University of Padova

Department of Information Engineering

Analysis of Parking Lot Occupancy

Computer Vision

Final Project - February 2025

Group CV24

Pietro Volpato - 2120825

Ali Esmaeili nasab - 2091086

23/02/2025

Abstract

This project develops a computer vision system for parking lot management, using surveillance camera images to detect and classify parking space occupancy. The system identifies parking spaces in empty lots, classifies them as "empty" or "occupied," and segments cars as "correctly" or "incorrectly" parked. The parking lot status is displayed in real time through a 2D top-down map. Tested under various conditions, the system demonstrates good performance in detecting spaces and vehicles. The approach provides a straightforward, image-processing-based solution for real-time parking space monitoring and management.

Introduction

Video surveillance systems are increasingly utilizing computer vision algorithms to track and identify objects in real-time, providing valuable semantic information. One such application is in parking lot management, where image processing techniques automate the detection of vehicle occupancy, ensuring accurate information on parking space availability. This information can be used to monitor parking usage over time or to identify vehicles parked incorrectly.

For this project, we developed a computer vision system designed to manage parking lot occupancy. The system processes images captured from surveillance cameras to detect vehicles and determine the occupancy status of parking spaces. The result is an overview of the parking lot's status, indicating which spaces are available or occupied, displayed through a 2D top-down view minimap, and indicating the incorrectly parked cars.

The system uses two types of images: one from an empty parking lot and another showing the lot with vehicles. The empty parking lot images help to detect the parking spaces in an unobstructed scenario, while the images with cars provide insight into parking space usage at various times throughout the day.

To achieve this, the system performs the following steps:

1. **Parking Space Localization**: Parking spaces in the empty parking lot images are detected and localized using rotated bounding boxes, aligned with the main dimensions of each parking space.

2. **Occupancy Classification**: The detected parking spaces are classified based on their occupancy status, indicating whether a space is “empty” or “occupied.”

3. **Car Segmentation**: The system segments cars into two categories:

    - **Correctly parked cars**: Vehicles parked within the boundaries of the detected parking spaces.

    - **Incorrectly parked cars**: Vehicles parked outside the designated spaces.

4. **2D Visualization**: The system generates a real-time 2D top-down visualization map of the parking lot, where each parking space is color-coded according to its occupancy status: blue for available and red for occupied spaces.

The system was tested using various image sequences extracted from a surveillance camera. These sequences varied in terms of weather conditions, time of day, the number of vehicles in the parking lot, and the presence of pedestrians.

To simplify the project scope, certain assumptions, given by the project description, were made:

- The camera pose remains fixed throughout the image sequences.
- The focus is placed on the two primary areas of the parking lot with the largest number of parking spaces, excluding the upper-right portion where fewer spaces are located.

We want to point out that we could not test our code in the vlab due to an error reported in the forum, so we are not sure if our code runs in the virtual lab.

Another important fact is that all our results that involve the use of bounding boxes of the correspondent parking spot are referred to the ground truth bounding boxes given from the assignment, since we were not able to obtain good results from the parking space detector method developed and all our performances would have been compromised by these fact(since few parking spaces are detected).
The group was made up of 3 people but one person decided to quit from the group during this period.

Methodology

The project involved multiple tasks, each building upon the previous one in a hierarchical manner. Below, we outline each step in detail, explaining the approach and methodology used.

Each stage of the project was implemented separately, with different team members working on individual components in separate files to ensure modularity and clarity. The integration of these components was done progressively, allowing for iterative improvements and debugging.

Each team member contributed to different stages of development, and individual contributions will be detailed at the end of the report.

Detection

The parking space detection, which needs to be done in the images with empty parking spaces, i.e. the frames in the sequence0 folder, starts with the loading and the preprocessing of those images. Here we made another strong assumption, that is applying a RoI(Region of Interest) mask in all those images, discarding two portions of the images: the upper-right part where the parking spaces that we are allowed to discard are located and the part of the image at the left of the first row of parking spaces. Doing that, we can focus only on the parking lines that are relevant for our project and discard all the others.

