Camera Based 2D Feature Tracking

This document explains the implementation of Camera Based 2D Feature Tracking. The program is implemented in C++ with a choice to choose from multiple Feature Detectors (SHITOMASI, HARRIS, FAST, BRISK, ORB, AKAZE and SIFT) and Descriptors (BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT) of OpenCV library.

To the end of the document performance evaluations are shown with different combinations of Detectors and Descriptors and top three combinations are mentioned.

1. Data Buffer Optimization:

The below code snippet shows the implementation of a Data Buffer that hold a maximum of two data frames (**on MidTermProject_Camera_Student.cpp**). It is implemented using a vector, if the size of the vector is **"2"** and a new data frame must be added to the vector, its first element is removed and then the new data frame is pushed at the back of the vector.

```
//// STUDENT ASSIGNMENT
//// TASK MP.1 -> replace the following code with ring buffer of size dataBufferSize

// push image into data frame buffer
DataFrame frame;
frame.cameraImg = imgGray;
if (dataBuffer.size() >= dataBufferSize)
{
    dataBuffer.erase(dataBuffer.begin());
}
dataBuffer.push_back(frame);

//// EOF STUDENT ASSIGNMENT
```

2. Keypoint Detection:

Multiple Keypoint detectors are implemented using the OpenCV library, namely SHITOMASI, HARRIS, FAST, BRISK, ORB, AKAZE and SIFT. Required detector can be specified as a string and the program uses the detector. The below code snippet shows the implementation that allows for choosing the detector needed as a string (**on MidTermProject_Camera_Student.cpp**).

```cpp
// extract 2D keypoints from current image
vector<cv::KeyPoint> keypoints; // create empty feature list for current image
string detectorType = "SHITOMASI";
// string detectorType = "HARRIS";
// string detectorType = "FAST";
// string detectorType = "BRISK";
// string detectorType = "ORB";
// string detectorType = "AKAZE";
// string detectorType = "SIFT";

//// STUDENT ASSIGNMENT
//// TASK MP.2 -> add the following keypoint detectors in file matching2D.cpp and enable string-based selection
//// -> HARRIS, FAST, BRISK, ORB, AKAZE, SIFT

if (detectorType.compare("SHITOMASI") == 0)
{
    detKeypointsShiTomasi(keypoints, imgGray, false);
}
else if (detectorType.compare("HARRIS") == 0)
{
    detKeypointsHarris(keypoints, imgGray, false);
}
else
{
    detKeypointsModern(keypoints, imgGray, detectorType, false);
}
//// EOF STUDENT ASSIGNMENT
```

The function "**detKeypointsShiTomasi**" (**on matching2D_Student.cpp**) implements the ShiTomasi detector. Below is its implementation.

```cpp
void detKeypointsShiTomasi(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)
{
    // compute detector parameters based on image size
    int blockSize = 4;       //  size of an average block for computing a derivative covariation matrix over each pixel
    double maxOverlap = 0.0; // max. permissible overlap between two features in %
    double minDistance = (1.0 - maxOverlap) * blockSize;
    int maxCorners = img.rows * img.cols / max(1.0, minDistance); // max. num. of keypoints

    double qualityLevel = 0.01; // minimal accepted quality of image corners
    double k = 0.04;

    // Apply corner detection
    double t = (double)cv::getTickCount();
    vector<cv::Point2f> corners;
    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);
    }
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << "Shi-Tomasi detection with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms" << endl;
```

The function "**detKeypointsHarris**" (**on matching2D_Student.cpp**) implements the Harris detector. Below is its implementation.

```cpp
// Detect keypoints in image using HARRIS detector
void detKeypointsHarris(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)
{
    // Harris Dector parameters
    int blockSize = 2; // for each pixel a blockSize*blockSIze neighbourhood is considered
    int apertureSize = 3; // aperture paameter for Sobel operator
    int minResponse = 100; // minimum value for a corner in the 8 bit scaled response matrix
    double k = 0.04; // Harris parameter

    double maxOverlap = 0.0; // max. permissible overlap between two features in %

    // Apply corner detection
    double t = (double)cv::getTickCount();
    cv::Mat dst, dst_norm, dst_norm_scaled;
    dst = cv::Mat::zeros(img.size(), CV_32FC1);
    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);

    for (int j = 0; j < dst_norm.rows; j++)
    {
        for (int i = 0; i < dst_norm.cols; i++)
        {
            int response = (int) dst_norm.at<float>(j, i);
            if (response > minResponse)
            {
                cv::KeyPoint newKeyPoint;
                newKeyPoint.pt = cv::Point2f(i, j);
                newKeyPoint.size = 2 * apertureSize;
                newKeyPoint.response = response;
                keypoints.push_back(newKeyPoint);
            }
        }
    }
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << "Harris detection with n=" << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms" << endl;
```

