SFND 2D Feature Tracking

The idea of the camera course is to build a collision detection system - that's the overall goal for the Final Project. As a preparation for this, you will now build the feature tracking part and test various detector / descriptor combinations to see which ones perform best. This mid-term project consists of four parts:

- First, you will focus on loading images, setting up data structures and putting everything into a ring buffer to optimize memory load.
- Then, you will integrate several keypoint detectors such as HARRIS, FAST, BRISK and SIFT and compare them with regard to number of keypoints and speed.
- In the next part, you will then focus on descriptor extraction and matching using brute force and also the FLANN approach we discussed in the previous lesson.
- In the last part, once the code framework is complete, you will test the various algorithms in different combinations and compare them with regard to some performance measures.

See the classroom instruction and code comments for more details on each of these parts. Once you are finished with this project, the keypoint matching part will be set up and you can proceed to the next lesson, where the focus is on integrating Lidar points and on object detection using deep-learning.

Dependencies for Running Locally

- cmake >= 2.8
  - All OSes: click here for installation instructions
- make >= 4.1 (Linux, Mac), 3.81 (Windows)
  - Linux: make is installed by default on most Linux distros
  - Mac: install Xcode command line tools to get make
  - Windows: Click here for installation instructions
- OpenCV >= 4.1
  - This must be compiled from source using the `-D OPENCV_ENABLE_NONFREE=ON` cmake flag for testing the SIFT and SURF detectors.
  - The OpenCV 4.1.0 source code can be found here
- gcc/g++ >= 5.4

- Linux: gcc / g++ is installed by default on most Linux distros
- Mac: same deal as make - [install Xcode command line tools]
- Windows: recommend using [MinGW]

Basic Build Instructions

1. Clone this repo.
2. Make a build directory in the top level directory: `mkdir build && cd build`
3. Compile: `cmake .. && make`
4. Run it: `./2D_feature_tracking`.

Solution - Camera Based 2D Feature Tracking

Part I: Solution Description

MP.1 Data Buffer Optimization

Task: 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.

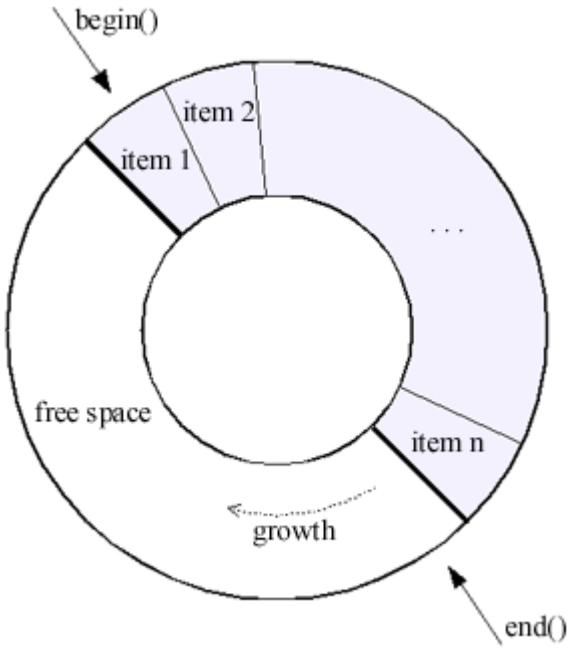
Implementation: The data buffer optimization is done by using a ring buffer provided by boost::circular_buffer. The term circular buffer (also called a *ring* or *cyclic buffer*) refers to an area in memory which is used to store incoming data. When the buffer is filled, new data is written starting at the beginning of the buffer and overwriting the old.

boost::circular_buffer is a STL compliant container.

