Project2: Camera Based 2D Feature Tracking

Write up

1st submit: October 14th, Kenta Kumazaki

1. Goal

In this "Feature tracking" project, I implemented a few detectors, descriptors, and matching algorithms.

The aim of implementing various detectors and descriptors combinations is to learn a wide range of options available in the OpenCV library.

2. Overview

The steps of this project are the following:

(1) **The Data Buffer**: I focused on loading images, setting up data structures and putting everything into a ring buffer to optimize memory load.

(2) **Keypoint Detection**: I integrated several keypoint detectors such as HARRIS, FAST, BRISK and SIFT and compare them with regard to number of keypoints and speed.

(3) **Descriptor Extraction & Matching**: I focused on descriptor extraction and matching using brute force and also the FLANN approach.

(4) **Performance Evaluation**: Once the code framework is complete, I tested the various algorithms in different combinations and compare them with regard to some performance measures.

3. Submission

(1) GitHub

https://github.com/kkumazaki/Sensor-Fusion_Project2_Camera-Based-2D-Feature-Tracking.git

(2) Directory

I cloned the basic repository from Udacity https://github.com/udacity/SFND_2D_Feature_Tracking and added/modified the following files.

- **Writeup_of_project2.pdf**: This file
- **README.md**: Read me file of this repository
- **src**
  - **MidTermProject_Camera_Student.cpp**: Main script to set the initial conditions and run the functions.
  - **matching2D_Student.cpp**: Script used to create the functions of detectors, descriptors and matchers.
- **result**
  - **Project2_result.xlsx**: The resulting list of calculating keypoint numbers and time to execute of all possible combinations with detectors/descriptors.
  - **detector_***_descriptor_***.txt**: The result of calculation with each combination of detectors/descriptors.
  - **detector_***_descriptor_***.jpg**: The image of matching the 1st and 2nd frame with each combination of detectors/descriptors.

4. Reflection

(1)The Data Buffer

Task MP.1

I focused on loading images, setting up data structures and putting everything into a ring buffer to optimize memory load. I modified "MidTermProject_Camera_Student.cpp" as following.

I added the logic to erase the oldest dataBuffer if data size of buffer is bigger than specified size. **(A)**
In this case, the specified size is 2 because we only need 2 pairs to execute matching process. **(B)**
As a result, I checked the buffer size never becomes more than 2. **(C)**

B

```
// misc
int dataBufferSize = 2;       // no. of images which are held in memory (ring buffer) at the same time
vector<DataFrame> dataBuffer; // list of data frames which are held in memory at the same time
```

```
//// STUDENT ASSIGNMENT
//// TASK MP.1 -> replace the following code with ring buffer of size dataBufferSize
cout << "-----imgIndex: " << imgIndex << "-----" << endl;

// push image into data frame buffer
DataFrame frame;
frame.cameraImg = imgGray;
dataBuffer.push_back(frame);

int dataSize = dataBuffer.size();

if (dataSize > dataBufferSize){
    dataBuffer.erase(dataBuffer.begin()); // There's no methos "pop_front"
    dataBuffer.shrink_to_fit(); // Release memory
}

// Debug
cout << "dataBuffer size: " << dataBuffer.size() << endl;
//cout << "dataBuffer capacity: " << dataBuffer.capacity() << endl;

//// EOF STUDENT ASSIGNMENT
cout << "#1 : LOAD IMAGE INTO BUFFER done" << endl;
```

A

C

(2)Keypoint Detection

Task MP.2

My second task is to focus on keypoint detection.

In the student version of the code there’s already an existing implementation of the Shi-Tomasi detector.

I implemented a selection of alternative detectors, which are HARRIS, FAST, BRISK, ORB, AKAZE, and SIFT in “matching2D_Student.cpp” as following.

HARRIS detector is a traditional method, so there’s the function only for HARRIS as below.

```
// Detect keypoints in image using the traditional Harris detector
void detKeypointsHarris(vector<cv::KeyPoint> &keypoints, cv::Mat &img, bool bVis)
{
    // compute detector parameters based on image size
    int blockSize = 4;       //  size of an average block for computing a derivative covariation matrix over each pix
    int apertureSize = 3;
    int minResponse = 100;
    double k = 0.04;

    // Apply corner detection
    double t = (double)cv::getTickCount();
    cv::Mat dst, dst_norm, dst_norm_scaled;
    dst = cv::Mat::zeros(img.size(), CV_32FC1);
    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);

    // remove the unnecessary points on 4 border lanes to make calculation easier
    int count = 0;
    int neighber = 3; // parameter to neglect the lower value point in this range
```

```
    for(int r = neighber; r < dst_norm.rows-neighber; r++) {
        for(int c = neighber; c < dst_norm.cols-neighber; c++) {
            int response = (int)dst_norm.at<float>(r, c);
            if (response > minResponse) {
                int comp_cnt = 0;
                int comp_thresh = (2 * neighber +1) * (2 * neighber +1) -1;
                for (int i = -neighber; i <= neighber; i++) {
                    for (int j = -neighber; j <= neighber; j++) {
                        if (response > (int)dst_norm.at<float>(r+i,c+j)) {
                            comp_cnt += 1;
                        }
                    }
                }

                if (comp_cnt == comp_thresh) {
                    cv::KeyPoint newKeyPoint;
                    newKeyPoint.pt = cv::Point2f(c, r);
                    //newKeyPoint.pt = cv::Point2f(r, c);
                    newKeyPoint.size = 2 * apertureSize;
                    keypoints.push_back(newKeyPoint);
                    count += 1;
                }

            }
        }
    }
```