The function "**detKeypointsModern**" (**on matching2D_Student.cpp**) implements the FAST, BRISK, ORB, AKAZE and SIFT detectors. They are implemented with default built-in parameters. Below is its implementation.

```cpp
// Detect Keypoints using modern detectors, namely FAST, BRISK, ORB, AKAZE, SIFT
void detKeypointsModern(std::vector<cv::KeyPoint> &keypoints, cv::Mat &img, std::string detectorType, bool bVis)
{
    cv::Ptr<cv::FeatureDetector> detector;
    double t = (double)cv::getTickCount();

    if (detectorType.compare("FAST") == 0)
    {
        // FAST detector with default parameters
        detector = cv::FastFeatureDetector::create();
    }
    else if (detectorType.compare("BRISK") == 0)
    {
        // BRISK detector with default parameters
        detector = cv::BRISK::create();
    }
    else if (detectorType.compare("ORB") == 0)
    {
        // ORB detector with default parameters
        detector = cv::ORB::create();
    }
    else if (detectorType.compare("AKAZE") == 0)
    {
        // AKAZE detector with default parameters
        detector = cv::AKAZE::create();
    }
    else if (detectorType.compare("SIFT") == 0)
    {
        // SIFT detector with default parameters
        detector = cv::xfeatures2d::SIFT::create();
    }
    else
    {
        throw std::invalid_argument("Wrong detector type supplied. Supported type are FAST, BRISK, ORB, AKAZE, SIFT");
    }

    detector->detect(img, keypoints);
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << detectorType.append(" detection with n=") << keypoints.size() << " keypoints in " << 1000 * t / 1.0 << " ms" << endl;
```

3. Keypoint Removal:

Out of the Keypoints extracted from the choice of detector, we only need the Keypoints on the preceding vehicle. So, an OpenCV's rectangle is defined and any keypoints outside the rectangle are removed from the vector of keypoints.

```
//// STUDENT ASSIGNMENT
//// TASK MP.3 -> only keep keypoints on the preceding vehicle

// only keep keypoints on the preceding vehicle
bool bFocusOnVehicle = true;
cv::Rect vehicleRect(535, 180, 180, 150);
if (bFocusOnVehicle)
{
    for (auto it = keypoints.begin(); it < keypoints.end(); it++)
    {
        if(!(vehicleRect.contains((*it).pt)))
        {
            keypoints.erase(it);
        }
    }
}
```

4. Keypoint Descriptor:

BRISK, BRIEF, ORB, FREAK, AKAZE and SIFT Detectors are implemented with a facility to choose the descriptor as a string (as in case of detectors). Below is the code snippet that shows the string variable which helps choose the descriptor.

```
//// STUDENT ASSIGNMENT
//// TASK MP.4 -> add the following descriptors in file matching2D.cpp and enable string-based selection based on descriptorType
//// -> BRIEF, ORB, FREAK, AKAZE, SIFT

cv::Mat descriptors;
string descriptorType = "BRISK"; // BRIEF, ORB, FREAK, AKAZE, SIFT
// string descriptorType = "BRIEF";
// string descriptorType = "ORB";
// string descriptorType = "FREAK";
// string descriptorType = "AKAZE";
// string descriptorType = "SIFT";
descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg, descriptors, descriptorType);
//// EOF STUDENT ASSIGNMENT
```

