Directed Project: Optimization of Tracking Framework

David Eduardo Moreno-Villamarín

***Abstract*—Extensive work has been conducted on designing object tracking frameworks that are robust enough to cope with video artifacts such as occlusion or varying background. In this work we analyze a video tracker implemented in Matlab that resolves this by using a support vector machine (SVM) and a mean shift tracker (MST). However, the robustness given by the SVM costs execution time. This is reflected in the number of frames per second (FPS) processed by the tracker, which ranges between 0.35 and 1. We attempt to speed up the processing time of the tracker by optimizing the algorithm that extracts the features fed to the SVM, obtaining a FPS value that ranges between 1.39 and 12.89. This was accomplished with the help of OpenCV's library and Matlab's mex interface. An implementation can be found at https://github.com/ujemd/HOG-Features-Tracking.**

I. Introduction

In this work, we attempt to optimize the object tracking framework proposed in [1] to obtain a faster processing time of a video. It consists of a Foreground-Background Tracker (FBT), used to detect and locate an object employing a support vector machine (SVM) [2]–[4], a Mean Shift Tracker (MST) that locates an object based on its RGB color histogram [5], [6], and updating modules for the SVM and the RGB histogram to adapt the background model.

The SVM of the FBT component is trained upon a Histogram of Oriented Gradients (HOG) [7] representation of object and background patches. It has been observed that most of the processing time is spent during the (HOG) feature computation. In particular, this routine takes between 60% and 80% of the total processing time, depending on the video. Since the algorithm is implemented in Matlab, it is possible to optimize the algorithm by reducing the computing time of the HOG features by using precompiled C++ code with help of Matlab's mex interface.

There has been extensive research done on developing features for object tracking [8]. One of the most used invariant features has been proposed by Dalal and Triggs [7]: the Histogram of Oriented Gradients. It was originally designed for a human detection algorithm, and had the same underlying idea of the Scale-Invariant feature transform (SIFT) of using oriented gradients [9]. The main difference is that HOG aims to recognize regions more than key points.

The first step of the process of HOG is to apply gamma correction or color normalization to the input image or ROI, this step is normally skipped, because of its modest effect on detection performance and its computational cost. The authors suggest using a detection window of size 64×128, which is later divided into overlapping regions called blocks, for the computation of gradients. The blocks consist of smaller regions called cells, which are preferred to have a size of 8×8 pixels. The gradients of cells are later binned into a n-bin histogram of edge orientations (typically 9 bins) using trilinear interpolation. Later, the histograms are normalized in overlapping blocks, where a size of 2×2 cells seems to give the best results. The normalization is usually done using a L2-norm, and provides illumination invariance and better detection. This process is illustrated in Figure 1.

Departmento de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana Seccional Cali, Cali, Colombia.
June 14, 2017

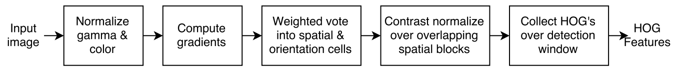

Input image
Normalize gamma & color
Compute gradients
Weighted vote into spatial & orientation cells
Contrast normalize over overlapping spatial blocks
Collect HOG's over detection window
HOG Features

Fig. 1. Overview of the feature extraction chain.

The length of a feature vector is given by the formula:

$$\text{length}(HOG) = (c \times c \times N_{bins}) \times (B_Y \times B_Y) \quad (1)$$

where $c \times c$ are the number of cells in a block, N_{bins} is the number of bins per cell, and $B_X \times B_Y$ are the number of blocks in a window:

$$B_X = (S_X - pc + L)/L \quad (2)$$
$$B_Y = (S_Y - pc + L)/L \quad (3)$$

where $S_X \times S_Y$ is the detection window (or ROI) size, $pc \times pc$ is the block size in pixels, and L is the block stride in pixels.

II. Methods

In this section, we explain the different approaches taken for implementing the HOG extraction procedure. We start describing the existing algorithms in C++ computer vision libraries OpenCV [10] and dlib [11], and how we also attempted to implement our own algorithm for the feature extraction with the help of OpenCV.

A. *OpenCV's HOG Algorithm*

As previously stated, Matlab's mex interface allows the joint use of Matlab functions and precompiled C/C++ code and libraries. Our first attempt at optimizing the tracking framework is using the OpenCV's own HOG implementation. This library includes the `cv::HOGDescriptor` class, which contains people detection methods based on HOG features, as well as the computation of HOG features.

