Camera Based 2D Feature Tracking ReadMe

Philipp Rapp

June 12, 2020

Mid-Term Report

MP.0 Mid-Term Report

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

The document at hand represents the readme file.

Data Buffer

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.

In order to solve this task, I decided to implement a ring buffer. A ring buffer can be implemented by creating a plain array with a fixed capacity (in this case 2 elements) and keeping track of

- the current first entry (the head) and
- the occupied slots (the size).

In order to have reusable code, I used a template for both the data type as well as for the capacity. In order to have the same ease of use as for the STL vector, I also implemented a (non-complete) iterator class for the ring buffer.

Keypoints

MP.2 Keypoint Detection

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

I implemented the selection via string by creating an if-else branch (with lots of else cases). The last else makes sure that a valid keypoint detector has been selected.

In each branch, the respective keypoint detection function is used. For the actual keypoint detection, I used the functions which are already provided in the OpenCV and stayed closely to the preceding exercises. When parameters had to be chose for the detectors, I used those from the preceding exercises.

MP.3 Keypoint Removal

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

In order to solve this task, I created a second vector to hold keypoints. This vector will contain all keypoints within the rectangle. Note that I created this vector outside the if branch, in order to ensure that it does not loose its scope. The STL function `copy_if` lends itself to copy only those keypoints to the new vector which are inside the rectangle. The function takes the original keypoint vector, the (in the beginning empty) keypoint vector for the points within the rectangle, as well as a lambda function that checks whether or not the keypoint is within the rectangle, which is computed by means of the OpenCV method `contains`. Finally, the keypoint vectors are swapped so that the original variable holds the keypoints within the rectangle.

Descriptors

MP.4 Keypoint Descriptors

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

I solved this task by implementing an if-else branch (with lots of else cases). While I was at it, I also added another output argument of type `DescriptorType` which specifies whether the descriptor is a binary descriptor or wheter it is a HOG (histogram of oriented gradients) descriptor. This comes in handy later, as it avoids the manual and therefore error-prone specification of the descriptor type when matching keypoints.

The descriptors itself are all taken directly from OpenCV. I used the default parameters in the respective create method.

MP.5 Descriptor Matching

Implement FLANN matching as well as k-nearest neighbor selection. Both methods must be selectable using the respective strings in the main function.

In order to implement FLANN, I used the `FlannBasedMatcher` which is available in OpenCV. It is important to ensure that the data type of the matrices which contain the descriptor data are of single-precision (32-bit) floating point.

In order to address the k-nearest neighbor selection, I first made sure that cross checking is disabled, as it turned out that this cannot be combined with KNN (an

error is thrown otherwise). Then I took advantage of the `knnMatch` method provided by OpenCV, which does the actual KNN matching. All you need to provide are the descriptors for query (source) and train (reference), the value k (in this case 2), and a vector of type `DMatch`.

MP.6 Descriptor Distance Ratio

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.

As the distances d_i are already sorted by `knnMatch` in the distance increasing order (according to the OpenCV documentation), the distance ratio ρ can be computed according to

$$\rho = \frac{d_0}{d_1}. \tag{1}$$

We have $0 \leq \rho \leq 1.0$. All KNN matches where $\rho \leq \rho_{\text{threshold}}$ holds true are pushed in the vector containing the final matches. In our case, $\rho_{\text{threshold}} = 0.8$.

Performance

Note: As stated by Andreas Haja, we do not take accuracy or the receiver operating characteristic (ROC) into account here.

MP.7 Performance Evaluation 1

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.

In order to complete this task, I added a command line argument which allows for the specification of the keypoint detector. That way, it is not necessary to recompile the program in order to change the detector. I also created the shell script `performance_eval_01.sh` which allows for batch processing. On top of that, I added the class `PerformanceEvaluation` in order to keep all performance evaluation related code in one place and not too clutter the main program too much.

Furthermore, I keep track of all keypoints for all 10 images and compute the mean as well as the standard deviation of their size in order to assess the distribution of the keypoint's neighborhood size. The results are shown in Table 1 as well as in Figures 1 to 7.

MP.8 Performance Evaluation 2

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.

Table 1: Keypoint number and size distribution for different detectors.

Keypoint Detector	Average no. of keypoints per image	Average keypoint size	Keypoint size standard deviation
SHITOMASI	117.9	4	0
HARRIS	24.8	6	0
FAST	149.1	7	0
BRISK	271.3	22.05	14.67
ORB	115	55.95	25.22
AKAZE	165.5	7.69	3.53
SIFT	137.2	5.05	6.01

Figure 1: Shi-Tomasi keypoints.

Figure 2: Harris keypoints.

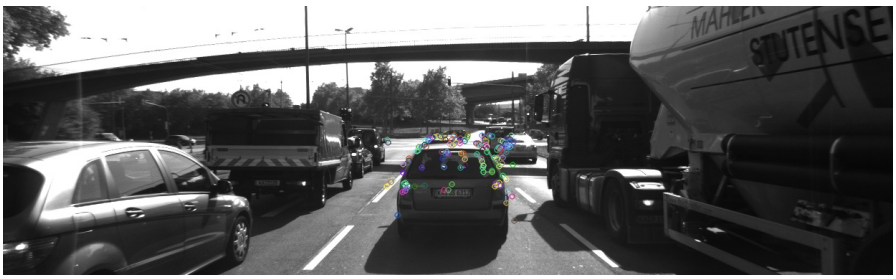

Figure 3: FAST keypoints.