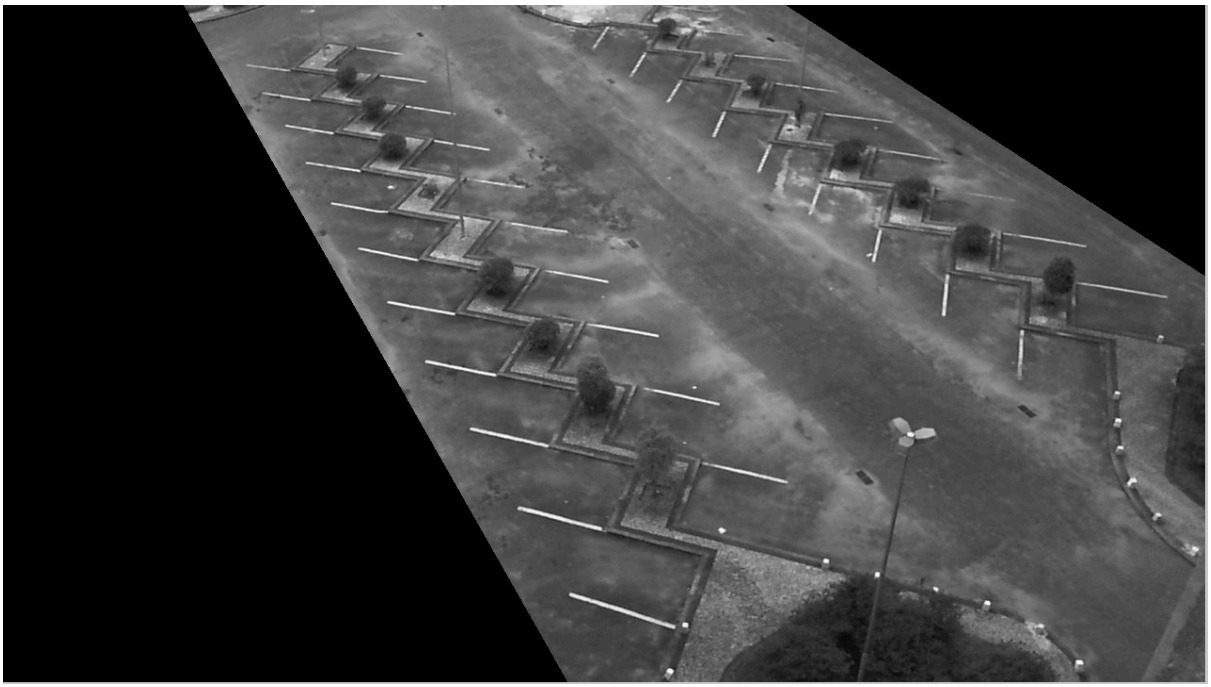


Fig 1: example of a preprocessed image of the application of the previous described ROI

After the storing of the images in a std::vector, they are preprocessed and converted in a grayscale image, to be used for the line detection. This line detection is performed by applying the LineSegmentDetector implemented in the OpenCV library. We choose that algorithm instead of the classic Hough transform approach since it has better performance. We also tried to use the cv::HoughLinesP function, but the performances in terms of parking line detection decreased significantly. For this purpose, we implemented in our code the gamma color correction algorithm and the CLAHE (Contrast Limited Adaptive Histogram Equalization) for equalizing the histogram of each image. Both of those algorithms are thought to handle the variation of the illumination changes in the image provided in the dataset. Although these approaches improved the detection of the lines using

the Hough transform, they reduced the performance of the Line Segment Detector; given that, we decided to keep the latter since it shows better performances overall. We also tried different approaches, one of the most noticeable was to use the cv::adaptiveThreshold function after the application of a Gaussian filter on the images and apply a series of morphological operations for improving the parking lines and combines the resulting mask with another mask obtained by using the Sobel operator for getting the edges. We also tried to apply a bilateral filter for comparing the result with the Gassian and it came out that the latter one was better. Unfortunately, these methods resulted in a loss of accuracy in the line detection(using the Hough transform).

After the preprocessing of the images, we apply the Line Segment Detector algorithm; we opted for keeping the default parameters and we decide for not refining the lines using the methods implemented in the algorithm since we wanted to do our filtering of the lines and because some important lines were lost by both the standard and the advanced refinement of the Line Segment Detector. The detected lines are defined as cv::Vec4i, which are the endpoints of the lines but we opted for storing the parameters of each line in a struct since for other methods we need to have the length and the orientation of the lines, so for computing them once we opted for that solution. Note that the angles are normalized in the range [0, 180) since it is useful to have a normalized range of angles for having the same angles for parallel lines(such as 0°-180° or 30°-210°). Talking about the filtering of those detected lines, we noticed that a large proportion of the parking lines have two major orientations, one is in the range of 15° and the other in the range [90°, 125°]. So after having discarded all the lines that are out of those ranges, we noticed that there were a considerable number of lines that were duplicated or were completely wrong for the detection of parking spaces. For discarding this lines, we decided to analyze group of lines that are close under a certain distance and apply the PCA(Principal Component Analysis) for computing the major orientation of the group of lines and keep only the “strongest lines”(the ones that have a similar orientation with respect to the major one) and finally merge the lines that lines that are close.

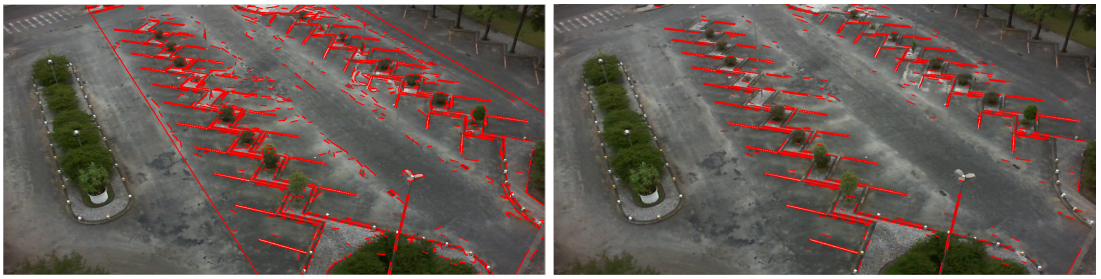


Fig 2-3: example of image with all the detected lines and image with the filtering based on angles

