Comparison of OpenCV's Feature Detectors and Feature Matchers

Frazer K. Noble
Centre for Additive Manufacturing
School of Engineering and Advanced Technology
Massey University
New Zealand
Email: f.k.noble@massey.ac.nz

***Abstract*— There exists a range of feature detecting and feature matching algorithms; many of which have been included in the Open Computer Vision (OpenCV) library. However, given these different tools, which one should be used? This paper discusses the implementation and comparison of a range of the library's feature detectors and feature matchers. It shows that the Speeded-Up Robust Features (SURF) detector found the greatest number of features in an image, and that the Brute Force (BF) matcher matched the greatest number of detected features in an image pair. Given a benchmark image set, OpenCV's SURF detector found, on average, 1907.20 features in 1538.61 ms, and OpenCV's BF matcher, on average, matched features in 160.24 ms. The combination of the Binary Robust Invariant Scalable Key-points (BRISK) detector and BF matcher was found to be the highest ranked combination of OpenCV's feature detectors and feature matchers; on average, detecting and matching 1132.00 and 80.20 features, respectively, in 265.67 ms. It was concluded that if the number of features detected is important, the SURF detector should be used; else, if the number of features matched is important, the BF matcher should be used; otherwise, the combination of the OpenCV's BRISK feature detector and BF feature matcher should be used.**

***Keywords*—OpenCV; Feature Detectors; Feature Matchers; Comparison**

I. Introduction

Reconstructing a three dimensional (3D) model of a scene from photographs taken from different view-points, i.e. Structure from Motion (SFM), starts with detecting features in each photograph and matching the detected features in a subsequent photograph or photographs [1].

There exists a wide range of feature detectors, e.g. Scale-Invariant Feature Transform (SIFT) [2], Speeded-Up Robust Features (SURF) [3], Binary Robust Invariant Scalable Key-points [4], Oriented FAST and Rotated BRIEF (ORB) [5], KAZE [6], and Accelerated KAZE (AKAZE) [7], and feature matchers, e.g. Brute Force (BF) and Fast Library for Approximate Nearest Neighbors (FLANN) [8]; many of which have been implemented in the Open Computer Vision (OpenCV) library [9].

Given these different detectors and matchers, the question: "which ones should be used?" arises. As such, this paper discusses the implementation and comparison of OpenCV's SIFT, SURF, BRISK, ORB, KAZE, and AKAZE feature detectors, and OpenCV's BF and FLANN feature matchers. Given a benchmark image set, the implemented detectors' and matchers' number of detected features and number of features matched, and the times taken to detect and match them, are presented. In addition, a novel comparison tool is described and used to rank the combination of different feature detectors and feature matchers.

The remainder of this paper is organised as follows: next is Section II, which describes the tools used and implementation of OpenCV's detectors and matchers; after that Section III, which presents the results of the comparison of the detectors and matchers; next is Section IV, which discusses the previous section's results; and lastly Section V, which concludes this paper.

II. Methodology

In this section, the implementation and comparison of OpenCV's feature detectors and feature matchers is described. First, the tools used are listed; then, how OpenCV's detectors and matchers have been implemented is described; lastly, a tool for comparing each combination of the detectors and matchers is presented.

A. Implementation

In order to have comparable results, a benchmark image set, "Sculpture", was chosen and downloaded from http://vision.princeton.edu/courses/SFMedu/. Sculpture's images were 480 $\times$ 640 pixels in size and of JPEG file type.

OpenCV 3.1 has been used and its source code was downloaded from https://github.com/Itseez/opencv/releases/tag/3.1.0. The opencv_contrib repository was also required and its source code was downloaded from https://github.com/Itseez/opencv_contrib. Instructions from https://github.com/Itseez/opencv_contrib were followed to build the library from its source code; where, CMAKE 3.5.1 was used and was downloaded from https://cmake.org/. This paper's main program was written in C++ using Microsoft's Visual Studio 2015 integrated development environment (IDE).

All measurements have been taken on a 1.70 GHz Intel Core i5-3317U, 4.00 GB RAM, x64 Windows 10 Professional personal computer (PC).