To compute the HOG feature vector using OpenCV, it is first required to create a `cv::HOGDescriptor` object. The constructor, as in the Matlab function, requires parameters such as: cell size, block size, block overlap, number of bins (for the histogram), and whether it uses signed ($[0, 360]$) or unsigned ($[0, 180]$) orientation of the gradients. In addition, it also requests the size of the window detection size and window stride, which can be set as the input image or ROI size. There are other optional parameters for specifying the process of some of the steps of the HOG computation.

After creating the object, the extraction of the features is made by using the `cv::HOGDescriptor.compute` function to compute the feature vector of type `float`. The feature vector is later converted to a single precision `mexarray` type for Matlab compatibility.

The first attempt was coding a C++ version of Matlab's `extractHOGFeatures` routine. It was intended for this function to be completely compatible with the existing code in the tracking framework, without the need of changing the syntax of the input arguments. This new feature extraction function was pretty similar to the previous implementation, with the single change being the use of the OpenCV class.

We used the same input parsing used in the Matlab version, and would call a mex function, which created the HOG object with such inputs each time it was called. It also computed the features, that were mapped to the original output. The single difference in OpenCV is that it expects an image I of size $S_X \times S_Y$ given by:

$$S_X = \text{floor}(\text{width}(I)/p) \times p \tag{4}$$
$$S_Y = \text{floor}(\text{height}(I)/p) \times p \tag{5}$$

where $p \times p$ is the cell size.

It was later observed that the input parsing using Matlab took a significant mount of processing time, and later removed. The next step was to create the HOG object once per frame, which showed a much better reduction in processing time. This was accomplished by creating a HOGDescriptor class in Matlab that models the C++ version. This class contains as properties some of the input parameters, and its methods are: the class constructor, the `compute` and function, and the class destructor. The class constructor would command the c++ HOGdescriptor class to construct the object using persistent memory with the input parameters. The `compute` method would use the object to extract the features in a single precision vector. And the destructor would free the memory used by the object, while releasing the Matlab object as well.

B. *dlib's FHOG Algorithm*

Another explored path was to use the dlib library. It contains the HOG implementation described in the work of Felzenszwalb et al [4]. The function `extract_fhog_features` takes as input an image or ROI and the cell size parameter p. The image is broken into cells, and within each cell a 31 dimensional feature vector ($FHOG$) is computed. The length of the vector can be calculated as:

$$\text{length}(FHOG) = 31 \times C_X \times C_Y \tag{6}$$
$$C_X = \max(\text{round}(S_X/p) - 2, 0) \tag{7}$$
$$C_Y = \max(\text{round}(S_Y/p) - 2, 0) \tag{8}$$

where $S_X \times S_Y$ is the image size, $p \times p$ is the cell size and $C_X \times C_Y$ are the number of cells in the image.

C. *C++ Algorithm*

An implementation from scratch of the feature extraction in C++ with the help of OpenCV's library was also considered. The reason, as further explained in section III, was that the success plots of the trackers implemented with OpenCV and dlib presented a smaller area under the curve. With our own implementation, it was possible to have a better control of some of the other HOG parameters such as the kernels used for the gradients computation, and how the histogram binning procedure was done.

III. Results

The performance of the different algorithms was tested on seven sequences: Car, Jumper, Pedestrian 2, Pedestrian 3, Charger, Camera, and Gurgaon; which consist of 945, 313, 338, 184, 460, 390, and 160 frames. Figure 2 display the first frame of the video sequences and the target object with its bounding box.

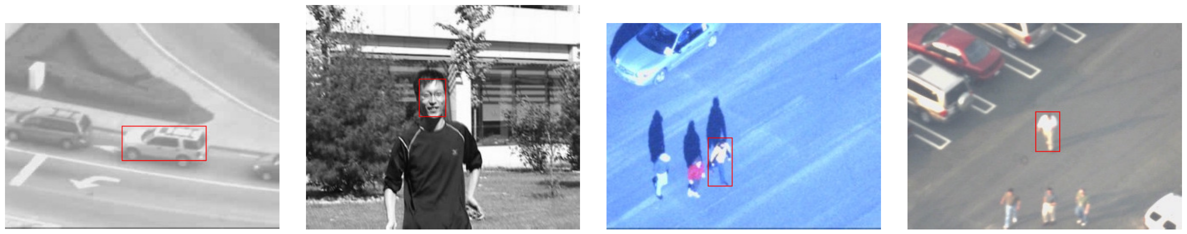
A. *Success Plots of Pristine Videos*

We started by testing how the different HOG algorithms affected the tracker performance. Figure 3 shows the success plots of the video sequences and each HOG implementation. This method of evaluation was selected as it is also used for assessing the effect of distortions on the tracker in [1]. The area under the curve of the success plot indicates how well the tracker performs in a video sequence.