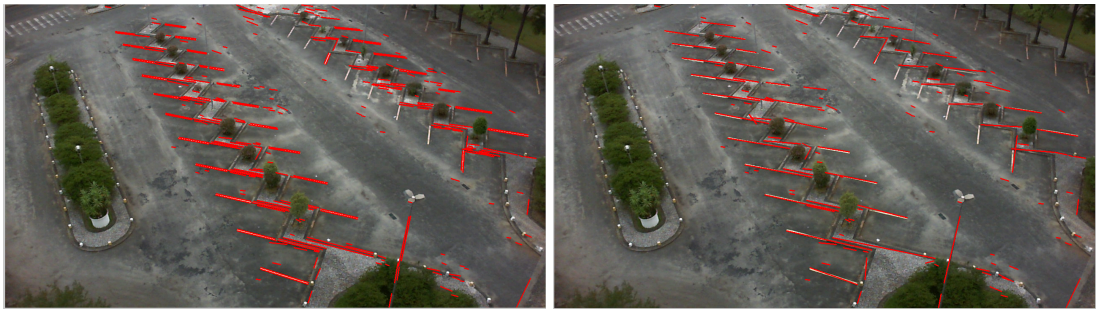


Fig 4-5: example of image with the filtering of lines based on PCA and image with merged lines

As shown in figure, we encountered a considerable issue: all the lines of the third row are wrong. The problem could be that PCA finds the predominant orientation of the lines that are the one that is not the wanted orientation but the orientation of the other lines. Unfortunately, we were not able to find a suitable solution to that problem, although we thought that maybe a filtering of the lines based on the region of the image where they are located could have produced better results but for time reason we were not able to develop the code for that solution.

After the refinement of lines, we implemented a clustering of these lines. It is needed for pairing the lines that have similar orientation and have a distance consistent with the dimension of a parking spot. For that purpose, first we opted for a greedy approach of processing all the lines that are parallel and at a certain distance and store these pairs in a vector, but then we tried a different and more complicated approach that is a modified version of the k-means clustering based on the angles(for details, see the code). After some testing, we saw that the brute-force approach was more effective and we kept that.

Given the clusters of pairs of lines, the detection of parking spaces is straightforward. For each pair of lines, check that they are parallel and are sufficiently distant and create a cv::RotatedRect in correspondence of these two lines. A refinement is also performed on these bounding boxes.

We have to admit that our method seems good in theory but in practice it does not work. The results are really bad in terms of parking space detection.

Classification

The classification step focuses on determining whether each parking space is occupied or free. This is done by processing the bounding boxes (represented as cv::RotatedRect objects) obtained from earlier steps and classifying them into two categories: **occupied** and **free** spaces. Since we could not achieve the parking space bounding boxes, we are going to use the BBoxes given in the ground truth to show the performance.

Initial Approach:

The first stage in the process, as with any image processing task, involves preprocessing the images. This includes converting the images to grayscale and applying a Gaussian blur to smooth the image, thereby reducing the effects of shadows and illumination. These steps are essential for enhancing the accuracy of further image processing.

The initial idea was to apply a thresholding technique to the edge map generated by the Canny edge detector. The rationale behind this approach was that, if a car were present in a given bounding box, the edge map would contain more white pixels representing the car's edges. However, this method proved ineffective, as the asphalt patterns and the parking lines were also detected in empty parking spaces, which created noise in the edge map.

Alternative Methods Explored:

The next attempt was to introduce a comparison to the asphalt color (set to cv::Scalar(127, 127, 127)). However, this approach had limitations because the asphalt color varies, and it's difficult to define a consistent threshold for it. To overcome this, the method was extended to use double thresholding to try and eliminate the asphalt color from the image, but it still didn't provide satisfactory results.

Another approach involved using **K-means clustering** on both the thresholded image and the edge map. This method aimed to segment the regions of interest, but it also had its drawbacks, as the clustering wasn't able to distinguish well between empty spaces and occupied ones due to the complex patterns of the parking lot.