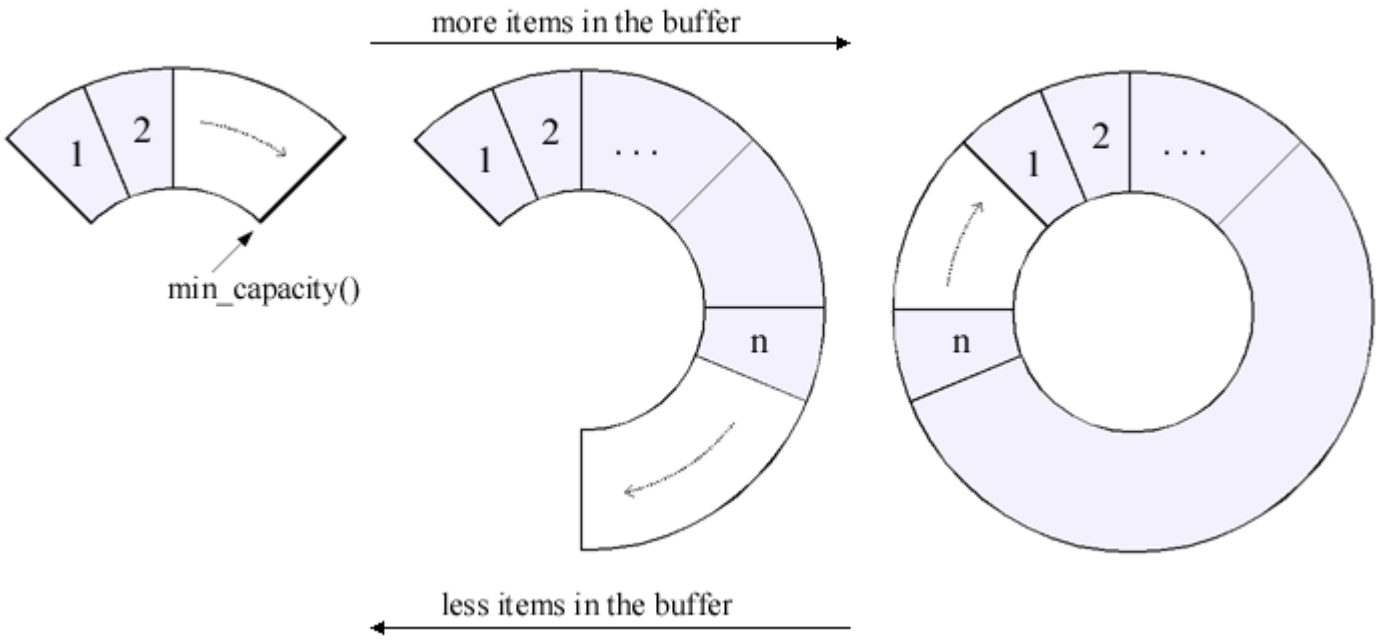
It is a kind of sequence similar to std::list or std::deque. It supports random access iterators, constant time insert and erase operations at the beginning or the end of the buffer and interoperability with std algorithms.

The circular_buffer is especially designed to provide **fixed capacity** storage. When its capacity is exhausted, newly inserted elements will cause elements to be overwritten, either at the beginning or end of the buffer (depending on what insert operation is used).

The circular_buffer only allocates memory when created, when the capacity is adjusted explicitly, or as necessary to accommodate resizing or assign operations.



There is also a circular_buffer_space_optimized version available.



circular_buffer_space_optimized is an adaptation of the circular_buffer which **does not allocate memory all at once when created**, instead it allocates memory as needed.

The predictive memory allocation is similar to typical std::vector implementation. Memory is automatically freed as the size of the container decreases.

The memory allocation process of the space-optimized circular buffer. The min_capacity of the capacity controller represents the minimal guaranteed amount of allocated memory. The allocated memory will never drop under this value. The default value of the min_capacity is set to 0.
The min_capacity can be set using the constructor parameter () capacity_control or the function set_capacity.

The space-optimized version is, of course, a little slower.

The instatiation of the ring buffer is done by

```
boost::circular_buffer<DataFrame> dataBuffer(2);  // ring buffer
```

, where 2 is the number of images which are held in memory (ring buffer) at the same time.

Here an example where int the ring buffer a data frame is pushed:

```
        //// 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;

        dataBuffer.push_back(frame);
```

MP.2 Keypoint Detection

Task: Implement detectors HARRIS, FAST, BRISK, ORB, AKAZE, and SIFT and make them selectable by setting a string accordingly.

