SFND2 - 2D Feature tracking

0. The submission

The project was run and compiled without problems on Udacity VM.
All the submission is under */home/workspace/SFND2D_Feature_Matching/*
Content of the submission:
-The c++ project
-The present writeup under *SFND2D_Feature_Matching*
-Code output files are under *SFND2D_Feature_Matching/build*
-An excel (*tasks_spreadsheet.xlsx*) summarizes the files output in a more readable format.
-Remarks regarding code structure in the file *MidTermProject_Camera_Student.cpp*:
I have added a loop *eval_mode* which works whenever we wish to grid-search all pairs of detectors/descriptors for all images. This loop is accessible by setting ***eval_mode=true*** in the parameters initiation.
-Tasks MP1..MP6 happen outside ***eval_mode*** because it's required that the user "hand-picks" the detector/descriptor. Tasks MP7..MP9 happen inside ***eval_mode*** because manual recording of the result of each pair is tedious!
Here's an outlook of these 2 loops

```
/* MAIN PROGRAM */
int main(int argc, const char* argv[])
{

    /* INIT VARIABLES AND DATA STRUCTURES */

    // data location
    string dataPath = "../";

    // camera
    string imgBasePath = dataPath + "images/";
    string imgPrefix = "KITTI/2011_09_26/image_00/data/000000"; // left camera, color
    string imgFileType = ".png";
    int imgStartIndex = 0; // first file index to load (assumes Lidar and camera names have identical nami
    int imgEndIndex = 9;   // last file index to load
    int imgFillWidth = 4;  // no. of digits which make up the file index (e.g. img-0001.png)

    // misc
    int dataBufferSize = 2;       // no. of images which are held in memory (ring buffer) at the same time
    bool bVis = false;            // visualize results
    bool eval_mode = true;        //if true an evaluation loop is added in the end
    bool verbose = false;         //prints messages to the console
    /* MAIN LOOP OVER ALL IMAGES */

    if (!eval_mode) {...
    } //eof non eval mode

    else { //evaluation mode loop...
    }//eof eval mode
    return 0;
}
```

-At the end of the code, the results are collected into .txt files which are next converted into xlsx ***tasks_spreadsheet.xlsx*** for a better readability. Below is a screenshot of the results recording section (file *MidTermProject_Camera_Student.cpp* line 303):

```
//task 9.1: duration of detection per image & detector
ofstream myfile("task91.txt");
cout << "task 91" << endl;
for (int k = 0; k < 7; k++) {
    for (int ii = 0; ii < 10; ii++) {
        cout<< computeDurationDetector[k][ii] << " ";
        myfile << computeDurationDetector[k][ii] << " ";
    }
    cout << endl;
    myfile << endl;
}
myfile.close();

//task 9.2: duration of description per image & descriptor
myfile.open("task92.txt");
cout << "task 92" << endl;
for (int k = 0; k < 6; k++) {
    for (int ii = 0; ii < 10; ii++) {
        cout<< computeDurationDescriptor[k][ii] << " ";
        myfile << computeDurationDescriptor[k][ii] << " ";
    }
    cout << endl;
    myfile << endl;
}
myfile.close();

//task 7.1: number of keypts per image & detector
myfile.open("task71.txt");
cout << "task 71" << endl;
for (int k = 0; k < 7; k++) {
    for (int ii = 0; ii < 10; ii++) {
        cout<< kptsPerFramePerDetector[k][ii][0] << " ";
        myfile << kptsPerFramePerDetector[k][ii][0] << " ";
    }
    cout << endl;
    myfile << endl;
}
myfile.close();

//task 7.2: average neighboring size per image & detector
myfile.open("task72.txt");
cout << "task 72" << endl;
for (int k = 0; k < 7; k++) {
    for (int ii = 0; ii < 10; ii++) {
        cout << kptsPerFramePerDetector[k][ii][1] << " ";
        myfile << kptsPerFramePerDetector[k][ii][1] << " ";
    }
    myfile << endl;
    cout<<endl;
}
myfile.close();

//task 8: number of macthed keypoints per descriptor & detector & image transition
for (int ii = 0; ii < 9; ii++) {
    string filename = "task8_transition" + to_string(ii) + ".txt";
    myfile.open(filename);
    for (int k = 0; k < 7; k++) {
        for (int jj = 0; jj < 6; jj++) {
            myfile << kptsPerFramePerDetector[k][jj][ii] << " ";
        }
        myfile << endl;
    }
```

1. Data Buffer

MP.1 Data buffer optimization

As requested, whenever a new image is pushed in and the size of the buffer exceeds 2, the buffer will erase the first entry.