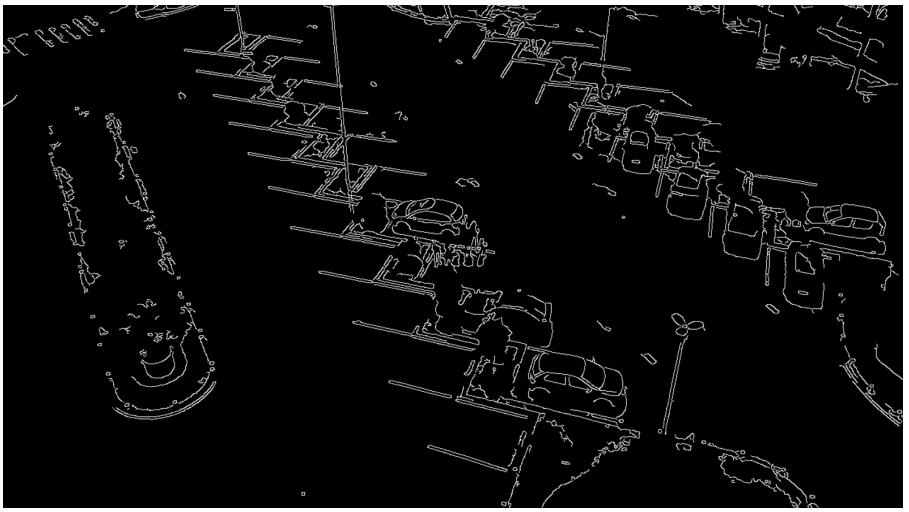
We also experimented with adaptive and Otsu thresholding on the edge maps, as well as applying **contrast stretching** and **histogram normalization** to enhance the image's dynamic range and improve the quality of detection.

Feature Detection Approach:

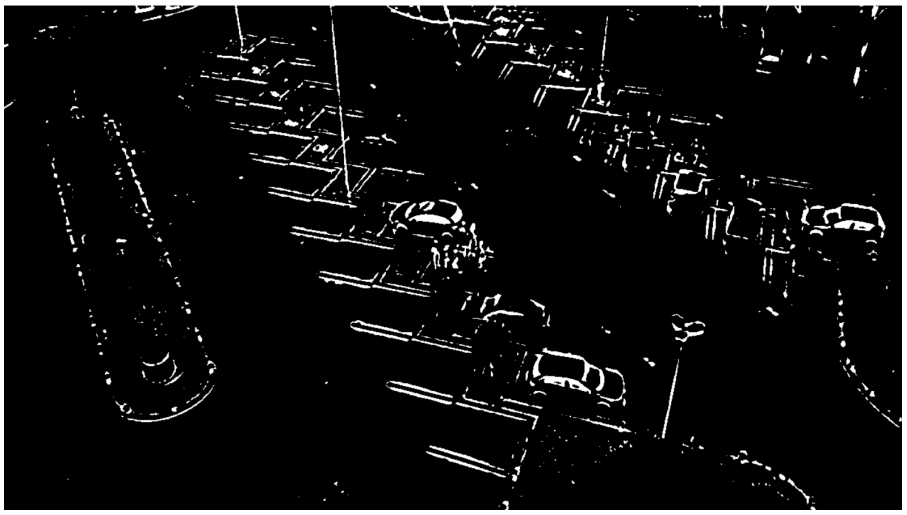
An additional idea was to use the empty parking space as a reference for classification. Initially, simple methods were employed to compare features, but later more sophisticated techniques like **ORB** (Oriented FAST and Rotated BRIEF) and **SIFT** (Scale-Invariant Feature Transform) were applied to detect distinctive features in the empty parking space. These features were then matched to each bounding box (BBox). The assumption was that if the bounding box contained features similar to the asphalt and ground (which would be present in an empty space), the space would likely be classified as free.

Addressing Cropping Issues:

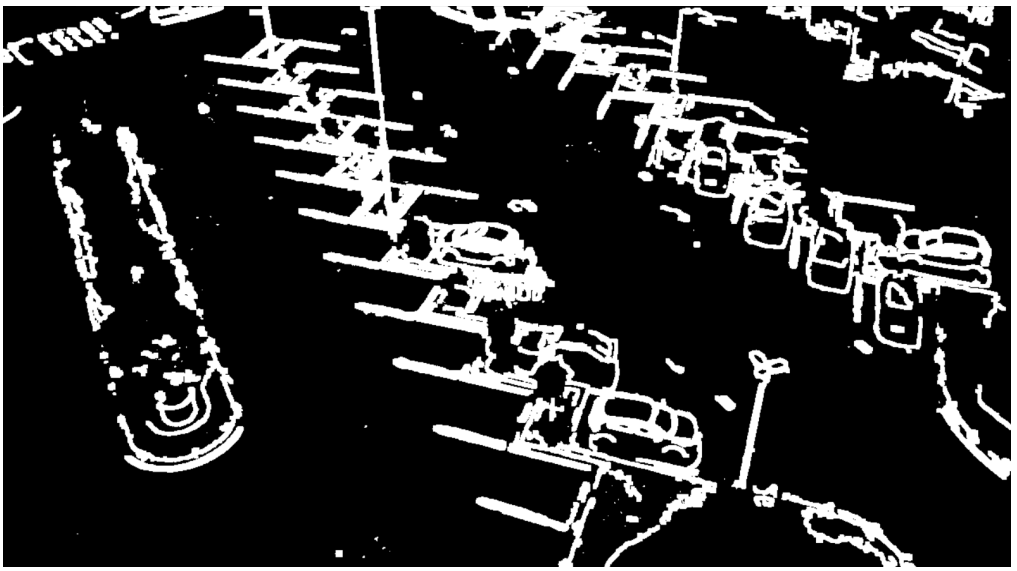
One of the challenges faced during the classification process was that the bounding boxes were not perfectly aligned with the parking spaces in the original image. To address this, a new method was implemented: rotating the entire image around the bounding box center using warpAffine. This approach ensured better alignment and more accurate classification.