Implementation of HARRIS detector:

```
// Detect keypoints in image using the traditional Harris detector
void detKeypointsHarris(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)
{
	int blockSize = 2;	// a blockSize x blockSize neighborhood for every pixel
	int apertureSize = 3;	// for sobel operator
	int minResponse = 100;	// minimum value for a corner in the 8-bit scaled response matrix
	double k = 0.04;	// Harris parameter

	cv::Mat dst, dst_norm, dst_norm_scaled;
	dst = cv::Mat::zeros(img.size(), CV_32FC1);

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

	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);

	double maxOverlap = 0.0;

	for (size_t i = 0; i < dst_norm.rows; i++)  // look for prominent corners and keypoints
	{
		for (size_t j = 0; j < dst_norm.cols; j++)
		{
			int response = (int)dst_norm.at<float>(i, j);

			if (response > minResponse)  // only store points above a threshold
			{
				cv::KeyPoint newKeypoint;

				newKeypoint.pt = cv::Point2f(j, i);
				newKeypoint.size = 2 * apertureSize;
				newKeypoint.response = response;
				newKeypoint.class_id = 1;

				bool bOverlap = false;
```

```
                        for (auto it = keypoints.begin(); it != keypoints.end(); ++it)  //
perform non-maximal suppression in local neighbourhood around new key point
                        {
                                double kptOverlap = cv::KeyPoint::overlap(newKeypoint, *it);

                                if (kptOverlap > maxOverlap)
                                {
                                        bOverlap = true;

                                        if (newKeypoint.response > (*it).response)
                                        {
                                                *it = newKeypoint;

                                                break;
                                        }
                                }
                        }

                        if (!bOverlap)
                        {
                                keypoints.push_back(newKeypoint);
                        }
                }
        }
    }

    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    cout << "Harris detector with n= " << keypoints.size() << " keypoints in " << 1000 * t / 1.0
<< " ms" << endl;

    // visualize results
    if (bVis)
    {
        cv::Mat visImage = dst_norm_scaled.clone();
        cv::drawKeypoints(dst_norm_scaled, keypoints, visImage, cv::Scalar::all(-1),
cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

        string windowName = "Harris corner detector results";

        cv::namedWindow(windowName, 5);
        imshow(windowName, visImage);

        cv::waitKey(0);
    }
}
```

Implementation of alternative keypoints detetors:

```
// Detect keypoints using alternative detectors such as FAST, BRISK, ORB, AKAZE, and SIFT
void detKeypointsAlt(vector<cv::KeyPoint> &keypoints, cv::Mat &img, string detectorType, bool bVis)
{
    cv::Ptr<cv::FeatureDetector> detector;
    double t = 0.0;

    if (detectorType.compare("FAST") == 0)
    {
        int threshold = 30;    // difference between intensity of the central pixel and pixels
of a circle around this pixel
        int bNMS = true;       // non-maximal suppression on keypoints

        cv::FastFeatureDetector::DetectorType type = cv::FastFeatureDetector::TYPE_9_16;
        detector = cv::FastFeatureDetector::create(threshold, bNMS, type);

        t = (double)cv::getTickCount();
        detector->detect(img, keypoints);
        t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
```

```
		cout << "FAST detector with n= " << keypoints.size() << " keypoints in " << 1000 * t /
1.0 << " ms" << endl;
	}
	else if (detectorType.compare("BRISK") == 0)
	{
		detector = cv::BRISK::create();

		t = (double)cv::getTickCount();
		detector->detect(img, keypoints);
		t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

		cout << "BRISK detector with n= " << keypoints.size() << " keypoints in " << 1000 * t
/ 1.0 << " ms" << endl;
	}
	else if (detectorType.compare("SIFT") == 0)
	{
		detector = cv::xfeatures2d::SIFT::create();

		t = (double)cv::getTickCount();
		detector->detect(img, keypoints);
		t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

		cout << "SIFT detector with n= " << keypoints.size() << " keypoints in " << 1000 * t /
1.0 << " ms" << endl;
	}
	else if (detectorType.compare("ORB") == 0)
	{
		detector = cv::ORB::create();

		t = (double)cv::getTickCount();
		detector->detect(img, keypoints);
		t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

		cout << "ORB detector with n= " << keypoints.size() << " keypoints in " << 1000 * t /
1.0 << " ms" << endl;
	}
	else if (detectorType.compare("AKAZE") == 0)
	{
		detector = cv::AKAZE::create();

		t = (double)cv::getTickCount();
		detector->detect(img, keypoints);
		t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();

		cout << "AKAZE detector with n= " << keypoints.size() << " keypoints in " << 1000 * t
/ 1.0 << " ms" << endl;
	}

	// avoid error with AKAZEFeatures
	for (auto &keypoint : keypoints)
	{
		keypoint.class_id = 1;
	}

	// visualize results
	if (bVis)
	{
		cv::Mat visImage = img.clone();
		cv::drawKeypoints(img, keypoints, visImage, cv::Scalar::all(-1),
cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

		string windowName = "Alternative detectors results";

		cv::namedWindow(windowName, 2);
		imshow(windowName, visImage);

		cv::waitKey(0);
	}
```

```
}
```

