Camera Based 2D Feature Tracking - Mid-Term Project

In order to design collision avoidance system, keypoint features must be tracked on the preceding vehicle. Therefore, in this project, we detect and compute the keypoints and match them with their correspondences across the image frames. Performance of various keypoint detectors and descriptors are analyzed to select a suitable one for our task. The following **rubric points** are addressed in the project.

MP.1| Data Buffer Optimization

In order to prevent memory overflow of the computer caused due to inserting of image sequences inside the data buffer, we implement a **ring buffer**, wherein we define the size of the buffer and once the buffer is full, the old image is popped out from the beginning and the new image is pushed back at the end. We have here defined the size of the buffer as 2. In order to remove the elements from the vector we use the STL function `erase()`.

```
 if (dataBuffer.size() >= dataBufferSize)
    {
        dataBuffer.erase(dataBuffer.begin());
    }
    dataBuffer.push_back(frame);
```

MP.2| Keypoints

We implement the SHITOMASI,HARRIS, FAST, BRISK, ORB, AKAZE and SIFT keypoint detectors. The selection of the detectórs is done by checking the string detectorType.

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

The detectors FAST, BRISK, ORB, AKAZE and SIFT are implemented in the **detKeypointsModern** function inside the file **matching2D_student.cpp** and their headers are defined in **matching2D.hpp** file.

MP.3| Keypoint Removal

Since our object of interest is the preceding vehicle, we will remove all other keypoints outside this vehicle which is defined by rectangular bounding box.The box parameters we used are defined by the parameters; cx = 535, cy = 180, w = 180, h =

150. We use the **contains()** function inside the class Rect of **OpenCV** to check if the keypoint lies inside the bounding box.

```
cv::Rect vehicleRect(535, 180, 180, 150);
if (bFocusOnVehicle)
{
    auto outResult = [vehicleRect](auto kPt) { return !vehicleRect.contains(kPt.pt);
};
    keypoints.erase(std::remove_if(keypoints.begin(), keypoints.end(), outResult),
keypoints.end());
}
```

MP.4| Descriptors

We implement the BRISK, BRIEF, ORB, FREAK, AKAZE and SIFT keypoint descriptors. The selection of the desciptors is done by checking the string **descriptorType**.

```
string descriptorType = "BRISK"; // BRIEF, ORB, FREAK, AKAZE, SIFT
descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg,
descriptors, descTime, descriptorType);
```

The descriptors BRISK, BRIEF, ORB, FREAK and SIFT are implemented in the function **descKeypoints** inside the file **matching2D_student.cpp**.

MP.5| Descriptor Matching

For matching the descriptors in the images **FLANN** matching method is implemented as an alternative to the **Brute Force** method. For selecting the "good" matches, **K-Nearest-Neighbor** method is used. Both the methods are selectable by the string matcherType. The FLANN method is implemented in the function matchDescriptors inside **matching2D_student.cpp**. As OpenCV requires us to convert the binary descriptors into floating point vectors, we convert the image type to CV_32F.

```
if (matcherType.compare("MAT_FLANN") == 0)
{
    if(descSource.type() != CV_32F)
    {
        descSource.convertTo(descSource, CV_32F);
        descRef.convertTo(descRef, CV_32F);
    }

    matcher = cv::FlannBasedMatcher::create();

}
```

After the matcher is created, the matches are computed using the function **matcher**.

```
matcher->match(descSource, descRef, matches);
```

MP.6| Descriptor Distance Ratio

To remove the false positive matches, K-NN method is implemented, in which for each keypoint in the source image, k (here, **k=2**) best matches are located in the refernce image. Then, a threshold is apllied to the ratio of the descriptor distances and the

amibigous matches are filtered out. For our purpose, we set the threshold value (**minDistRatio = 0.8**). We loop across all the knn matches and store only those matches which passes the distance ratio test.

```
for (auto it = knn_matches.begin(); it != knn_matches.end(); it++)
{
    if((*it)[0].distance < minDistRatio * (*it)[1].distance)
    {
        matches.push_back((*it)[0]);
    }
}
```

Performance Evaluations

In the following rubric points, we discuss the performance of various keypoint detectors and descriptors and select the best possible combination for our task. **Note**: All the computations and processing are done locally in **INTEL-CORE I7** processor with **NVIDIA GeForce MX150**.

MP.7| Performance Evaluation 1

All the feature detectors we implemented, i.e; HARRIS, SHI-TOMASI, SIFT, FAST, BRISK, ORB and AKAZE are compared based on the number of keypoints detected on the preceding vehicle for all the 10 image frames. The results are stored in the spreadsheet **PerformanceResults.csv**. The neighborhood size of the keypoint is computed using `keypoint.size` parameter. The size is the region around the point of interest that is used to describe the keypoint. The larger it is, more regional information is used to describe the feature point. We have computed the mean of the neighborhood size for all the keypoints in the image. Based on the observed number of detections, the keypoint detectors are ranked below:

1. **BRISK**
2. **AKAZE**
3. **FAST**
4. **SIFT**
5. **SHI-TOMASI**
6. **ORB**
7. **HARRIS**

MP.8| Performance Evaluation 2

Number of matched keypoint is counted for all the 10 images using all the possible combination of detectors and descriptors we have implemented. In the matching step, **Brute-Force** matching method is used and selector type as **K-Nearest-Neighbor**, with the descriptor distance ratio threshold set to **0.8**. The distance between the descriptors in case of **HOG descriptors** is computed using SSD or **NORM_L2 method** and that for the **binary descriptors** is done using the **Hamming Distance** method. The results are listed in the **PerformanceResults.csv** file.

MP.9| Performance Evaluation 3

Finally, speed of the keypoint detectors and descriptors is analyzed by computing the time taken for the keypoint detection and descriptor extraction. We use OpenCV's `cv::getTickCount()` to compute the time. All the observations are recorded in the spreadsheet. According to the performance analysis of various detector+descriptor

pairs, TOP 3 pairs are listed below based on #matched keypoints, compuational time and matches/millisecond.

TOP 3 pairs based on no. of matched keypoints

Detector+Descriptor pair	mean # matched keypoints
BRISK + BRISK	174
AKAZE + BRISK	135
BRISK + SIFT	121

TOP 3 pairs based on computational time

Detector+Descriptor pair	mean time (in ms)
FAST + BRISK	2.22
SHI-TOMASI + BRISK	11.8
HARRIS + BRISK	12

Overall TOP 3 pairs taking into account both the above factors

Detector+Descriptor pair	approx. mean matches / ms
FAST + BRISK	45
ORB + BRISK	10
SHI-TOMASI + BRISK	7

It is pretty clear that **FAST + BRISK** pair outperforms all other pairs. It computes the maximum number of matches in 1 millisecond. Therefore our ideal choice would be this pair. However, if we are looking for maximum number of matched keypoints, **BRISK + BRISK** can also be chosen, however it is quite slow compared to FAST+BRISK pair. Also note, we do not take into account accuracy of the matching in this analysis.

RESULT

We look into the matched keypoints on the preceding vehicle for **FAST+BRISK** detector/descriptor pair.