Feature Detection and Tracking in OpenCV

An implementation using Lucas-Kanade method complemented with different feature detectors

Luigi Riz
Master's Degree in Artificial Intelligence Systems
DISI - University of Trento
Trento, Italy
luigi.riz@studenti.unitn.it
Mat. 223823

***Abstract*—The following report describes the implementation details of a system that performs feature detection and tracking over the frames of a video. In particular the project consists in three main components, *i.e.* a wrapper file, used to parse command line arguments and run configuration operations, the `Detector` class, handling feature detection on single frames of the video source employing different methods, and the `Tracker` class, that is "detector-agnostic" and simply tracks the keypoints returned by the detector.**
Moreover, this report describes some qualitative and quantitative results, that underline the pros and the cons of each of the detectors used.

The whole project has been developed in Python 3.10, and it is inspired by the OpenCV tutorial about Optical Flow [1] and other tutorials provided by the same organization [2].

I. Implementation details

The `README.md` file provided in the repo explains in detail how to run the code and reports the different available flags to customize its behaviour. In particular, the project is structured in a modular way, so that the user can simply call the wrapper file, that instantiates a `Detector` object and a `Tracker` object, that interact to produce the required output, as reported in Figure 1.

A. Wrapper file

The wrapper file, namely `DetectAndTrack.py`, is a simple script that manages the parsing of command line arguments and instatiates the objects for subsequent processing.
In particular, the most important flag this piece of code manages is the `--detector` one, that allows the user to select a feature detector among the four available ones, namely `ORB`, `SIFT`, `STAR` and `FAST`. Another important flag, used in particular for validation, is the `--DEBUG` one, that displays additional information (*e.g.* the frames on which feature detection is performed, the number of spotted tracks, ...) and prints in the console some benchmark information.

Once the parsing is completed, the wrapper script instantiates a `Detector` object, that is passed to the constructor of the `Tracker` object, which than starts the processing of the video.

B. Detector class

The `Detector` class, declared in the homonymous file, is dedicated to the detection of keypoints inside the frames it is provided with. In particular, its constructor is able to instatiate the following types of detectors:

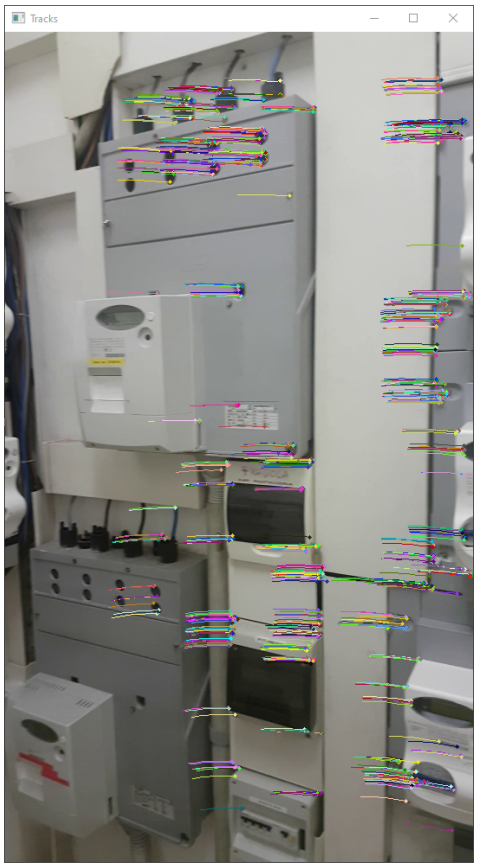

Fig. 1: Output produced by the code.

- SIFT: introduced by Lowe *et al.* [3], it tries to find features at different scales in a way to find *Scale-Invariant Keypoints*.
- FAST: proposed by Rosten *et al.* [4], it is mainly focused on the speed of detection.
- ORB: designed by Rublee and other members of the "OpenCV Labs" [5], it combines methods proposed previously to provide a patent-free alternative to SIFT and SURF.
- STAR: derived from CenSurE [6] and used to compute

BRIEF descriptors [7], it has been created to combine speed and accuracy of computation.

The `detect` method of this class takes in input an image (in greyscale) and eventually a binary mask, depicting the areas of the image in which to look for features. This idea has been borrowed from the tutorial by OpenCV [1], with the aim to mask out the areas of the image that already contain some tracks (sequence of coordinates of tracked keypoints) and concentrate the search of new interesting points in other locations. Figure 2 shows an example of masked image, that can be displayed in real-time when the whole script is run with the `--DEBUG` flag.