(a) (b) (c) (d) (e)

Fig. 1. The Sculpture benchmark image set. This benchmark images have been used to compare the number of features detected and matched across each image and image pair by the implemented OpenCV detectors and matchers. Their image pairs are formed as follows: [(a),(b)], [(b),(c)], [(c),(d)], and [(d),(e)]

B. Comparison Algorithms

In order to access OpenCV's feature detectors and feature matchers, the library and opencv_contrib repository's source code was downloaded and then built using CMAKE. A C++ project was created in the IDE and a header file, main.h, and source file, main.cpp, included. Within main.h, OpenCV's headers features2d.h and xfeatures2d.h were included, and the namespace cmp, continaing global variables and classes allFrames and pairFrames, defined. Within main.cpp, the main program's algorithm was implemented. Here, its data-types, keywords, and important variables are differentiated by type-setting them as **`int`**, `define` and `variable`, respectively.

Algorithm 1 illustrates the organisation of the namespace cmp; where, variables `detectFlag` and `matchFlag` are enumerations of the different feature detectors and feature matchers tested, the class `allFrames` stores a set of images, and class `pairFrames` stores a pair of images, their key points, descriptors, and matched key points.

Class `allFrames`'s member variable `frames` is a vector of OpenCV **`Mat`** objects, and stores all the images whose features are to be detected and matched. The member variable `frames` is populated using `allFrames`'s membership function `load()`.

Class `pairFrames`'s member variable `frames` is a vector of OpenCV Mat objects, and stores a pair of images whose features are detected and matched. The member variable `frames` is populated using `pairFrames`'s member function `load()`. The class's member variables `keyPoints` and `descriptors` are vectors of vectors of OpenCV **`KeyPoint`** and OpenCV **`Mat`** objects and stores detected key points and descriptors, respectively. The member variables are populated using the member function `detect()`, which takes a feature detector named in the enumeration `detectFlag` as a parameter. Class `pairFrames`'s member variable `goodMatches` is a vector of OpenCV **`DMatch`** objects and stores indices of the matched key points that passed a ratio test. The member variable is populated using the member function `match()`, which takes a feature matcher named in the enumeration `matchFlag` as a parameter. The member variables `dFlag` and `mFlag` are instances of the `detectFlag` and `matchFlag` enumeration and keep track of which feature detector and feature matcher were used. Class `pairFrames`' member variable `matchTime` is a double that holds the time taken to match the key points in `keyPoints`'s indices. The member variable `detectTime` is a vector of doubles that holds the time taken to detect the key points in `frames`'s indices. Both `matchTime` and `detectTime` are populated by the member function `getTime()`, which measures the time elapsed since member function `startTimer()` was called.

Algorithm 1

```
enum detectFlag
enum matchFlag

class allFrames {
public:
   vector<Mat> frames
   allFrames()
    void load(fileName)
}

class pairFrames {
public:
   vector<Mat> frames
   vector<vector<KeyPoint> keyPoints
   vector<Mat> descriptors
   vector<DMatch> goodMatches
   vector<double> detectTimes
   double matchTimes
   detectFlag dFlag
   matchFlag mFlag
   pairFrames()
   void load(Mat im1, Mat im2)
   void detect(detectFlag d)
   void match(matchFlag m)
   void startTimer(void)
   double getTime(void)
}
```

Algorithm 2 illustrates class `allFrames`'s `load()` member function. First, the member function creates a file stream to open a text file that contains a list of the file names of images that need to be loaded. Each line of the file is then read in and OpenCV's `imread()` function is used to store the image as a **`Mat`** object. The image is then pushed to the back of the `allFrame`'s member variable `frames`. Afterwards, the file stream is closed.

Algorithm 2

```
input: string fileName
create file stream f;
use f to open fileName.txt
for i= 0 to end of file do
  read line
  instantiate an OpenCV Mat object as image
  read image listed in line into image
  push image to back of frames
end for
close f
```

Algorithm 3

```
input: Mat i1, Mat i2
push i1 to back of frames
push i2 to back of frames
```

Algorithm 3 illustrates class `pairFrames`'s `load()` member function, which takes two OpenCV **`Mat`** objects as parameters and pushes them to the back of `pairFrames`'s member variable `frames`.