Below shows the implementation of the descriptors using OpenCV implemented in the function **"descKeypoints"** (**on matching2D_Student.cpp**). All use the default built in OpenCV parameters.

```cpp
void descKeypoints(vector<cv::KeyPoint> &keypoints, cv::Mat &img, cv::Mat &descriptors, string descriptorType)
{
    // select appropriate descriptor
    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; // apply this scale to the pattern used for sampling the neighbourhood of a keypoint.

        extractor = cv::BRISK::create(threshold, octaves, patternScale);
    }
    else if (descriptorType.compare("BRIEF") == 0)
    {
        // BRIEF descriptor with default parameters
        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();
    }
    else if (descriptorType.compare("ORB") == 0)
    {
        // ORB descriptor with default parameters
        extractor = cv::ORB::create();
    }
    else if (descriptorType.compare("FREAK") == 0)
    {
        // FREAK descriptor with default parameters
        extractor = cv::xfeatures2d::FREAK::create();
    }
    else if (descriptorType.compare("AKAZE") == 0)
    {
        // AKAZE descriptor with default parameters
        extractor = cv::AKAZE::create();
    }
    else if (descriptorType.compare("SIFT") == 0)
    {
        // SIFT descriptor with default parameters
        extractor = cv::xfeatures2d::SIFT::create();
    }
    else
    {
        throw std::invalid_argument("Wrong descriptor type supplied. Supported type are BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT");
    }

    // perform feature description
    double t = (double)cv::getTickCount();
    extractor->compute(img, keypoints, descriptors);
```

5. Descriptor Matching:

FLANN matching is implemented. Using a string variable FLANN or Brute Force matching can be chosen. FLANN is implemented in the function **“matchDescriptors”** (**on matching2D_Student.cpp**).

```cpp
/* MATCH KEYPOINT DESCRIPTORS */

vector<cv::DMatch> matches;
// string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN
string matcherType = "MAT_FLANN";
string descriptorType = "DES_BINARY"; // DES_BINARY, DES_HOG
// string descriptorType = "DES_HOG";
// string selectorType = "SEL_NN";       // SEL_NN, SEL_KNN
string selectorType = "SEL_KNN";

//// STUDENT ASSIGNMENT
//// TASK MP.5 -> add FLANN matching in file matching2D.cpp
//// TASK MP.6 -> add KNN match selection and perform descriptor distance ratio filtering with t=0.8 in file matching2D.cpp

matchDescriptors((dataBuffer.end() - 2)->keypoints, (dataBuffer.end() - 1)->keypoints,
                 (dataBuffer.end() - 2)->descriptors, (dataBuffer.end() - 1)->descriptors,
                 matches, descriptorType, matcherType, selectorType);

//// EOF STUDENT ASSIGNMENT
```

```cpp
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 matcher
    bool crossCheck = false;
    cv::Ptr<cv::DescriptorMatcher> matcher;

    if (matcherType.compare("MAT_BF") == 0)
    {
        int normType = cv::NORM_HAMMING;
        if (descriptorType.compare("DES_BINARY") == 0)
        {
            normType = cv::NORM_HAMMING;
        }
        else {
            normType = cv::NORM_L2;
        }
        matcher = cv::BFMatcher::create(normType, crossCheck);
    }
    else if (matcherType.compare("MAT_FLANN") == 0)
    {
        if (descSource.type() != CV_32F)
        { // convert descSource to float
            descSource.convertTo(descSource, CV_32F);
        }

        if (descRef.type() != CV_32F)
        { // convert descRef to float
            descRef.convertTo(descRef, CV_32F);
        }
        matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
    }
```