Other detectors are modern methods, so they are implemented in one function as below.

I use IF Statements to switch the detector type.

```
void detKeypointsModern(vector<cv::KeyPoint> &keypoints, cv::Mat &img, string detectorType, bool bVis)
{
    // Refer 2D Features Framework in OpenCV
    // Reference: https://docs.opencv.org/master/d9/d97/tutorial_table_of_content_features2d.html
    cv::Ptr<cv::FeatureDetector> detector;
    // (1)FAST
    if (detectorType.compare("FAST") == 0) {
        // Reference: Udacity Lesson solution
        int threshold = 30;  // difference between intensity of the central pixel and pixels of a circle around this
        bool bNMS = true;    // perform non-maxima suppression on keypoints
        cv::FastFeatureDetector::DetectorType type = cv::FastFeatureDetector::TYPE_9_16; // TYPE_9_16, TYPE_7_12, TYP
        detector = cv::FastFeatureDetector::create(threshold, bNMS, type);
   }
    // (2)BRISK
    else if (detectorType.compare("BRISK") == 0) {
        detector = cv::BRISK::create();
    }
    // (3)ORB
    else if (detectorType.compare("ORB") == 0) {
        detector = cv::ORB::create();
    }
    // (4)AKAZE
    else if (detectorType.compare("AKAZE") == 0) {
        detector = cv::AKAZE::create();
    }
    // (5)SIFT
    else if (detectorType.compare("SIFT") == 0) {
        //Reference: Udacity Lesson solution
        detector = cv::xfeatures2d::SIFT::create();
    }
```

There are some codes to select the type of detector in "MidTermProject_Camera_Student.cpp" as below.

For logging purpose in Task MP.7, 8, 9, I changed the location of the selection in the beginning of Main Function.

(Same thing to the selection of descriptors and matchers)

```
    // Change the location for efficiency
    string detectorType = "FAST";  // Task MP.2 Modern fast methods: FAST, BRISK, ORB, AKAZE, SIFT // SIFT detector
    string descriptorType = "ORB"; // BRIEF, ORB, FREAK, AKAZE, SIFT, BRISK // SIFT/AKAZE descriptor is only good wi
    string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN // Basically use BF because it's assigned in Lesson i
    // Change the string name to avoid duplication
    string descType = "DES_BINARY"; // DES_BINARY, DES_HOG // Basically use BINARY because it's faster
    //string descType = "DES_HOG"; // DES_BINARY, DES_HOG // Only use HOG with SIFT
    string selectorType = "SEL_KNN";       // SEL_NN, SEL_KNN // Use KNN with minDescDistRatio: 0.8
```

In the Main Function, I use the 3 functions to calculate detection of keypoints as following.

```
        //// STUDENT ASSIGNMENT
        //// TASK MP.2 -> add the following keypoint detectors in file matching2D.cpp and enable string-based selecti
        //// -> HARRIS, FAST, BRISK, ORB, AKAZE, SIFT

        // Calculate time for logging in main function
        double t1 = (double)cv::getTickCount();

        if (detectorType.compare("SHITOMASI") == 0) {
            detKeypointsShiTomasi(keypoints, imgGray, bVis);
            //detKeypointsShiTomasi(keypoints, imgGray, false);
        }
        else if (detectorType.compare("HARRIS") == 0) {
            detKeypointsHarris(keypoints, imgGray, bVis);
            //detKeypointsHarris(keypoints, imgGray, false);
        }
        else {
            detKeypointsModern(keypoints, imgGray, detectorType, bVis);
            //detKeypointsModern(keypoints, imgGray, detectorType, false);
        }
        //// EOF STUDENT ASSIGNMENT
```

Task MP.3

My third task is to remove all keypoints outside of a bounding box around the preceding vehicle.

Box parameters you should use are : cx = 535, cy = 180, w = 180, h = 150.