Edge



Threshold



Threshold + Edge

Filled (Final image to be used)

Final Approach:

After evaluating various methods, the final approach combines the edge map and the threshold map to classify the parking spaces more effectively. The function preprocesses the parking lot image by converting it to grayscale, applying Gaussian blur, and detecting edges using the Canny edge detector. It then compares these edges with a reference image of an empty parking lot, subtracting background features to focus on the parking spaces. Morphological operations are used to refine the detected regions, and the bounding boxes are classified based on the ratio of foreground pixels. Finally, the results are stored in a vector, indicating whether each parking space is occupied or free.

Here are the key steps:

1. **Combining Edge and Threshold Maps:** The edge map (generated from the Canny edge detector) and the threshold map (obtained from adaptive thresholding) are combined using a **bitwise OR** operation. This allows the two types of information to complement each other and highlight both edges and areas of interest.
2. **Morphological Operations:** To improve the segmentation, morphological operations such as dilation are applied to expand the white regions in the binary image. This helps in connecting fragmented regions and making the boundaries of the cars and parking spaces more continuous.
3. **Subtracting Empty Space Features:** A reference edge map of the empty parking space is created using the hue channel of the HSV color space. This is particularly useful for capturing patterns in the asphalt and distinguishing them from objects (like cars) in the parking lot. The empty space edge map is subtracted from the combined edge and threshold map to remove features that correspond to the empty spaces, leaving only the occupied areas.
4. **Filling Contours:** The contours of the resulting image are filled using the `cv::drawContours` function, which fills closed regions (such as cars) with white pixels. This ensures that the occupied spaces are filled, making them easier to detect.
5. **Morphological Closing:** To further refine the results, morphological closing is applied, which combines dilation and erosion to close small gaps and connect nearby regions. This helps ensure that the detected cars and other features are solid and well-defined.
6. **Classifying Parking Spaces:** Finally, each bounding box is analyzed by computing the ratio of white pixels (representing potential cars) inside the bounding box to the total number of pixels. A predefined threshold value (e.g., `empty = 0.4`) is used to determine whether the parking space is occupied. If the ratio exceeds the threshold, the space is classified as **occupied**; otherwise, it is classified as **free**.

This final approach improves classification accuracy by using a combination of edge detection, thresholding, morphological operations, and reference-based subtraction. The use of feature detection in empty spaces and morphological techniques ensures that both occupied and free parking spaces can be identified with higher precision.

Additionally, a visualization method was implemented where bounding boxes were drawn around parking spaces, with their outlines colored red for occupied and green for empty spaces. This provided an intuitive way to verify the system's performance.

Furthermore, a manual contrast stretching method was developed to enhance image clarity, but it was ultimately not used in the final implementation due to limited impact on classification accuracy.

I also implemented a method to compare the classification results with the ground truth and compute key performance metrics, including True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). This allowed for a detailed assessment of the system's performance and the accuracy of the classification for each parking space. These metrics provided valuable insights into the areas where the system performed well and where improvements were needed.

Segmentation

This task requires us to segment the cars in the images and label them as correctly parked in a parking space or not. For this purpose, the images are preprocessed converting them in grayscale

and applying a RoI. The RoI is similar to the ones previously discussed but in this case we discard only the upper-right portion of the images. So now we have tried two different approaches:

1. Take all the images with the empty parking spaces and create an “average image” by taking the median of each pixel in the set of 5 images with empty parking spaces and then apply the cv::absdiff function for creating a mask with the differences in the images. This approach showed poor results since the cars with dark colors were hard to detect in the mask
2. Use the Background Subractor for creating the mask. First, we train the model using the images with empty parking spaces adding the images with gamma color correction and images equalized using CLAHE.

We opted for the second approach that showed better results than the other one.

After the creation of the mask, we enhance the quality of it by applying two morphological operations, an opening followed by a closing, and a threshold of 200. After that we tried two methods for segmenting the cars:

1. Given the mask, we find the contours and get the corresponding rotated bounding box using the function cv::minAreaRect. After that a filtering and merging algorithm is applied to all the bounding boxes created; the logic is the following: detect the large bounding boxes that are likely to represent a car, merge them if some of them are overlapping, and iteratively merge the large bounding boxes with the smallest one, since there are some small bounding boxes that represent only a portion of the car. Avoid merging the large bounding boxes in this phase of the algorithm since we want to merge the small with the large bounding boxes.
2. The second technique we experimented with is based on the OpenCV function *connectedComponentsWithStats* that taken as input the mask computes the connected components labeled image and produces in output some statistics. In particular, we can create the mask of each labeled portion of the image that theoretically represents a car and use the statistics produced by the algorithm for creating a cv::Rect that represents the car.

In the end, we decided to use the first approach, since it has good results.