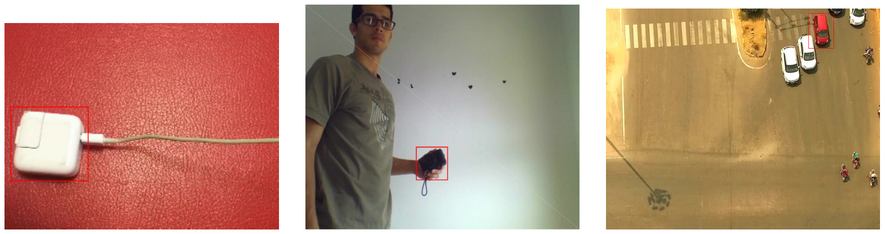
As mentioned in the previous section, OpenCV's and dlib's versions of HOG features diminish the tracker's performance when compared to the results given by Matlab's HOG features. Specifically, this effect is most noticeable when analyzing the Jumper and Camera sequences. However, our HOG implementation, does not perform better than the other algorithms, as shown in the fourth column of Figure 3. Therefore, we did some exploration of the OpenCV function of HOG features.

As we observed that OpenCV's function strictly required that the sizes of an input image to be a multiple of the cell size, we experimented by modifying how the tracking algorithm generated the background and object patches. This resulted in some improvements of the performance on the Jumper and Pedestrian 2 videos, but also negatively affected the other videos.

We also identified that the values in OpenCV's HOG feature vector were lower than those of Matlab's HOG feature vector. We proceeded to calculate the RMSE of both vectors obtained from an image of 256×256 pixels, using a cell size of 8×8 pixels, a block size of 16×16, and block overlap of 8×8. This resulted in a vector with a length of 34596. The RMSE



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



(e) (f) (g)

Fig. 2. First frame of the video sequences: (a) Car, (b) Jumper, (c) Pedestrian 2, (d) Pedestrian 3, (e) Charger, (f) Camera, (g) Gurgaon.

between Matlab's vector and OpenCV's vector was 0.910. Due to the lack of information about the other input parameters of OpenCV's function, we experimented by varying some of them and observed that higher values of `winSigma` yielded higher values of the feature vector. With `winSigma`=8 we obtained a RMSE of 2.89×10^{-5} between both vectors. This modification produced better results, as illustrated in the last column of Figure 3.

B. *Execution Times of Pristine Videos*

Later, we tested the execution time of the tracking framework for each video. The tests were carried out using a computer with a 64-bit operating system, a RAM of 4GB, and a processor Intel Core i5-4210M with 2.60GHz. As Table I shows, we estimated the time required for the tracker when using the different feature extraction procedures, the percentage of time required by these procedures, and the number of frames per second. It is possible to observe that the fastest execution times were given by dlib's implementation, being in average 13 times faster than the original algorithm. This result was followed by OpenCv's function, which executed in average 8 times faster than Matlab's implementation. Nonetheless, the previous success plots demonstrated that the tracker performs better when using the HOG algorithm from OpenCV.

It is also important to mention that the execution time of the tracker for each video depends on the video's resolution. In particular, the tracker works by extracting HOG features from a predefined number of patches of each frame in the video. Hence, a high-resolution video means a high resolution object, which translates into bigger patches that require more time for the feature extraction. This also affects the length the feature vectors, increasing the time required by the training and prediction algorithms of the support vector machine.

On the other hand, the content of each video will also define whether the tracker works in SVM mode or Mean Shift mode at determined moments. In the case of Mean Shift mode, the tracking algorithm will not require HOG feature computation. Therefore, it also influences the total execution time required for an individual video sequence.

TABLE I
TRACKING EXECUTION TIMES FOR EACH VIDEO SEQUENCE, PERCENTAGE OF TIME REQUIRED TO CALCULATE HOG FEATURES, AND ESTIMATED FRAMES PER SECOND. THE COLUMNS REPRESENT EACH OF THE HOG IMPLEMENTATIONS: MATLAB, OPENCV, DLIB, AND OUR C++ ALGORITHM.