Algorithm 4 illustrates class `pairFrames`'s `detect()` member function, which takes an instance of the enumeration `detectFlag`, `d`, as a parameter. First, the member function tests `d`, comparing it against all the types of feature detectors in enumeration `detectFlag`. If `d` is a given type, member variable `dFlag` is set to `d`, and a pointer to the detector's `create()` member function created. The member function `startTimer()` is called, and the detector's `detectAndCompute()` member function called; where, member variables `frames`'s first index, `keyPoints` and `descriptors` are passed as parameters. The member function `getTime()` returns the time elapsed detecting and computing the key points and descriptors and is push to the back of member variable `detectTimes`. The previous three steps are repeated for member variable `frames`' second index. Note: the previous logic described for `d` equal to SIFT is the same for `d` equal to SURF, BRISK, ORB, KAZE, and AKAZE, but for creating a point of the given detector's type. For brevity's sake, we have not included the additional logic.

Algorithm 5 illustrates class `pairFrames`'s member function `match()`, which takes an instance of the enumeration `matchFlag`, `m`, as a parameter. First, the member function tests `m`, comparing it against all the types of feature matchers in enumeration `matchFlags`. If `m` is equal to a given type, member variable `mFlag` is set to `m`, and member variable `dFlag` is tested. If `m` is equal to BF and `dFlag` is equal to either SIFT, SURF, or KAZE, then the feature matcher will expect the descriptors to be floating point values and the distance will be calculated using OpenCV's `NORM_L2` flag (the square root of sum of squares); otherwise, if `m` is equal to BF and `dFlag` is equal to either BRISK, ORB, or AKAZE, then the feature match will expect the descriptors to be unsigned integers and the distance will be calculated using

Algorithm 4

```
input: enum detectFlag d
if d == SIFT then
  dFlag = d
  create pointer to SIFT's create()
  call startTimer()
  detect frames[0]'s key points
  compute descriptor
  time = getTime()
  push time to back of detectTime
  push key points to back of keyPoints
  push descriptor to back of descriptors
  call startTimer()
  detect frames[1]'s key points
  compute descriptor
  time = getTime()
  push time to back of detectTime
  push key points to back of keyPoints
  push descriptor to back of descriptors
else if d == ... then
  ⋮
end if
```

OpenCV's `NORM_HAMMING` flag. If `m` is equal to FLANN, then the feature matcher will expect the descriptors to be floating point values and the distance will be calculated using OpenCV's `NORM_L2` flag. Afterwards, the descriptors are converted and if `m` is equal to BF, the matcher is instantiated with the distance measurement flag. Class `pairFrames`'s member function `startTimer()` is called, and the matcher's `match()` member function called; where, member variables `descriptors` is passed as a parameter. A ratio test is then applied to identify good matches, as described in [2]; key points that pass are pushed to the back of member variable `goodMatches`. Note: the previous logic described for `m` equal to BF is the same for `m` equal to FLANN, but the FLANN-based matcher will not need to be instantiated using the distance flag. For brevity's sake, we have not included the additional logic.

Algorithm 6 illustrates this paper's main program's `main()` function. The function first instantiates the class `allFrames`. Next, it calls instantiated object's member function `load()`; populating its member variable `frames` with the reference images listed in a text file. Then, for all the feature detectors and feature matchers types, the class `pairFrames` instantiated and a pair of images loaded into its member variable `frames` using the member function `load()`. The image pair's features are then detected and matched using the member functions `detect()` and `match()`. This is repeated for all images to be detected and matched.

C. Comparison Tool

Given an application, the decision to use a specific feature detector or feature matcher may be based on a single objective, e.g. detect the greatest number of features or match features in

Algorithm 5

```
input: enum matchFlag m
if m == BF then
  mFlag == m
  if dFlag == SIFT || SURF || KAZE then
    type = float
    dist = NORM_L2
  else if dFlag == BRISK || ORB || AKAZE then
    type = uint8
    dist = NORM_HAMMING
  end if
  convert descriptors to type
  instantiate BFMatcher with dist
  call startTimer()
  match descriptors
  matchTime = getTime()
  for i= 0 to matches do
    if matches[i] <ratio then
      push matches[i] to back of good_matches
    end if
  end for
else if mats = ... then
  ⋮
end if
```