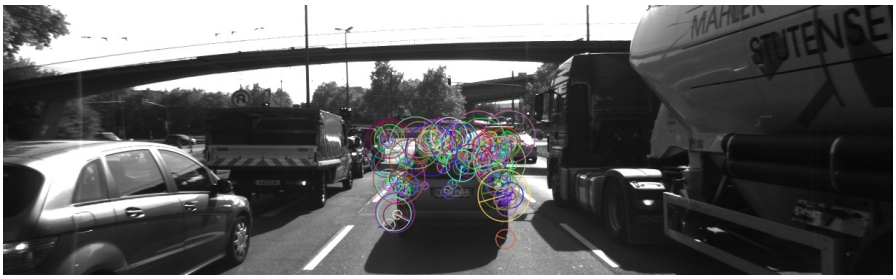

Figure 4: BRISK keypoints.

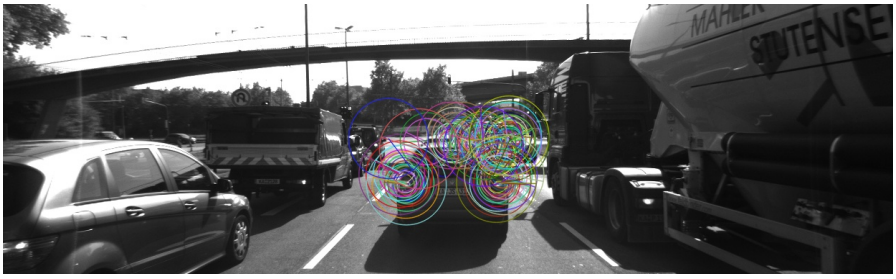

Figure 5: ORB keypoints.

Figure 6: AKAZE keypoints.

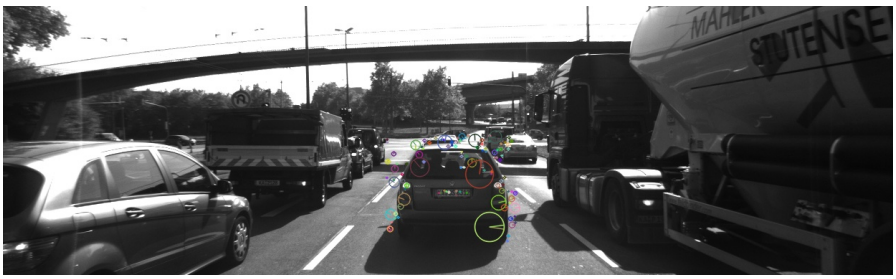

Figure 7: SIFT keypoints.

Table 2: Number of matched keypoints for all 10 images (in total). The rows are the detectors, and the columns are the descriptors.

	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
SHITOMASI	770	947	908	770	N/A	927
HARRIS	143	174	163	145	N/A	163
FAST	901	1103	1079	881	N/A	1046
BRISK	1549	1694	1488	1499	N/A	1617
ORB	744	553	759	416	N/A	756
AKAZE	1206	1267	1181	1184	1249	1263
SIFT	588	708	N/A	593	N/A	792

In order to complete this task, I added a second command line argument which allows for the specification of the descriptor. That way, it is not necessary to recompile the program in order to change the descriptor. I also created the shell script `performance_eval_02.sh` which allows for batch processing. I also took advantage of my `PerformanceEvaluation` class again.

The number of *matched keypoints* for all 10 images (in total) is depicted in Table 2.

Note: According to the OpenCV documentation, the AKAZE descriptors can only be used with KAZE or AKAZE keypoints. This is why the AKAZE descriptor column contains N/As everywhere except for the AKAZE detector.

Second note: There was also a problem when using the SIFT detector in conjunction with the ORB descriptor, as OpenCV raises an error in this case.

MP.9 Performance Evaluation 3

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.

In order to complete this task, I extended my performance evaluation by means of measuring the time. Actually, the time is already measured in the functions which detect the keypoints and compute the descriptors. I only need to add them to the performance evaluation class and output the results. Again, in order to allow for batch processing without the need for recompiling, I created a shell script `performance_eval_03.sh`.

The average processing time (per image frame) for keypoint detection plus descriptor computation is shown in Tables 3 for all valid detector and descriptor combinations.

In my opinion, the TOP3 detector / descriptor combinations for the purpose of detection keypoints on a vehicle for designing a collision avoidance system are

1. FAST keypoint detector with BRIEF descriptor
2. FAST keypoint detector with ORB descriptor
3. FAST keypoint detector with BRISK descriptor

Table 3: Average processing time for keypoint detection plus descriptor computation per image frame in milliseconds. The rows are the detectors, and the columns are the descriptors.

	BRISK	BRIEF	ORB	FREAK	AKAZE	SIFT
SHITOMASI	9.12	7.18	7.19	24.43	N/A	21.06
HARRIS	9.81	9.37	9.39	26.07	N/A	20.71
FAST	1.99	1.01	1.30	18.12	N/A	13.04
BRISK	34.82	32.77	34.86	53.99	N/A	65.38
ORB	9.57	6.90	9.12	23.61	N/A	38.17
AKAZE	50.24	45.77	48.59	66.06	83.66	66.16
SIFT	71.99	67.03	N/A	87.33	N/A	119.35

The reason is that speed is very important, especially for an automotive application, where the electronic control unit is usually of limited computational power. Apart from that, a collision avoidance system needs to react very fast. Those TOP3 combinations, which have been selected solely on computational speed, also have enough keypoints (around 1000), as can be seen in Table 2. (In reality, we would also need to check the quality of the keypoints, that is, how well they lend themselves for a high number of true positive and a low number of false positive matches).