I implemented it in "MidTermProject_Camera_Student.cpp" as following.

At first I tried to create another KeyPoint object and push_bach the cropped keypoints, but I found that it's wasting the memory resources. **(A)**

Instead of that, I erased the keypoints, whose positions are out of range, from the original KeyPoint object. **(B)**

```
    //// STUDENT ASSIGNMENT
    //// TASK MP.3 -> only keep keypoints on the preceding vehicle
    // Use it only in this project
    bool bFocusOnVehicle = true;
    vector<float> vehicleRect{535, 180, 180, 150}; // Roughly define rectangle area.
    //cv::Rect vehicleRect(535, 180, 180, 150); // Roughly define rectangle area.
    vector<cv::KeyPoint> keypointsCropped; // create empty feature list for after cropping
    int keypointSize = keypoints.size();

    if (bFocusOnVehicle)
    {
```

A

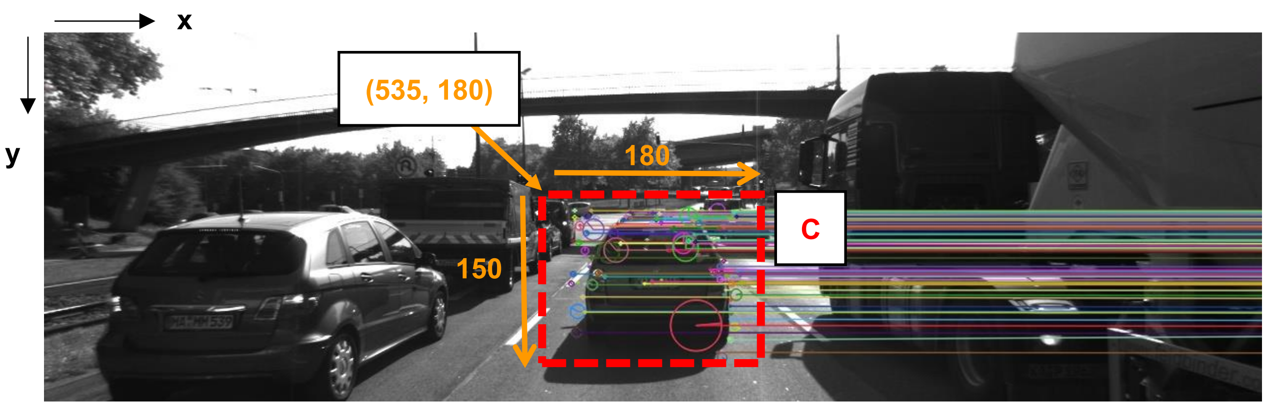
```
        /*
        // (i) push back
        for (int i = 0; i < keypointSize; i++){
            if ((keypoints[i].pt.x > vehicleRect[0]) && (keypoints[i].pt.x < (vehicleRect[0]+vehicleRect[2])) &&
                (keypoints[i].pt.y > vehicleRect[1]) && (keypoints[i].pt.y < (vehicleRect[1]+vehicleRect[3]))) {
                    keypointsCropped.push_back(keypoints[i]);
                    cout << "cropped keypoints[" << i << "]: x=" << keypoints[i].pt.x << ", y=" << keypoints[i].
            }
        }
        */
```

B

```
       // (ii) erase (be careful about for iteration!)
        for (int i = keypointSize; i > 0; i--){
            if ((keypoints[i].pt.x < vehicleRect[0]) || (keypoints[i].pt.x > (vehicleRect[0]+vehicleRect[2])) ||
                (keypoints[i].pt.y < vehicleRect[1]) || (keypoints[i].pt.y > (vehicleRect[1]+vehicleRect[3]))) {
                    keypoints.erase(keypoints.begin() + i);
                    //cout << "cropped keypoints[" << i << "]: x=" << keypoints[i].pt.x << ", y=" << keypoints[i
            }
        }
```

The resulting keypoints are shown below. (e.g. detector = SIFT)

I was able to remove the keypoints outside the specified range. **(C)**



(3)Descriptor Extraction & Matching

Task MP.4

My fourth task is to implement a variety of keypoint descriptors to the already implemented BRISK method and make them selectable using the string 'descriptorType'. The methods I must integrate are BRIEF, ORB, FREAK, AKAZE and SIFT. The SURF is not a part of the mid-term project.

I implemented the source codes of each descriptor in "matching2D_Student.cpp" as below.