Algorithm 6

```
instantiate allFrames as ref
load images with ref's load()
vector<string> dets = ["SIFT",...]
vector<string> mats = ["BF",...]
for i = 0 to dets's size do
  for j = 0 to mats's size do
    for k = 0 to ref's frames's size−1 do
      instantiate pairFrames as p
      load images with p's load()
      detect with p's detect()
      match with p's match()
    end for
  end for
end for
```

the shortest time, and, as such, the choice of which detector or matcher to use is obvious. A non-trivial problem, however, is choosing a detector and matcher given conflicting objectives, e.g. detect the most number of features and match them in the shortest time possible. Therefore, a comparison tool is proposed here to assist in choosing a combination of detector and matcher. The tool is defined as follows:

$$N = k * \left[\frac{(fd + fm) \,/\, (td + tm)}{(fd_{\max} + fm_{\max}) \,/\, (td_{\min} + tm_{\min})} \right]; \quad (1)$$

where, N is the comparison tool's value, k is a constant, fd is the number of features detected by a given feature detector, $fd_{\max}$ is the maximum number of features detected, fm is the number of features matched in an image pair, $fm_{\max}$ is the maximum number of features matched, td is the time taken to detect fd features, $td_{\min}$ is the minimum time taken to detect features, tm is the time taken to match fd features in an image pair, and $tm_{\min}$ is the minimum time taken to match features. In this paper, $k = 100$ has been used and was empirically determined.

TABLE I
AVERAGE FEATURES DETECTED AND RUN TIME (MS)

Feature Detector	Features Detected	Run Time (ms)
SIFT	1368.20	2093.42
SURF	**1907.20**	1538.61
BRISK	1132.00	**150.94**
ORB	**500.00**	206.93
KAZE	933.20	**7134.63**
AKAZE	769.00	2748.33

III. COMPARISON RESULTS

In this section, the results from the comparison on OpenCV's feature detectors and feature matchers is presented. First, the number of features detected in each benchmark image, and the time taken, is presented. Then, the number of features matched across the benchmark set's image pairs, and the time taken, is presented. Lastly, the comparison tool's computed values for each combination of OpenCV's detectors and matchers are presented.

A. Feature Detection Comparison

The number of features detected and the time to detect these features are presented in this section. All results are computed on the benchmark image set (see Figure 1).

Table I tabulates the average number of features detected in each benchmark image by the feature detectors, and the average time to detect these features.

Figure 2 illustrates the number of features detected in each benchmark image by the feature detectors.

Figure 3 illustrates the time taken to detect features in each benchmark image by the feature detectors.

B. Feature Matching Comparison

The number of features matched and the time to match these features are presented in this section. All results are computed based of the features detected in the previous section.

Table II tabulates the average number of features matched across the benchmark set's image pairs by the feature matchers, and the time to match these features.

Figure 4 illustrates the number of features matched across the benchmark set's image pairs by the feature matchers.

Figure 5 illustrates the time taken to match the features across the benchmark set's image pairs by the feature matchers.

