Camera Based 2D Feature Tracking

Seymur Dadashov

July 17, 2020

Abstract

The purpose of the project is to compare and contrast the different available keypoint detectors and descriptors provided by the OpenCV framework. Observations with regards to number of keypoints detected in the image; the elapsed time of both detectors and descriptors for given methods; and overall performance of given method when matching keypoints is monitored. Descriptors based on the principle of histogram of gradients and binary feature creation are tested.

1 Task MP.1 Data Buffer Optimization

The implementation of the ring buffer was done by the use of a two if conditions. The first if condition is checking if the initial vector of type `DataFrame` was populated such that it would be of size 2. The second if statement was used for the remainder of the sequence frames as long as the `dataBuffer` had size of 2. The element in the 2nd position i.e. $i = 1$ was shifted to position $i = 0$ and the new incoming image was assigned index $i = 1$ once it was available.

```
// push image into data frame buffer
DataFrame frame;
frame.cameraImg = imgGray;
if(dataBuffer.size() == dataBufferSize)
{
    dataBuffer[0] = dataBuffer[1]; //cycle the ring buffer
    dataBuffer[1] = frame;
}
if(dataBuffer.size() < dataBufferSize)
{
    dataBuffer.push_back(frame);
}
```

Listing 1: Ring Buffer

2 Task MP.2 Keypoint Detection

Within an if, else if, and else conditions, the Shi-Tomasi, Harris, ORB, FAST, BRISK, AKAZE, and SIFT detectors are executed. On line 119 of `MidTermProject_Camera_Student.cpp`, the `detectorType` is compared against the string of HARRIS, and if the strings are equal, the detKeypointsHarris function is called. Otherwise, the detKeypointsModern function is executed, which contains a series of if conditions that compare the detectorType string against other string literals.

Within `detKeypointsModern`, a `cv::Ptr` of `FeatureDetector` type is assigned the proper detector based on the chosen detector. Once assigned, the detector-¿detect method is executed, which takes the grayscale image as input, and stores the detected keypoints in the keypoints vector. The results are displayed if the user has chosen to visualize them.

The function `detKeypointsModern` also takes in a `std::vector<double>& time` that records the elapsed time required for running the operation of detector-¿detect. The value is then appended to the vector for later use to be passed the CSV file.

3 Task MP.3 Keypoint Removal

The keypoints found within the bounding box of the preceding vehicle are extracted within the if statement beginning on line 135 of `MidTermProject_Camera_Student.cpp`. A temporary vector `cropPoints` is initiated along with the OpenCV bounding box `vehicleRect`. For each point in the keypoints vector, the `vehicleRect.contains` method is called to determine if that point exists within the boundary. If the condition is true, that point is copied to the `cropPoints` vector. When the for-loop finishes, the `keypoints` vector is assigned the contents of the `cropPoints` vector.

```
// only keep keypoints on the preceding vehicle
bool bFocusOnVehicle = true;
if (bFocusOnVehicle)
{
    // Define Crop Region
    cv::Rect vehicleRect(535, 180, 180, 150);
    vector<cv::KeyPoint> cropPoints;
    // Filter points
    for(auto point : keypoints)
    {
        if(vehicleRect.contains(cv::Point2f(point.pt))) { cropPoints.push_back(point); }
    }
    // Assign cropped KeyPoints
    keypoints = cropPoints;
    dataLogger.push_back(to_string(keypoints.size())); // Vehicle Keypoints
    cout << "The number of keypoints in Image " << imgIndex << " is " << keypoints.size() << endl;
}
```

Listing 2: Preceding Vehicle Keypoint Isolation

4 Task MP.4 Keypoint Descriptors

The BRIEF, ORB, FREAK, AKAZE, and SIFT descriptors were implemented in a manner similar to the detectors from MP.2. Beginning on line 64 of `matching2D_Student.cpp`, a series of if conditions are used to compare the string input by the user. If the string matches one of the hard-coded descriptor types, the generic extractor variable is assigned the appropriate descriptor, and the compute method is called. Additionally, the elapsed time is measured during execution and stored in a vector.

5 Task MP.5 Descriptor Matching

FLANN matching was implemented on line 20 of matching2D_Student.cpp in the function matchDescriptors. The matcher cv:Ptr of cv::DescriptorMatcher type is assigned a pointer to a descriptor matcher constructed with a FLANNBASED type. Additionally, two if conditions are used to determine if the descriptor matrices are not CV_32F type, and if so, they are converted in order to avoid an existing bug in OpenCV.

6 Task MP.6 Descriptor Distance Ratio

In the function matchDescriptors, K-Nearest-Neighbor selection is implemented in beginning on line 40. A vector of vectors of cv::DMatch type is used to store the matches after calling matcher-¿knnMatch, using a value of 2 for k. Next, for each match in the knnMatches vector, the descriptor distance ratio test is performed. The distance threshold is set to 0.8, and each point falling within the threshold distance is copied to the matches vector.

7 Task MP.7 Performance Evaluation 1

The number of keypoints found on the preceding vehicle is part of the process of isolating the keypoints in Task MP.3 as the keypoint count on the preceding vehicle is equal to the number of keypoints isolated in Task MP.3. In the same conditional statement, the number of keypoints is stored onto a vector dataLogger which is later used to append data for each combination of detectors-descriptors to a CSV file.

8 Task MP.8 Performance Evaluation 2

The number of matched keypoints is directly available from the computation done in Task MP.6, where matched keypoints where initially stored into a vector knn_matches on line 43 in matching2D_Student.cpp. These points were filtered using the threshold value of 0.8 and appended to the final vector of matched keypoints. The number can be extracted from matches.size(). Additionally, this value is stored onto the vector dataLogger and later added to the CSV file for every combination run of detector-descriptor.