Below code snapshot describes the buffer rotation

```
//// 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);
if (dataBuffer.size() > 2) dataBuffer.erase(dataBuffer.begin());
```

2. Keypoints

MP.2 Keypoints detection

HARRIS, FAST, BRISK, ORB, AKAZE, and SIFT are built and selectable by "***detectorType***"
Below codes describe how detector name is passed as parameter:
MidTermProject_Camera_Student.cpp line 74

```
//// STUDENT ASSIGNMENT
//// TASK MP.2 -> add the following keypoint detectors in file matching2D.cpp and enable strin
//// -> HARRIS, FAST, BRISK, ORB, AKAZE, SIFT
float duration;
if (detectorType.compare("SHITOMASI") == 0) {
    detKeypointsShiTomasi(keypoints, imgGray, duration, false,verbose);
}
else if (detectorType.compare("HARRIS") == 0) {
    detKeypointsHARRIS(keypoints, imgGray,duration, false, verbose);
}
else {
    detKeypointsModern(keypoints, imgGray, detectorType,duration, false, verbose);
}
```

Below the code for FAST, BRISK, ORB, AKAZE, and SIFT
matching2D_student.cpp line 170

```
void detKeypointsModern(vector<cv::KeyPoint>& keypoints, cv::Mat& img, std::string detectorType, float& duration, bool bVis, bool verbose)
{
    // compute detector parameters based on image size
    cv::Ptr<cv::FeatureDetector> detector;
    if (detectorType.compare("BRISK") == 0) {
        detector = cv::BRISK::create();
    }
    else if (detectorType.compare("AKAZE") == 0) {
        detector = cv::AKAZE::create();
    }
    else if (detectorType.compare("FAST")) {
        detector = cv::FastFeatureDetector::create();
    }
    else if (detectorType.compare("ORB")) {
        detector = cv::ORB::create();
    }
    else if (detectorType.compare("SIFT")) {
        detector = cv::xfeatures2d::SIFT::create();
    }


    double t = (double)cv::getTickCount();
    detector->detect(img, keypoints);
    t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
    duration = 1000 * t / 1.0;
    if(verbose)
        cout << detectorType << " detector with n= " << keypoints.size() << " keypoints in " << duration << " ms" << endl;
    // 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 = detectorType + " Detector Results";
        cv::namedWindow(windowName, 6);
        imshow(windowName, visImage);
        cv::waitKey(0);
    }
}
```

Below the code for Harris detector (*matching2D_student.cpp* line 209)

```
void detKeypointsHARRIS(vector<cv::KeyPoint>& keypoints, cv::Mat& img, float& duration, bool bVis, bool verbose)
{
    // Detector parameters
    int blockSize = 2;     // for every pixel, a blockSize × blockSize neighborhood is considered
    int apertureSize = 3;  // aperture parameter for Sobel operator (must be odd)
    int minResponse = 100; // minimum value for a corner in the 8bit scaled response matrix
    double k = 0.04;       // Harris parameter (see equation for details)

    // Detect Harris corners and normalize output
    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());

    double maxOverlap = 0.0; // max. permissible overlap between two features in %, used during non-maxima suppression
    for (size_t j = 0; j < dst_norm.rows; j++)
    {
        for (size_t i = 0; i < dst_norm.cols; i++)
        {
            int response = (int)dst_norm.at<float>(j, i);
            if (response > minResponse)
            { // only store points above a threshold
                cv::KeyPoint newKeyPoint;
                newKeyPoint.pt = cv::Point2f(i, j);
                newKeyPoint.size = 2 * apertureSize;
                newKeyPoint.response = response;
                // perform non-maximum suppression (NMS) in local neighbourhood around new key point
                bool bOverlap = false;
                for (auto it = keypoints.begin(); it != keypoints.end(); ++it)
                {
                    double kptOverlap = cv::KeyPoint::overlap(newKeyPoint, *it);
                    if (kptOverlap > maxOverlap)
                    {
                        bOverlap = true;
                        if (newKeyPoint.response > (*it).response)
                        {                      // if overlap is >t AND response is higher for new kpt
                            *it = newKeyPoint; // replace old key point with new one
                            break;             // quit loop over keypoints
                        }
                    }
```

e: Debug: Ready   No Kit Selected   Build: [all]   matchDescriptors(std::vector<cv::KeyPoint>& kPtsSource,