MP.3 Keypoint Removal

Task: Remove all keypoints outside of a pre-defined rectangle and only use the keypoints within the rectangle for further processing.

In the solution this task is solved by using a lambda expression looping through detected keypoints:

```
        // only keep keypoints on the preceding vehicle
        bool bFocusOnVehicle = true;
        cv::Rect vehicleRect(535, 180, 180, 150);
        if (bFocusOnVehicle)

             {
                   keypoints.erase(std::remove_if(keypoints.begin(), keypoints.end(),
[&vehicleRect](const cv::KeyPoint &r) {return !(vehicleRect.contains(r.pt)); }), keypoints.end());
             }
```

MP.4 Keypoint Descriptors

Task: Implement descriptors BRIEF, ORB, FREAK, AKAZE and SIFT and make them selectable by setting a string accordingly.

Implementation:

```
// Use one of several types of state-of-art descriptors to uniquely identify keypoints
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("SIFT") == 0)
       {
             extractor = cv::xfeatures2d::SiftDescriptorExtractor::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("BRIEF") == 0)
       {
             extractor = cv::xfeatures2d::BriefDescriptorExtractor::create();
       }
```

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

        t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
        cout << descriptorType << " descriptor extraction in " << 1000 * t / 1.0 << " ms" << endl;
}
```

MP.5+6 Descriptor Matching + Descriptor Distance Ratio

Task: Implement FLANN matching as well as k-nearest neighbor selection. Both methods must be selectable using the respective strings in the main function. Use the K-Nearest-Neighbor matching to implement the descriptor distance ratio test, which looks at the ratio of best vs. second-best match to decide whether to keep an associated pair of keypoints.

Implementation:

```
// Find best matches for keypoints in two camera images based on several matching methods
void matchDescriptors(vector<cv::KeyPoint> &kPtsSource, vector<cv::KeyPoint> &kPtsRef, cv::Mat
&descSource, cv::Mat &descRef,
        vector<cv::DMatch> &matches, string descriptorType, string matcherType, string selectorType)
{
        // configure matcher
        bool crossCheck = false;
        cv::Ptr<cv::DescriptorMatcher> matcher;

        if (matcherType.compare("MAT_BF") == 0)
        {

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

                matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
        }

        // 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 << "NN with n=" << matches.size() << " matches in " << 1000 * t / 1.0 << " ms" <<
endl;
        }
        else if (selectorType.compare("SEL_KNN") == 0)
        {
                // k nearest neighbors (k=2)
                vector<vector<cv::DMatch>> knn_matches;
                double t = (double)cv::getTickCount();

                matcher->knnMatch(descSource, descRef, knn_matches, 2);
```

```
            t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
            cout << "KNN with n = " << knn_matches.size() << " matches in " << 1000 * t / 1.0 << "
ms" << endl;

            // Implement k-nearest-neighbor matching and filter matches using descriptor distance
ratio test
            double minDescDistRatio = 0.8;