6. Descriptor Distance Ratio:

K-NN matching is implemented with number of nearest neighbors as two (k = 2). A string variable can used to perform either the Nearest Neighbor or K-Nearest Neighbor matching. K-NN is implemented in the function **“matchDescriptors”** (**on matching2D_Student.cpp**).

```cpp
/* MATCH KEYPOINT DESCRIPTORS */

vector<cv::DMatch> matches;
// string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN
string matcherType = "MAT_FLANN";
string descriptorType = "DES_BINARY"; // DES_BINARY, DES_HOG
// string descriptorType = "DES_HOG";
// string selectorType = "SEL_NN";       // SEL_NN, SEL_KNN
string selectorType = "SEL_KNN";

//// STUDENT ASSIGNMENT
//// TASK MP.5 -> add FLANN matching in file matching2D.cpp
//// TASK MP.6 -> add KNN match selection and perform descriptor distance ratio filtering with t=0.8 in file matching2D.cpp

matchDescriptors((dataBuffer.end() - 2)->keypoints, (dataBuffer.end() - 1)->keypoints,
                 (dataBuffer.end() - 2)->descriptors, (dataBuffer.end() - 1)->descriptors,
                 matches, descriptorType, matcherType, selectorType);

//// EOF STUDENT ASSIGNMENT
```

```
// perform matching task
if (selectorType.compare("SEL_NN") == 0)
{ // nearest neighbor (best match)
    double t = (double)cv::getTickCount();
    matcher->match(descSource, descRef, matches); // Finds the best match for each descriptor in desc1
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << " SEL_NN with number of matches = " << matches.size() << " found in " << 1000 * t / 1.0 << " ms" << endl;
}
else if (selectorType.compare("SEL_KNN") == 0)
{ // k nearest neighbors (k=2)
    double minDescDistRatio = 0.8;

    double t = (double)cv::getTickCount();

    vector<vector<cv::DMatch>> knn_matches;
    matcher->knnMatch(descSource, descRef, knn_matches, 2);

    for (auto it = knn_matches.begin(); it != knn_matches.end(); it++)
    {
        if ((*it)[0].distance < minDescDistRatio * (*it)[1].distance)
        {
            matches.push_back((*it)[0]);
        }
    }
    cout << "Number of matches removed: " << knn_matches.size() - matches.size() << endl;

    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << " SEL_KNN with number of matches = " << matches.size() << " found in " << 1000 * t / 1.0 << " ms" << endl;
}
```

7. Performance Evaluation 1:

Number of Keypoints detected using each detector and the size of their neighborhood is tabulated on all the ten images. Below is the table.

Number Of KeyPoints Detected and Neighborhood Size for Each Detector														
Image	SHITOMASSI		HARRIS		FAST		BRISK		ORB		AKAZE		SIFT	
	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood	No. of Keypoints	Neighborhood
Image 1	715	4	190	6	2711	7	1472	12.2	280	23.8	727	3.2	771	7.6
Image 2	680	4	162	6	2660	7	1494	12.6	277	23.6	710	3.1	731	7.7
Image 3	716	4	201	6	2606	7	1480	12.2	280	23.5	705	3	734	5.4
Image 4	705	4	202	6	2604	7	1467	12	282	23.8	724	3	714	5.6
Image 5	699	4	299	6	2587	7	1487	12.5	284	23.6	733	3.1	702	5.1
Image 6	672	4	1454	6	2622	7	1450	12.8	288	23.8	724	3.1	741	5.7
Image 7	690	4	115	6	2611	7	1463	12.4	288	23.6	736	3.2	749	7.4
Image 8	711	4	461	6	2609	7	1412	12.6	289	23.6	720	3.2	745	7.1
Image 9	722	4	328	6	2663	7	1417	12.6	287	23.8	736	3.1	791	6.6
Image 10	698	4	823	6	2669	7	1429	12.2	292	23.6	725	3.1	757	6.1
Average	700.8	4	423.5	6	2634.2	7	1457.1	12.41	284.7	23.67	724	3.11	743.5	6.43

As can be seen from the above table, the BRISK Detector detects highest number of keypoints, whereas ORB detects the least.

Also, the size of neighborhood is constant for ShoTomasi, HARRIS and FAST Detectors with a value of 4, 6, 7 respectively. BRISK, ORB, AKAZE and SIFT detectors have neighborhood of 12.41, 23.67, 3.11, 6.43 respectively. Hence, AKAZE has the least neighborhood.

8. Performance Evaluation 2:

Here the number of keypoints matched for the keypoints found (as shown in the above table in section 7). The below are the tabulations for all combinations of Detectors and Descriptors.

Also, here FLANN matching is used with K-NN with number of nearest neighbors as 2 and descriptor distance ratio is set to 0.8.