MP.3 Keypoints cropping to vehicle

After keypoints are selected for the whole image, we crop these to keep only those within the provided car rectangle. Below the code:

```
//// TASK MP.3 -> only keep keypoints on the preceding vehicle
bool bFocusOnVehicle = true;
cv::Rect vehicleRect(535, 180, 180, 150);
if (bFocusOnVehicle)
{
    vector<cv::KeyPoint> newKeypoints;
    float w = vehicleRect.width;
    float h = vehicleRect.height;
    float x = vehicleRect.x;
    float y = vehicleRect.y;
    for (auto it = keypoints.begin(); it != keypoints.end(); ++it) {
        float x0 = (*it).pt.x;
        float y0 = (*it).pt.y;
        if (x0 >= x && x0 < (x + w) && y0 >= y && y0 < (y + h)) {
            newKeypoints.push_back(*it);
        }
    }
    keypoints = newKeypoints;
}
```

3. Descriptors

MP.4 Keypoint descriptor

Descriptors are selectable by a string and callable using a single function.
Below the code for selecting the desired descriptor (*MidTermProject_Camera_Student.cpp* line 131)

```
/* EXTRACT KEYPOINT DESCRIPTORS */
//// 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;
//"BRISK", "BRIEF", "ORB", "FREAK", "AKAZE", "SIFT"
string descriptorName = "BRIEF"; // BRIEF, ORB, FREAK, AKAZE, SIFT

descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg, descriptors, descriptorName,duration,verbose);
```

Below the code for calling the descriptors (*matching2D_student.cpp* line 60)

```
// 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, float& duration,bool verbose)
{
    // 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) {
        bool    orientation = true; // use orientation or not
        int     bytes = 32; //length of description in bytes: 16,23,64
        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create(bytes, orientation);
    }
    else if (descriptorType.compare("ORB") == 0) {
        int     nfeatures = 100;
        float   scaleFactor = 1.2f;
        int     nlevels = 8;
        int     edgeThreshold = 31;
        int     firstLevel = 0;
        int     WTA_K = 2;
        cv::ORB::ScoreType  scoreType = cv::ORB::HARRIS_SCORE;
        int     patchSize = 31;
        int     fastThreshold = 20;
        extractor = cv::ORB::create(nfeatures, scaleFactor);
    }
    else if (descriptorType.compare("FREAK") == 0) {
        bool    orientationNormalized = true;
        bool    scaleNormalized = true;
        float   patternScale = 22.0f;
        int     nOctaves = 4;
        extractor = cv::xfeatures2d::FREAK::create(orientationNormalized, scaleNormalized, patternScale, nOctaves);
    }
    else if (descriptorType.compare("AKAZE") == 0) {
        cv::AKAZE::DescriptorType   descriptor_type = cv::AKAZE::DESCRIPTOR_MLDB;
        int     descriptor_size = 0;
        int     descriptor_channels = 3;
        float   threshold = 0.001f;
        int     nOctaves = 4;
        int     nOctaveLayers = 4;
        //extractor = cv::AKAZE::create(descriptor_type, descriptor_size, descriptor_channels, threshold, nOctaves);
        extractor = cv::AKAZE::create();

    }
    else if (descriptorType.compare("SIFT") == 0) {
        int     nfeatures = 0;
        int     nOctaveLayers = 3;
        double  contrastThreshold = 0.04;
        double  edgeThreshold = 10;
```

MP.5 & MP.6 Descriptor Matching / Descriptor Distance ratio

Below the code implementing FLANN matching as well as descriptor distance ratio for KNN algorithm (*matching2D_studen.cpp* line 7)

```
    std::vector<cv::DMatch>& matches, std::string descriptorType, std::string matcherType, std::string selectorType, bool verbose)
{
    // 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;
        int normType = cv::NORM_HAMMING;
        if(descriptorType.compare("DES_HOG"))
            normType = cv::NORM_L2;

        matcher = cv::BFMatcher::create(normType, crossCheck);
    }

    else if (matcherType.compare("MAT_FLANN") == 0)
    {
        if (descSource.type() != CV_32F)
        { // OpenCV bug workaround : convert binary descriptors to floating point due to a bug in current OpenCV implementation
            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)
        matcher->match(descSource, descRef, matches); // Finds the best match for each descriptor in desc1
    }

    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); // finds the 2 best matches
        t = ((double)cv::getTickCount() - t) / cv::getTickFrequency();
        if (verbose)
            cout << " (KNN) with n=" << knn_matches.size() << " matches in " << 1000 * t / 1.0 << " ms" << endl;
        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]);
            }
        }
    }
}
```