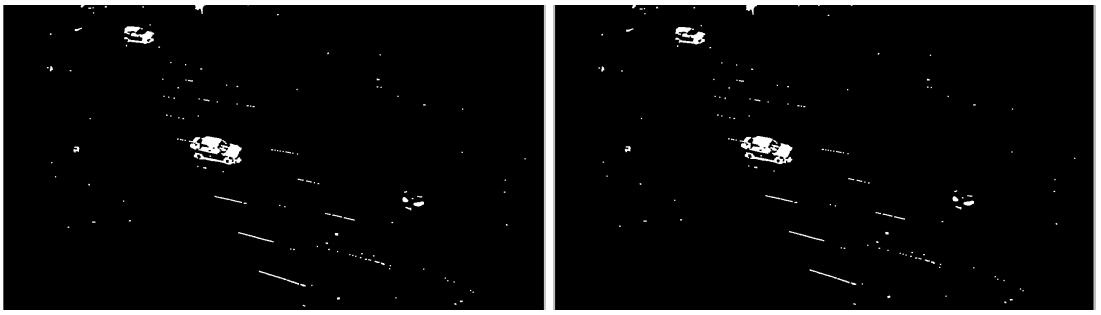


Fig 6-7: example of the output of the Background Subtractor and the corresponding enhancement

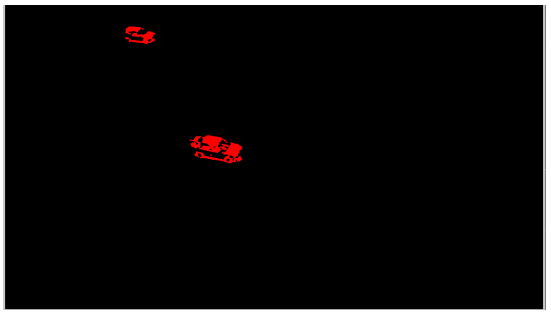


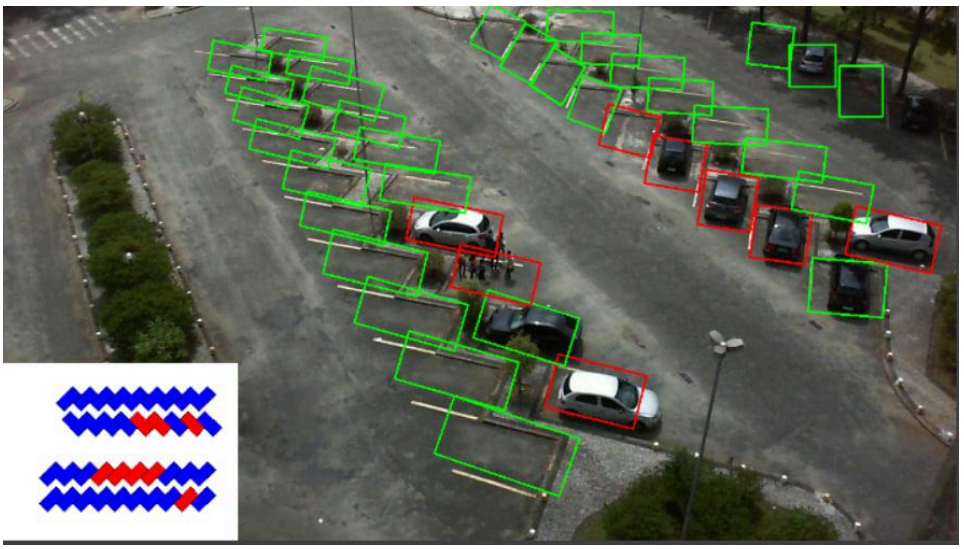
Fig 8: mask with the segmented carrs

To decide if the car is correctly parked, we took the bounding boxes that represent the parking spaces(theoretically computed in the first part of the project) and calculated if the center of the bounding box that represents the car is inside the bounding box of the parking spot.

Finally, we create the mask for the evaluation phase, where each pixel is labeled as requested by the assignment(0 the background, 1 car parked and 2 car out of the parking space).

