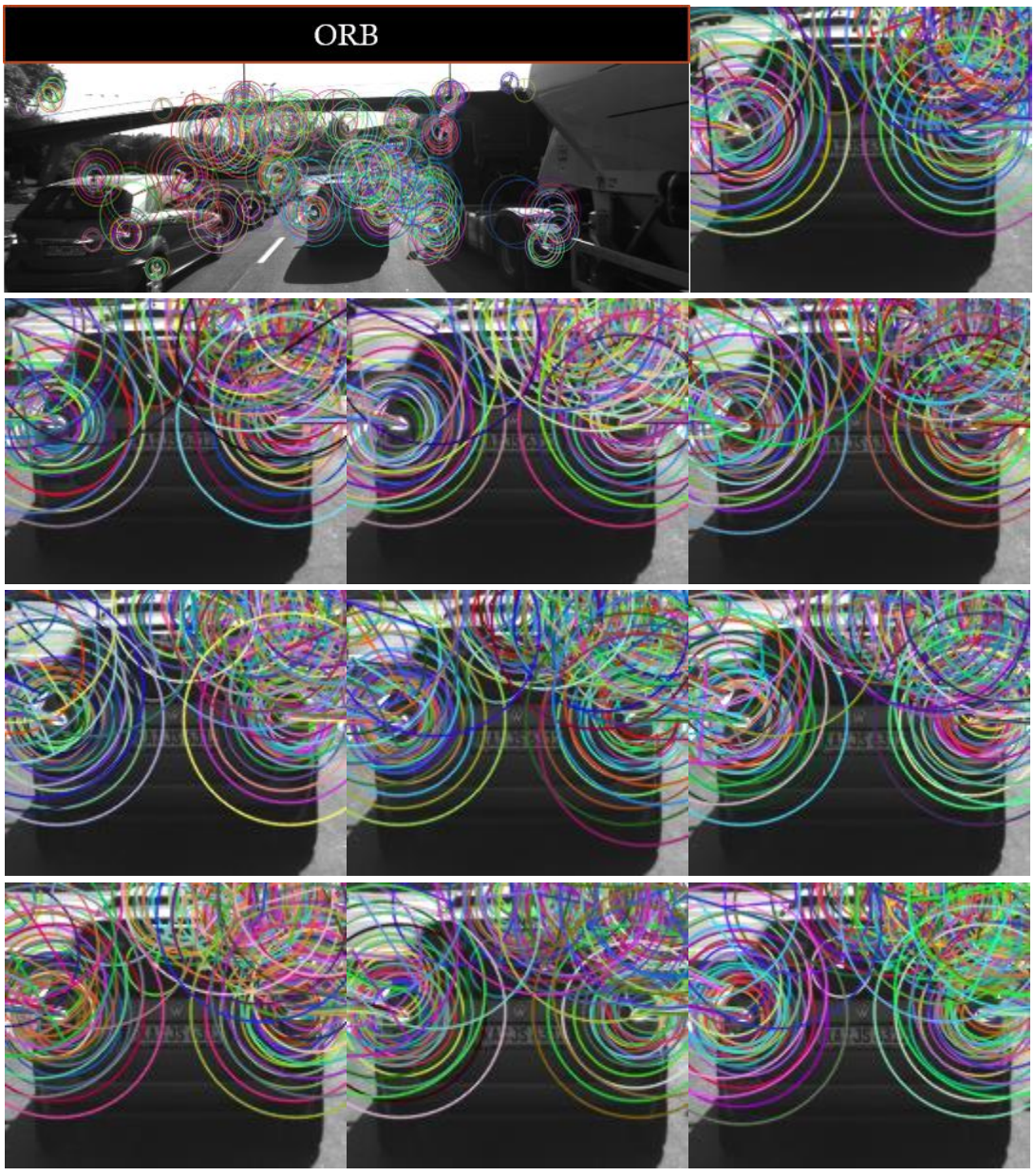
Accelerated KAZE (AKAZE)

AKAZE is meant to produce results similar to SIFT but on a sparse nonlinear scale space with reduced computational complexity (and without the “nonfree” licensing constraints). Functionally, it seems to behave like a noisy edge detector, with keypoints following the overall silhouette of the car. However, while it is not always clear to a human eye what details the keypoints are picking up on, there does seem to be an impressive consistency in the estimated position, size/scale and orientation of keypoints across images – ultimately, that is what matters for our application. Visibly, the approach does produce distinct, nicely centered, scaled and oriented keypoints on the tail lights of the car, with a degree of precision only observed in SIFT.



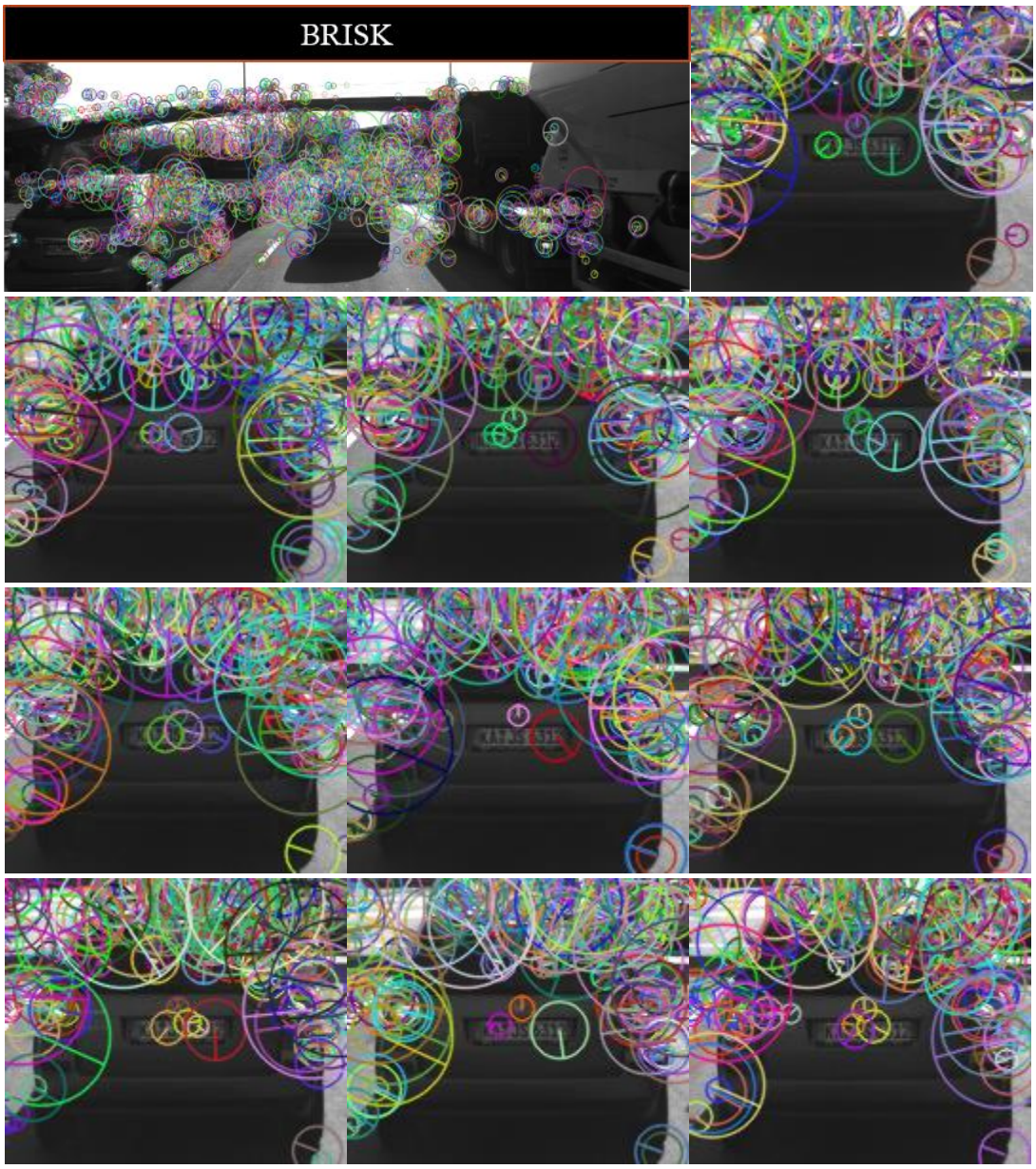
Oriented FAST and Rotated BRIEF (ORB)

ORB keypoints appear frequently as groups of (somewhat) concentric circles, probably as a result of running FAST in parallel across octaves without downselect. Thus, it seems that (more or less) the same keypoints are consistently identified and selected across collection regions of varying size. Orientation estimates are also observed to vary at different scales on the same keypoint. These effects combine to form what looks like uncertainty distributions in size/scale and orientation that could be helpful in performing frame-to-frame keypoint correlation under real-world uncertainty.



BRISK (Binary Robust Invariant Scalable Keypoints)

BRISK keypoints use a multi-scale approach similar to ORB, but with increased sub-pixel granularity from intra-octave layer sampling and interpolation to maximum-saliency “true scale” keypoint. The result is a set of “optimal’ overlapping (but not concentric) keypoints of varying size/scale. Unfortunately for our application, this approach produces somewhat unstable keypoints with lower repeatability across frames, making it a sub-optimal candidate for frame-to-frame keypoint matching.

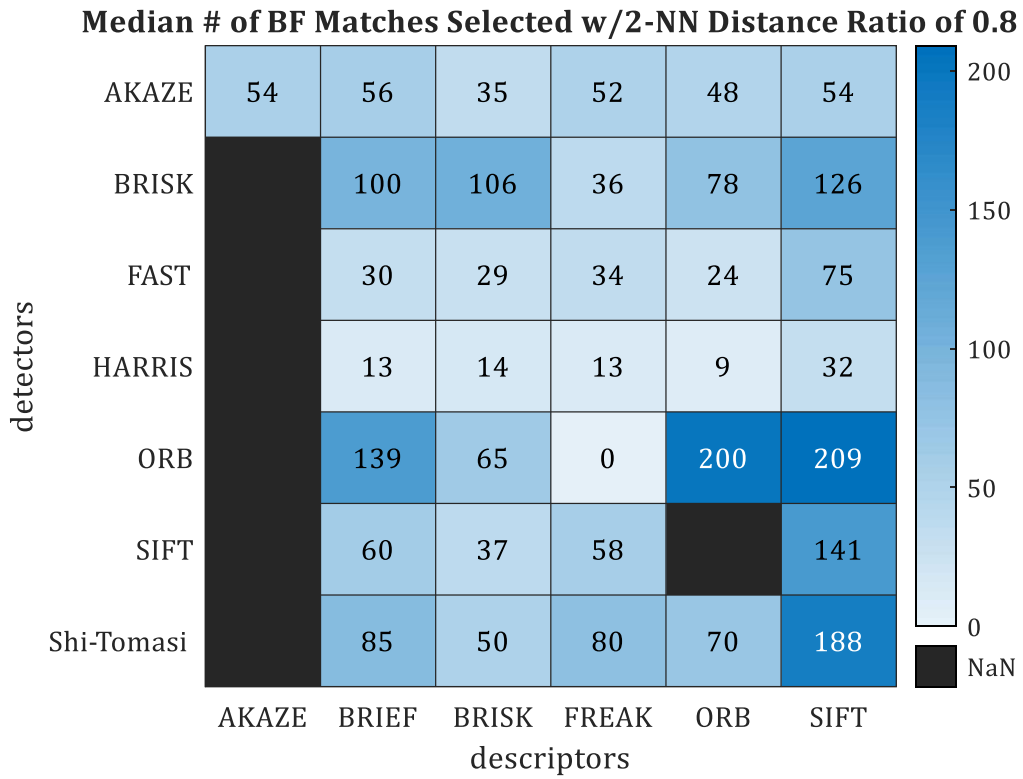


MP.8 PERFORMANCE EVALUATION, PART II

Using the Brute Force matching approach with descriptor distance ratio threshold set to 0.8, count the number of matched keypoints for all 10 images using all possible combinations of detectors and descriptors.

1. Overview: Median Number of Matched Keypoints

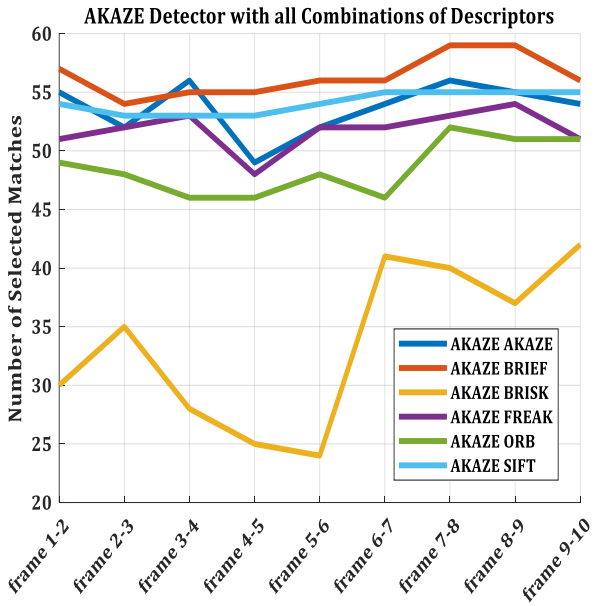
We can look at the median number of keypoint matches for a quick comparison across the different configurations.