I mainly referred to OpenCV reference pages.

```cpp
// 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 keypoi

        extractor = cv::BRISK::create(threshold, octaves, patternScale);
    }
    else if  (descriptorType.compare("BRIEF") == 0)
    {
        // Reference: https://docs.opencv.org/master/d1/d93/classcv_1_1xfeatures2d_1_1BriefDescriptorExtractor.html#a
        int bytes = 32;
        bool use_orientation = false;

        extractor = cv::xfeatures2d::BriefDescriptorExtractor::create(bytes, use_orientation);
    }

    else if  (descriptorType.compare("ORB") == 0)
    {
        // Reference: https://docs.opencv.org/master/db/d95/classcv_1_1ORB.html
        int     nfeatures = 500;
        float   scaleFactor = 1.2f;
        int     nlevels = 8;
        int     edgeThreshold = 31;
        int     firstLevel = 0;
        int     WTA_K = 2;
        auto    scoreType = cv::ORB::HARRIS_SCORE;
        int     patchSize = 31;
        int     fastThreshold = 20;

        extractor = cv::ORB::create(nfeatures, scaleFactor, nlevels, edgeThreshold, firstLevel,
                                    WTA_K, scoreType, patchSize, fastThreshold);
    }

    else if  (descriptorType.compare("FREAK") == 0)
    {
        // Reference: https://docs.opencv.org/master/df/db4/classcv_1_1xfeatures2d_1_1FREAK.html
        bool    orientationNormalized = true;
        bool    scaleNormalized = true;
        float   patternScale = 22.0f;
        int     nOctaves = 4;
        const std::vector<int> & selectedPairs = std::vector<int>();

        extractor = cv::xfeatures2d::FREAK::create(orientationNormalized, scaleNormalized,
                                                    patternScale, nOctaves, selectedPairs);
    }
```

```cpp
    else if  (descriptorType.compare("AKAZE") == 0)
    {
        // Reference: https://docs.opencv.org/master/d8/d30/classcv_1_1AKAZE.html
        auto    descriptor_type = cv::AKAZE::DESCRIPTOR_MLDB;
        int     descriptor_size = 0;
        int     descriptor_channels = 3;
        float   threshold = 0.001f;
        int     nOctaves = 4;
        int     nOctaveLayers = 4;
        auto    diffusivity = cv::KAZE::DIFF_PM_G2;

        extractor = cv::AKAZE::create(descriptor_type, descriptor_size, descriptor_channels,
                                      threshold, nOctaves, nOctaveLayers, diffusivity);
    }
```

I referred to the solution source codes in Udacity Lesson for SIFT as below.

```cpp
    else if  (descriptorType.compare("SIFT") == 0)
    {
        // Reference: https://docs.opencv.org/master/d7/d60/classcv_1_1SIFT.html
        /*
        int     nfeatures = 0;
        int     nOctaveLayers = 3;
        double  contrastThreshold = 0.04;
        double  edgeThreshold = 10;
        double  sigma = 1.6;
        extractor = cv::xfeatures2d::SIFT::create(nfeatures, nOctaveLayers, contrastThreshold,
                                                  edgeThreshold, sigma);
        */

        // Reference: Udacity Lesson solution
        extractor = cv::xfeatures2d::SiftDescriptorExtractor::create();
    }
```

There are some codes to select the type of descriptor in "MidTermProject_Camera_Student.cpp" as below.

As already written in detector section, there's the selection part in the beginning of Main Function.

```cpp
    // Change the location for efficiency
    string detectorType = "FAST";  // Task MP.2 Modern fast methods: FAST, BRISK, ORB, AKAZE, SIFT // SIFT detector
    string descriptorType = "ORB"; // BRIEF, ORB, FREAK, AKAZE, SIFT, BRISK // SIFT/AKAZE descriptor is only good wi
    string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN // Basically use BF because it's assigned in Lesson i
    // Change the string name to avoid duplication
    string descType = "DES_BINARY"; // DES_BINARY, DES_HOG // Basically use BINARY because it's faster
    //string descType = "DES_HOG"; // DES_BINARY, DES_HOG // Only use HOG with SIFT
    string selectorType = "SEL_KNN";       // SEL_NN, SEL_KNN // Use KNN with minDescDistRatio: 0.8
```

Same as detector, in main function I run the function to calculate description as following.

```cpp
        //// STUDENT ASSIGNMENT
        //// TASK MP.4 -> add the following descriptors in file matching2D.cpp and enable string-based selection base
        //// -> BRIEF, ORB, FREAK, AKAZE, SIFT

        // Calculate time for logging in main function
        double t2 = (double)cv::getTickCount();

        cv::Mat descriptors;
        // Change the location of descriptorType for efficiency
        //string descriptorType = "ORB"; // BRIEF, ORB, FREAK, AKAZE, SIFT, BRISK
        //string descriptorType = "BRISK"; // BRIEF, ORB, FREAK, AKAZE, SIFT
        descKeypoints((dataBuffer.end() - 1)->keypoints, (dataBuffer.end() - 1)->cameraImg, descriptors, descriptorTy
        //// EOF STUDENT ASSIGNMENT
```

Task MP.5

My fifth task focuses on the matching part. The current implementation uses Brute Force matching combined with Nearest-Neighbor selection. I must now add FLANN as an alternative to brute-force as well as the K-Nearest-Neighbor approach.