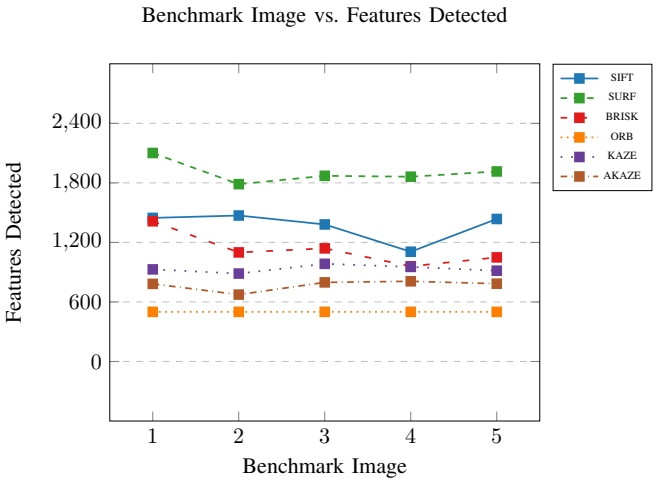

Fig. 2. Benchmark Image vs. Features Detected

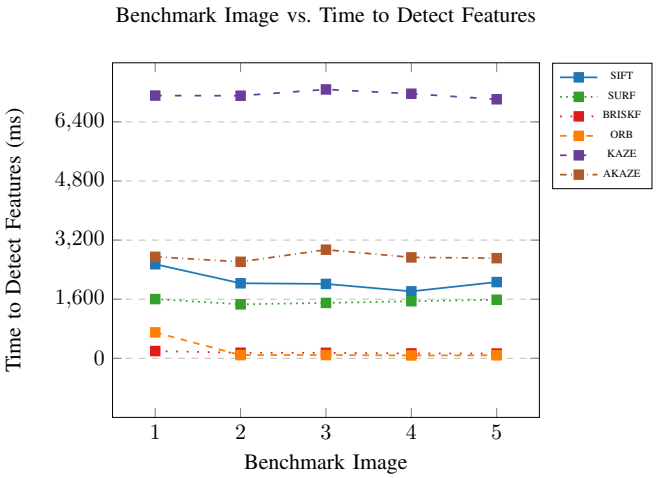

Fig. 3. Benchmark Image vs. Time to Detect Features.

TABLE II
AVERAGE FEATURES MATCHED AND RUN TIME.

Feature Detector/Matcher	Features Matched	Run Time (ms)
SIFT/BF	169.00	331.26
SIFT/FLANN	**172.75**	**2186.69**
SURF/BF	168.75	352.11
SURF/FLANN	178.00	2759.64
BRISK/BF	80.25	114.73
BRISK/FLANN	38.25	1573.29
ORB/BF	23.50	**15.33**
ORB/FLANN	**9.00**	620.41
KAZE/BF	108.75	92.89
KAZE/FLANN	115.00	1306.96
AKAZE/BF	94.25	55.13
AKAZE/FLANN	42.00	1020.02

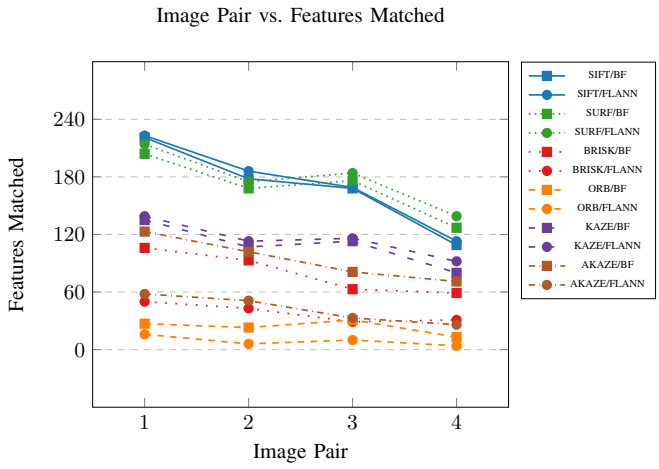

Fig. 4. Image Pair vs. Features Matched

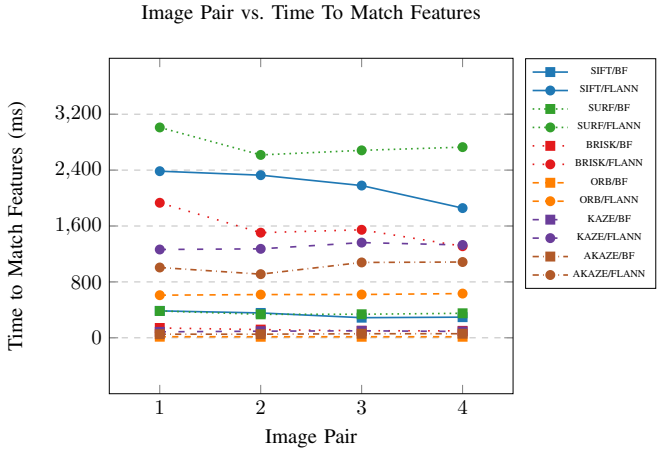

Fig. 5. Image Pair vs. Time to Match Features

C. Comparison Tool's Computed Values

The comparison tool's computed values are presented in this section. All results are computed on the previous sections' results.

Table III tabulates the comparison tool's computed values for each combination of feature detector and feature matcher.

IV. DISCUSSION

In this section, the results presented in Section III are discussed. First, the number of features detected, and the time to detect these features, is discussed; then, the number of features matched, and the time to match these features, is discussed; lastly, the comparison tool's computed values for combinations of OpenCV's detectors and matchers are discussed.

A. Feature Detection

From Figures 2 and 3, and Table I, it is evident that OpenCV's SURF feature detector detected the most features; detecting, on average, 1907.20 features in 1538.61 ms. In comparison, OpenCV's BRISK feature detector detected 1132.00

TABLE III
COMPUTED FEATURE DETECTOR AND FEATURE MATCHER COMPARISON TOOL VALUES.