            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 << "# keypoints removed = " << knn_matches.size() - matches.size() << endl;
      }
}
```

Part II: Performance Evaluation

MP.7 Performance Evaluation 1

Task: Count the number of keypoints on the preceding vehicle for all 10 images and take note of the distribution of their neighborhood size. Do this for all the detectors you have implemented.

Here the number of keypoints count per image:

SHITOMASI	125	118	123	120	120	113	114	123	111	112
HARRIS	17	14	18	21	26	43	18	31	26	34
FAST	419	427	404	423	386	414	418	406	396	401
BRISK	264	282	282	277	297	279	289	272	266	254
ORB	92	102	106	113	109	125	130	129	127	128
AKAZE	166	157	161	155	163	164	173	175	177	179
SIFT	138	132	124	138	134	140	137	148	159	137

MP.8 Performance Evaluation 2

Task: Count the number of matched keypoints for all 10 images using all possible combinations of detectors and descriptors. In the matching step, the BF approach is used with the descriptor distance ratio set to 0.8.

Here the number of matched keypoints count per image:

SHITOMASI/BRISK	84	80	73	77	74	70	79	81	72
SHITOMASI/BRIEF	96	93	92	89	92	93	85	91	85
SHITOMASI/ORB	86	84	87	91	87	76	81	88	88
SHITOMASI/FREAK	66	66	64	63	62	64	61	65	63
SHITOMASI/SIFT	96	86	83	91	86	81	87	92	86
HARRIS/BRISK	11	9	10	11	16	14	12	21	17
HARRIS/BRIEF	12	12	14	17	17	16	12	20	21
HARRIS/ORB	11	11	14	17	19	19	13	21	20

HARRIS/FREAK	11	9	13	14	13	18	10	17	18
HARRIS/SIFT	15	12	15	18	22	28	15	21	20
FAST/BRISK	213	216	187	205	185	200	215	203	208
FAST/BRIEF	229	253	233	247	224	243	251	260	238
FAST/ORB	226	220	218	226	220	235	251	226	239
FAST/FREAK	178	181	156	182	159	179	196	164	171
FAST/SIFT	257	246	237	236	232	251	259	252	255
BRISK/BRISK	125	143	129	138	108	129	154	157	136
BRISK/BRIEF	138	166	129	141	148	155	158	161	148
BRISK/ORB	132	146	122	144	130	156	159	149	150
BRISK/FREAK	95	111	91	102	95	105	119	124	117
BRISK/SIFT	137	151	145	146	144	151	156	165	166
ORB/BRISK	36	37	36	43	40	50	53	61	55
ORB/BRIEF	37	38	37	53	42	64	58	62	59
ORB/ORB	42	39	55	44	39	61	57	65	56
ORB/FREAK	30	30	41	33	33	37	43	43	42
ORB/SIFT	51	49	51	48	50	68	65	74	54
AKAZE/BRISK	118	109	112	110	109	116	126	137	127
AKAZE/BRIEF	108	116	110	109	116	129	133	135	131
AKAZE/ORB	113	106	108	110	117	122	123	136	131
AKAZE/FREAK	104	98	91	87	87	104	120	117	112
AKAZE/SIFT	130	116	102	109	111	125	122	133	134
SIFT/BRISK	55	60	59	61	52	49	50	62	64
SIFT/BRIEF	63	72	65	67	52	57	72	67	84
SIFT/ORB	63	57	61	62	52	63	62	59	74
SIFT/FREAK	61	62	56	64	49	50	46	56	58
SIFT/SIFT	75	72	65	76	59	64	64	69	81

MP.9 Performance Evaluation 3

Task: Log the time it takes for keypoint detection and descriptor extraction. The results must be entered into a spreadsheet and based on this data, the TOP3 detector / descriptor combinations must be recommended as the best choice for our purpose of detecting keypoints on vehicles.

Here is the time for keypoint detection and descriptor extraction per image:

SHITOMASI/BRISK	2450.48	2503.61	2485.75	2435.05	2437.2	2448.82	2423.85	2419.57	2417.51
SHITOMASI/BRIEF	159.536	167.679	160.549	157.398	158.98	158.644	158.502	159.153	155.746
SHITOMASI/ORB	221.804	218.185	216.546	216.112	217.228	218.675	218.323	216.626	216.217
SHITOMASI/FREAK	221.793	221.114	220.715	222.32	220.903	218.786	220.829	219.373	221.274
SHITOMASI/SIFT	191.93	192.814	190.734	191.454	191.385	189.799	194.515	193.813	190.922
HARRIS/BRISK	3006.81	2998	2996.67	2993.7	3235.81	3049.64	3027.22	3040.18	3156.38