I implemented the source codes of each matcher and selector in "matching2D_Student.cpp" as below.

To implement FLANN, I mainly referred to the solution source codes in Udacity Lesson. **(A)**

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

    if (matcherType.compare("MAT_BF") == 0)
    {
        // Reference: Udacity Lesson solution
        int normType = descriptorType.compare("DES_BINARY") == 0 ? cv::NORM_HAMMING : cv::NORM_L2;
        //int normType = cv::NORM_HAMMING;
        matcher = cv::BFMatcher::create(normType, crossCheck);
    }
    else if (matcherType.compare("MAT_FLANN") == 0)
    {
        // Reference: Udacity Lesson solution
        // Solution by the past mentor: https://knowledge.udacity.com/questions/211123
        if (descSource.type() != CV_32F || descRef.type() != CV_32F)
        //if (descSource.type() != CV_32F)
        { // OpenCV bug workaround : convert binary descriptors to floating point due to a bug in current OpenCV impl
            descSource.convertTo(descSource, CV_32F);
            descRef.convertTo(descRef, CV_32F);
        }

        matcher = cv::DescriptorMatcher::create(cv::DescriptorMatcher::FLANNBASED);
        cout << "FLANN matching";
    }
```

There are some codes to select the type of matcher in "MidTermProject_Camera_Student.cpp" as below.

As already written before, there's the selection part in the beginning of Main Function.

The "descriptorType" becomes duplication, so I changed the variable name as following. (B)

```
    // Change the location for efficiency
    string detectorType = "FAST";  // Task MP.2 Modern fast methods: FAST, BRISK, ORB, AKAZE, SIFT // SIFT detector
    string descriptorType = "ORB"; // BRIEF, ORB, FREAK, AKAZE, SIFT, BRISK // SIFT/AKAZE descriptor is only good wi
    string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN // Basically use BF because it's assigned      son i
    // Change the string name to avoid duplication
    string descType = "DES_BINARY"; // DES_BINARY, DES_HOG // Basically use BINARY because it's faster
    //string descType = "DES_HOG"; // DES_BINARY, DES_HOG // Only use HOG with SIFT
    string selectorType = "SEL_KNN";        // SEL_NN, SEL_KNN // Use KNN with minDescDistRatio: 0.8
```

Matcher needs more than 1 image frame, so it runs only when dataBuffer size is more than 1.

```
        if (dataBuffer.size() > 1) // wait until at least two images have been processed
        {

            /* MATCH KEYPOINT DESCRIPTORS */

            vector<cv::DMatch> matches;
            /* // Change the location of matcher parameters for efficiency
            //string matcherType = "MAT_FLANN";        // MAT_BF, MAT_FLANN
            string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN
            // Change the string name to avoid duplication
            string descType = "DES_HOG"; // DES_BINARY, DES_HOG
            //string descriptorType = "DES_BINARY"; // DES_BINARY, DES_HOG
            //string descriptorType = "DES_BINARY"; // DES_BINARY, DES_HOG
            string selectorType = "SEL_KNN";       // SEL_NN, SEL_KNN
            //string selectorType = "SEL_NN";       // SEL_NN, SEL_KNN
            */

            //// STUDENT ASSIGNMENT
            //// TASK MP.5 -> add FLANN matching in file matching2D.cpp
            //// TASK MP.6 -> add KNN match selection and perform descriptor distance ratio filtering with t=0.8 in f

            // Calculate time for logging in main function
            double t3 = (double)cv::getTickCount();

            matchDescriptors((dataBuffer.end() - 2)->keypoints, (dataBuffer.end() - 1)->keypoints,
                             (dataBuffer.end() - 2)->descriptors, (dataBuffer.end() - 1)->descriptors,
                             matches, descType, matcherType, selectorType);
                             //matches, descriptorType, matcherType, selectorType);
```

Task MP.6

As my sixth task, I then implement the descriptor distance ratio test as a filtering method to remove bad keypoint matches.

I implemented the source codes of each matcher and selector in "matching2D_Student.cpp" as below.
To implement KNN and filter matches using descriptor distance ratio test, I mainly referred to the solution source codes in Udacity Lesson. **(A), (B)**

```
    // perform matching task
    if (selectorType.compare("SEL_NN") == 0)
    { // nearest neighbor (best match)
        // Reference: Udacity Lesson solution
        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)
        // Reference: Udacity Lesson solution
        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();
        cout << " (KNN) with n=" << knn_matches.size() << " matches in " << 1000 * t / 1.0 << " ms" << endl;

