Keypoint Detectors, Feature Descriptors and Applications

I. Objective
The objective of this project is to introduce students to keypoint detectors and feature descriptors and their computer vision applications. Keypoint detectors are also referred to as interest point detectors or salient point detectors. After completing this assignment the student should become familiar with:

- Well-known feature detectors and descriptors
- How to apply feature detectors and descriptors in various computer vision applications

Furthermore, students should be able to investigate more state-of-the-art feature detectors and descriptors by themselves.

II. Introduction to keypoint detectors and feature descriptors in OpenCV
Features can refer to specific locations in images, such as mountain peaks, building corners, or other object characteristics depending on the considered application. These kinds of localized features are often called key-point features or interest points and are often described by the local appearance of patches of pixels surrounding the point location. So finding these image features is called keypoint detection. Once a keypoint is detected, the region surrounding the detected keypoint is used to describe this keypoint’s local characteristics by forming a keypoint descriptor, also called feature descriptor. This descriptor is typically formed by extracting features describing the local characteristics of the neighborhood of the detected keypoint. Descriptors that are invariant to illumination, rotation, translation and/or scaling are typically desirable. Once you have the features (keypoints) and their corresponding descriptors, you can find the same features in other images and match them, stitch them, track them, to name a few applications. In this project, OpenCV will be used to implement feature detectors and descriptors and applications. Some popular feature detectors and descriptors are described briefly below.

Please read Chapter 16 “Keypoints and Descriptors” of the Learning OpenCV 3 book.

II.1 Keypoint detector and descriptor

- SIFT (Scale-Invariant Feature Transform)
So now let us examine the SIFT implementation in OpenCV. Let’s start by detecting keypoints and drawing them. First we have to construct a SIFT object. We can set different parameters corresponding to this object. Consider the C++ code below.

```cpp
#include <stdio.h>
#include <iostream>
#include "opencv2/core.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "opencv2/highgui.hpp"

using namespace cv;
using namespace cv::xfeatures2d;

/** @function main */
int main(int argc, char** argv)
{

Mat img_1 = imread("lena.jpg");
Mat img_2 = imread("lena.jpg");

//-- Step 1: Detect the keypoints using SIFT Detector
Ptr<SIFT> detector = SIFT::create();
std::vector<KeyPoint> keypoints_1, keypoints_2;
detector->detect(img_1, keypoints_1);
detector->detect(img_2, keypoints_2);

//-- Draw keypoints
Mat img_keypoints_1; Mat img_keypoints_2;
drawKeypoints(img_1, keypoints_1, img_keypoints_1, Scalar::all(-1),
DrawMatchesFlags::DEFAULT);
drawKeypoints(img_2, keypoints_2, img_keypoints_2, Scalar::all(-1),
DrawMatchesFlags::DEFAULT);

//-- Show detected (drawn) keypoints
imshow("Keypoints 1", img_keypoints_1);
imshow("Keypoints 2", img_keypoints_2);

waitKey(0);
return 0;
}
```

In the code above, the `detector->detect()` function finds the keypoint in the considered image. You can pass a mask if you want to search only a part of an image. Each keypoint is a special structure which has many attributes like its (x,y) coordinates, size of the meaningful neighborhood, angle which specifies its orientation, response that specifies strength of keypoints etc.

Now to calculate the descriptor, OpenCV provides two methods. The first method is as follows:
1) Since you already found keypoints, you can call \texttt{compute()} which computes the descriptors from the keypoints. Example: \texttt{detector->compute(gray, keypoints, descriptors);} 

2) If you didn’t find keypoints, you can directly find keypoints and descriptors in a single step with the function, \texttt{detectAndCompute()} as shown in the sample code below.

\begin{verbatim}
Ptr<Feature2D> f2d = xfeatures2d::SIFT::create();
f2d->detectAndCompute(img, Mat(), keypoints, descriptors);
\end{verbatim}

Here \texttt{keypoints} will be a vector of keypoints and \texttt{descriptors} is a matrix of size Number_of_Keypoints \times 128.

After computing the keypoints and descriptors, students should be able to detect and characterize, by means of a descriptor, keypoints for images in OpenCV.