HARRIS/BRIEF	139.541	141.676	142.471	152.011	312.044	137.14	166.462	153.513	211.9
HARRIS/ORB	208.987	210.86	211.128	220.562	378.738	206.562	239.57	220.402	280.429
HARRIS/FREAK	284.816	286.764	289.017	296.082	461.552	284.222	314.661	297.686	360.949
HARRIS/SIFT	175.142	176.554	179.886	187.13	347.479	173.154	210.879	199.685	248.391
FAST/BRISK	2339.3	2337.59	2351.08	2349.84	2350.17	2432.4	2422.28	2440.3	2345.68

FAST/BRIEF	106.345	977.519	101.772	992.907	100.846	985.107	994.776	103.088	993.881
FAST/ORB	134.302	137.709	138.738	146.521	137.367	139.411	137.713	137.414	135.944
FAST/FREAK	156.909	162.565	162.946	154.972	154.042	158.286	158.799	154.673	154.542
FAST/SIFT	199.077	201.969	196.277	190.144	189.749	200.423	192.735	187.638	186.929
BRISK/BRISK	4708.48	4725.57	4741.73	4991.29	4727.49	4642.99	4659.2	4657.73	4648.93
BRISK/BRIEF	2452.13	2441.65	2448.31	2440.92	2446.81	2447.67	2452.56	2481.92	2453.98
BRISK/ORB	2497.85	2485.27	2496.14	2493.69	2489.53	2500.99	2523.02	2549.24	2489.71
BRISK/FREAK	2517.62	2519.16	2503.06	2505.01	2508.05	2490.6	2499.4	2506.52	2487.24
BRISK/SIFT	2500.24	2492.04	2501.51	2499.66	2494.36	2501.26	2483.89	2483.17	2482.55
ORB/BRISK	2363.52	2362.57	2399.19	2395.47	2374.69	2356.52	2350.73	2362.01	2350.43
ORB/BRIEF	132.63	133.231	134.117	136.758	137.484	138.227	150.604	140.101	136.309
ORB/ORB	231.976	220.537	203.3	200.411	213.401	216.745	214.012	202.378	210.047
ORB/FREAK	219.242	218.804	222.248	219.186	210.551	212.485	211.433	217.432	229.763
ORB/SIFT	179.295	176.933	186.684	179.335	182.711	182.382	213.53	186.016	179.854
AKAZE/BRISK	2770.26	2773.06	2755.18	2785.9	2781	2780.38	2770.74	2814.12	2761.59
AKAZE/BRIEF	515.765	531.016	556.665	513.472	513.343	532.451	527.263	517.882	514.435
AKAZE/ORB	568.044	574.525	568.207	682.544	613.988	678.542	611.317	614.453	592.3
AKAZE/FREAK	583.496	582.875	601.202	579.336	591.51	600.186	603.449	604.695	567.844
AKAZE/SIFT	576.19	597.444	551.73	552.013	571.703	554.417	568.231	609.633	581.862
SIFT/BRISK	2591.73	2627.91	2560.45	2675.84	2608.37	2627.07	2603.35	2585.54	2571.19
SIFT/BRIEF	291.512	286.857	282.29	289.515	300.796	334.852	291.821	291.816	309.547
SIFT/ORB	371.74	350.902	347.841	342.48	355.4	350.203	361.593	354.549	355.096
SIFT/FREAK	387.366	361.615	368.542	370.232	369.649	357.91	355.96	359.074	352.744
SIFT/SIFT	328.683	321.887	336.258	331.479	340.954	334.259	327.564	339.855	324.46

Recommended TOP3 detector/descriptor combinations the best choice for my purpose of detecting keypoints on vehicles:

The most matched keypoints will be scored by using the FAST detector. Also taking account the time for keypoint detection and descriptor extraction per image my TOP3 detector/descriptor combinations are:

1. FAST/ORB:

   - Matched keypoints per image:

FAST/ORB	226	220	218	226	220	235	251	226	239

   - time for keypoint detection and descriptor extraction per image:

FAST/ORB	134.302	137.709	138.738	146.521	137.367	139.411	137.713	137.414	135.944

2. FAST/SIFT:

   - Matched keypoints per image:

FAST/SIFT	257	246	237	236	232	251	259	252	255

   - time for keypoint detection and descriptor extraction per image:

FAST/SIFT	199.077	201.969	196.277	190.144	189.749	200.423	192.735	187.638	186.929

3. FAST/BRIEF:

   - Matched keypoints per image:

FAST/BRIEF	229	253	233	247	224	243	251	260	238

   - time for keypoint detection and descriptor extraction per image:

FAST/BRIEF	106.345	977.519	101.772	992.907	100.846	985.107	994.776	103.088	993.881