Number of Matched Keypoints for Detector: SHITOMASSI							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	191	283	252	162	Cannot be computed	353
Image 2	Image 3	193	267	231	181	Cannot be computed	369
Image 3	Image 4	211	266	263	185	Cannot be computed	370
Image 4	Image 5	195	265	245	164	Cannot be computed	354
Image 5	Image 6	197	302	256	161	Cannot be computed	373
Image 6	Image 7	198	276	233	169	Cannot be computed	357
Image 7	Image 8	204	260	243	176	Cannot be computed	371
Image 8	Image 9	202	284	261	176	Cannot be computed	378
Image 9	Image 10	206	285	236	146	Cannot be computed	360
Average		199.6667	276.4444	246.6666667	168.8889	Cannot be computed	365

Number of Matched Keypoints for Detector: HARRIS							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	66	77	77	56	Cannot be computed	99
Image 2	Image 3	66	80	76	63	Cannot be computed	86
Image 3	Image 4	84	98	96	68	Cannot be computed	110
Image 4	Image 5	83	109	94	64	Cannot be computed	99
Image 5	Image 6	128	139	141	113	Cannot be computed	131
Image 6	Image 7	184	282	275	171	Cannot be computed	398
Image 7	Image 8	59	49	49	46	Cannot be computed	53
Image 8	Image 9	186	218	210	138	Cannot be computed	255
Image 9	Image 10	155	182	166	134	Cannot be computed	181
Average		112.3333	137.1111	131.5555556	94.77778	Cannot be computed	156.8889

Number of Matched Keypoints for Detector: FAST							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	609	944	788	539	Cannot be computed	1729
Image 2	Image 3	642	970	798	545	Cannot be computed	1695
Image 3	Image 4	581	985	806	557	Cannot be computed	1667
Image 4	Image 5	644	968	788	545	Cannot be computed	1672
Image 5	Image 6	650	940	807	560	Cannot be computed	1642
Image 6	Image 7	626	919	790	549	Cannot be computed	1649
Image 7	Image 8	617	969	808	534	Cannot be computed	1669
Image 8	Image 9	595	976	797	587	Cannot be computed	1644
Image 9	Image 10	647	930	780	540	Cannot be computed	1659
Average		623.4444	955.6667	795.7777778	550.6667	Cannot be computed	1669.556

Number of Matched Keypoints for Detector: BRISK							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	443	605	341	367	Cannot be computed	810
Image 2	Image 3	423	635	320	361	Cannot be computed	838
Image 3	Image 4	433	590	334	351	Cannot be computed	786
Image 4	Image 5	453	593	341	345	Cannot be computed	824
Image 5	Image 6	424	600	323	362	Cannot be computed	817
Image 6	Image 7	415	582	367	367	Cannot be computed	819
Image 7	Image 8	416	603	328	356	Cannot be computed	820
Image 8	Image 9	405	588	302	365	Cannot be computed	754
Image 9	Image 10	410	553	305	369	Cannot be computed	748
Average		424.6667	594.3333	329	360.3333	Cannot be computed	801.7778

Number of Matched Keypoints for Detector: ORB							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	123	119	101	61	Cannot be computed	174
Image 2	Image 3	141	116	113	54	Cannot be computed	177
Image 3	Image 4	119	114	104	50	Cannot be computed	177
Image 4	Image 5	123	121	97	60	Cannot be computed	165
Image 5	Image 6	129	119	104	50	Cannot be computed	170
Image 6	Image 7	137	125	111	58	Cannot be computed	186
Image 7	Image 8	132	132	115	59	Cannot be computed	188
Image 8	Image 9	129	128	92	54	Cannot be computed	189
Image 9	Image 10	117	123	108	58	Cannot be computed	177
Average		127.7778	121.8889	105	56	Cannot be computed	178.1111

Number of Matched Keypoints for Detector: AKAZE							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	316	359	270	251	345	394
Image 2	Image 3	288	360	240	262	328	404
Image 3	Image 4	314	335	235	235	344	411
Image 4	Image 5	287	330	223	240	311	402
Image 5	Image 6	297	339	221	239	337	410
Image 6	Image 7	316	377	239	256	345	419
Image 7	Image 8	311	367	224	281	323	395
Image 8	Image 9	327	386	262	268	342	430
Image 9	Image 10	311	369	272	270	331	413
Average		307.4444	358	242.8888889	255.7778	334	408.6667