9 Task MP.9 Performance Evaluation 3

A complete log of all the data can be found in Midterm_Data_Results.csv as the direct output of the code and further processed data with median calculations of matches and run-time can be found in Midterm_Data_Log_Processed.c Additionally, the results for my top three detector-descriptor combination can be found in Midterm_Data_TopModels.csv.

The combinations top three detector-descriptor combinations for this project are:

1. Detector: **FAST** Descriptor: **BRIEF**
2. Detector: **FAST** Descriptor: **ORB**
3. Detector: **FAST** Descriptor: **BRISK**

From an analytical point of view, the above selection make a lot of sense when it comes to the task of creating a collision-avoidance-system (CAS) as the we would like to find many keypoints with a high True-Positive Ratio (TPR) in the least amount of time.

The combinations seen in Table 1 had the fastest elapsed time when searching for keypoints during the detection phase. All the combinations listed completed a keypoint search in under $1ms$. Although the FAST detector did not find the most number of points, compared to BRISK, FAST was able to find a median of 1794 keypoints, while BRISK found 2725. However, the one large advantage is that FAST was able to do so almost 40 times faster than BRISK. The models that followed took at least 100 times longer and were all below 1400 keypoints discovered overall.

Observing the behaviour of the descriptors, it is easy to see that the binary descriptors out perform the histogram of gradients (HOG) based descriptors in terms of elapsed time. This result was so significant

Top Model Summary					
Rank	Det-Desc Combination	True-Positive (%)	Detector Run-Time (ms)	Descriptor Run-Time (ms)	Matched Keypoint
1	FAST-BRIEF	83.22%	0.98	0.75	121
2	FAST-ORB	81.33%	0.98	1.07	121
3	FAST-BRISK	68.59%	0.99	1.88	100

Table 1: Select data for top three detector-descriptor combinations.

that all binary descriptors performed the task in under $5.66ms$, while the next best HOG descriptor took $18.10ms$ all the way to $96.70ms$. As seen in Table 1, the chosen descriptors all perform under $2ms$. The True-Positive percentage reported was comparable to some of the HOG based approaches as it is seen in literaturem, the use of histograms tends to result in greater successful matching i.e. higher TPR value. The reason FAST-BRISK was chosen as the third best model over some close competitors is highly due to the fact that the next best combination dealt with ORB as the detector, which resulted in a much higher detection run-time of up to $7ms+$ which is a significant increase when compared to FAST. At the cost of some accuracy, I believe it would be better for a system to respond more quickly when making the decision between 10% boost in succesful keypoint matching or 7 times increase in run-time in the program.

Image ID	Detector	Descriptor	Total Keypoints	Vehicle Keypoints	Detector Runtime	Descriptor Runtime	Matched Keypoints	Matcher Runtime	True-Positive %
0	FAST	BRISK	1824	149	1.08051	2.34480	0	0.00000	-
1	FAST	BRISK	1832	152	0.98583	2.00264	97	0.60877	63.82%
2	FAST	BRISK	1810	150	1.32056	1.92632	104	0.64082	69.33%
3	FAST	BRISK	1817	155	1.29836	2.02502	101	0.64590	65.16%
4	FAST	BRISK	1793	149	1.35785	1.73277	98	0.62744	65.77%
5	FAST	BRISK	1796	149	0.96784	2.42674	85	0.60188	57.05%
6	FAST	BRISK	1788	156	1.32022	1.82537	107	0.65565	68.59%
7	FAST	BRISK	1695	150	1.31600	1.74103	107	0.63627	71.33%
8	FAST	BRISK	1749	138	1.29301	1.61742	100	0.56914	72.46%
9	FAST	BRISK	1770	143	1.63258	1.68421	100	0.56336	69.93%
0	FAST	BRIEF	1824	149	0.97918	1.34101	0	0.00000	-
1	FAST	BRIEF	1832	152	0.98343	0.78142	119	0.59142	78.29%
2	FAST	BRIEF	1810	150	1.27420	1.03771	130	0.58633	86.67%
3	FAST	BRIEF	1817	155	0.96331	0.80646	118	0.57597	76.13%
4	FAST	BRIEF	1793	149	0.94673	0.74320	126	0.56473	84.56%
5	FAST	BRIEF	1796	149	1.00122	0.70369	108	0.54499	72.48%
6	FAST	BRIEF	1788	156	0.95994	0.74164	123	0.82038	78.85%
7	FAST	BRIEF	1695	150	0.93502	0.76586	131	0.65776	87.33%
8	FAST	BRIEF	1749	138	0.97493	0.66525	125	0.53634	90.58%
9	FAST	BRIEF	1770	143	1.01446	0.72334	119	0.49331	83.22%
0	FAST	ORB	1824	149	1.00912	1.05428	0	0.00000	-
1	FAST	ORB	1832	152	1.42056	1.63702	118	0.63483	77.63%
2	FAST	ORB	1810	150	0.94949	1.04889	123	0.62292	82.00%
3	FAST	ORB	1817	155	0.95808	1.08122	112	0.62040	72.26%
4	FAST	ORB	1793	149	0.96628	1.04632	126	0.62453	84.56%
5	FAST	ORB	1796	149	0.98720	0.99858	106	0.58813	71.14%
6	FAST	ORB	1788	156	0.95385	1.08884	122	1.03032	78.21%
7	FAST	ORB	1695	150	0.96812	1.01701	122	0.63629	81.33%
8	FAST	ORB	1749	138	1.02761	1.13351	123	0.57729	89.13%
9	FAST	ORB	1770	143	1.34473	1.63135	119	0.52593	83.22%

Figure 1: Table of data for top three models.