Faculty of Applied Sciences
University of Liege
September 9, 2019

Ben Mariem	Chiodio	Derroitte	Girineza	Testouri
Sami	Adrien	Natan	Guy	Mehdi

Computer Vision :

Task 1: Main modules development

1 Abstract

The overview goal of this application is to produce a panoramic view given a sequence of images. This first part, more precisely, aims to implement the basic module of the project.
Firstly, this paper will describe how the video is first acquired from the NVIDIA Jetson TX2 and how the images are modified in order to reduce the computation time of the algorithms.
Secondly, the camera calibration part will be developed. An *OpenCV* algorithm is used to recognize a checkered paper. This allows to detect the deformations of this checkered paper. Therefore, the camera can be calibrated.
From those results, detecting the camera motion can be done by computing the angle of rotation to which the TX2 is subjected. This is achieved by decomposing the homogrpahy matrix between each frame into a rotation matrix.
Finally, this report will contain explanations on how the panoramic images are created from the sequences captured. Indeed, the frames will be projected into the cylindrical plane and then will be stitched together to create a *"cylindrical panorama"*.

2 Module Explanation

2.1 Image/Video acquisition and calibration

2.1.1 Description of the algorithm

Image/Video acquisition Using *OpenCV* existing algorithms, camera reading and image/video writing is a relatively easy task. No major implementation choice have to be made other than setting the parameters accordingly with the result wanted as described in the next subsection.
It should be noted that while recording, the display becomes gray and the time of recording appears on the screen. It simply helps us to have a more practical tool to record sequences and does not modify the images taken.

Camera calibration : Current practice is to model the camera thanks to the pinhole camera model. An advantage, explained in [1], of this model is that it will consider only one equivalent lens whereas there could be several. Thus, this model will be very convenient to use. Unfortunately, it is very rare that a camera perfectly fits the characteristics of this ideal model which causes accuracy problems that have to be taken into account while modeling.
Camera calibration will be used to correct the model of the camera used for an application. It will permit an estimation of the intrinsic parameters of the camera (e.g: the relation between camera's natural units (pixels) and the real world units, the focal length, ...).
The internal geometric and optical camera characteristics (type of camera) are intrinsic parameters while the position and the orientation of the camera are extrinsic parameters (location of the camera ?).
Those intrinsic parameters can be represented by the camera matrix, a 3-by-3 matrix. This matrix can then be used to correct distortion, to compute camera's rotation angles, etc.

2.1.2 Implementation

Image/Video acquisition: *OpenCV* tutorial[1] on how to read the stream from the camera can be found pretty easily. Reading from the Jetson TX2 onboard camera requires precise arguments, such as the *width, height, frame rate*. Those arguments can be fixed with the one from the statement : 1280, 720 and 25 FPS. Once the window is opened, capturing sequences of 500 or 1500 frames, or capturing images can be simply done with an input listener which does the corresponding actions. The key to press can be found in the **README.md** attached with the code.
The rest of the code is mostly composed of windows handling, frame reading, etc.

Camera calibration: *OpenCV* is already composed of functions that allow one to perform calibration from a chessboard. The only part left is the configuration of the parameters according to the chessboard used.

The way the code is structured is in two main parts:
On one hand, all the images that will be used for calibration are captured. In order to achieve this, the algorithm tries to find the corners of the checkerboard (using `cv2.findChessboardCorners([...])`) in real-time. Once the corners are found on the live video stream, it informs the user that the image can be used for calibration and then saved. The calculation of corners in real-time is performed imprecisely in this first approach using the `cv2.CALIB_CB_FAST_CHECK flag`.
On the other hand, the algorithm takes all the pre-selected images, recalculates the corners accurately this time and applies the calibration from these points, with `cv2.calibrateCamera([...])`.

This approach has advantages and drawbacks. Firstly, the algorithm is slow because the computation of chessboard corners is performed twice: once, live and in an imprecise way to detect if the image can be used and then again on the chosen images but in a precise way this time.
However, this method allows not to work "blindly", and to be sure that the images used for calibration are relevant.

The end of the code also contains the formatting of the result in a *json* file that will be used by the latter algorithms.

2.1.3 Validation test for the implementation

Image/Video acquisition: From the opened windows resulting from the code, it can be directly seen whether or not the camera reading is correctly performed. The recorded images and videos can be displayed to ensure that the result files are correct.

[1] See [2] for instance.

Camera calibration: In [3], a condition for the camera matrix to be valid is that the focal length in the x-axis has to be the same, or nearly the same, as for the y-axis. The computation of the camera matrix shows that these two focal lengths are nearly the same.