		Matlab	OpenCV	dlib	C++
Car	Time (s)	2246.96	421.61	220.04	2822.39
	HOG %	0.92	0.67	0.50	0.94
	FPS	0.42	2.24	4.29	0.33
Jumper	Time (s)	324.72	47.71	24.32	212.77
	HOG %	0.90	0.44	0.49	0.90
	FPS	0.96	6.56	12.87	1.47
Pedestrian 2	Time (s)	409.18	26.22	24.62	417.93
	HOG %	0.93	0.46	0.32	0.95
	FPS	0.83	12.89	13.73	0.81
Pedestrian 3	Time (s)	184.13	15.39	13.17	118.04
	HOG %	0.95	0.43	0.28	0.92
	FPS	1.00	11.96	13.97	1.56
Charger	Time (s)	1331.42	330.13	132.94	2797.03
	HOG %	0.98	0.92	0.85	0.99
	FPS	0.35	1.39	3.46	0.16
Camera	Time (s)	388.86	43.39	27.49	254.68
	HOG %	0.96	0.56	0.51	0.94
	FPS	1.00	8.99	14.19	1.53
Gurgaon	Time (s)	448.71	78.66	34.34	562.01
	HOG %	0.97	0.81	0.71	0.98
	FPS	0.36	2.03	4.66	0.28

C. *Success Plots of Distorted Videos*

We applied the same four types of distortion to the video sequences used in [1] and analyzed how the tracking framework performed when using OpenCV's and dlib's feature extraction algorithms. Table II shows the parameters used for generating the degraded videos for each severity level. We then obtained the success plot for each distortion type, sequence, and severity level. After averaging the success plots