Number of Matched Keypoints for Detector: SIFT							
		Descriptor Used					
Image Number	Image Number	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
Image 1	Image 2	236	272	Cannot be computed	217	Cannot be computed	292
Image 2	Image 3	241	277	Cannot be computed	210	Cannot be computed	289
Image 3	Image 4	237	287	Cannot be computed	204	Cannot be computed	293
Image 4	Image 5	237	280	Cannot be computed	202	Cannot be computed	282
Image 5	Image 6	232	262	Cannot be computed	189	Cannot be computed	286
Image 6	Image 7	219	277	Cannot be computed	177	Cannot be computed	268
Image 7	Image 8	243	262	Cannot be computed	178	Cannot be computed	294
Image 8	Image 9	251	279	Cannot be computed	194	Cannot be computed	302
Image 9	Image 10	259	323	Cannot be computed	202	Cannot be computed	322
Average		239.4444	279.8889	Cannot be computed	197	Cannot be computed	292

9. Performance Evaluation 3:

The time taken for detection, description and matching in milliseconds are logged in a table, as shown below. The average of number of Keypoints detected and matched by their description are considered (**from section 7 and 8 tabulations**) in the table and percentage of match is also tabulated.

Percentage of Keypoints Matched and Total Execution Time in Milliseconds.																				
Detector / Descriptor		BRISK			BRIEF			ORB			FREAK			AKAZE			SIFT		
	Detected Count	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)	Matched Count	Percentage Match	Total time Detector +Descriptor+Matching (ms)
SHITOMASSI	700	199	28.428571	39.5	276	39.428571	32.49	246	35.142857	31.7	168	24	82.7	Cannot be computed	Cannot be computed	Cannot be computed	365	52.142857	68.9
HARRIS	423	112	26.477541	29.5	137	32.387707	27.8	131	30.969267	26.06	94	22.222222	75.2	Cannot be computed	Cannot be computed	Cannot be computed	156	36.879433	81.1
FAST	2634	623	23.65224	80.3	955	36.256644	52.1	795	30.182232	47.57	550	20.88079	120	Cannot be computed	Cannot be computed	Cannot be computed	1669	63.363705	305.6
BRISK	1457	424	29.100892	488.1	594	40.768703	413.3	329	22.580645	468.9	360	24.708305	522	Cannot be computed	Cannot be computed	Cannot be computed	801	54.975978	755.3
ORB	284	127	44.71831	15.85	121	42.605634	14.25	105	36.971831	19.5	56	19.71831	63.5	Cannot be computed	Cannot be computed	Cannot be computed	178	62.676056	176.1
AKAZE	724	307	42.403315	140.2	358	49.447514	135.1	242	33.425414	135	255	35.220994	188	334	46.132597	244.9	408	56.353591	213
SIFT	743	239	32.166891	159.7	279	37.550471	171.3	Cannot be computed	Cannot be computed	Cannot be computed	197	26.514132	247	Cannot be computed	Cannot be computed	Cannot be computed	292	39.300135	277.1

It is important in our scenario that the Keypoints must be accurately matched, and the entire process runs as quickly as possible. The below table shows the top ten Detector and Descriptor combinations with highest percentage of matching at the top.

Dectector + Descriptor	Matching Percentage	Total Execution Time Per Image (ms)
FAST + SIFT	63.36	305
ORB + SIFT	62.67	176
AKAZE + SIFT	56.35	213
BRISK + SIFT	54.97	755
SHITOMASI + SIFT	52.14	68.9
AKAZE + BRIEF	49.44	135.1
AKAZE + AKAZE	46.13	244.9
ORB + BRISK	44.7	15.85
ORB + BRIEF	42.6	14.25
AKAZE + BRISK	42.4	140.2

As discussed before, it is also important to have the process run as quickly as possible. As can be seen from the above table, the quickest of the top ten combinations with highest matching are given below. So, it is these three combinations of Detectors and Descriptors that best match our need.

I. ORB + BRIEF
II. ORB + BRISK
III. SHITOMASI + SIFT