        // 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;
    }
}
```

(4)Performance Evaluation

Task MP.7

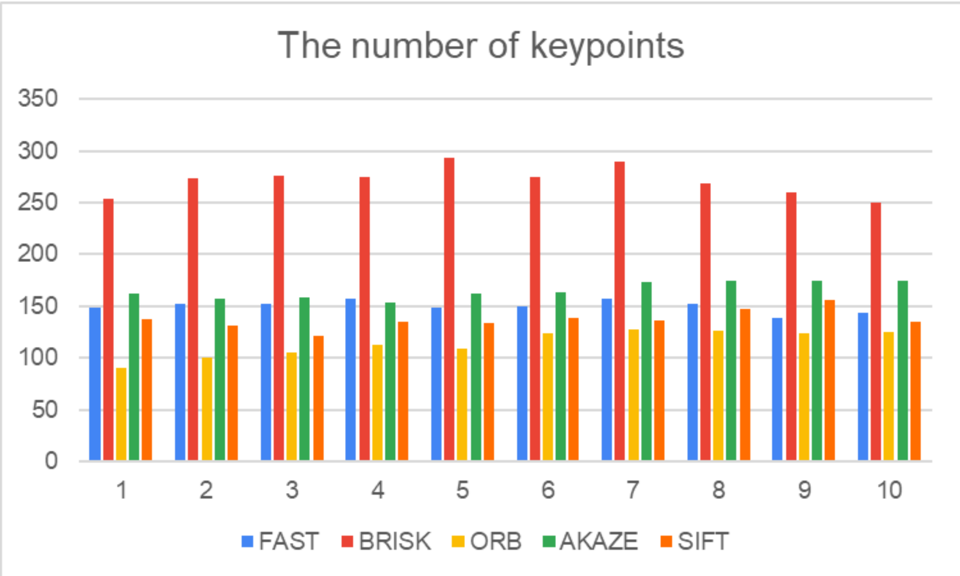
My seventh task is to count the number of keypoints on the preceding vehicle for all 10 images and take note of the distribution of their neighborhood size. Do this for all the detectors you have implemented.

I followed the mentor's instruction "SIFT detector only works well with SIFT descriptor whose type is DES_HOG". I used DES_HOG only for this combination because DES_BINARY is faster.
https://knowledge.udacity.com/questions/323235

The number of keypoints with each detector is shown below.
BRISK has the most number, and ORB has the smallest.

The number of keypoints — *: minDescDistRatio = 0.8

Detector	Descriptor	Matcher	Descriptor Type	Selector Type *	Image										Average	Standard deviation
					1	2	3	4	5	6	7	8	9	10		
FAST	ORB	BF	BINARY	KNN	149	152	152	157	149	150	157	152	139	144	150.1	5.4660569
BRISK	ORB	BF	BINARY	KNN	254	274	276	275	293	275	289	268	260	250	271.4	13.874037
ORB	ORB	BF	BINARY	KNN	91	101	106	113	109	124	128	127	124	125	114.8	12.769756
AKAZE	ORB	BF	BINARY	KNN	162	157	159	154	162	163	173	175	175	175	165.5	8.1955272
SIFT	SIFT	BF	HOG	KNN	137	131	121	135	134	139	136	147	156	135	137.1	9.2790086



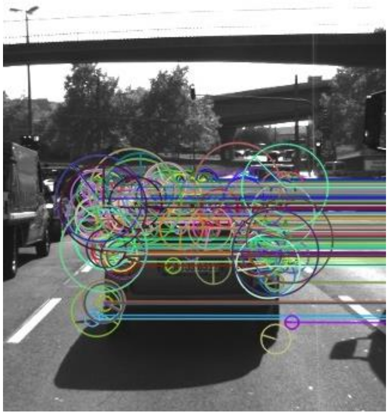
I take note of the distribution of their neighborhood size of each detector as below.
As written in the mentor's instruction, I just check them in each image file. (No Calculation)
https://knowledge.udacity.com/questions/106021

FAST and AKAZE have mostly small neighborhood sizes, and the dispersion looks small.
BRISK and SIFT have small and large neighborhood sizes, and the dispersion looks large.
ORB have mostly large neighborhood sizes. Only ORB has the mismatched keypoint on the tree. **(A)**

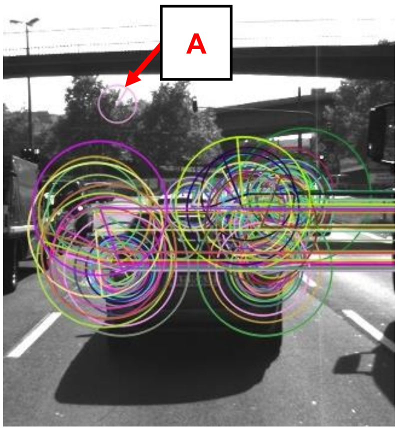
<u>Overall, **FAST** and **AKAZE** look good detector compared with others.</u>

<FAST>

<BRISK>



<ORB>



<AKAZE>

<SIFT>

Task MP.8

My eighth task is to count the number of matched keypoints for all 10 images using all possible combinations of detectors and descriptors.

In the matching step, use the BF approach with the descriptor distance ratio set to 0.8.

As written above, I ran SIFT detector only with SIFT descriptor to avoid error.

I also followed the mentor's instruction "AKAZE descriptor only works well with AKAZE detector.

(We need to implement AKAZE detector with other descriptors, though)

https://knowledge.udacity.com/questions/163998

I show the result of number of matched keypoints as following.

The number of matched keypoints — *: minDescDistRatio = 0.8

Detector	Descriptor	Matcher	Descriptor Type	Selector Type *	Image										Average	Standard deviation
					1	2	3	4	5	6	7	8	9	10		
FAST	BRIEF	BF	BINARY	KNN	0	119	129	119	126	109	123	131	124	119	122.11111	6.5849154
FAST	ORB	BF	BINARY	KNN	0	118	123	113	126	107	122	123	123	119	119.33333	5.9791304
FAST	FREAK	BF	BINARY	KNN	0	98	99	92	98	86	99	104	101	105	98	5.8736701
FAST	AKAZE	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
FAST	SIFT	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
FAST	BRISK	BF	BINARY	KNN	0	97	104	102	98	85	107	108	100	100	100.11111	6.8088994