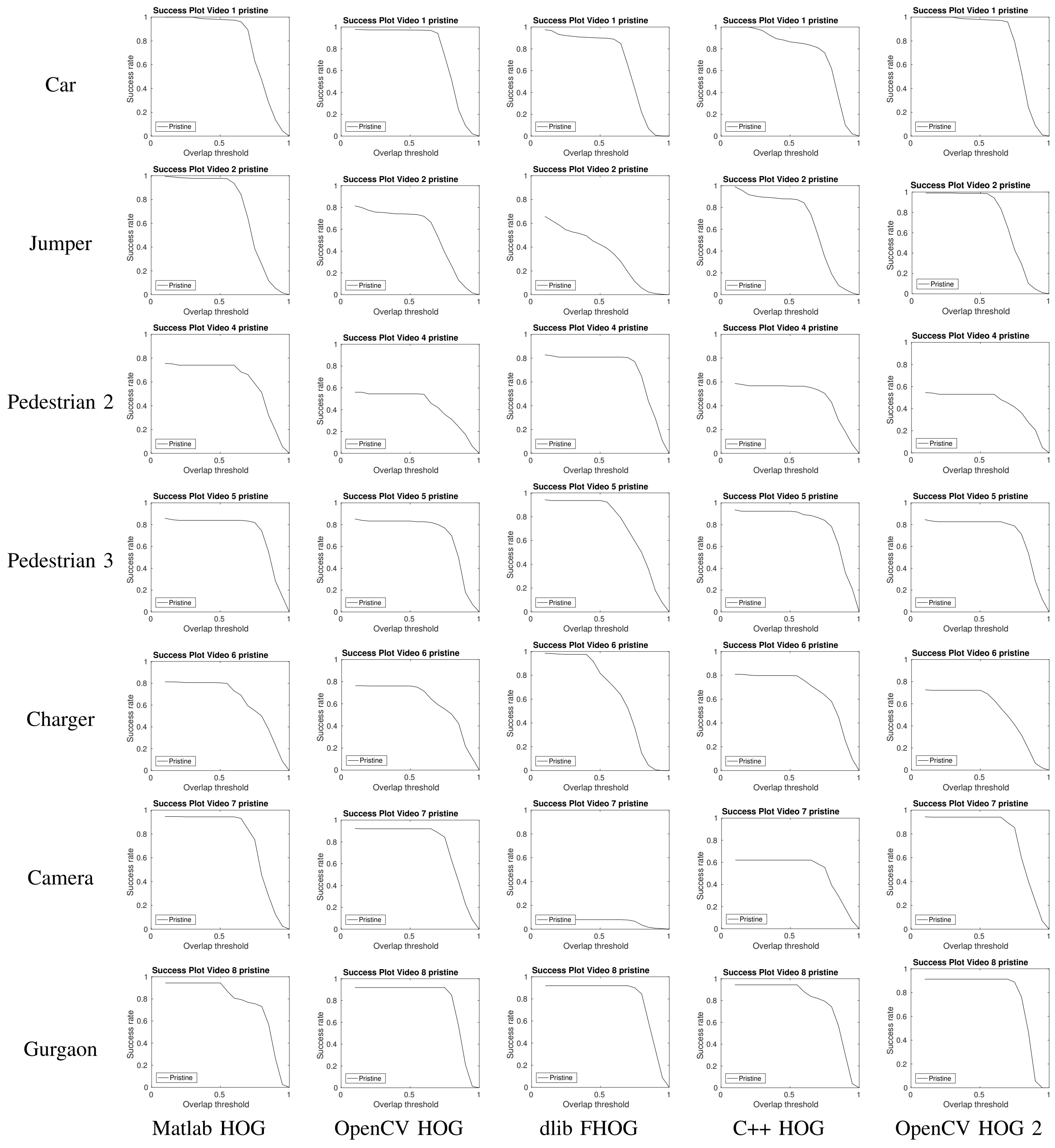


Fig. 3. Success plots of pristine videos. Ordered column wise by HOG implementation: the first column is Matlab's original HOG extraction implementation; the second column is OpenCV's implementation; the third column is dlib's implementation; the fourth column is our implementation; and the fifth column is OpenCV's implementation with modified `winSigma`.

for each video, we generated two success plots for each level of each distortion by using the both aforementioned HOG implementations. Figure 4 displays the resulting success plots.

As expected, the results we obtained from the implementation of OpenCV's HOG features were more consistent with the effects of each level of degradation than those of dlib's implementation. For example, Figure 4g shows a greater effect from distortion level 2 than 1, and Figure 4h displays a very similar behavior between pristine digrated videos with salt and pepper levels 1 and 2. In addition, the success plots generated with OpenCV are more similar to those generated using Matlab entirely.

One important issue with dlib's FHOG, is that its feature length (6) is different than the feature vectors obtained with the other implementations (1), and the tracker is designed to work with the latter vector length. Because the tracking framework guarantees a separability of object and background patches by the SVM, by setting the number of background patches

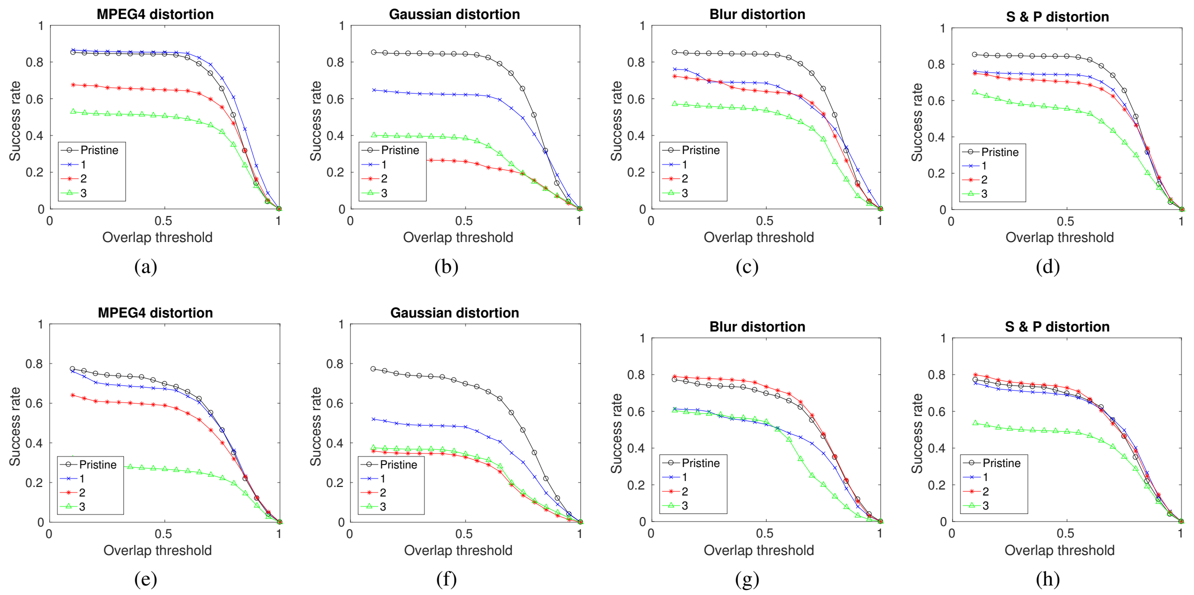


Fig. 4. Success plots for degraded videos. (a-d) are obtained when using OpenCV's HOG features, and (e-h) when using dlib's FHOG features. The plots represent the average result for all pristine videos (black), and levels of distortion 1 (blue), 2 (red) and 3 (green) applied, being 3 the most severe degradation.