4. Performance

MP.7 Performance evaluation 1

The number of keypoints + summary of neighboring are recorded in 3D array called ***kptsPerFramePerDetector***, the results of the evaluation are stored in the end of the code into a txt file. For a better readability, the table is included in the xlsx file ***tasks_spreadsheet.xlsx*** included with the submission in the spreadsheets "task71" & "task72". Below a screenshot of the code

```
//Task 7 & task 9.1: record number of keypoints + average neighboring size + detection duration
if (i == 0) {
    kptsPerFramePerDetector[j][imgIndex][0] = keypoints.size();
    float avgNeighSize = 0;
    for (auto it = keypoints.begin(); it != keypoints.end(); ++it)
    {
        avgNeighSize += (*it).response;
    }
    avgNeighSize /= keypoints.size();
    kptsPerFramePerDetector[j][imgIndex][1] = avgNeighSize;
    computeDurationDetector[j][imgIndex] = duration;
}
```

Number of the keypoints per image per detector:

Detector	Image 1	Image 2	Image 3	Image 4	Image 5	Image 6	Image 7	Image 8	Image 9	Image 10
SHITOMASI	125	118	123	120	120	113	114	123	111	112
HARRIS	17	14	18	21	26	43	18	31	26	34
FAST	91	102	106	113	109	124	129	127	124	125
BRISK	254	274	276	275	293	275	289	268	259	250
ORB	419	427	404	423	386	414	418	406	396	401
AKAZE	162	157	159	154	162	163	173	175	175	175
SIFT	419	427	404	423	386	414	418	406	396	401

Average neighboring size

Detector	Image 1	Image 2	Image 3	Image 4	Image 5	Image 6	Image 7	Image 8	Image 9	Image 10
SHITOMASI	0	0	0	0	0	0	0	0	0	0
HARRIS	128.588	130.714	132.333	129.81	136.308	136.209	132.333	140	143.154	142.294
FAST	0.00373891	0.00374825	0.00378307	0.00354859	0.0035164	0.00350128	0.00340757	0.00364616	0.0035798	0.00365361
BRISK	85.0452	82.3313	82.8232	86.0668	81.0522	83.3371	82.0299	83.3334	88.6208	84.0628
ORB	38.2387	37.8993	39.3564	39.2577	40.4793	39.4227	37.8493	40.4089	40.8056	39.8354
AKAZE	0.00706422	0.00697579	0.00683655	0.00711349	0.00674738	0.00669098	0.00679974	0.00712348	0.00691204	0.00701516
SIFT	38.2387	37.8993	39.3564	39.2577	40.4793	39.4227	37.8493	40.4089	40.8056	39.8354

MP.8 Performance evaluation 2

The results are stored in 3D array called ***`matchedKptsPerFramePerDescr`*** containing the number of matched keypoints per descriptor, detector and frame transition. the results of the evaluation are stored in the end of the code into a txt file. For a better readability, the table is included in the xlsx file ***tasks_spreadsheet.xlsx*** included with the submission in the spreadsheet “task8”. Below a screenshot of the code. A copy of the result would take a lot of space since we have 9 matrices each of size #_descriptors x #_detectors. So I kindly refer you to the excel file.

```
// 4. Match keypoints
vector<cv::DMatch> matches;
string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN
string descriptorType = "DES_BINARY"; // DES_BINARY, DES_HOG
if (descriptorName.compare("SIFT") == 0) descriptorType == "DES_HOG";
string selectorType = "SEL_KNN";       // SEL_NN, SEL_KNN
matchDescriptors((dataBuffer.end() - 2)->keypoints, (dataBuffer.end() - 1)->keypoints,
    (dataBuffer.end() - 2)->descriptors, (dataBuffer.end() - 1)->descriptors,
    matches, descriptorType, matcherType, selectorType,verbose);
(dataBuffer.end() - 1)->kptMatches = matches;
if(verbose)
    cout << "#4 : MATCH KEYPOINT DESCRIPTORS done" << endl;
//task 8 number of matched keypoints
matchedKptsPerFramePerDescr[j][i][imgIndex - 1] = matches.size();
```