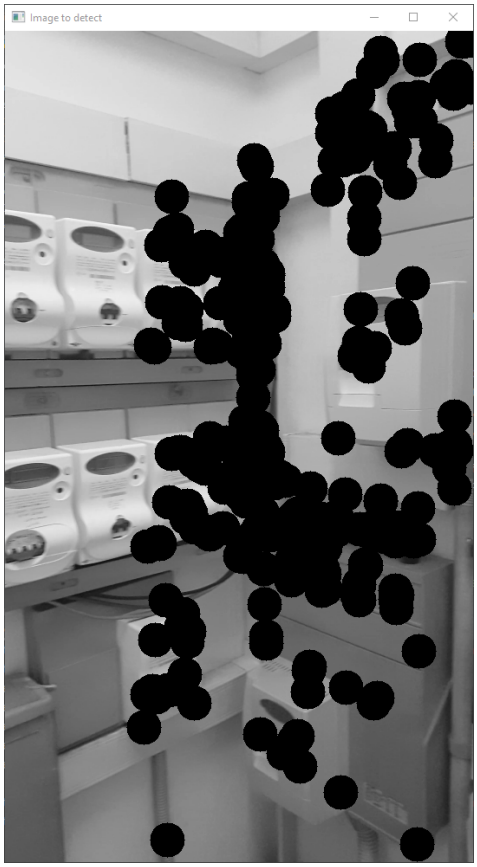

Fig. 2: Example of masked frame, on which feature detection is performed.

The `detect` function in the end returns a list of coordinates, representing the new keypoints to be tracked, that is passed to the `Tracker` for further processing.

C. Tracker class

The `Tracker` class is responsible for the tracking of the keypoints returned by the `Detector`, using Lucas-Kanade method. In particular, its `run()` method follows the pseudo-code in Algorithm 1.

As the pseudo-code highlights, the function is divided into two main parts: the first one is devoted to the application of the optical flow method, to the evaluation of "good" and "bad" tracks and to the drawing of tracks on the image to display; the second one is dedicated to the interaction with the detector passed at instantiating-time. It is worth noting that, as already said, the tracker is totally agnostic with respect to the type of detector involved in keypoints extraction.

Algorithm 1 Run

$tracks \leftarrow$ empty list
while $True$ **do**
  Read $frame$
  if $frame$ is too high **then**:
    Resize $frame$
  end if
  if len($tracks$) ≥ 0 **then**
    $pts \leftarrow$ last point from each $track$ in $tracks$
    $new\_pts \leftarrow$ optical flow from $prev\_frame$ to $frame$ considering pts
    $backwards\_pts \leftarrow$ optical flow from $frame$ to $prev\_frame$ considering new_pts
    $dist \leftarrow$ distance(pts, $backwards\_pts$)
    Remove $track \in tracks$ for which $dist \geq 1$
    for each $pt \in new\_pts$ **do**
      Append pt to relative $track$
      Append $track$ to $tracks$
      if $track$ too long **then**:
        Remove first pair of coordinates
      end if
      Draw updated track on the image to display
    end for
  end if
  if $frame\_idx$ mod $detect\_interval = 0$ **then**
    $mask \leftarrow$ as Figure 2
    $pts \leftarrow detector.detect(frame\_grey, mask)$
    Add pts to $tracks$ as single-element tracks
  end if
  $prev\_frame \leftarrow frame$
  Show image with tracks
end while

Other pieces of code included in the class file but not reported in Algorithm 1 are devoted to debugging operations, that keep track of different factors, such as the average time required for a detection step, the average number of tracks per frame and others. All these aspects will be discussed and formalized in Section II.

II. Results

All the experiments have been performed with the following set of parameters:

- `detect_interval` $= 30$, meaning that a detection step is performed once every 30 frames;
- `track_len` $= 10$, that is to say the max number of points inside a track is 10;
- `--DEBUG` flag, to get debugging information.

Moreover, all the videos inside the `material` folder have been used, since they differ in terms of motion (camera

motion in the case of *"Contesto_industriale1.mp4"*, object motion in *"Cars_On_Highway.mp4"* and a mixture of the two in *"Robotic_Arm.mov"*), in terms of location of the scene (indoor or outdoor), in terms of illumination and in terms of presence of occlusions. Furthermore, the parameters of the Lucas-Kanade tracker are maintained as in the code taken as reference and the configuration of the detectors is kept as in the OpenCV documentation default.

A. Qualitative considerations

Just by looking at the real-time output each different detector produces, some differences and considerations can be stated.

The SIFT detector seems to extract robust keypoints, since the produced tracks are rather stable. However, sometimes, some outliers are detected on flat surfaces, causing incoherent tracks. There is no difference of performance when dealing with apparent and real motion or when considering different kinds of illumination.

The FAST detector extracts a huge amount of keypoints in its default configuration, and the resulting tracks are extremely robust. However, this amount of data causes the slowing down of the script, that is probably caused by the tracking part of the script, since speed is one of the masterpieces of this detector. Also in this case, illumination and motion differences do not affect the performances of the detector.

The ORB detector sits in between SIFT and FAST: it extracts many points while remaining fast, flowing and extremely robust. In fact, no outliers are detected as keypoints and the number of tracks is consistent over all the frames of the video. Illumination changes and different motion types do not affect the performances of this detector.

The STAR detector is not so good in its default configuration: few keypoints are extracted at each detection step and the resulting tracks are also quite incoherent. However, its performances are stable when considering the different settings of the three proposed videos.