II.2 Keypoint (Interest point) detectors

- FAST (Features from Accelerated Segment Test)

Well known feature detectors like SIFT and SURF are not usually fast enough (unless accelerated using specific hardware) to be used in real-time applications such as SLAM or resource constrained embedded platforms like cellular devices. The FAST feature/ keypoint detector was introduced in the paper: Edward Rosten and Tom Drummond, “Machine learning for high speed corner detection” in 9th European Conference on Computer Vision, vol. 1, 2006, pp. 430–443, as a solution for high-speed keypoint detection in real-time computer-vision and robotics applications. This keypoint detector uses a high-speed test to exclude a large number of non-corners in an image through the use of a learnt decision tree. The FAST feature detector was later improved in the follow-up paper: Edward Rosten, Reid Porter, and Tom Drummond, “Faster and better: a machine learning approach to corner detection” in IEEE Trans. Pattern Analysis and Machine Intelligence, 2010, vol 32, pp. 105-119. \textbf{Note: Please download and read the papers.}

So now let us examine a sample code for computing image keypoints with the FAST feature detector. Let’s start by detecting keypoints and drawing them. First we have to instantiate an object for the FAST feature detector. We can set different parameters corresponding to this object. Consider the following C++ code:
```c++
#include <stdio.h>
#include <iostream>
#include "opencv2/core.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"

using namespace cv;
using namespace cv::xfeatures2d;

/** @function main */
int main(int argc, char** argv)
{
    Mat img_1 = imread("lena.jpg");
    Mat img_2 = imread("lena.jpg");

    //-- Step 1: Detect the keypoints using a FAST Detector
    Ptr<FastFeatureDetector> detector = FastFeatureDetector::create();
    std::vector<KeyPoint> keypoints_1, keypoints_2;
    detector->detect(img_1, keypoints_1);
    detector->detect(img_2, keypoints_2);

    //-- Draw keypoints
    Mat img_keypoints_1; Mat img_keypoints_2;
    drawKeypoints(img_1, keypoints_1, img_keypoints_1, Scalar::all(-1),
                  DrawMatchesFlags::DEFAULT);
    drawKeypoints(img_2, keypoints_2, img_keypoints_2, Scalar::all(-1),
                  DrawMatchesFlags::DEFAULT);

    //-- Show detected (drawn) keypoints
    imshow("Keypoints 1", img_keypoints_1);
    imshow("Keypoints 2", img_keypoints_2);

    waitKey();
    return 0;
}
```

In the code above, the FAST feature detector object is first instantiatated and called `detector`. The `detector->detect()` function finds the keypoint in the considered image. You can pass a mask if you want to search only a part of an image. Each keypoint/interest point is stored in a special structure called `<KeyPoint>` which has many attributes like its (x,y) coordinates, size of the meaningful neighborhood, angle which specifies its orientation, response that specifies strength of keypoints etc. Please refer to the OpenCV documentation at [https://docs.opencv.org/3.3.0/](https://docs.opencv.org/3.3.0/) for
additional information. Note: The docs and the code discussed is for OpenCV 3.3.0. If using a different version, please change the version on the docs page.

OpenCV also provides a `drawKeyPoints()` function which draws the small circles on the keypoints. If you pass a flag, `cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS` to it, it will draw a circle with the size of the keypoint’s neighborhood and it will even show its orientation. Please see the function description below.

```
<table>
<thead>
<tr>
<th>Function: drawKeypoints()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Arguments</strong></td>
</tr>
<tr>
<td>Image</td>
</tr>
<tr>
<td>keypoints</td>
</tr>
<tr>
<td>outImage</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>flags = DrawMatches::DEFAULT</td>
</tr>
</tbody>
</table>
```

A detailed documentation for all methods and data structures supported by the FAST keypoint detector can be found here:

https://docs.opencv.org/3.3.0/d7/d19/classcv_1_1AgastFeatureDetector.html#aa467aa45e824c0d3c8a9a2033c47fe987