2.2 Camera motion estimation

2.2.1 Description of the algorithm

In order to track the rotation angle of the camera, the rotation angle between each frame is computed and the sum of each angle is computed to get the total angle. The computation of the angle between two frames can be decomposed into four steps:

1. Features detection
2. Features matching
3. Computation of the homography matrix
4. Extraction of the rotation from the homography matrix

1. In order to track the differences between two frames, one needs to identify feature points in the frame and search for them in the second frame. The features are detected in the first and second frame using the ORB algorithm which is an efficient alternative to SIFT or SURF. Indeed, it produces a fair number of feature points and good matches as well as being the fastest of the three algorithms in terms of times per feature, as developed in [4]. In addition to that, SIFT and SURF are patented.

2. Once the features are extracted in both frames, one needs to search for features that are in both frames to establish a transformation relation between them. The matches are computed using the FLANN method because it seems that it is faster than the brute-force method (also implemented) while preserving the accuracy.
Indeed, as stand in [5], *"FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than Brute-Force Matcher for large datasets"*.

3. Given the camera matrix previously computed (see camera calibration) and the matches, the homography matrix can be computed between both frames.

4. The homography matrix can be decomposed into translation and rotation matrix. Finally the Euler angles are computed from the rotation matrix.

These steps are executed between each frame of the video. The total angle is given by the sum of all the angles between each frame.

2.2.2 Implementation

1. As said in the previous section, the features are detected using the ORB algorithm. Indeed, *OpenCV* provides a class obtained thanks to the function `cv2.ORB_create` that allows us to detect the feature in an image thanks to the function `cv2.ORB_create().detectAndCompute`.

2. As said in the previous section, the feature are matched using either the FLANN or the Brute-Force approach (both implemented). Indeed, *OpenCV* provides two classes obtained thanks to `cv2.FlannBasedMatcher` and `cv2.BFMatcher` respectively for having a FLANN or a Brute-Force matcher. Then, features matching can be done using the functions `cv2.FlannBasedMatcher.knnMatch` or `cv2.BFMatcher.match` respectively for both approaches.

3. As said in the previous section, the homography matrix can be computed given the camera matrix previously computed (see camera calibration) and the matches. Indeed, Opencv provides a function cv2.findHomography to compute the homography matrix using RANSAC algorithm[2].

4. The homography matrix can be decomposed into translation and rotation matrices using `cv2.decomposeHomographyMat` also provided by *OpenCV*

2.2.3 Validation test for the implementation

In order to test whether our angle calculation was correctly performed, we conducted several experiments: The first one, which would be qualified as more empirical, consisted of simply making an image sequence of a known angle and comparing it to the calculated angle. Using this method, an average error of 8° was found.

However, a variance due to experimental errors was present. More thoughtful tests were then considered: calculating the angle on a sequence of images making a "round trip". The final angle, provided that the camera returns to its original position, should be 0°. The resulting angles had an error, using this method, of a maximum of 5°, which was considered satisfactory.

2.3 Panoramic image construction

2.3.1 Description of the algorithm

Panoramic image construction can be done by applying a cylindrical warping on each frame and then stitching them together. Indeed as developed in [7], projecting the frame into the cylindrical plane allows the stitching only to consist of applying a pure translation to the frame one wants to add to the panorama in construction. Thus, panorama construction can be done by following those two steps for each frame :

1. Cylindrical Warping
2. Image Stitching

1. In this part, the frame will be projected into the cylindrical plane by applying the following formula:

$$x' = s * arctan(\frac{x - x_c}{f}) + x_c \tag{1}$$

$$y' = \frac{y - y_c}{\sqrt{(x - x_c)^2 + f^2}} + y_c \tag{2}$$

where f is the focal length (obtained thanks to Camera Calibration) and s is an arbitrary scaling factor (sometimes called the radius of the cylinder) that is set to $s = f$ to minimize the distortion.

2. In this part, it is needed to compute the translation that must be applied to one frame to stitch it to the panorama in construction given the feature matches between the panorama in construction and the current frame.
Then, one has tp apply this translation to the current frame and superpose the result of this translation to the panorama in construction.

2.3.2 Implementation

1. In this part, it is simply required to implement the previously mentioned formula.

2. In this part, the translation obtained thanks to the function `cv2.estimateRigidTransform` provided by *Opencv* given the feature matches between the panorama in construction and the current frame has to be computed. Then, one must apply this translation to the current frame thanks to the function `cv2.warpAffine` also provided by *OpenCV*.
Finally, one has to superpose the result of this translation to the panorama in construction.