Other general qualitative considerations arise when looking at the produced outputs: occlusions cause some dragging around of keypoints that should remain static since they are part of the background and fast movement of objects cause the loss of tracking of points. However, these problems seem to be related to the tracking part of the code, since the detector, working on single frames, has no notion about object movement and possible occlusions.

B. Quantitative results

By using the `--DEBUG` flag, the code extracts also some quantitative measures during the execution. This data can then be used to compare the different types of detectors in an analytical way. In particular the aspects taken into consideration are:

- Average number of detected keypoints;
- Average number of tracks per frame;
- Average number of deleted tracks per frame;
- Average time for a detection step;
- Overall execution time.

Table I summarizes these information for all the detectors and all the videos considered also in previous Paragraph.

As Table I underlines, some of the intuitions reported in Section II-A are confirmed also by the collected data. For example, the detector with highest average number of detected points is FAST, followed by ORB and SIFT that show similar performances. As already said, STAR has limited detection capabilities in the tested configuration.

The FAST detector is also by far the method with lowest average time per detection step, being from 10 to 14 times faster than SIFT. The STAR detector is also rather fast, but this is probably due to the scarce amount of computations it carries out in its default configuration. The ORB method is slightly faster than SIFT, being, on average, 1.5 times faster than its direct competitor.

The evaluation on the average number of tracks per frame also highlights that FAST outputs rather good keypoints, that are tracked in a robust way. ORB and SIFT have also in this case comparable performances, whereas the STAR detector has limited efficiency also in this case, which is probably a consequence of the low amount of keypoints it is able to find.

The analysis on the average number of deleted tracks per frames is not so informative, since the ratio $\frac{deleted\_tracks}{tracks\_per\_frame}$ is nearly constant for the four detectors in the case of the first and the third video (1.3% and 0.8% respectively). The situation is instead different in the case of the second video: in this case SIFT and FAST seem to behave better than STAR and ORB (0.1% VS 0.4%), probably because the first two detectors are able to detect strong keypoints from the background, that remain stable throughout all the duration of the video.

As already said and confirmed also by the data reported above, the overall execution time is not descriptive for the behaviour of the detectors, but is more related to the tracking part of the code. In fact, FAST, as stated before, is the fastest detector, but the version of the script using it is the slowest among the four tested, probably due to the huge number of the keypoints to be tracked.

III. Conclusions

The code developed for this assignment is focused on the analysis of the behaviour of different feature detectors in their default configuration. The experiments show that FAST is an excellent extractor, since it is able to derive keypoints in a fast and robust way; SIFT and ORB are valid alternatives, whereas STAR has to be better fine-tuned to have decent performances.

A possible field of investigation could be the fine-tuning of the detectors so that they extract roughly the same amount of keypoints: doing so, it could be possible to even better evaluate the average time required for a detection step and assess the robustness of the extracted point by looking at the produced tracks.

Another possibility could include the use of different tracking methods, in a way to evaluate the goodness of the feature detectors also changing this parameter in the system.

	Avg. detected points	Avg. tracks per frame	Avg. deleted tracks per frame	Avg. time per detection	Overall execution Time
	Contesto_industriale1.mp4				
SIFT	227.6	527.3	7.0	0.1211 sec	62.9261 sec
FAST	533.5	1251.1	16.0	0.0094 sec	77.3131 sec
ORB	420.5	1016.7	12.6	0.0739 sec	73.7250 sec
STAR	88.6	175.6	2.7	0.0230 sec	55.1620 sec
	Cars_On_Highway.mp4				
SIFT	107.8	1896.4	2.4	0.1118 sec	96.8314 sec
FAST	250.0	5996.7	4.6	0.0078 sec	200.4557 sec
ORB	190.9	1553.0	5.3	0.0482 sec	83.6430 sec
STAR	43.0	268.6	1.2	0.0217 sec	39.7072 sec
	Robotic_Arm.mov				
SIFT	335.8	1439.1	11.3	0.1230 sec	23.9591 sec
FAST	705.6	2964.5	23.7	0.0118 sec	34.2138 sec
ORB	393.1	1487.7	13.2	0.1560 sec	24.9968 sec
STAR	111.5	467.4	3.7	0.0280 sec	14.4474 sec

TABLE I: Quantitative results produced by the different detectors.

REFERENCES

[1] OpenCV, "OpenCV: Optical Flow." Accessed Jun. 06, 2022 [Online].

[2] OpenCV, "OpenCV: Feature Detection and Description." Accessed Jun. 06, 2022 [Online].

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *European conference on computer vision*, pp. 430–443, Springer, 2006.

[5] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*, pp. 2564–2571, Ieee, 2011.

[6] M. Agrawal, K. Konolige, and M. R. Blas, "Censure: Center surround extremas for realtime feature detection and matching," in *European conference on computer vision*, pp. 102–115, Springer, 2008.

[7] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*, pp. 778–792, Springer, 2010.