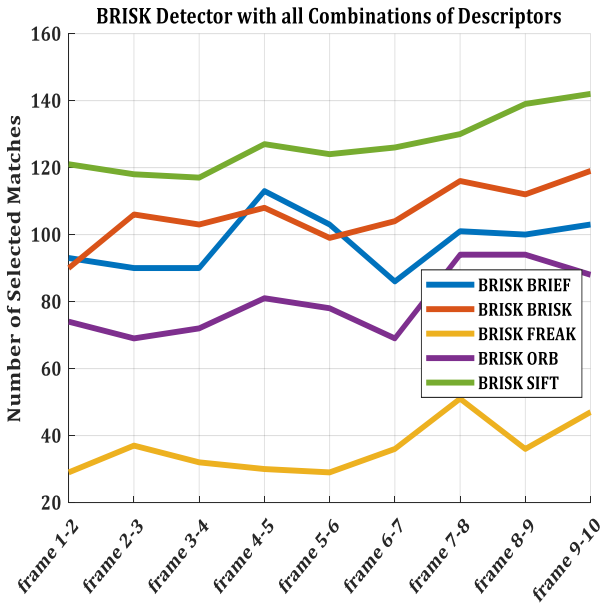


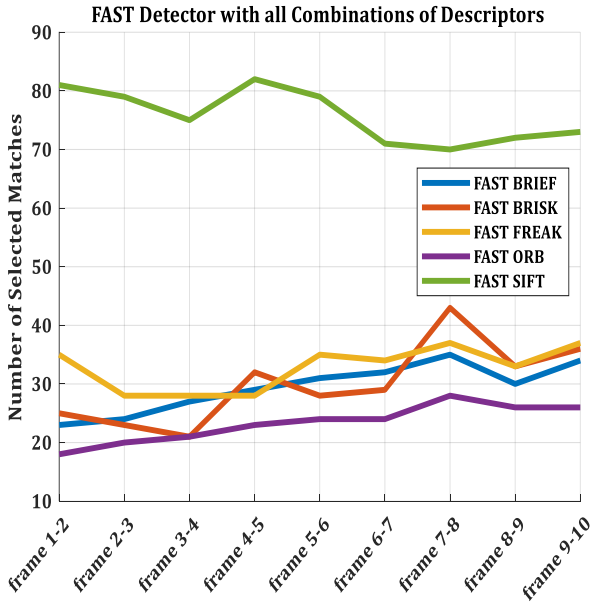
Note that AKAZE descriptors can only be used with KAZE or AKAZE detectors, and ORB descriptors cannot be computed on SIFT detectors. Also the ORB-FREAK combination produced no matches.

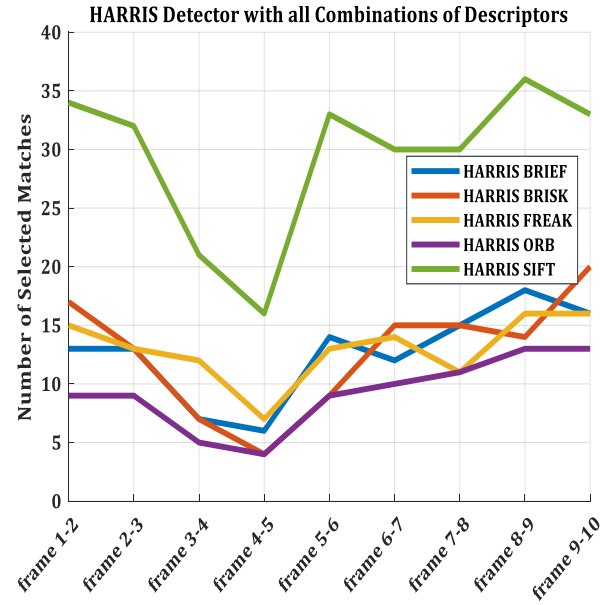
2. Number of Matched Keypoints for all 10 Images

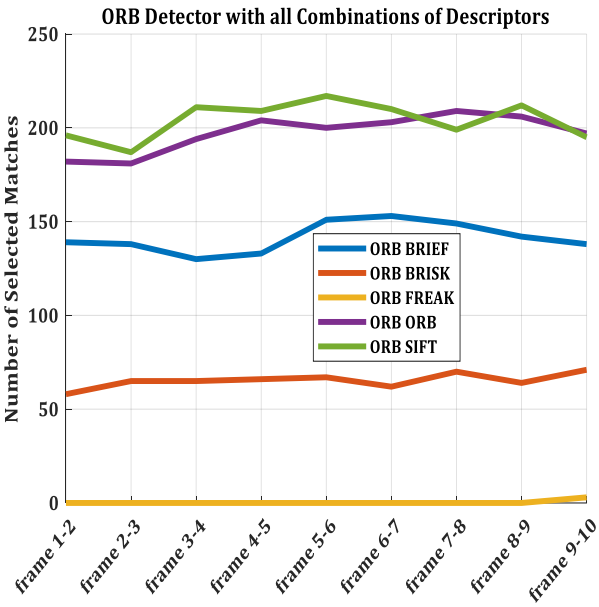
The following charts show the (between-frame) time histories of number of keypoint matches generated by each valid detector and descriptor combination in our study.

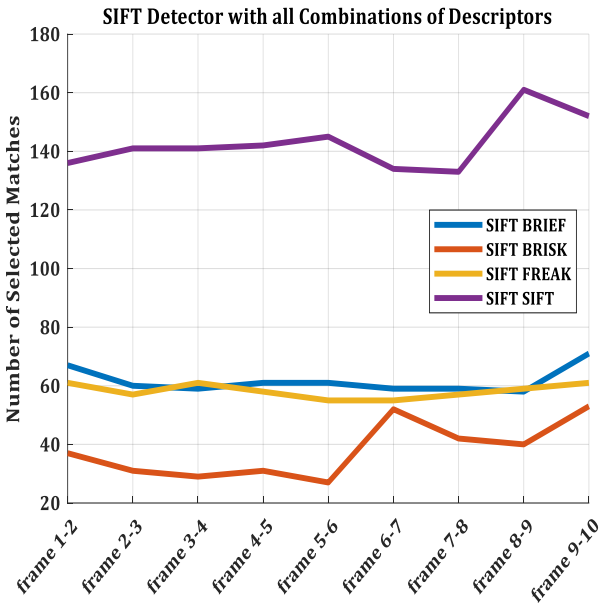


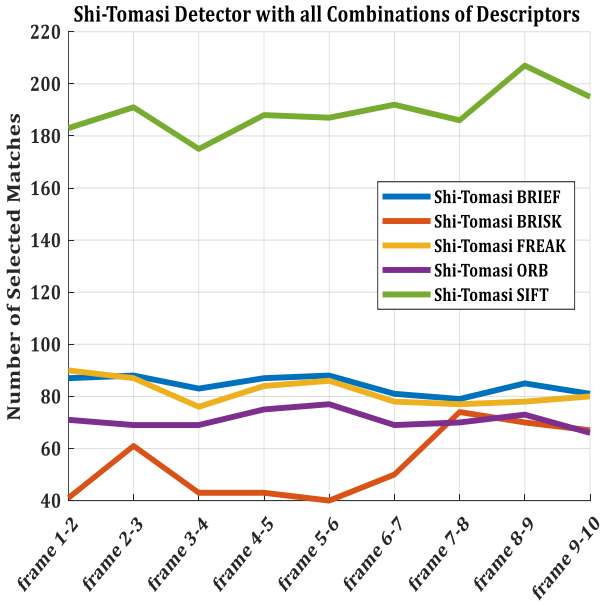







ORB Detector with all Combinations of Descriptors
Number of Selected Matches
250
200
150
100
50
0
ORB BRIEF
ORB BRISK
ORB FREAK
ORB ORB
ORB SIFT
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10

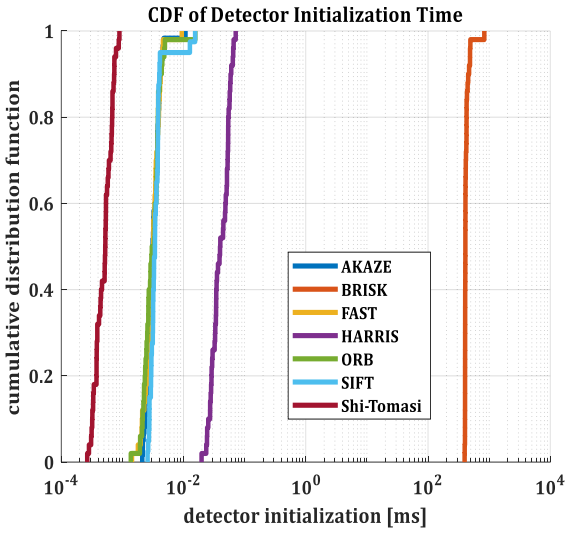

SIFT Detector with all Combinations of Descriptors
Number of Selected Matches
180
160
140
120
100
80
60
40
20
SIFT BRIEF
SIFT BRISK
SIFT FREAK
SIFT SIFT
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10

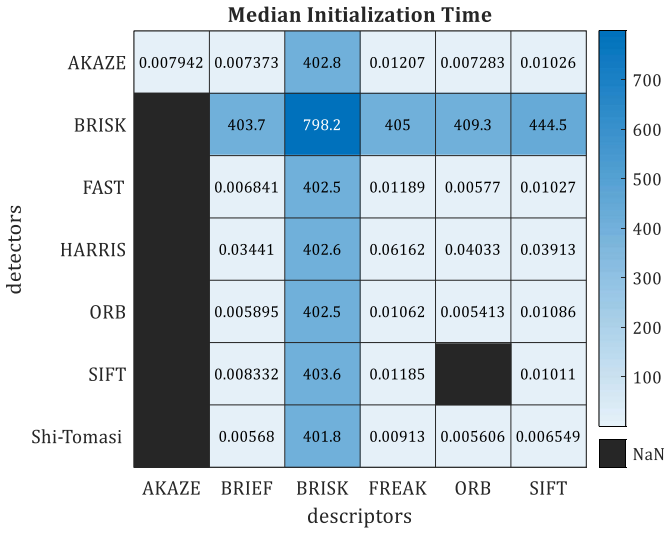

Shi-Tomasi Detector with all Combinations of Descriptors
Number of Selected Matches
220
200
180
160
140
120
100
80
60
40
Shi-Tomasi BRIEF
Shi-Tomasi BRISK
Shi-Tomasi FREAK
Shi-Tomasi ORB
Shi-Tomasi SIFT
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10

MP.9 PERFORMANCE EVALUATION, PART III

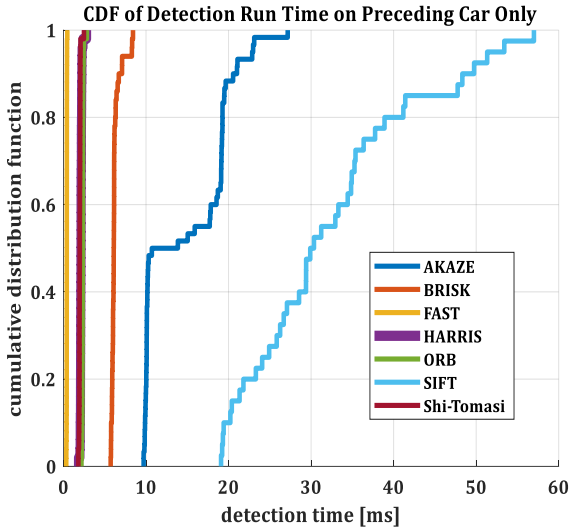
For all combinations of keypoint detection and descriptor extraction methods, record the time to complete. Based on this data, provide a recommendation for the top 3 detector/descriptor combinations for this application.

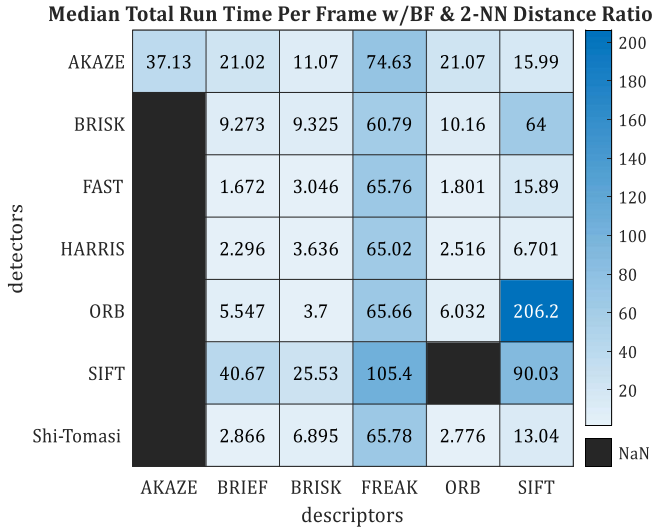
During the timing analysis, it was noted that the BRISK constructor was taking an extraordinarily long time to complete. This ran counter to expectations (and to what the literature states) that BRISK is more computationally efficient than say, SIFT. The timing issue did not seem to extend to the actual BRISK run-time execution routine, which ran at a rate that seemed more in line with expectations. Hence, under the assumption that initialization would only occur once at power-up, I extracted the initialization time out of the run-time estimates. For documentation purposes, the constructor initialization times are included here, but they are not included in the run-time estimates:





These next figures show the (1) CDFs for run-time in milliseconds across all frames for each keypoint detection method, and (2) the median total run time in milliseconds per frame, which includes Brute Force Matching and 2-Nearest Neighbor Distance Ratio Thresholding.





Based on these estimates, I generated a list of detector-descriptor candidates and performed a rough tradeoff between number of keypoint matches, subjective matching accuracy and computational cost to select my “top 3” detector-descriptor pairs for our application.