- **AGAST (Adaptive and Generic Accelerated Segment Test)**
  Although the FAST feature detector is able to detect keypoints much faster than well-known keypoint detectors like SIFT and SURF, through the use of the Accelerated Segment Test (AST), its corner configuration evaluation process slows down significantly when the camera rotates or even moves. The robustness of the FAST feature detector is largely dependent on the corner configurations used to train its AST decision tree. If certain configurations are missing from the training set, the false positive rate for detected keypoints increases. As a consequence, the decision tree has to be learnt for each new imaging environment. The AGAST keypoint detector was introduced to overcome the need to retrain the AST decision tree each time, in the paper: Elmar Mair, Gregory D. Hager, Darius Burschka, Michael Suppa, and Gerhard Hirzinger, “Adaptive and generic corner detection based on the accelerated segment test” In *Proceedings of the European Conference on Computer Vision (ECCV’10)*, September 2010. Note: Please download and read the paper. Additional information related to this work can be found here:
OpenCV provides an implementation of the AGAST keypoint detector. The functions provided are similar to the ones discussed in the FAST feature detector part and we use the `detect()` function for computing the keypoints. A detector object for the AGAST keypoint detector can be created using the following `create()` function.

```
Function: AgastFeatureDetector::create()
```

<table>
<thead>
<tr>
<th>Input Arguments</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold = 10</td>
<td>int</td>
</tr>
<tr>
<td>nonmaxSuppression = true</td>
<td>bool</td>
</tr>
<tr>
<td>type = OAST_9_16</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Arguments</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>detector_obj</td>
<td>Ptr&lt;AgastFeatureDetector&gt;</td>
</tr>
</tbody>
</table>

The default parameters work well for most images but you may need to adjust the threshold and AST properties for improved results. Please refer to the paper mentioned above for the effect of these parameters. Complete documentation for the AGAST feature detector class can be found here: https://docs.opencv.org/3.3.0/d7/d19/classcv_1_1AgastFeatureDetector.html

II.3 Feature Descriptors

- **FREAK (Fast Retina Keypoint)**

The Fast Retina Keypoint (FREAK) descriptor was introduced in the paper: Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst, “FREAK: Fast retina keypoint” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 510–517., 2012. **Note: Please download and read the paper.** FREAK proposes a novel fast binary keypoint descriptor inspired by the human visual system, which can be deployed on resource-constrained embedded vision platforms and is also robust to scale changes, rotation and image noise. The descriptor is comprised of a set of binary strings evaluated by comparing image intensities of points on a retinal sampling grid, in a way that best captures local intensity relationships. The FREAK descriptor has been shown to be faster than SIFT, SURF and BRISK features.

OpenCV provides an implementation for the **FREAK** feature descriptor. Since **FREAK** is only a feature descriptor, we need to first use a keypoint detector to identify keypoints in the image (e.g., FAST, AGAST, SIFT’s keypoint detector, etc.) and then compute feature descriptors for
these keypoints. Below is a code snippet that shows how to compute feature descriptors for keypoints that have been already detected.

```cpp
// Step-1: Create object for feature descriptor
Ptr<FREAK> desc_comp = FREAK::create();

// Step-2: Create destination array for descriptors
Mat desc1, desc2;

// Step-3: Compute feature descriptors for all points in keypoints_i
desc_comp->compute(img1, keypoints_1, desc1);
desc_comp->compute(img2, keypoints_2, desc2);
```

The `compute()` function is used to compute the actual feature descriptor and is stored in an array of size $N_i \times F_{dim}$, where $N_i$ is the number of keypoints detected and $F_{dim}$ is the actual size of the feature descriptor for each keypoint. For example, $F_{dim} = 512$.

Below is a description of all the input parameters used to instantiate the FREAK feature descriptor:

<table>
<thead>
<tr>
<th>Function: xfeatures2d::FREAK::create()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Arguments</strong></td>
</tr>
<tr>
<td>orientationNormalized = true</td>
</tr>
<tr>
<td>scaleNormalized = true</td>
</tr>
<tr>
<td>patternScale = 22.0</td>
</tr>
<tr>
<td>nOctaves = 4</td>
</tr>
<tr>
<td>selectedPairs</td>
</tr>
<tr>
<td><strong>Output Arguments</strong></td>
</tr>
<tr>
<td>descriptor_obj</td>
</tr>
</tbody>
</table>

Additional details about Parameters:
- orientationNormalized: Enable orientation normalization
- scaleNormalized: Enable scale normalization
- patternScale: Scaling of the descriptor pattern
- nOctaves: Number of octaves covered by the detected keypoints
- selectedPairs: (optional) user defined selected pairs indexes
The complete documentation for the **FREAK** feature descriptor can be found here: https://docs.opencv.org/3.3.0/df/db4/classcv_1_1xfeatures2d_1_1FREAK.html