Visualization

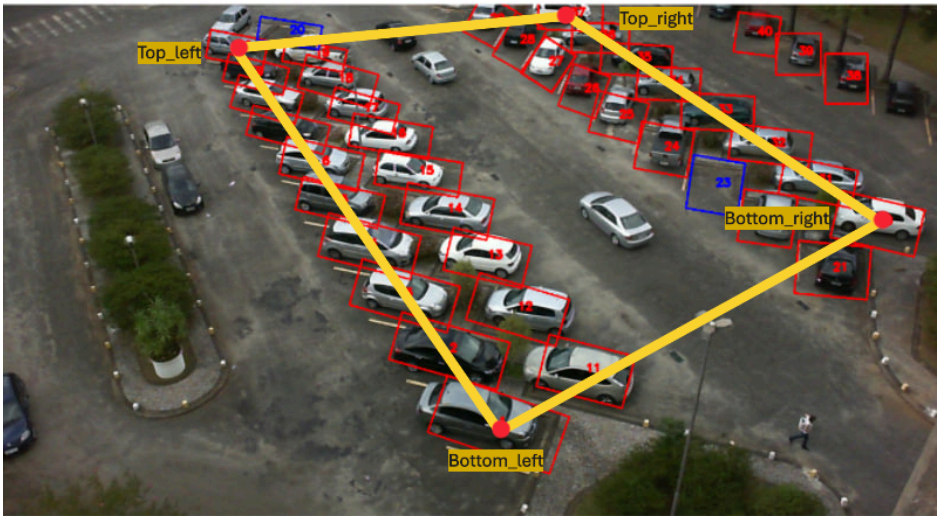
The final step of the project was to create a 2D top-view visualization map that represents the current state of the parking lot. Initially, a manual method was attempted, where a 2D map was drawn, and parking spaces were color-coded based on the occupancy vector obtained from the classification step. This required sorting and associating each parking space with its corresponding bounding box based on location. While this method provided a visually appealing representation, it was not well-suited for general use, as it depended heavily on predefined positions and lacked adaptability. Additionally, at the time of development, the full set of parking spaces was not available, making it uncertain whether all spaces were correctly detected.



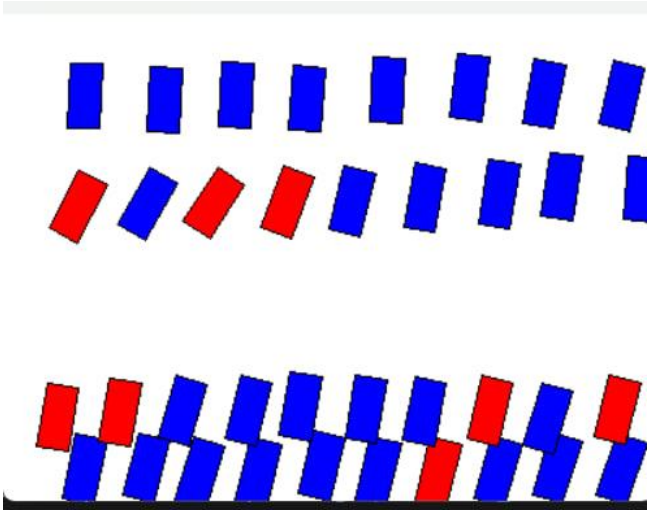
fig

To overcome these limitations, **homography** was used to generate the 2D map automatically. Homography requires four corresponding points from the original image and the target visualization space to compute the transformation matrix. The most reliable points were selected at the ends of the parking rows, representing the first and last parking spaces in two main rows. These points were determined using a method called `findExtremePoints()`, which identifies:

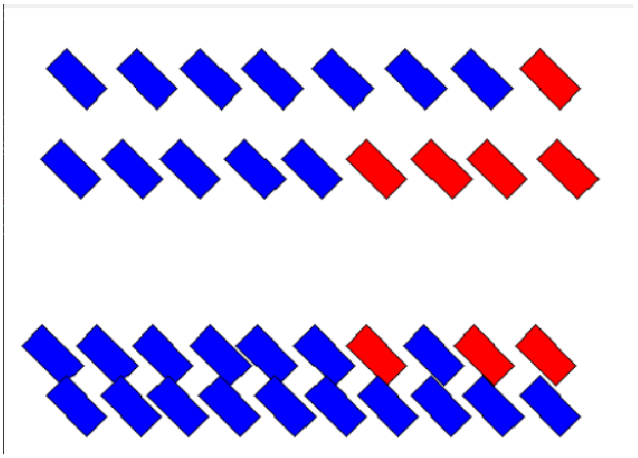
- **Bottom-left point:** Bounding box center with the highest y-coordinate.
- **Top-left point:** Bounding box center with the lowest x-coordinate.
- **Bottom-right point:** Bounding box center with the highest x-coordinate.
- **Top-right point:** Bounding box center with the lowest x-coordinate.



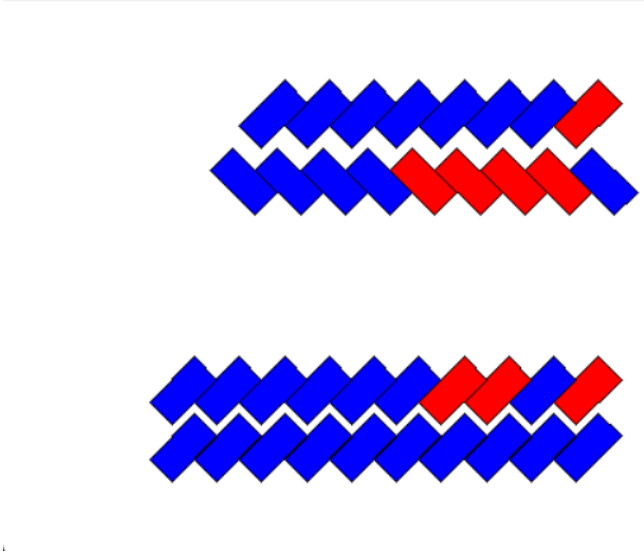
These four points were then mapped to a fixed rectangular space `{380, 280}, {50, 280}, {380, 50}, {50, 50}` to compute the **homography matrix (H)** using `getPerspectiveTransform(srcPoints, dstPoints)`. This transformation matrix was then applied to all bounding box centers using `perspectiveTransform(originalCenters, transformedCenters, H)`, ensuring an accurate mapping of parking spaces from the input image to the 2D top-view visualization.