[2] As explained in [6]

2.3.3 Validation test for the implementation

In order to test if the features matching works correctly during the algorithm, it was decided to add a *matching_ demo* mode to the code. As described in the **README.md** file, this allows, from a sequence of video, the choice of the features matched between two frames.
An example of this matching_demo option can be seen in the following image :

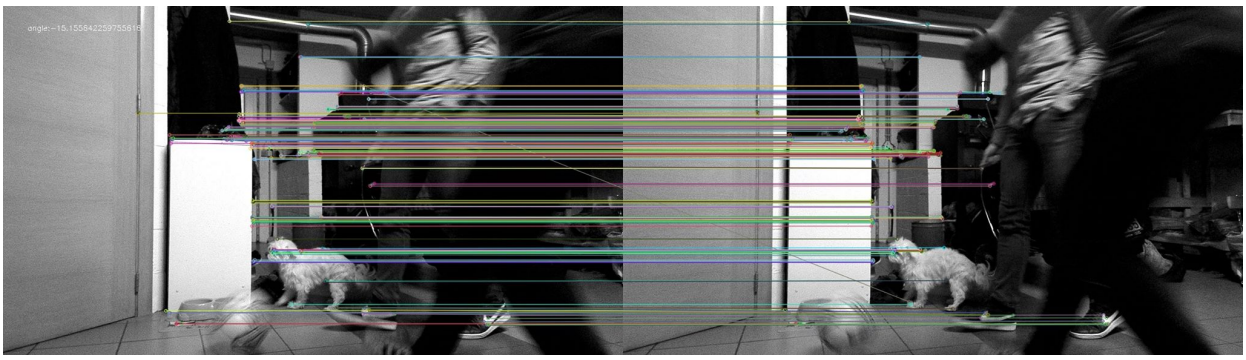

Figure 1: Feature matching between 2 consecutive frames

The final result of this first part can now be displayed : a panoramic image created from a sequence of image containing no moving object.

Figure 2: Panorama created

3 Image database

3.1 Description

It is required to film 2 mains dynamic sequences for this project containing at most 3 persons and 2 non-human moving objects, one indoor and one outdoor. Each of these needed to be accompanied with one companion sequence, which served as a reference to validate our algorithms.

Those dynamics sequences contain 1500 frames (25 FPS during 60 sec).
It was decided to have 2 people on each dynamic sequence. As non-human moving entity, a football ball and a white dog, *Grappa* were included.

3.2 Implementation

Implementing this last module is the easiest piece of code so far. In only a few modifications of our code in charge of image acquisition, satisfactory result can be obtained.
However, it should be noted that the images in the database are all loaded into a buffer and written only after the sequence capture was completed. Indeed, writing images being a time-consuming operation, doing it live is not a solution that allowed to keep the constraint of 25 FPS. Without this operation, some frames do not have time to be taken into account and this results in a jumpy sequence.

In the same optic, it is also decided to increase the performance of the Jetson TX2 by turning off power saving via clock throttling. This is further explain in the **README.md** attach to the code.

To capture the exact number of frames, a counter is also added to the code, allowing not to exceed this requested number, either for the reference sequences (500 frames) or the dynamic sequence (1500 frames).

4 Contribution

In this last section, it will be explained how the work was divided between the different team member.

The following description shows a strong delineation between the different parts of the project. This is a high vision of the work done that does not accurately reflect the used working method: although the main sub-tasks have been distributed between the members of the team, all team members have all been working on other parts of the project at one point or another. Thus, this delimitation should not be seen as a major decomposition of the work but rather on who were the main contributors of each part.

- **Image and video acquisition**: Chido Adrien and Derroitte Natan
- **Camera calibration**: Ben Mariem Sami and Chiodo Adrien
- **Camera motion detection**: Ben Mariem Sami and Testouri Mehdi
- **Panoramic image construction**: Ben Mariem Sami, Derroitte Natan, Girimeza Guy and Testouri Mehdi
- **Database generation**: Chido Adrien, Derroitte Natan and Girimeza Guy
- **Report writing**: Everyone

Let us end this report by saying that we have the advantage, for most of us, of being used to work together. The distribution of the workload was not a problem between us, on the contrary: everyone was able to contribute in a relevant way to this project.

References

[1] Comparing two new camera calibration methods with traditional pinhole calibrations - *OSA Publishing*

[2] Getting Started with Videos - *OpenCV 3.0.0-dev documentation*

[3] Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix - **Kyle Simek**

[4] Examensarbete utfört i Datorseende vid Tekniska högskolan i Linköping - *Niklas Rydholm*, page 23

[5] OpenCV-Python Tutorials - *Alexander Mordvintsev*

[6] `findHomography` documentation - *OpenCV*

[7] Image Alignment and Stitching - *Richard Szeliski1* - 2006