1. Top 3 Detector-Descriptor Pairs

In the absence of ground truth for this problem space, we are unable to quantitatively assess the relative accuracy of the competing detector-descriptor combinations. We can count the number of matches, but this does not give us accuracy metrics such as true positive or false positive rates.

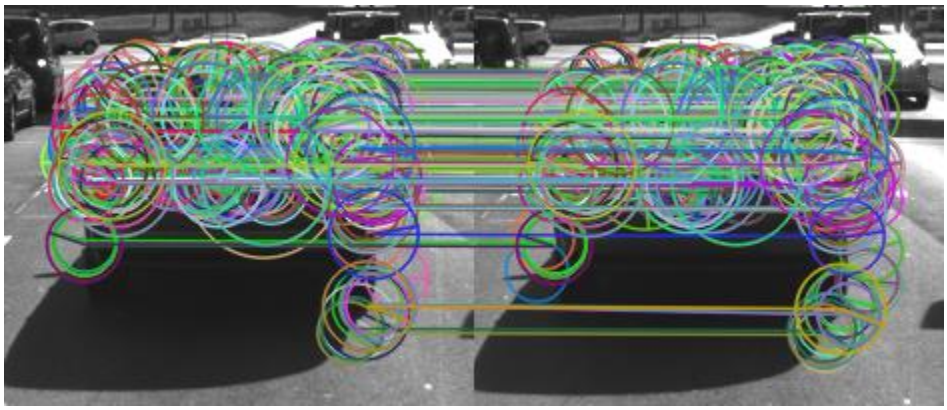
Hence, based on a subjective visual assessment of accuracy in keypoint identification, matching & selection (see images below), combined with the algorithm timing information, here are my top 3 choices:

Rank	Detector-Descriptor Pair	Median Number of Matches	Median Run-Time Per Frame [ms]
1	ORB - ORB	200	6.032
2	ORB - BRIEF	139	5.547
3	BRISK - BRISK	106	9.325
3	BRISK - BRIEF	100	9.273

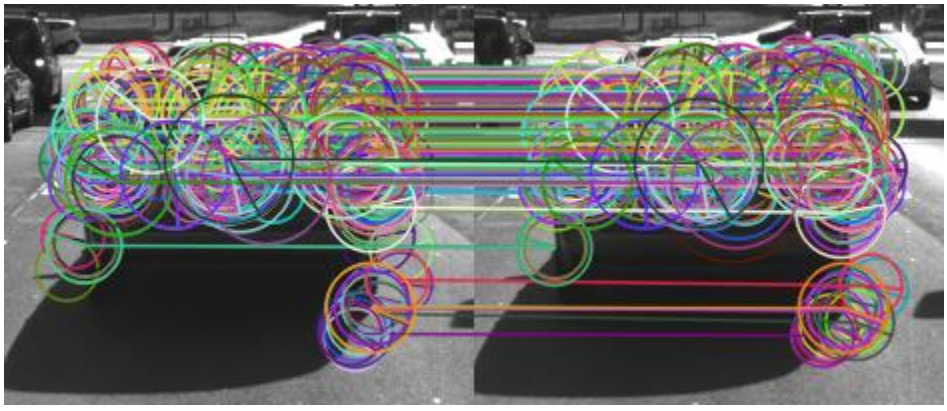
2. Visualization of Keypoint Matches on Video Frames

First Place: ORB-ORB

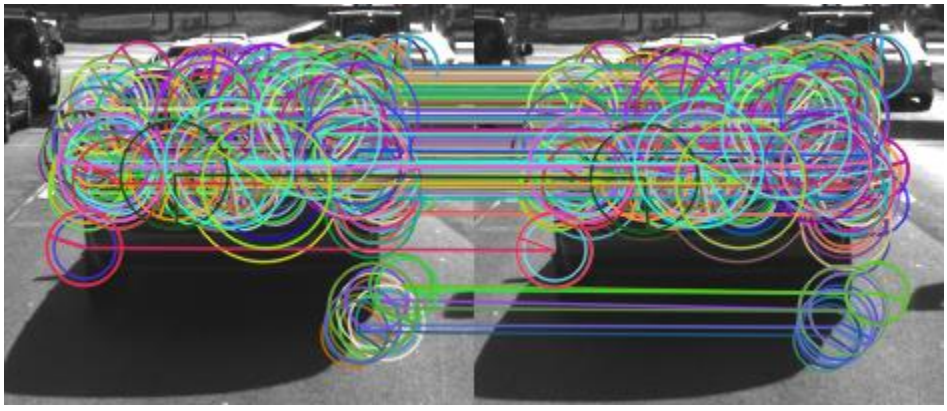
Frame 1-2



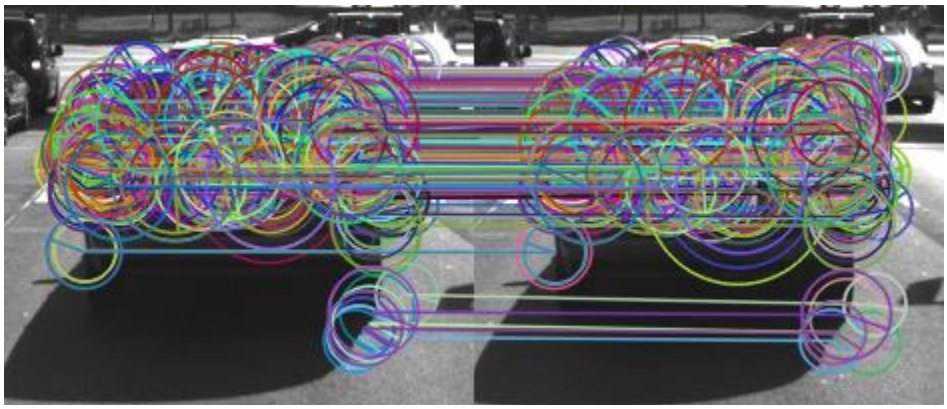
Frame 2-3



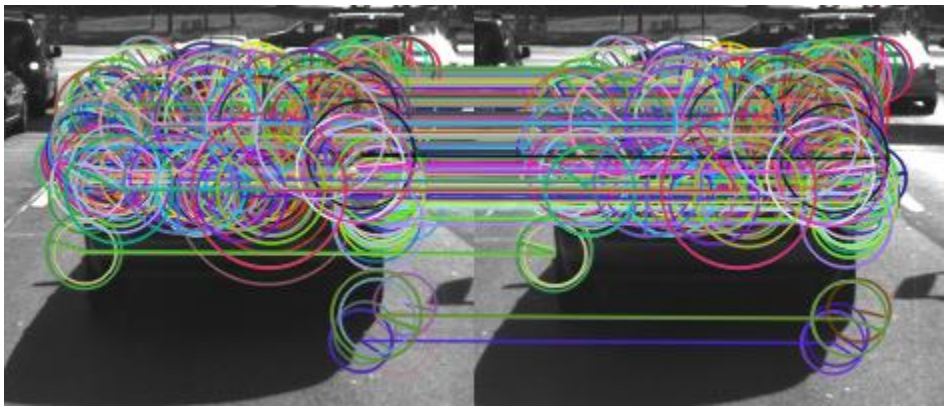
Frame 3-4



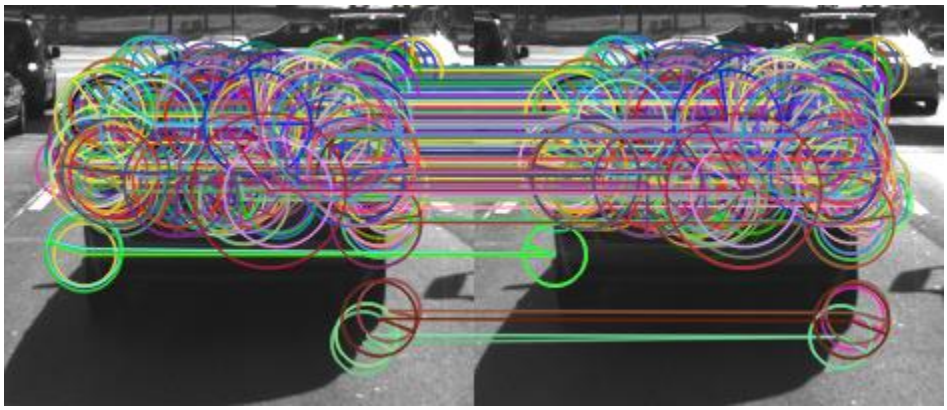
Frame 4-5



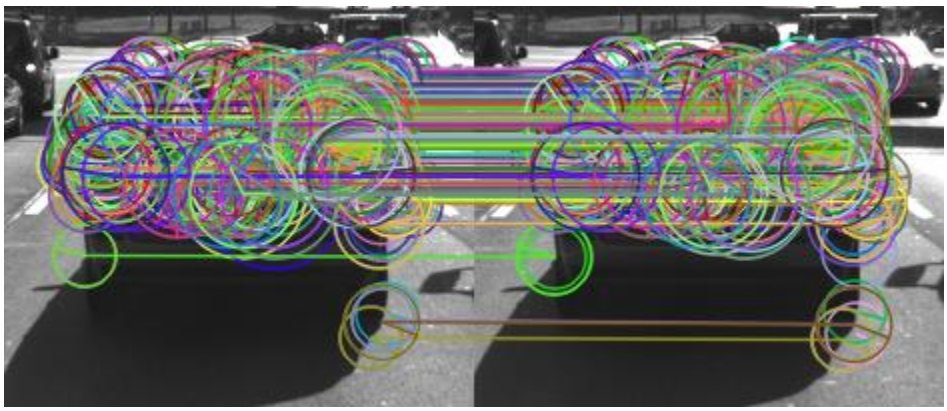
Frame 5-6



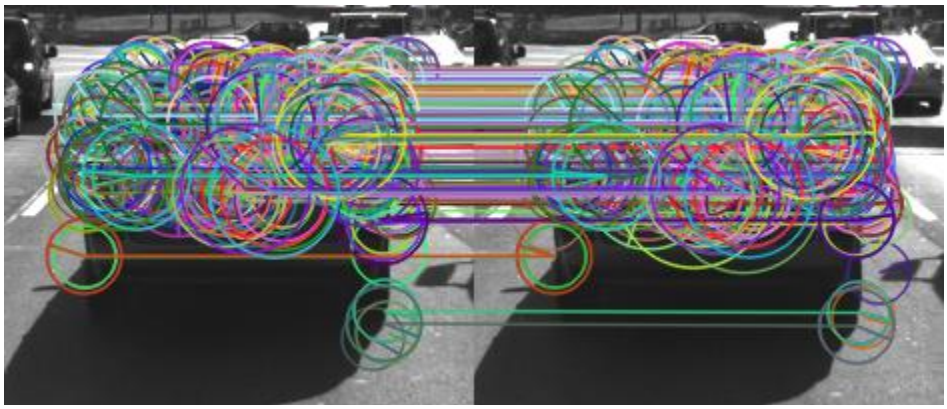
Frame 6-7



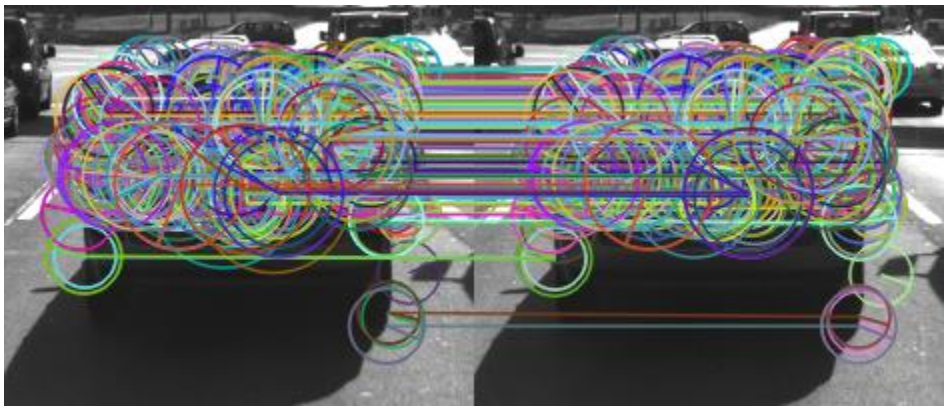
Frame 7-8



Frame 8-9

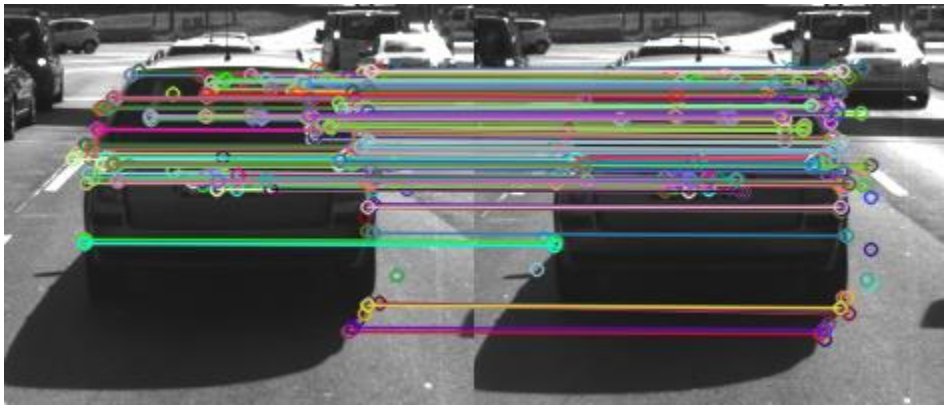


Frame 9-10

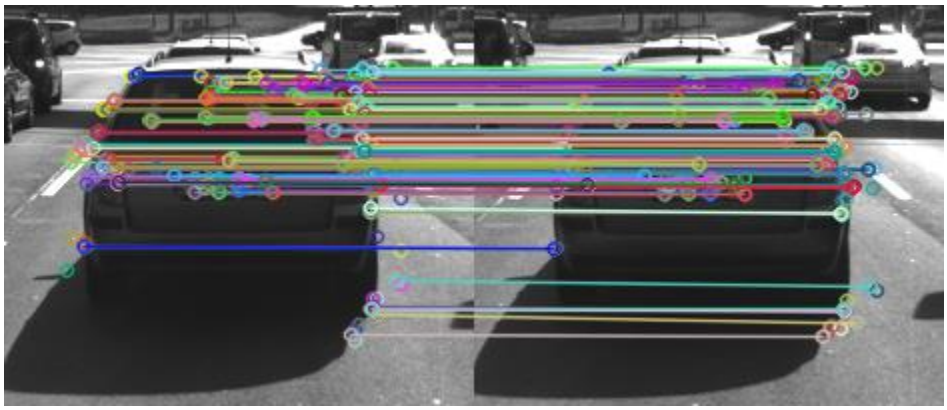


For easier viewing, here is ORB-ORB again with default keypoints:

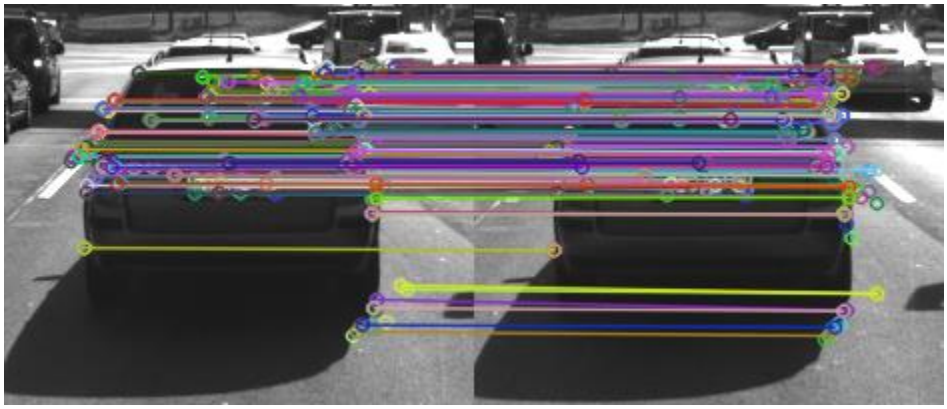
Frame 1-2



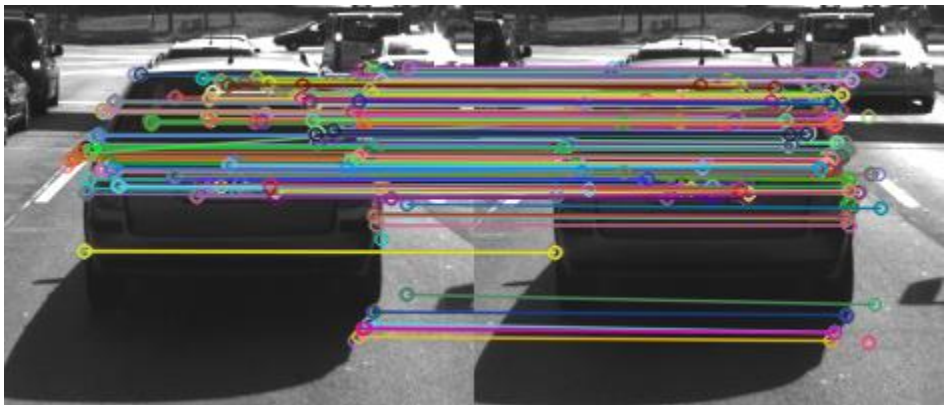
Frame 2-3



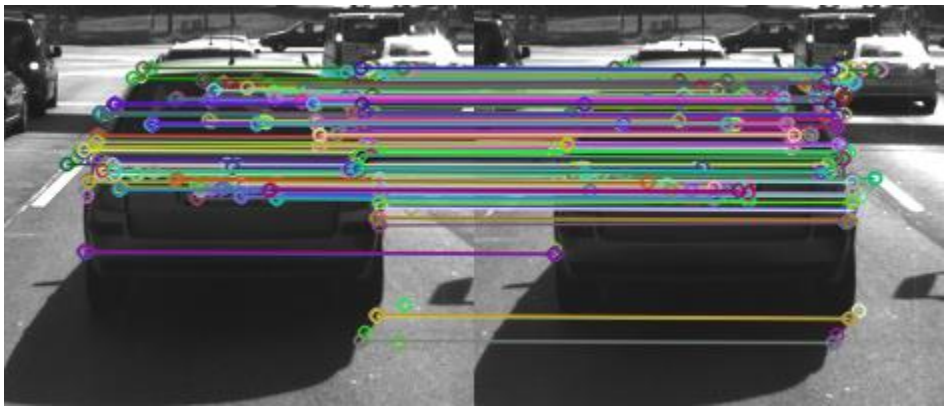
Frame 3-4



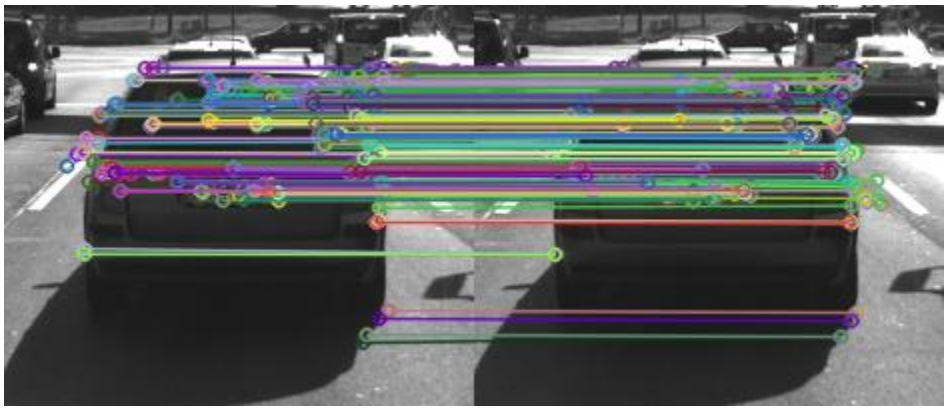
Frame 4-5



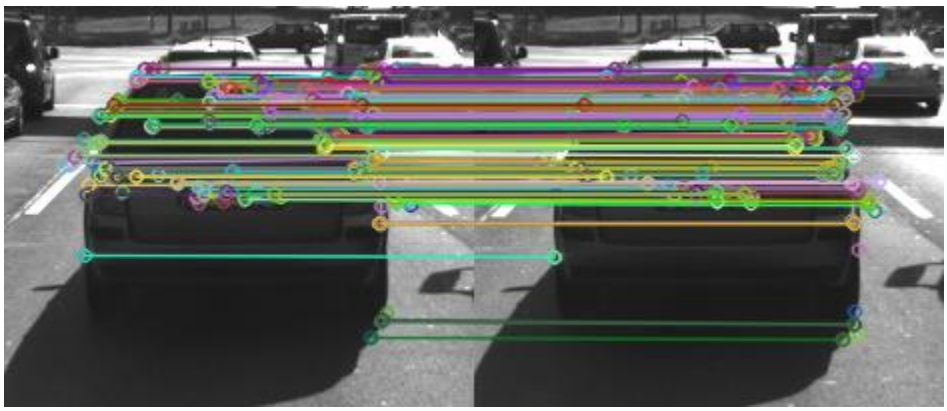
Frame 5-6



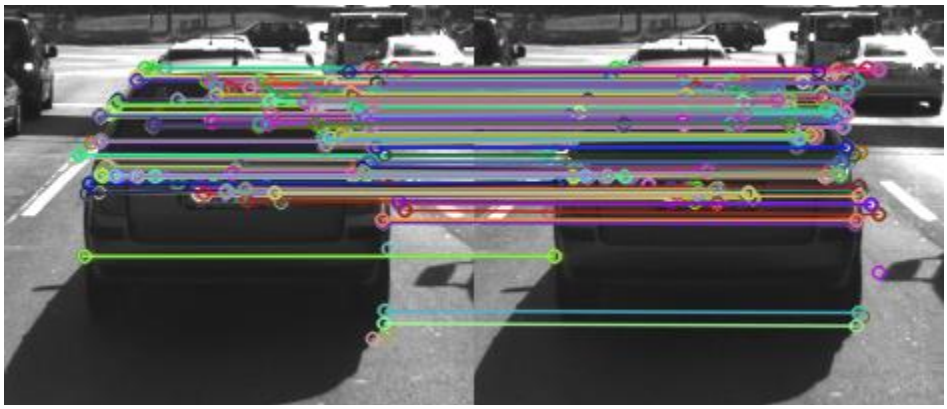
Frame 6-7



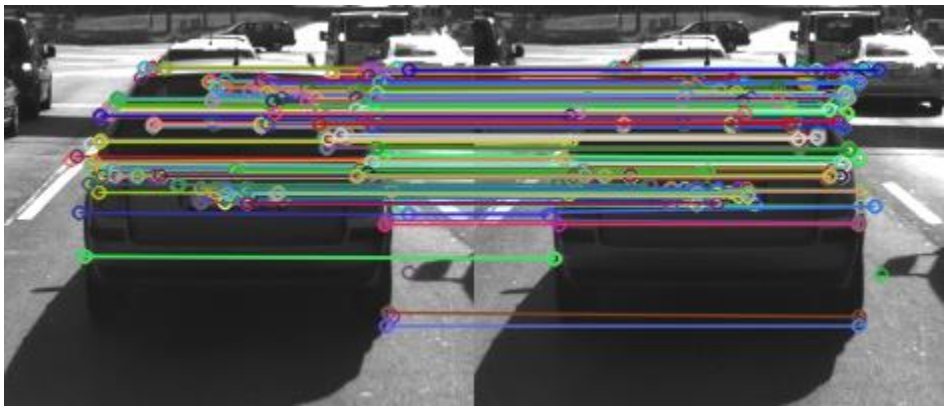
Frame 7-8



Frame 8-9

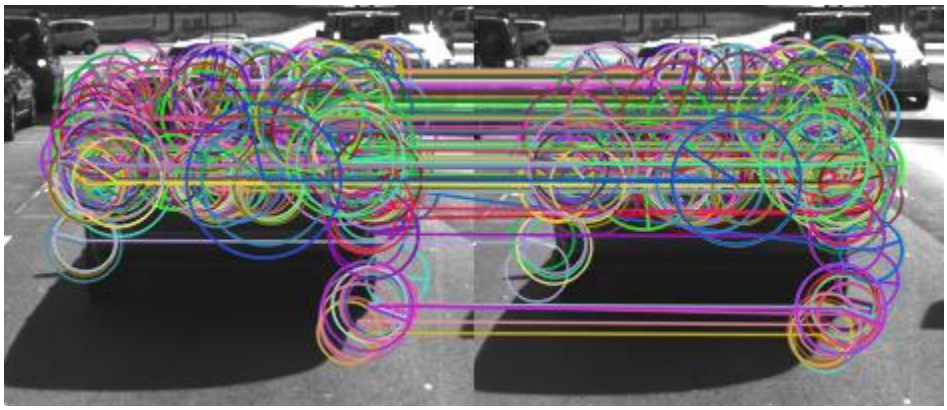


Frame 9-10

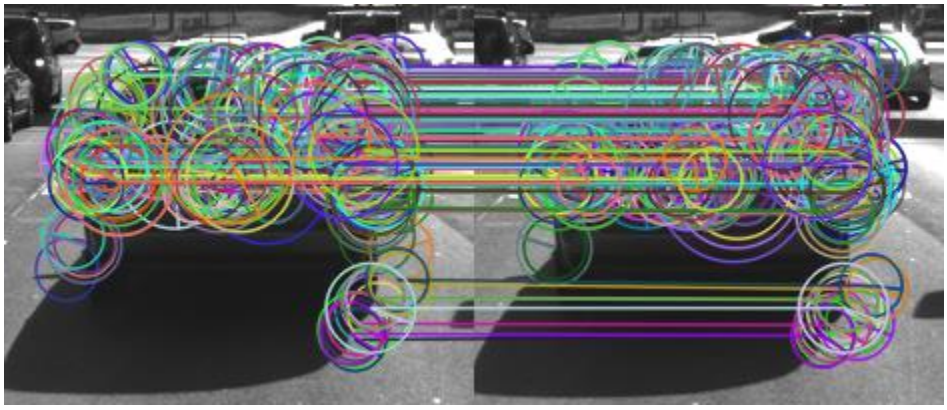


Second Place: ORB-BRIEF

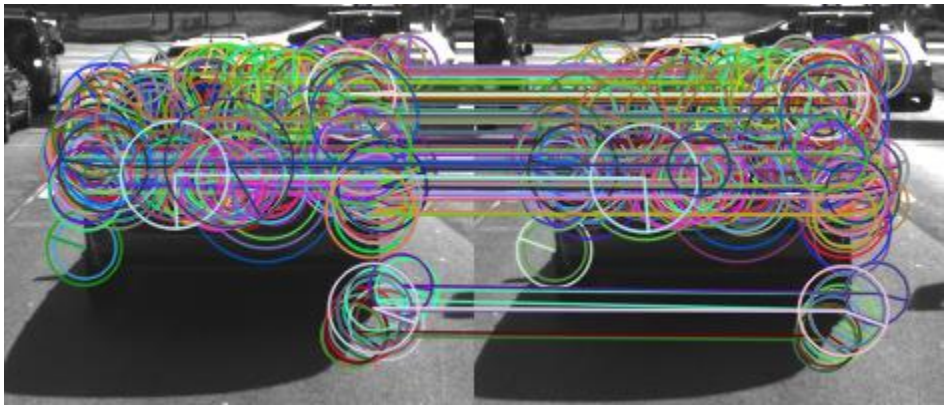