- **LUCID (Locally Uniform Comparison Image Descriptor)**

LUCID is also a fast region descriptor computation alternative to the more time consuming SIFT and SURF features. Fast descriptor alternatives like BRIEF and BRISK, use a random sampling grid for computing pairwise pixel differences and are less accurate. LUCID was proposed in the paper: Andrew Ziegler and Eric Christiansen and David Kriegman and Serge J. Belongie, “Locally Uniform Comparison Image Descriptor” in Advances in Neural Information Processing Systems, pp 1--9, 2012. *Note: Please download and read the paper.* LUCID uses a simple description method based on permutation distances between the ordering of intensities of RGB values between two patches and is computable in linear time with respect to patch size and does not require floating point computation. LUCID is faster than BRIEF, and its accuracy is directly comparable to SURF while being more than an order of magnitude faster.

OpenCV provides an implementation for the **LUCID** keypoint descriptor. The functions provided are similar to the ones discussed in the **FREAK** descriptor part and we use the **compute()** function for computing the keypoints. A descriptor object for the LUCID keypoint descriptor can be created using the following **create()** function.

<table>
<thead>
<tr>
<th>Function: xfeatures2d::LUCID::create()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Arguments</strong></td>
</tr>
<tr>
<td>lucid_kernel = 1</td>
</tr>
<tr>
<td>blur_kernel = 2</td>
</tr>
<tr>
<td><strong>Output Arguments</strong></td>
</tr>
<tr>
<td>descriptor_obj</td>
</tr>
</tbody>
</table>

Additional parameter details:

lucid_kernel: kernel for descriptor construction, where 1 = 3x3, 2 = 5x5, 3 = 7x7 and so on.
blur_kernel: kernel for blurring image prior to descriptor construction, where 1 = 3x3, 2 = 5x5, 3 = 7x7 and so on.
Complete documentation for LUCID can be found here:
https://docs.opencv.org/3.3.0/d4/d86/classcv_1_1xfeatures2d_1_1LUCID.html

Please refer to the paper mentioned above for additional details regarding descriptor parameters.

Note 1: Instructions to add the “xfeatures2d” OpenCV library for FERAK and LUCID

Please refer to Appendix 1 for the details of re-compiling the OpenCV library.

Note 2: List of all feature detectors and descriptors supported by OpenCV can be found here:
https://docs.opencv.org/3.3.0/d0/d13/classcv_1_1Feature2D.html

Note 3: Additional tutorials for feature detection and matching can be found here:
https://docs.opencv.org/3.3.0/d9/d97/tutorial_table_of_content_features2d.html

Note 4: Link to Drawing functions for keypoints and matches:
https://docs.opencv.org/3.3.0/d4/d5d/group__features2d__draw.html

III. Computer Vision Applications

- **Stereo correspondence algorithms**

Stereo matching is one of the most active research areas in computer vision. Over the years, a large number of algorithms for stereo correspondence have been developed. The Middlebury Stereo Vision Dataset, which is available at http://vision.middlebury.edu/stereo/, provides a taxonomy of existing stereo algorithms that allows the dissection and comparison of individual algorithm components design decision, and also a testbed for the quantitative evaluation of stereo algorithms. Please review and read the papers related to the top 5 algorithms in the Middlebury Stereo Vision Dataset. Then students should have a basic understanding about the two-frame stereo correspondence algorithms and also the algorithm evaluation method.

The task for this course project is to find the disparity of keypoints between two-frames. The specific requirements are as follows:

a) The keypoints should be generated by using the aforementioned feature detectors and descriptors in OpenCV.

b) In addition to the keypoint detectors and descriptors assigned, each group should also compute perform these steps for the SIFT keypoint detector and descriptor, which is to be used as a baseline.
c) Select a good matching algorithm to match the keypoints of two-frames.
d) Calculate the disparity between the obtained matched keypoints of two-frames.
e) Use the Middlebury Stereo Dataset for evaluation since it provides the ground-truth disparity map.
f) Evaluate the errors in terms of root mean square error (RMSE) and the percentage of incorrectly matched pixels. For more details about the error and accuracy, please read the paper: D. Scharstein and R. Szeliski, “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms,” *International Journal of Computer Vision*, 2002.
g) Compare results obtained by using different feature descriptors.