MP.9 Performance evaluation 3

Before discussing the details of the result, I present a summary of the result:
Top2 fastest detectors: ORB then SIFT
Top2 fastest descriptors: ORB then BRIEF
⇒ Top4 fastest pair detector/descriptor

1. ORB/ORB
2. ORB/BRIEF
3. SIFT/ORB
4. SIFT/BRIEF

The results are recorded at 2 points separately: After detection and after Description. Below is a snapshot of the code:

```
//Task 7 & task 9.1: record number of keypoints + average neighboring size + detection duration
if (i == 0) {
    kptsPerFramePerDetector[j][imgIndex][0] = keypoints.size();
    float avgNeighSize = 0;
    for (auto it = keypoints.begin(); it != keypoints.end(); ++it)
    {
        avgNeighSize += (*it).response;
    }
    avgNeighSize /= keypoints.size();
    kptsPerFramePerDetector[j][imgIndex][1] = avgNeighSize;
    computeDurationDetector[j][imgIndex] = duration;
}
if(verbose)
    cout << "#2 : DETECT KEYPOINTS done" << endl;

//3. Describe keypoints
cv::Mat descriptors;
//float duration;
descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg, descriptors, descriptorName,durati
//task 9.2 record description time
if (j == 0)
    computeDurationDescriptor[i][imgIndex] = duration;
if(descriptorName.compare("AKAZE")==0 && detectorType.compare("AKAZE") == 0)
    computeDurationDescriptor[i][imgIndex] = duration;
// push descriptors for current frame to end of data buffer
```

For a better readability, the table is included in the xlsx file ***tasks_spreadsheet.xlsx*** included with the submission in the spreadsheet “task91” & “task92”. Below is a copy of the results table for detection duration:

Detector	image 1	image 2	image 3	image 4	image 5	image 6	image 7	image 8	image 9	image 10	Average
SHITOMASI	45.641	16.3825	16.4594	16.2049	15.6305	16.6866	16.1139	15.9323	16.3401	16.4579	19.18491
HARRIS	15.95	16.14	16.1836	15.6627	17.1018	30.8063	12.5281	17.773	16.9343	21.9324	18.10122
FAST	17.9147	7.91268	7.19038	6.98465	7.97092	7.57889	7.91792	7.46005	7.66842	7.78123	8.637984
BRISK	43.1868	42.971	40.5335	41.2384	40.7268	40.8777	40.7522	41.7544	41.1798	40.4456	41.36662
ORB	2.1396	2.09833	1.9872	2.21042	2.02516	2.12394	2.11884	2.10686	2.01708	2.03972	2.086715
AKAZE	87.4141	79.3737	110.874	88.2284	95.2664	96.6135	97.8915	94.5526	117.154	149.294	101.66622
SIFT	3.05302	2.08878	1.94613	2.04158	3.12426	2.95866	3.14357	3.04612	3.31653	2.39156	2.711021

Below is a snapshot of the description duration

Descriptor	Image 1	Image 2	Image 3	Image 4	Image 5	Image 6	Image 7	Image 8	Image 9	Image 10	Average
BRISK	5.98808	2.36233	2.33834	2.22604	2.35277	2.38395	2.33239	2.27718	2.18077	2.26729	2.670914
BRIEF	4.36282	1.40473	0.762843	0.780715	1.47812	0.828526	1.44284	1.38129	1.41608	1.35876	1.5216724
ORB	0.992859	0.932464	1.05498	0.96216	0.94883	0.938094	0.917282	0.918169	0.942073	1.01928	0.9626191
FREAK	42.2938	43.5155	41.6527	40.8088	40.9735	44.5077	48.9105	52.1498	57.2271	55.6196	46.7659
AKAZE	69.4115	66.8348	68.5809	66.038	75.1443	70.1657	53.7495	55.1826	59.7429	58.923	64.37732
SIFT	17.9667	15.8425	15.2857	14.6158	14.6251	14.8734	15.2179	14.1698	15.621	14.6349	15.28528