TABLE II
PARAMETERS OF VIDEO DISTORTIONS APPLIED TO THE SEVEN TEST SEQUENCES [1].

	MPEG4	AWN	Blur	Salt & Pepper
Parameter	Quality(%)	Variance	Std. Deviation	Noise Intensity
Level 1	60	0.01	5	0.01
Level 2	30	0.05	10	0.05
Level 3	0	0.1	15	0.1

(N_{bg}) to 90% of the number of HOG features representing each patch (N_{HOG}), and by not allowing that the number of object patches N_{obj} surpasses 10% of N_{HOG} [1]. Hence, we expect to obtain better results using other implementations than dlib's FHOG features.

IV. Conclusion and Future Work

We experimented with three different HOG feature extraction algorithms: OpenCV's HOG features, dlib's FHOG features, and our own implementation. The previous tests show that the fastest method for the calculation of HOG features is dlib's FHOG function, which achieves a high speed at the expense of lower reliability of the tracker. One option for making this method perform better with the tracker was to adapt the framework to dlib's FHOG feature length. However, the aim of this work was not to modify the tracking algorithm, because such modification is problematic due to the complexity of the code. On the other hand, the results given by OpenCV's implementation show better performance at a significantly faster execution time than Matlab's HOG implementation. Unfortunately, our HOG algorithm from scratch did not offer a significant improvement over the other results.

One limitation of this work was that some of OpenCV's documentation, especially the `cv::HOGDescriptor` class. Having a clear definition of what some of the constructor parameters were, would have helped adapting the code to obtain more similar results to Matlab's implementation. Conversely, dlib's algorithm is based upon a different computation of HOG features, and offered fewer control parameters for the computation of the features.

One of the encountered problems is that the tracking algorithm did not guarantee to generate that the size of the image patches ($S_X \times X_Y$) was a multiple of the cell size ($p \times p$). In this context, the size of the image patches was set to that of the bounding box of the object to track, and also to be an odd number. With the cell size being calculated as $p = \text{floor}(S_X/4)$, there might be some loss of information when computing the HOG features, because the extraction expects the input image or ROI size described in equations 4 and 5.

There is still room for optimizing our own implementation. For example, Kim and Cho [12] propose a faster algorithm for HOG feature computation by removing some of the redundancy in overlapping blocks. Regarding the OpenCV's HOG extraction, it is possible to increase further the speed of the feature calculation by using its GPU counterpart. Unfortunately, this option could not be explored, as the entirety of this work was carried out using a virtual machine, which did not have a GPU module.

References

[1] H. D. Benítez-Restrepo, J. F. Flórez, and C. G. Rodríguez, "Object tracking on visually distorted videos," Pontificia Universidad Javeriana, Cali, Tech. Rep., 2016.
[2] M. B. Blaschko and C. H. Lampert, "Learning to localize objects with structured output regression," in *European conference on computer vision*. Springer, 2008, pp. 2–15.
[3] A. Vedaldi, V. Gulshan, M. Varma, and A. Zisserman, "Multiple kernels for object detection," in *2009 IEEE 12th international conference on computer vision*. IEEE, 2009, pp. 606–613.
[4] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[5] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 24, no. 5, pp. 603–619, 2002.

[6] D. Comaniciu, V. Ramesh, and P. Meer, "Kernel-based object tracking," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 25, no. 5, pp. 564–577, 2003.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 886–893.

[8] V. Argyriou, D. Rincón, J. Martínez, B. Villarini, and A. Roche, "Registration for motion estimation and object tracking," *Image, Video & 3D Data Registration: Medical, Satellite & Video Processing Applications with Quality Metrics*, pp. 53–78.

[9] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2. Ieee, 1999, pp. 1150–1157.

[10] G. Bradski, *Dr. Dobb's Journal of Software Tools*.

[11] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[12] S. Kim and K. Cho, "Fast calculation of histogram of oriented gradient feature by removing redundancy in overlapping block." *J. Inf. Sci. Eng.*, vol. 30, no. 6, pp. 1719–1731, 2014.