Frame 1-2



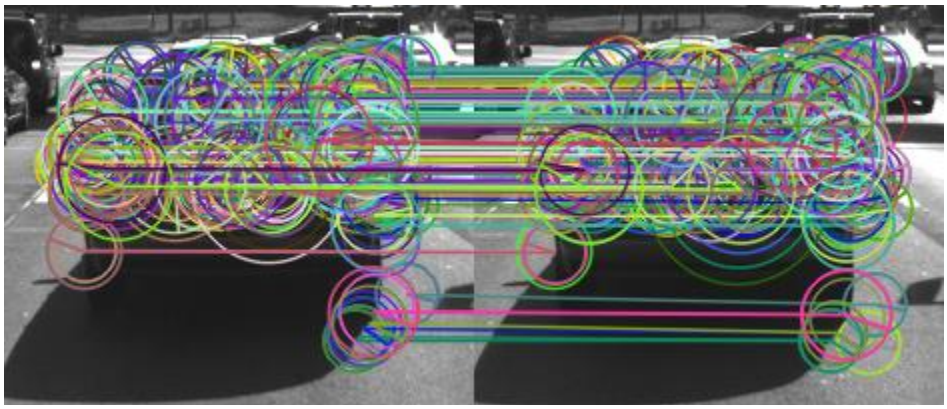
Frame 2-3



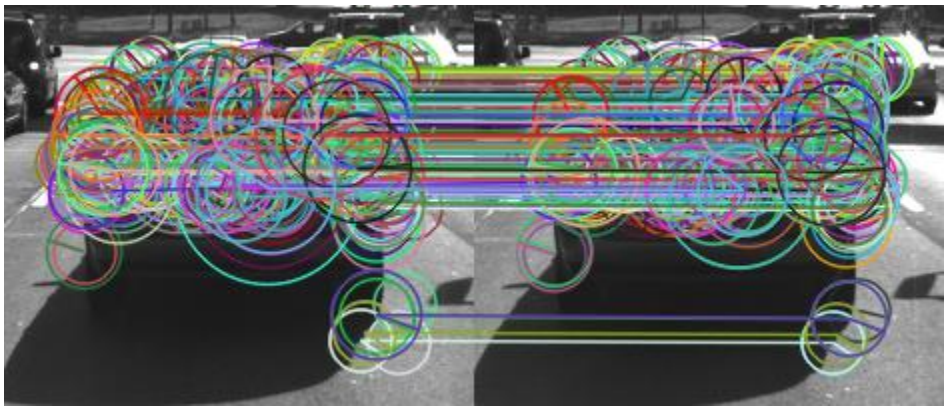
Frame 3-4



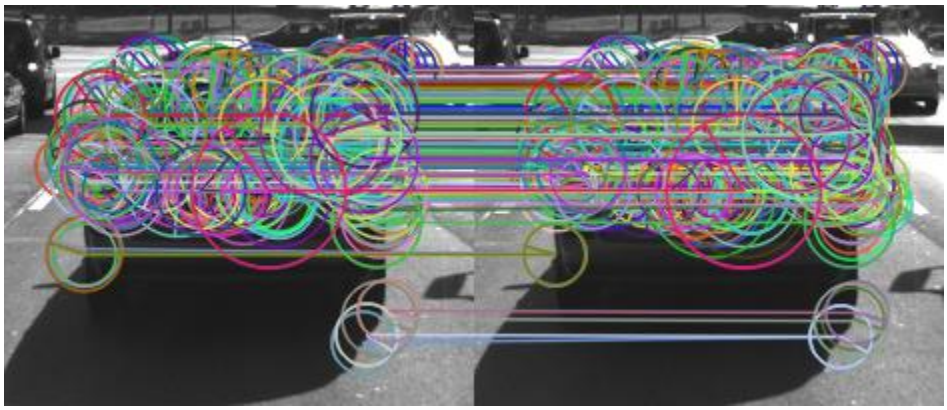
Frame 4-5



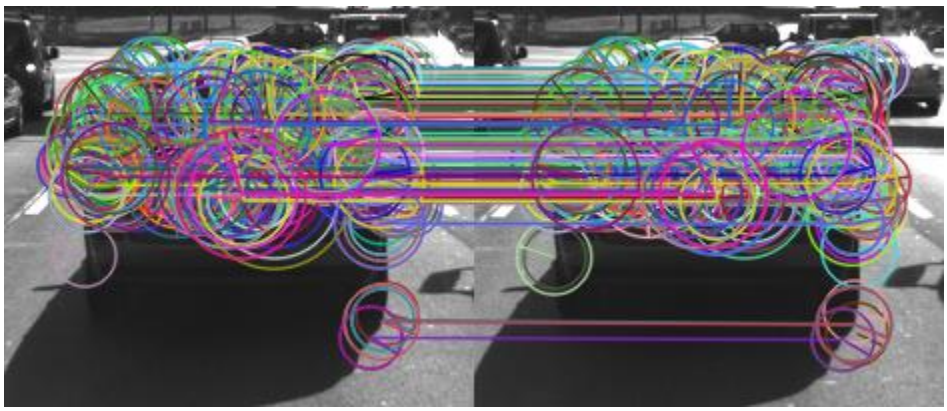
Frame 5-6



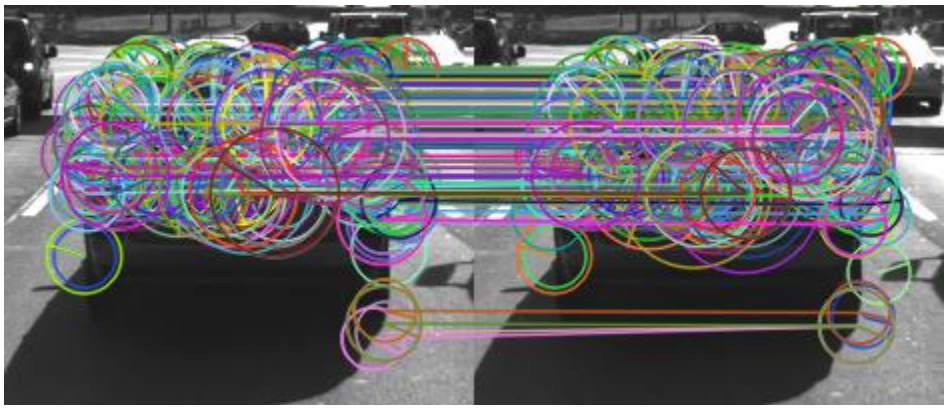
Frame 6-7



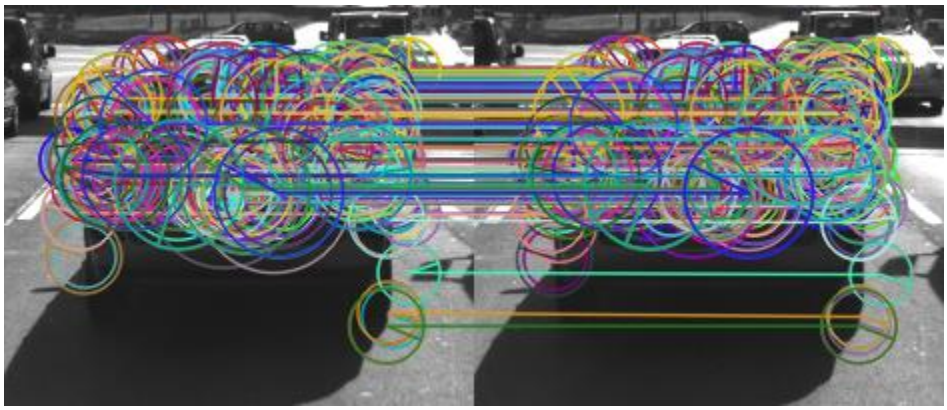
Frame 7-8



Frame 8-9

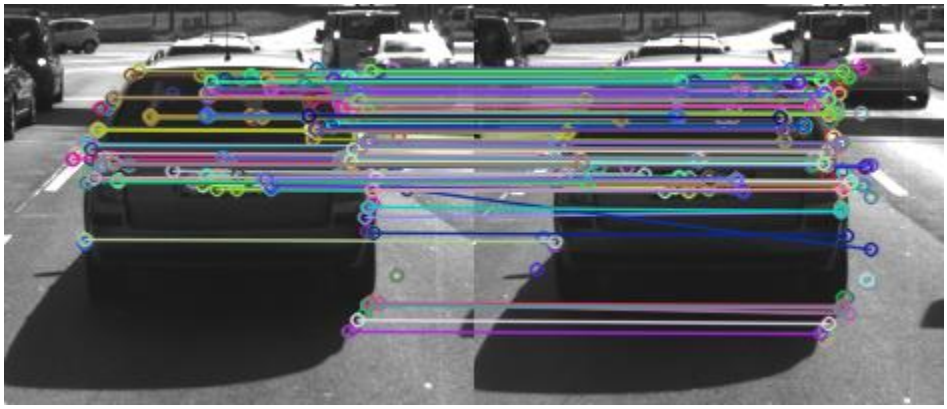


Frame 9-10

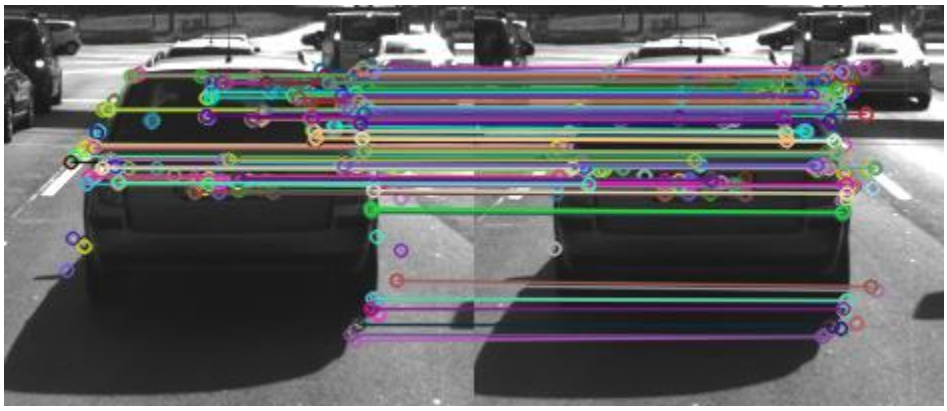


For easier viewing, here is ORB-BRIEF again with <u>default</u> keypoints:

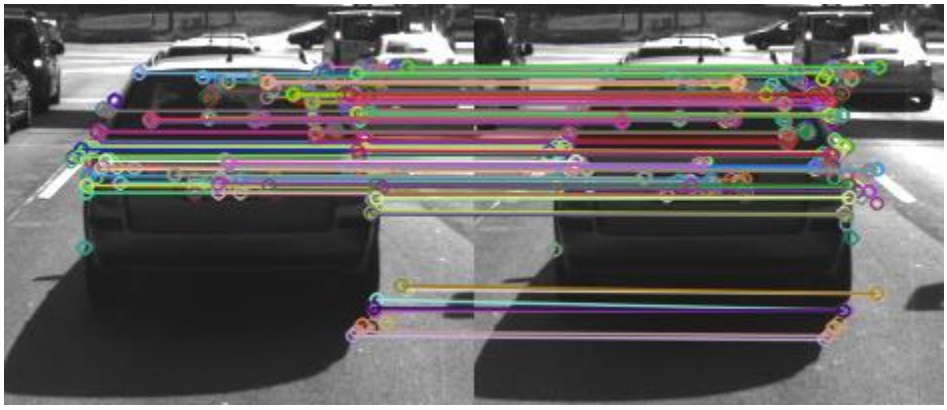
Frame 1-2



Frame 2-3

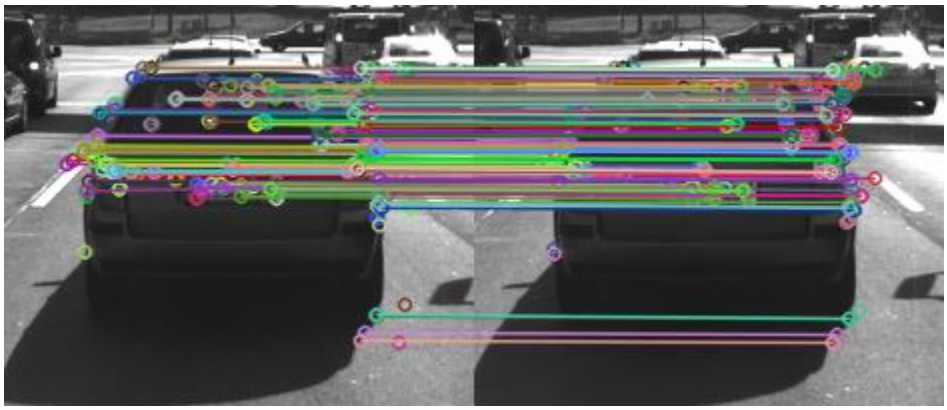


Frame 3-4



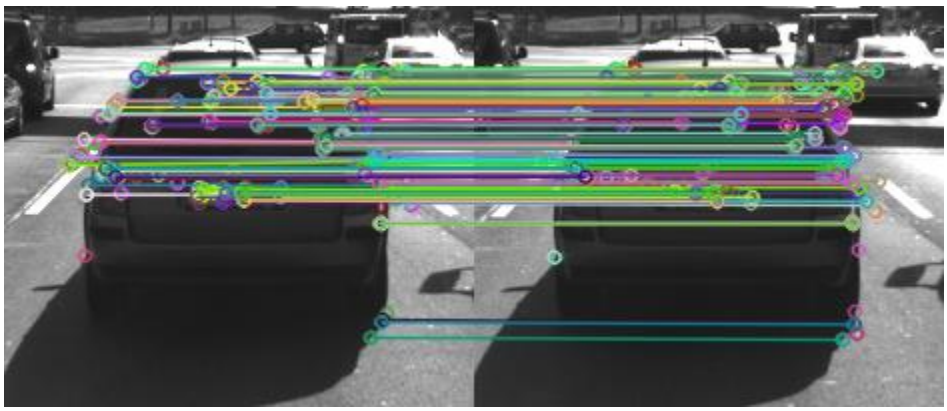
Frame 4-5

Frame 5-6



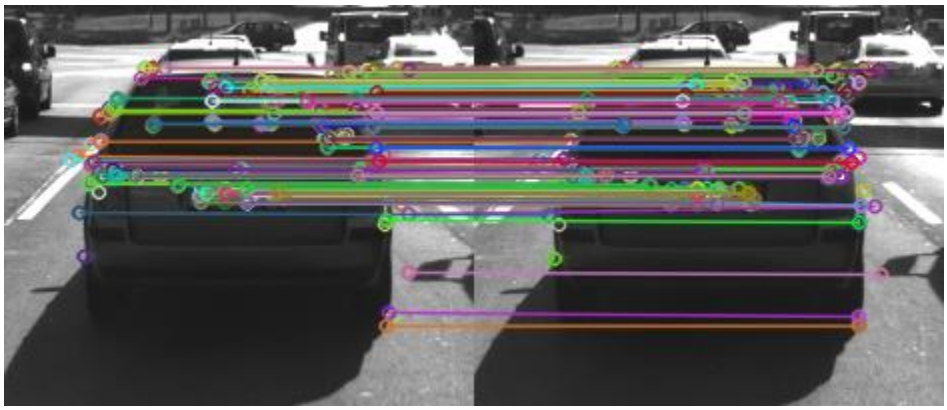
Frame 6-7

Frame 7-8



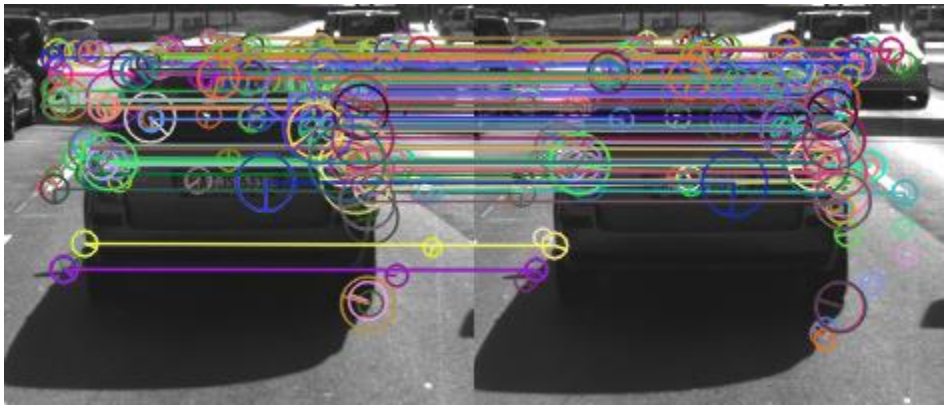
Frame 8-9

Frame 9-10

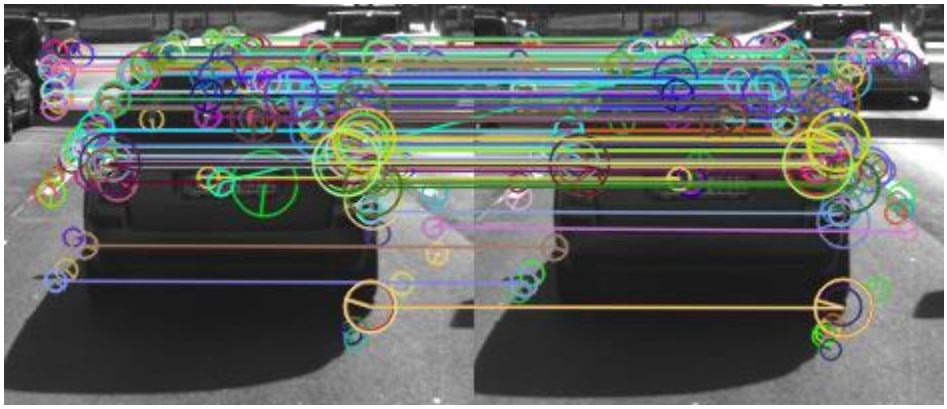


Tied for Third Place: BRISK-BRISK

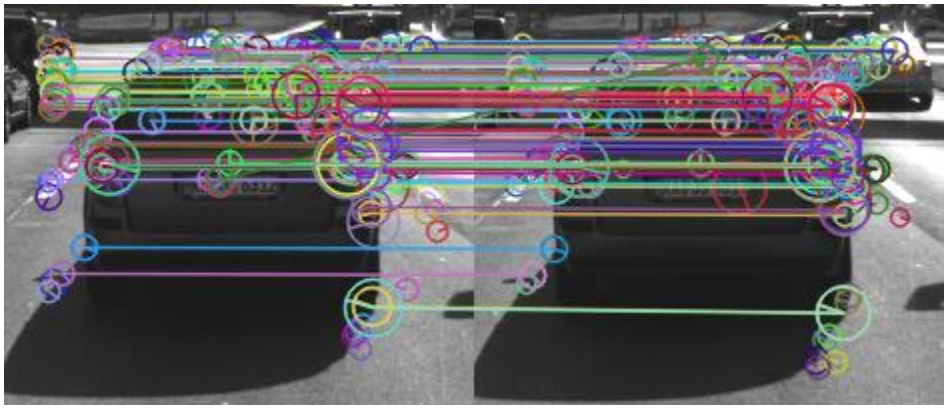
Frame 1-2



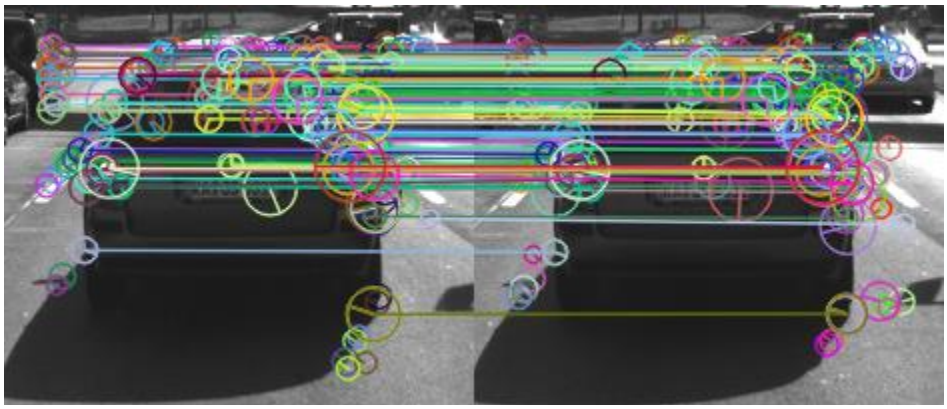
Frame 2-3



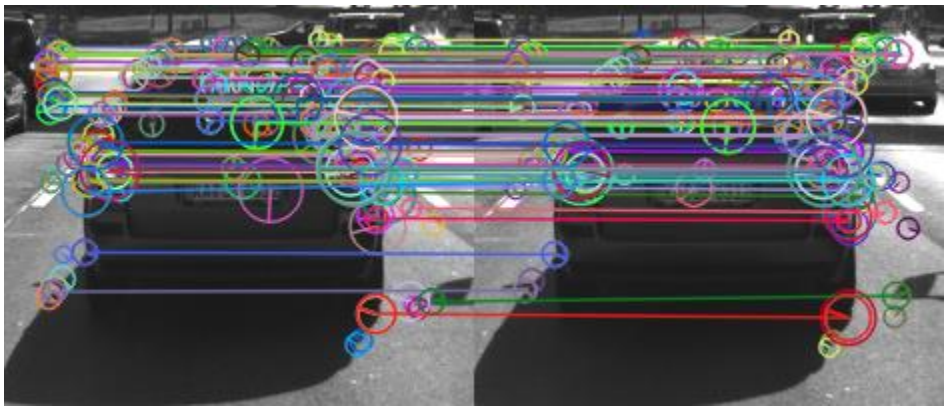
Frame 3-4



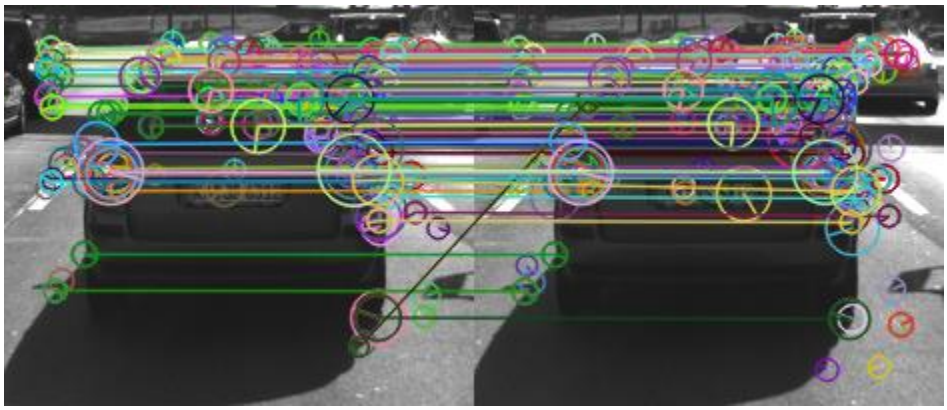
Frame 4-5



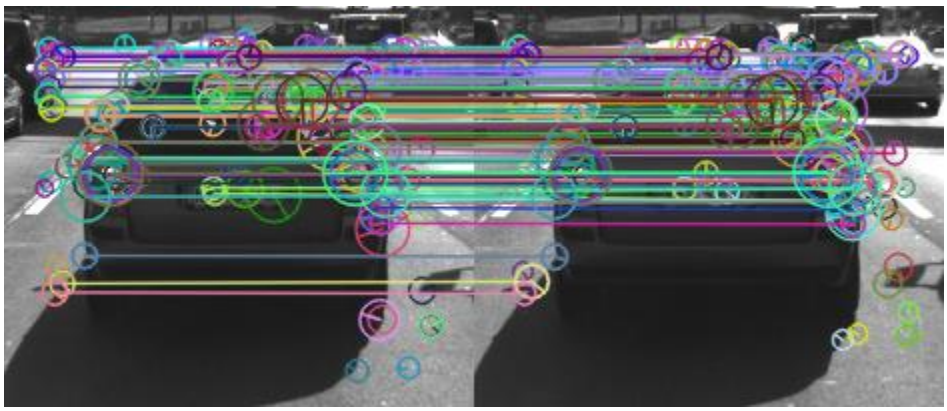
Frame 5-6



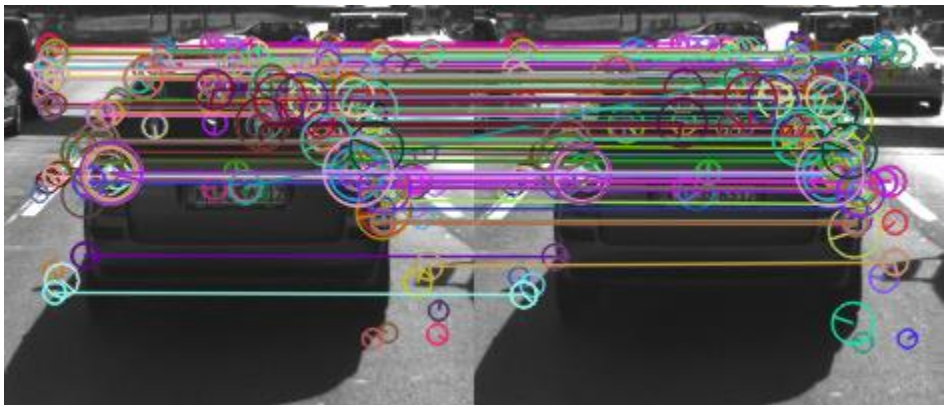
Frame 6-7



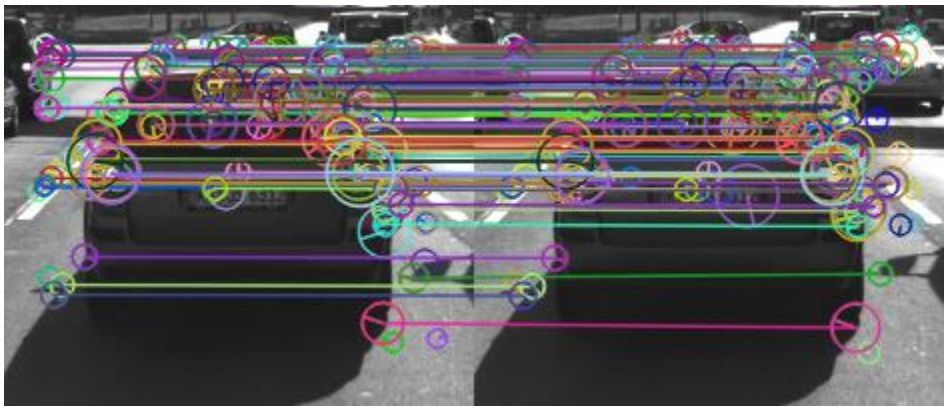
Frame 7-8



Frame 8-9

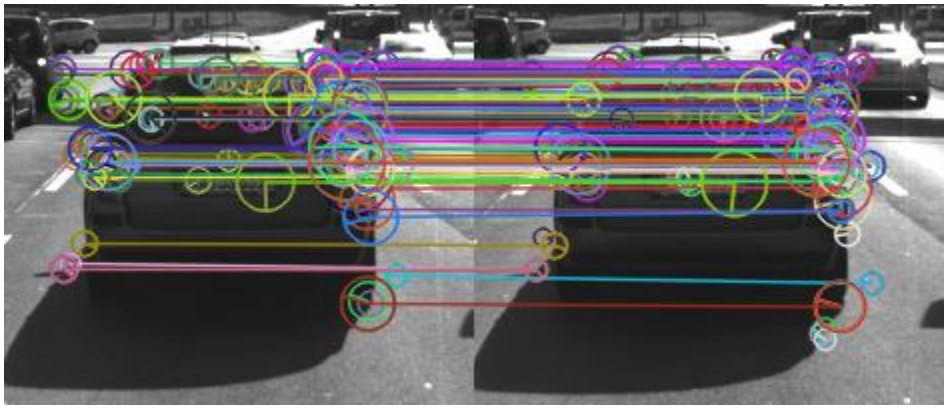