**Motion Estimation**

Motion estimation is the process of determining motion vectors that describe the transformation from one 2D image to another; usually from adjacent frames in a video sequence. Usually, optical flow is used for this application. Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. It is a 2D vector field where each vector is a displacement vector showing the movement of points from the first reference frame to a second frame. The Lucas-Kanade (LK) and the Kanade-Lucas-Tomasi (KLT) algorithms are popular optical flow computation methods.

The task for this course project is to track objects in a video by using the Lucas-Kanade or KLT optical flow method. Please read chapter 17 of the “Learning OpenCV3” book ([Link](#)) for tracking.

The specific requirements are as follows:

a) Read papers related to the Lucas-Kanade and Kanade-Lucas-Tomasi optical flow methods:


b) Use the optical flow method to track keypoints in a video. First, use the aforementioned feature detectors to detect keypoints in the first frame. Then iteratively track those points in subsequent frames using the Lucas-Kanade or KLT optical flow method. A pyramid implementation of the KLT algorithm is available in OpenCV (The function is `calcOpticalFlowPyrLK`). Plot the resulting optical flow as shown in the image below:

![Optical Flow Image](image.png)


c) Use the feature descriptors to track keypoints in a video. Use the aforementioned feature detectors to detect keypoints in the video frames and use their descriptors to match these and to generate the optical flow. Highlight matched keypoints for each frame and use different color for each set of matched keypoints. Plot the resulting optical flow as shown in the image above and compare with the optical flow method in (b) in terms of tracking accuracy and timing.

Note 1: Useful links for this application

- [https://docs.opencv.org/3.3.0/dc/d6b/group__video__track.html](https://docs.opencv.org/3.3.0/dc/d6b/group__video__track.html)
- [http://www.ces.clemson.edu/~stb/klt/](http://www.ces.clemson.edu/~stb/klt/)

Note 2: Tutorial for using optical flow in OpenCV:

- [https://docs.opencv.org/3.3.0/d2/d0a/lkdemo_8cpp-example.html](https://docs.opencv.org/3.3.0/d2/d0a/lkdemo_8cpp-example.html)

- **Image Stitching**

In this project, you will implement a system to combine a series of photographs into a panorama (see panorama below). Your software will detect discriminating features in the images, find the best matching features in the other images, automatically align the photographs (determine their overlap
and relative positions) and then blend the resulting photos into a single seamless panorama. You will use the aforementioned feature detectors to detect discriminating features in the images, and associated feature descriptors to find the best matching features in the other images. The main focus here is to automatically align the photographs (determine their overlap and relative positions) and then blend the resulting photos into a single seamless panorama as shown in the figure below.

**Input images:**

![Input images](image1.jpg)

**Resulting stitched panorama:**

![Resulting stitched panorama](image2.jpg)

The task for this course project is to combine a series of photographs into a 360º panorama. The specific requirements are as follows:

a) Compute the alignment of the images in pairs.
b) Select a good stitching algorithm to stitch and crop the resulting aligned images – OpenCV has built-in tools for stitching – refer to online OpenCV sample tutorial on stitching and Stitcher class.

c) Save the obtained panorama in JPEG format.

Some groups will be also tasked to compare the results of using feature detectors with those obtained using template matching. A useful reference for stitching using template matching and image blending is the paper by Burt & Adelson, “A Multiresolution Spline With Application to Image Mosaics,” ACM Transactions on Graphics, Vol. 2. No. 4, October 1983, Pages 217-236.

Note 1: Useful link for this application:
http://docs.opencv.org/modules/stitching/doc/stitching.html
https://docs.opencv.org/3.3.0/d2/d8d/classcv_1_1Stitcher.html#details

Note 2: Tutorial on image stitching using OpenCV:
https://docs.opencv.org/3.4.2/d8/d19/tutorial_stitcher.html

IV. Requirements

For this project, students in the class are divided into groups. The tasks for each group are different from other groups in terms of the feature detectors/descriptors and computer vision application. The Table below shows the specific tasks for each group.