BRISK	BRIEF	BF	BINARY	KNN	0	172	194	181	176	181	192	207	185	178	185.11111	10.867894
BRISK	ORB	BF	BINARY	KNN	0	154	166	154	160	156	180	164	169	170	163.66667	8.660254
BRISK	FREAK	BF	BINARY	KNN	0	154	173	153	168	158	181	169	175	165	166.22222	9.653727
BRISK	AKAZE	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
BRISK	SIFT	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
BRISK	BRISK	BF	BINARY	KNN	0	167	168	156	169	172	184	173	167	183	171	8.5732141
ORB	BRIEF	BF	BINARY	KNN	0	49	42	44	58	53	75	65	82	65	59.222222	13.727507
ORB	ORB	BF	BINARY	KNN	0	66	68	70	84	90	98	90	90	89	82.777778	11.680944
ORB	FREAK	BF	BINARY	KNN	0	41	36	44	47	44	51	52	47	54	46.222222	5.6960025
ORB	AKAZE	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
ORB	SIFT	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
ORB	BRISK	BF	BINARY	KNN	0	73	73	78	85	79	88	87	87	90	82.222222	6.5722481
AKAZE	BRIEF	BF	BINARY	KNN	0	136	132	129	132	135	145	149	148	151	139.66667	8.5146932
AKAZE	ORB	BF	BINARY	KNN	0	127	127	125	119	129	130	135	136	143	130.11111	7.0257463
AKAZE	FREAK	BF	BINARY	KNN	0	123	127	128	122	123	133	145	145	135	131.22222	8.9830087
AKAZE	AKAZE	BF	BINARY	KNN	0	134	137	131	127	128	145	146	149	148	138.33333	8.8034084
AKAZE	SIFT	BF	BINARY	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
AKAZE	BRISK	BF	BINARY	KNN	0	133	123	128	129	130	131	141	144	140	133.22222	6.9602043
SIFT	BRIEF	BF	HOG	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	ORB	BF	HOG	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	FREAK	BF	HOG	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	AKAZE	BF	HOG	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	SIFT	BF	HOG	KNN	0	81	78	82	91	89	80	82	99	100	86.888889	8.2831824
SIFT	BRISK	BF	HOG	KNN	0	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!

Task MP.9

My ninth task is to log the time it takes for keypoint detection and descriptor extraction.

The results must be entered into a spreadsheet and based on this information I then suggest the TOP3 detector / descriptor combinations as the best choice for our purpose of detecting keypoints on vehicles.

Finally, in a short text, I justify my recommendation based on my observations and on the data I collected.

I used the same matcher (BF) and selector (KNN with minDescDistRatio = 0.8) as MP.7, 8.

The summation of detection time and description time is shown below.

Log time of detection & description				*: minDescDistRatio = 0.8													
Detector	Descriptor	Matcher	Descriptor Type	Selector Type *	Image										Average	Standard deviation	
					1	2	3	4	5	6	7	8	9	10			
FAST	BRIEF	BF	BINARY	KNN	13.78482	2.63205	2.589599	1.918801	1.715051	2.0994	1.860771	1.922764	1.803991	2.65278	3.2980027	3.7023798	
FAST	ORB	BF	BINARY	KNN	3.91497	2.46766	2.83399	3.36009	2.379488	2.115638	2.42344	2.17239	2.14725	2.378315	2.6193231	0.589425	B
FAST	FREAK	BF	BINARY	KNN	50.0574	47.47236	47.129002	44.7902	46.460801	44.75397	44.31788	44.368731	44.23166	43.67391	45.725591	2.0113239	
FAST	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
FAST	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
FAST	BRISK	BF	BINARY	KNN	341.99349	334.44043	333.39725	336.80917	333.8677	335.38144	337.70074	337.07355	330.86572	333.43304	335.49625	3.0762553	
BRISK	BRIEF	BF	BINARY	KNN	379.58215	375.14927	383.07944	370.36348	373.95816	375.89489	371.29072	378.383	369.48902	370.16587	374.7356	4.5695051	