Frame 9-10

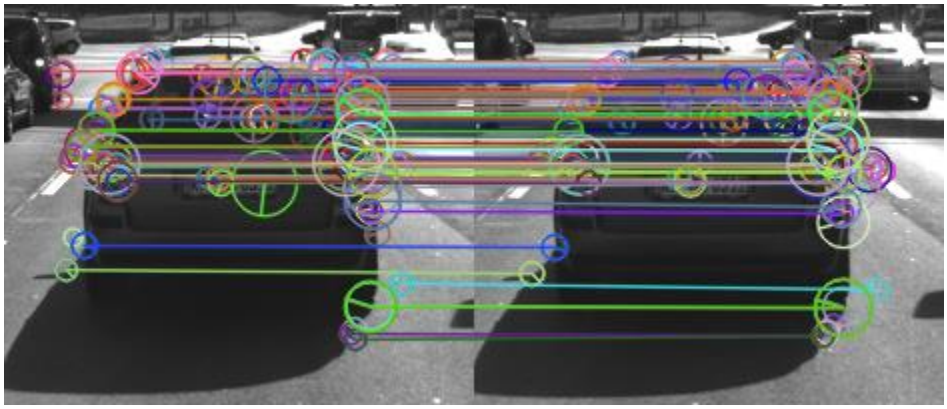


Tied for Third Place: BRISK-BRIEF

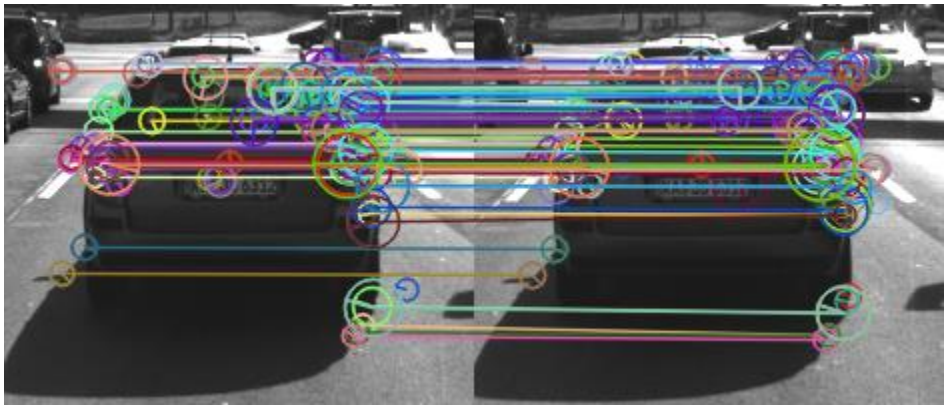
Frame 1-2

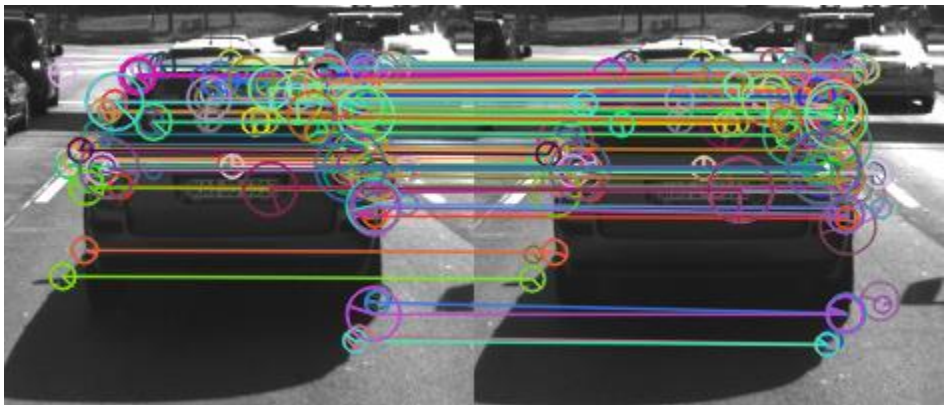


Frame 2-3

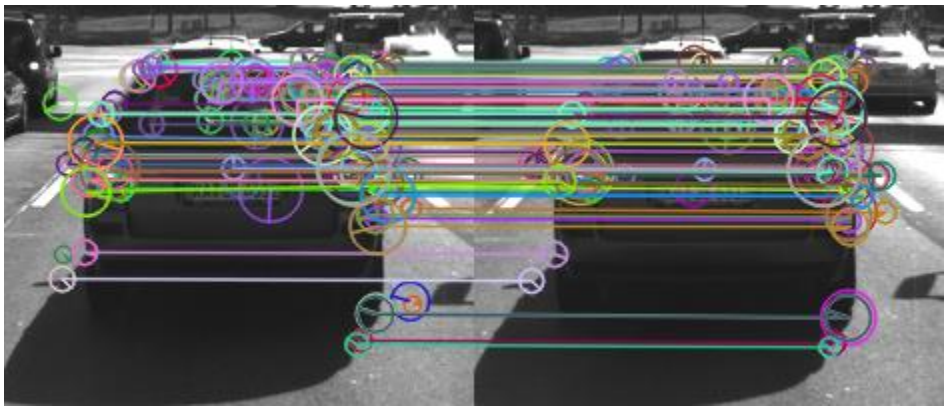


Frame 3-4



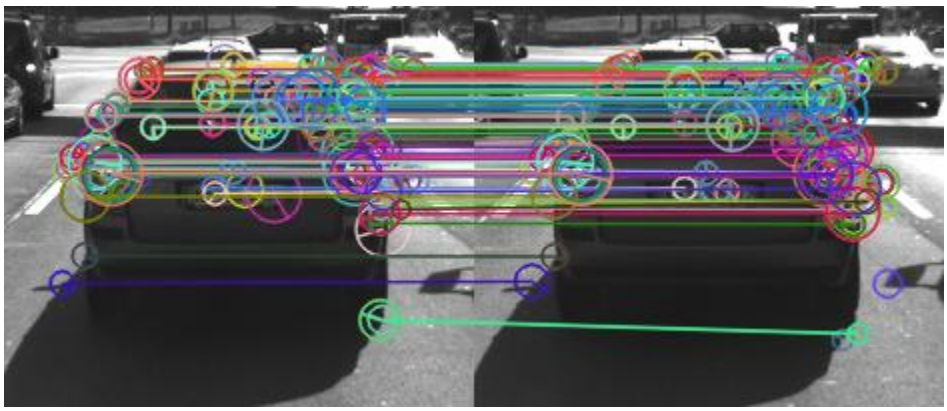


Frame 4-5



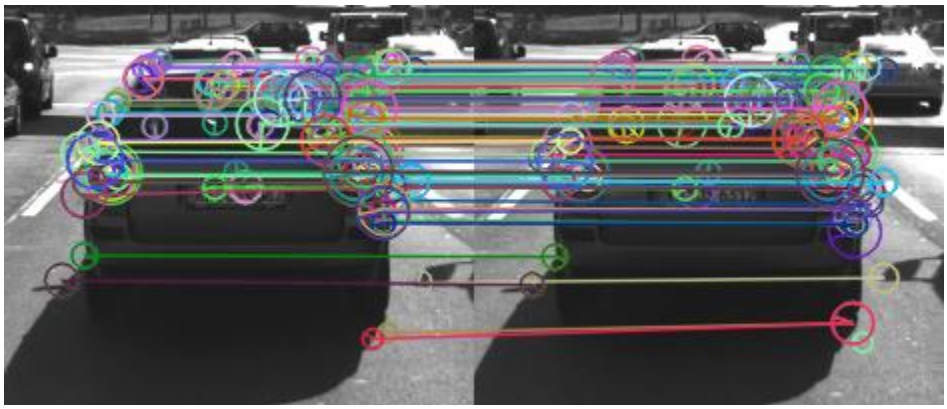
Frame 5-6

Frame 6-7

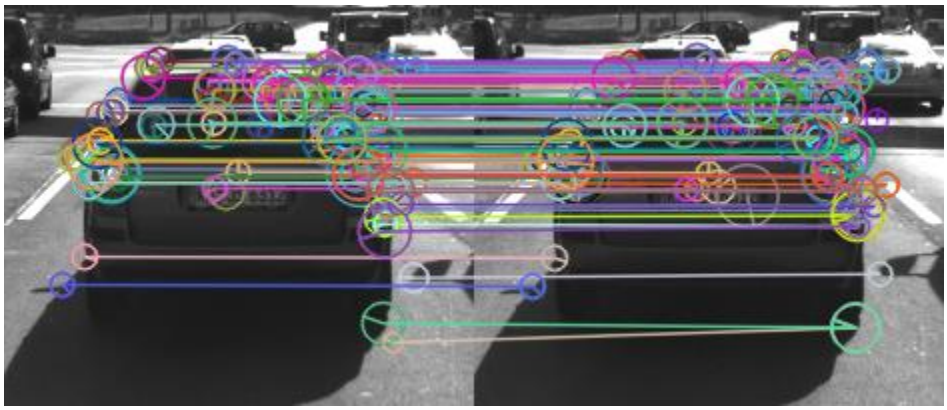


Frame 7-8

Frame 8-9

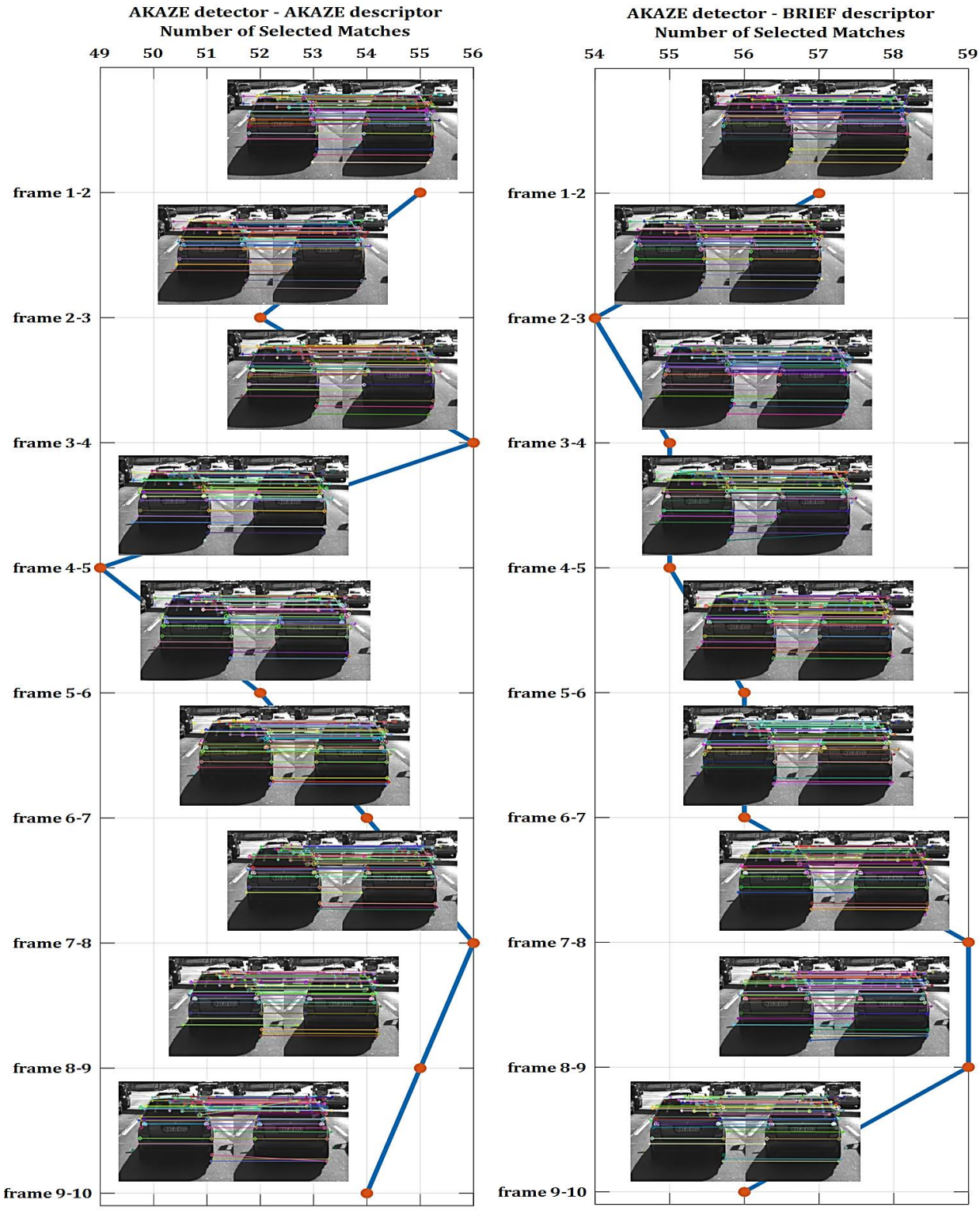


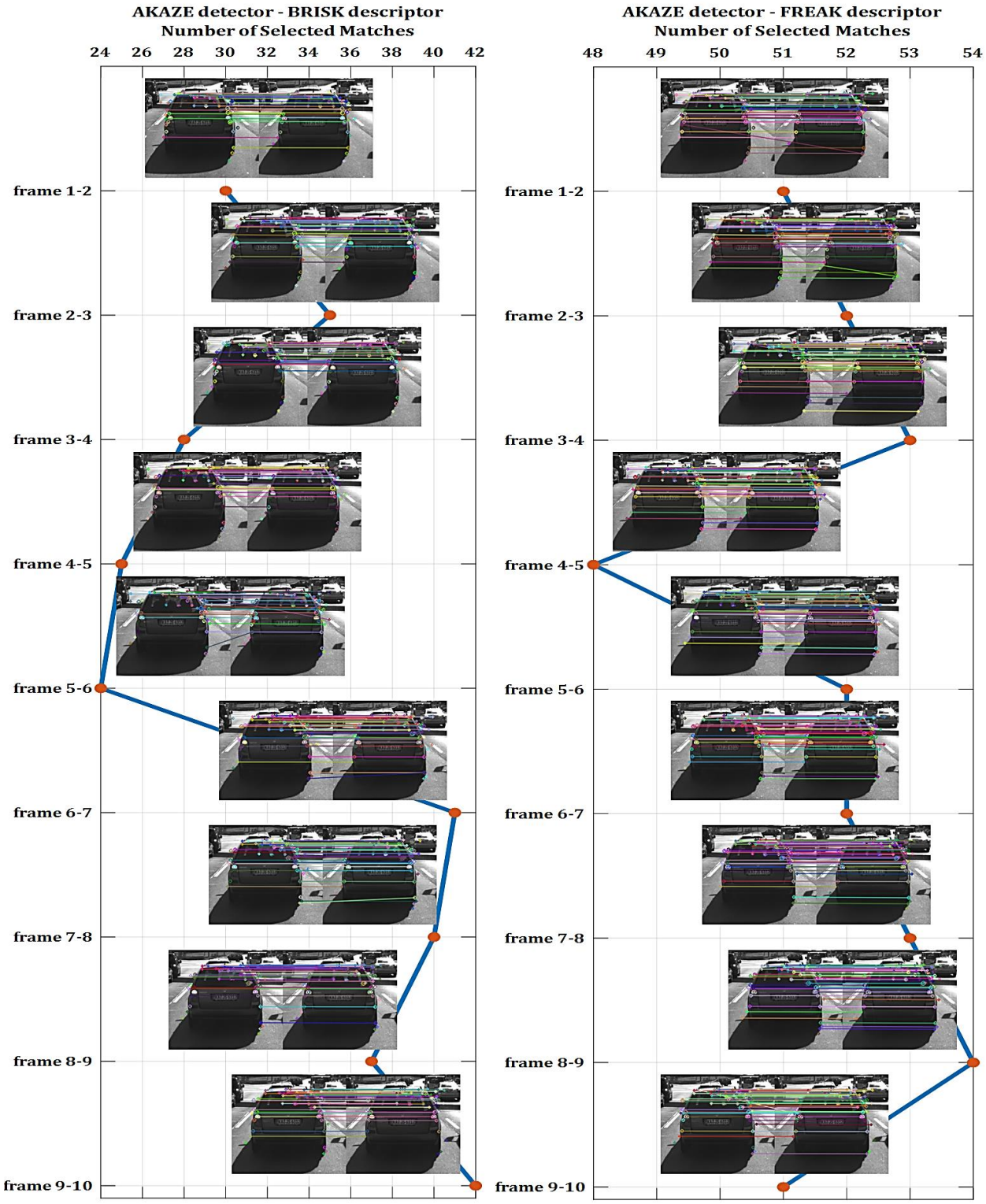
Frame 9-10

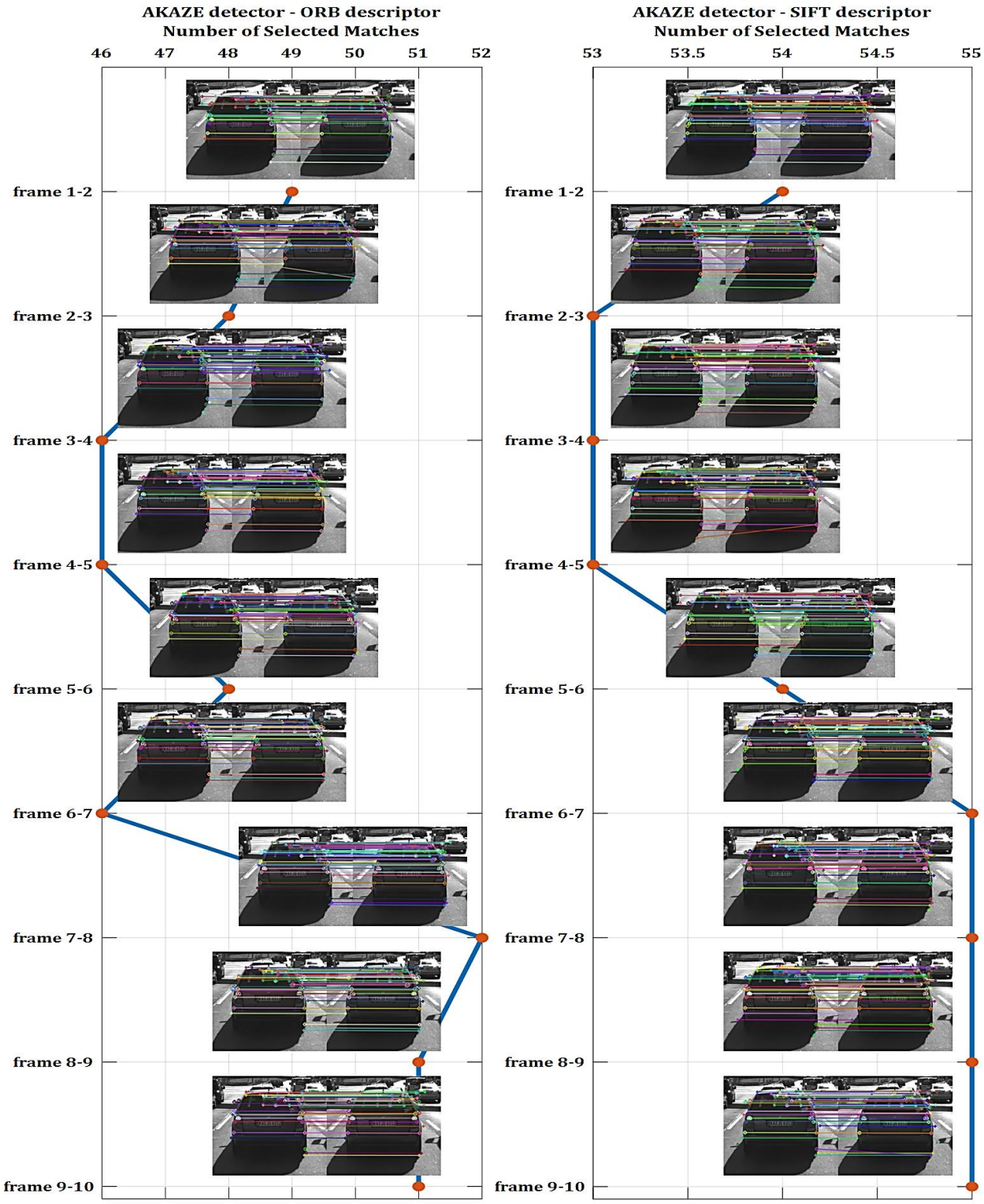


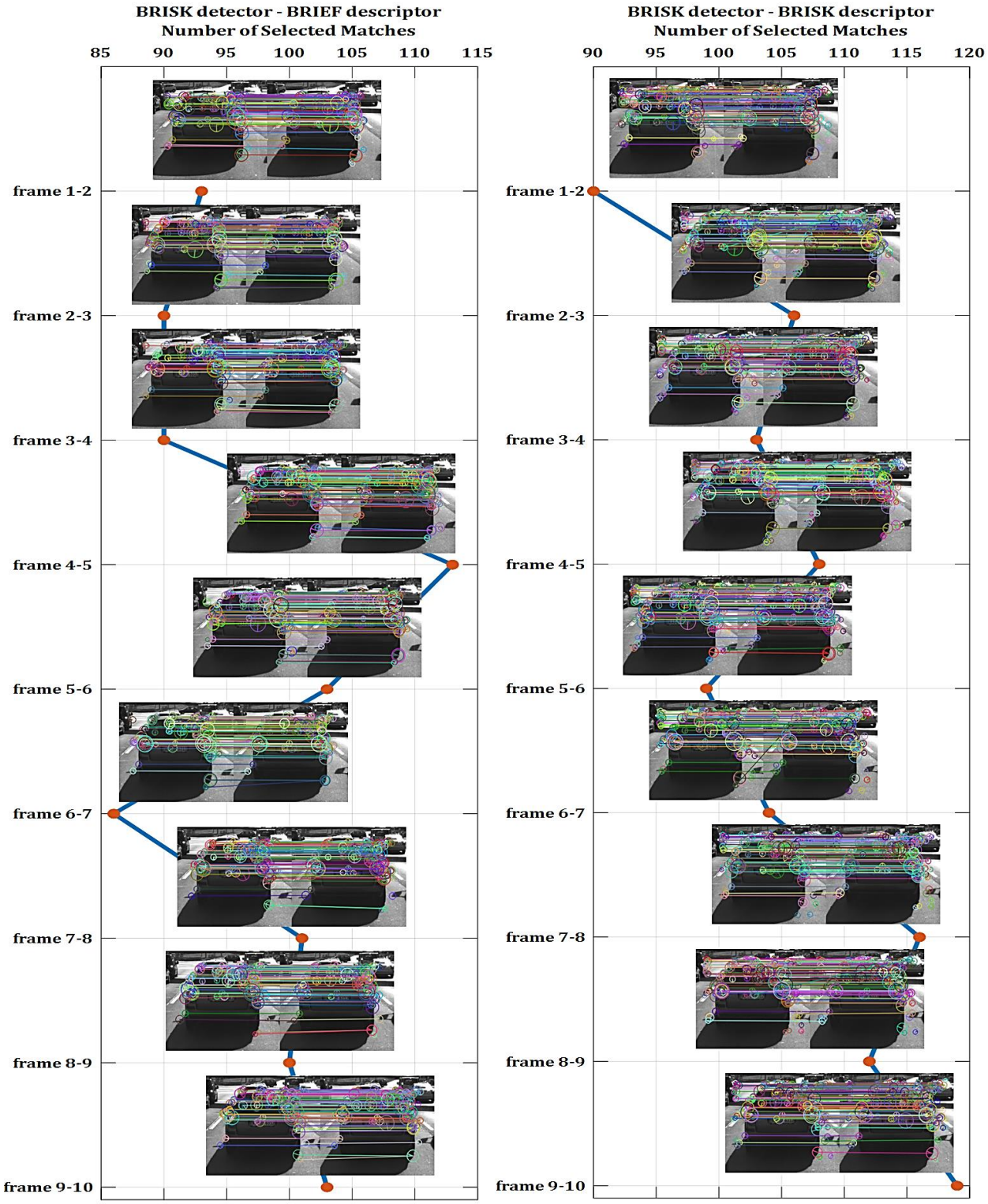
MP.10 APPENDIX

Saved images of keypoints and keypoint matches for all combinations of detectors and descriptors.




AKAZE detector - BRISK descriptor
Number of Selected Matches
24 26 28 30 32 34 36 38 40 42
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10
AKAZE detector - FREAK descriptor
Number of Selected Matches
48 49 50 51 52 53 54
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10


AKAZE detector - ORB descriptor
Number of Selected Matches
46
47
48
49
50
51
52
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10
AKAZE detector - SIFT descriptor
Number of Selected Matches
53
53.5
54
54.5
55
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10


BRISK detector - BRIEF descriptor
Number of Selected Matches
85
90
95
100
105
110
115
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10
BRISK detector - BRISK descriptor
Number of Selected Matches
90
95
100
105
110
115
120
frame 1-2
frame 2-3
frame 3-4
frame 4-5
frame 5-6
frame 6-7
frame 7-8
frame 8-9
frame 9-10