V. Submission Instructions

Submit by the due date via Blackboard under Projects a zipped folder named LastNames_Project1.zip, where LastNames are the last names of the members in your group. The submitted zipped folder should contain: 1) Code folder containing Readme file on how to compile and run the code and citing any existing code or part of code used, source code with comments, executable code, input data on which to run code, and corresponding output data that is generated by the code; 2) Report: Report should include title, authors’ names, introduction, description of adopted approaches, results and discussion, conclusion, and references; report should be formatted as single-column, double-spaced, 12-point Times New Roman font.

The following guidelines are suggested in preparing the final report:
(a) Introduction: clearly identify the problem of keypoint detection and keypoint descriptors, the importance of it, and the areas of application, and present the report organization. (0.75 to 1 page).

(b) Description of Adopted Feature Detectors and Descriptors: describe the feature detectors/descriptors assigned to your group and compare these. Each of the chosen feature detectors/descriptors should be described briefly and compared by stating what are the main contributions of this detector/descriptor as compared to other ones, what are the main ideas proposed and/or tools used (without details since the reader can refer to the referenced original paper for details), its advantages and disadvantages as compared to other detector/descriptor. The description should be concise (3 to 5 pages).

(c) Results: Show and discuss the results. The requirements are specified in each computer vision application as described previously in this document (2 to 5 pages).

(d) Conclusion: discuss unsolved problems (things that still need to be solved/improved), impediments to further progress, and future directions for possible improvements (0.75 to 1 page).
Please refer to the provided Project 1 Grading Sheet as a guide to satisfy the requirements of this project.

<table>
<thead>
<tr>
<th>Group</th>
<th>Feature Detector/Descriptors</th>
<th>Computer Vision Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>FAST+FREAK &amp; AGAST+LUCID</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 2</td>
<td>AGAST+LUCID &amp; FAST+LUCID</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 3</td>
<td>FAST+FREAK &amp; FAST+LUCID</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 4</td>
<td>FAST+FREAK &amp; AGAST+FREAK</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 5</td>
<td>AGAST+LUCID &amp; AGAST+FREAK</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 6</td>
<td>AGAST+FREAK &amp; FAST+LUCID</td>
<td>Stereo Correspondence</td>
</tr>
<tr>
<td>Group 7</td>
<td>FAST+FREAK &amp; AGAST+LUCID</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 8</td>
<td>AGAST+LUCID &amp; FAST+LUCID</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 9</td>
<td>FAST+FREAK &amp; FAST+LUCID</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 10</td>
<td>FAST+FREAK &amp; AGAST+FREAK</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 11</td>
<td>AGAST+LUCID &amp; AGAST+FREAK</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 12</td>
<td>AGAST+FREAK &amp; FAST+LUCID</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 13</td>
<td>FAST+FREAK &amp; AGAST+LUCID</td>
<td>Image Stitching</td>
</tr>
<tr>
<td>Group 14</td>
<td>AGAST+LUCID &amp; FAST+LUCID</td>
<td>Image Stitching</td>
</tr>
<tr>
<td>Group 15</td>
<td>FAST+FREAK &amp; FAST+LUCID</td>
<td>Image Stitching</td>
</tr>
<tr>
<td>Group 16</td>
<td>FAST+FREAK &amp; AGAST+FREAK</td>
<td>Image Stitching</td>
</tr>
<tr>
<td>Group 17</td>
<td>SIFT &amp; AGAST+FREAK</td>
<td>Image Stitching</td>
</tr>
<tr>
<td>Group 18</td>
<td>SIFT &amp; FAST+LUCID</td>
<td>Motion Estimation</td>
</tr>
<tr>
<td>Group 19</td>
<td>SIFT &amp; FAST+FREAK</td>
<td>Stereo Correspondence</td>
</tr>
</tbody>
</table>
Appendix 1. Instructions for Re-compiling the OpenCV library with additional libraries (xfeatures2d)

For the purpose of this project, we need to add the xfeatures2d library module to the existing OpenCV library. There are two approaches as follows:

1. Re-compile the OpenCV library using CMake.
   If the first approach does not apply to your system or programming language, you need to re-compile the libraries and add the opencv-contrib-modules as follows. The following illustration uses the CMake GUI for Windows System.