BRISK	ORB	BF	BINARY	KNN	398.5169	371.75964	375.5496	372.39759	372.07871	371.91483	371.24304	373.17009	371.8695	379.15854	375.76584	8.3490715	
BRISK	FREAK	BF	BINARY	KNN	428.0999	417.6433	429.0905	415.5593	416.184	419.347	425.2221	408.478	413.0412	419.2958	419.19611	6.5911802	
BRISK	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
BRISK	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
BRISK	BRISK	BF	BINARY	KNN	707.415	703.477	718.42	708.213	713.271	710.001	720.133	705.321	699.717	707.737	709.3705	6.3725998	
ORB	BRIEF	BF	BINARY	KNN	18.0461	11.22003	9.540118	8.499755	8.955175	8.478594	9.057613	8.442334	9.426711	8.567693	10.023412	2.9398764	A
ORB	ORB	BF	BINARY	KNN	24.38513	15.47827	16.14907	15.02095	15.52066	15.37132	15.01398	14.2217	15.01864	15.24786	16.142758	2.937269	
ORB	FREAK	BF	BINARY	KNN	56.7768	53.8343	52.68927	49.62825	50.42939	51.10066	49.662	50.742	49.93431	51.09253	51.588951	2.2609773	
ORB	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
ORB	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
ORB	BRISK	BF	BINARY	KNN	344.4093	341.66439	347.37533	340.89892	345.59549	339.69733	345.60187	339.40693	339.70318	338.60915	342.29619	3.159592	
AKAZE	BRIEF	BF	BINARY	KNN	123.07029	111.03576	112.47584	107.87008	107.17069	108.26384	108.0801	115.04378	111.16318	109.51094	111.36845	4.779381	
AKAZE	ORB	BF	BINARY	KNN	118.53384	111.58489	114.82577	109.64297	112.58801	110.95995	109.88618	110.30779	110.02497	112.61345	112.09678	2.7758205	
AKAZE	FREAK	BF	BINARY	KNN	152.2798	149.398	157.9001	150.7148	147.4728	145.9482	158.9215	147.145	140.1395	145.9262	149.58459	5.6875114	
AKAZE	AKAZE	BF	BINARY	KNN	202.7055	188.451	194.6408	196.3547	196.0294	191.6899	191.2015	194.823	193.0745	194.249	194.32193	3.8061133	
AKAZE	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
AKAZE	BRISK	BF	BINARY	KNN	454.75	449.233	443.869	450.253	438.771	448.095	441.489	449.89	443.54	435.508	445.5398	5.9234794	
SIFT	BRIEF	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
SIFT	ORB	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
SIFT	FREAK	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
SIFT	AKAZE	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	
SIFT	SIFT	BF	HOG	KNN	309.244	234.2365	256.557	240.33	247.321	234.1698	233.4812	233.8341	234.4652	233.9066	245.75454	23.586508	
SIFT	BRISK	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	

The time of ORB detector is short **(A)**, but keypoints of ORB are not stable as I wrote in Task MP.7.

Besides ORB, the TOP 3 fast methods are following. **(B)**

(1) FAST detector & ORB descriptor: average 2.6 msec, standard deviation: 0.6 msec

(2) FAST detector & BRIEF descriptor: average 3.3 msec, standard deviation: 3.7 msec

(3) FAST detector & FREAK descriptor: average 45.7 msec, standard deviation: 2.0 msec

(3) is more than 10 times slower than (1), (2), so it's not a good solution.

(2) is a little slower than (1), but the standard deviation is much larger than (1) so it's not good either.

As a conclusion, (1)FAST detector & ORB descriptor is the best combination in this project images.

The matching image of this combination is shown below. It looks there's no mismatch.

"result/detector_FAST_descripter_ORB_image.jpg"