Feature Detector/Matcher	Comparison Tool Value (N)
SIFT/BF	5.05
SIFT/FLANN	2.87
SURF/BF	8.75
SURF/FLANN	3.87
BRISK/BF	**36.38**
BRISK/FLANN	5.41
ORB/BF	18.78
ORB/FLANN	4.91
KAZE/BF	1.15
KAZE/FLANN	**0.99**
AKAZE/BF	2.46
AKAZE/FLANN	1.72

features in 150.94 ms: 59.35% of the number of SURF's detected features, but in 9.81% of the time. Considering Table I, if the greatest number of features detected was desired, SURF would be an ideal detector to use. Otherwise, if the time to detect features needed to be short, BRISK would be an ideal detector.

B. Feature Matching

From Figures 4 and 5, and Table II, it is evident that the combination of OpenCV's SIFT feature detector and FLANN feature matcher matched the most features; matching, on average, 172.75 features in 2186.69 ms. In comparison, the combination of OpenCV's ORB feature detector and BF feature matcher matched 23.50 features in 15.34 ms: 13.60% of the number of SIFT and FLANN's matched features, but in 0.70% of the time. Considering Table II, if only the combination of SIFT, SURF, and FLANN were evaluated, it would seem that FLANN matched the most features. However, this is only true for two of 12 combinations; of the remaining ten combinations, BF matched a greater number of features nine times. Therefore, if the greatest numbers of features matched was desired, BF would be an ideal matcher to use.

C. Comparison Tool's Computed Values

From Table III, it is evident that the combination of BRISK and BF has the greatest N value: 36.38. Considering the number of features detected, SURF and SIFT tend to have a greater number of features detected; however, they take a fairly long time to detect these features. BRISK, in contrast, detects a comparable number of features, but in nearly a tenth of the time. Considering the number of features matched, the combination of SIFT and FLANN matched the most features, but, again, taking a considerable time to do so. The combination of BRISK and BF matched a comparable number of features, but, again, in nearly the tenth of the time. With respect to the BF and FLANN feature matchers, BF typically matched features in the order of 100's of milliseconds; whereas, FLANN's run time was in the order of 1000's of milliseconds, and, with the exception of SIFT and SURF, typically matched less features than BF. Overall, the combination of BRISK and BF provided a reasonable balance between the number of features detected and features matched, and the time to do so.

V. CONCLUSION

In this paper, a range of feature detectors and feature matchers have been identified. Given these tools, this paper set out to answer the question: "which one should be used?". Subsequently, OpenCV's SIFT, SURF, BRISK, ORB, KAZE, and AKAZE feature detectors, and BF and FLANN feature matchers have been implemented and compared.

This paper's main program's algorithm's structure, and operation, has been presented and described. Results illustrating the performance of OpenCV's feature detectors, i.e. the number of features detected and time to detect these features; and OpenCV's feature matchers, i.e. the number of features matched and the time to match these features, have been presented. In addition, a comparison tool has been described, and the comparison tool's computed values, based on the previous results, presented.

Based on the results, it was evident that the SURF detected the most features and that the BF matched the most features. Based on the comparison tool's computed values, the combination of OpenCV's BRISK feature detector and BF feature matcher was the optimum combination.

ACKNOWLEDGEMENT

The author would like to acknowledge the support of Massey University's School of Engineering and Advanced Technology's Centre of Additive Manufacturing in carrying out the work described in this paper.

REFERENCES

[1] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.
[2] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.
[3] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417.
[4] S. Leutenegger, M. Chli, and R. Y. Siegwart, "Brisk: Binary robust invariant scalable keypoints," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2548–2555.
[5] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: an efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2564–2571.
[6] P. F. Alcantarilla, A. Bartoli, and A. J. Davison, "Kaze features," in *Computer Vision–ECCV 2012*. Springer, 2012, pp. 214–227.
[7] P. F. Alcantarilla and T. Solutions, "Fast explicit diffusion for accelerated features in nonlinear scale spaces," *IEEE Trans. Patt. Anal. Mach. Intell*, vol. 34, no. 7, pp. 1281–1298, 2011.
[8] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, pp. 331–340, 2009.
[9] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.