   - Download the opencv & opencv_contrib repositories
     [https://github.com/Itseez/opencv_contrib](https://github.com/Itseez/opencv_contrib)
   - Unzip the file, go to “opencv_contrib-master\modules\”, copy the folder “contrib_world” and “xfeatures2d”, and paste the folders to “C:\opencv\sources\modules\”.
   - Download and Install CMake from [https://cmake.org/download/](https://cmake.org/download/)
   - Open Cmake-GUI, and choose source as “C:/OpenCV3.3/opencv/sources\”. Create a new folder named “new_build” in “C:\opencv\”. Choose the build path as “C:/OpenCV3.3/opencv/new_build\”. This will store the newly built OpenCV files.
   - In Cmake-GUI, press the button “Configure\”. A new window will pop up to select the compiler. Select the proper compiler, such as “Visual Studio 14 2015 Win64”.

   ![Configure CMake](image)

   - Click on “Finish”, and wait for the analysis to be done. A list of build options are shown as follows.
   - Check the options: “BUILD_opencv_contrib_world”, “BUILD_opencv_world” and “BUILD_opencv_xfeatures2d\”.
   - If you don’t have python in your system uncheck the BUILD_PYHTON_EXAMPLES
Press the button “Generate”, and wait for the process to finish as shown in the message window below:

- In Visual Studio, open the solution “OpenCV.sln” at “C:\OpenCV3.3\opencv\new_build\OpenCV.sln”. Wait for the project to load.
- In “Debug” configuration and “x64” platform, build the project “ALL_BUILD” and then the project build “INSTALL”. Both build processes should finish with no errors.
- In “Release” configuration and “x64” platform, build the project “ALL_BUILD” and then the project “INSTALL”. Both build processes should finish with no errors. This will take a while.
Now we have re-compiled the OpenCV library by adding the “xfeatures2d” library. Next we will copy the updated files to the OpenCV folder:

1. In “C:\opencv\new_build\lib\Debug”, copy the files “opencv_ts330d.lib” and “opencv_world330d.lib”, and paste the files to “C:\opencv\build\x64\vc14\lib”.

2. In “C:\opencv\new_build\bin\Debug”, copy the files “opencv_contrib_world330d.dll” and “opencv_world330d.dll”, and paste the files to “C:\opencv\build\x64\vc14\bin”.

3. In “C:\opencv\new_build\lib\Release”, copy the files “opencv_ts330.lib” and “opencv_world330.lib”, and paste the files to “C:\opencv\build\x64\vc14\lib”.

4. In “C:\opencv\new_build\bin\Release”, copy the files “opencv_contrib_world330.dll” and “opencv_world330.dll”, and paste the files to “C:\opencv\build\x64\vc14\bin”.

5. Copy the files in “opencv\new_build\install\include\opencv2”, and paste these files to “C:\opencv\build\include\opencv2”.

Now the xfeature2d library and re-compiled OpenCV library have been updated in the OpenCV installation path. You can create a new C++ solution in Visual Studio to work on the project. Set the property settings to link the OpenCV library paths (same procedure as in the OpenCV installation guide).

In your project solution property setting, under Linker -> Input -> Additional Dependencies, add the “opencv_world330d.lib” and “opencv_ts330d.lib” for Debug mode. Add the “opencv_world330.lib” and “opencv_ts330.lib” for Release mode. As shown in the figure below:
Note 1: Useful Links for re-compiling the OpenCV libraries in Windows

http://audhootchavancv.blogspot.in/2015/08/how-to-install-opencv-30-and.html

https://nishantnath.wordpress.com/2015/10/19/open-cv-3-x-adding-contrib-support-to-default-installation/

https://putuyuwono.wordpress.com/2015/04/23/building-and-installing-opencv-3-0-on-windows-7-64-bit/

Note 2: OpenCV install on Linux

https://www.pyimagesearch.com/2015/06/22/install-opencv-3-0-and-python-2-7-on-ubuntu/

https://www.learnopencv.com/install-opencv3-on-ubuntu/

Note 3: OpenCV install on Mac OS

https://www.pyimagesearch.com/2016/12/19/install-opencv-3-on-macos-with-homebrew-the-easy-way/

**Additional Reading Material**

The following link provides additional useful material about OpenCV for Computer Vision and Image Processing applications.

http://opencv.org/books.html