However, directly applying the transformation to the bounding box centers did not yield satisfactory results. The transformed positions were not well-aligned with the expected grid of parking spaces. Initially, we attempted to calculate the average y-value of each cluster and use this to draw the parking spaces, but this approach still did not produce satisfactory results.



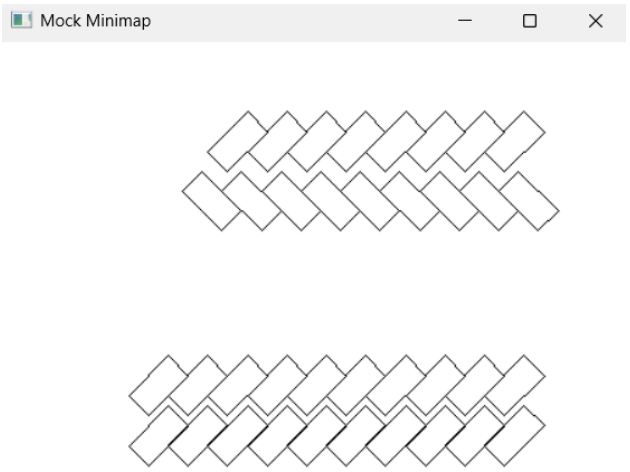
To refine the process, a more effective solution was implemented by combining the homography-based method with clustering. The transformed centers were clustered based on their y-axis values into four distinct clusters, corresponding to the four parking rows. This clustering helped group each bounding box into its relevant row, making it possible to determine how many bounding boxes exist in each row. Once the centers were grouped, they were sorted by both their x and y values, ensuring a consistent order of parking spaces in each row.



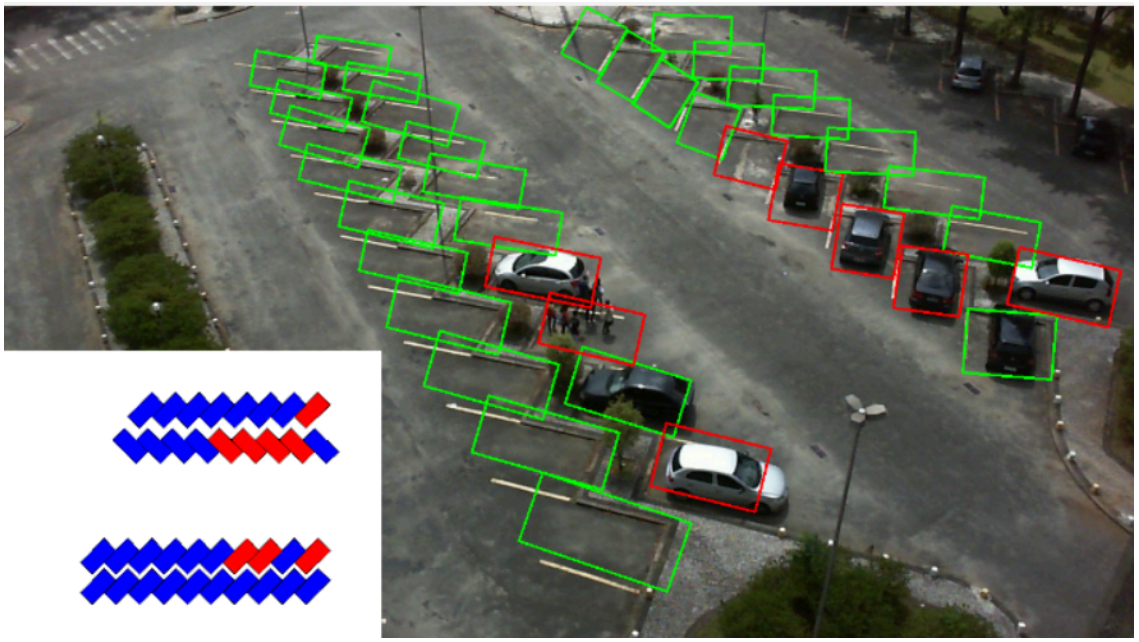
By sorting the clusters and knowing the predefined width, height, and angle of the parking spaces, rectangles could be drawn with these dimensions in each row. This approach is more general, as it adapts to varying numbers of parking spaces per row and the overall parking layout. However, it does require the number of rows to be predefined. Once the parking spaces were drawn, the occupancy status for each space was determined by matching the index of each center in the clustered version to its corresponding index in the original input. This final method allowed the system to accurately map parking spaces and their occupancy in the 2D visualization, ensuring a robust and adaptable solution.

Using this structured approach, a minimap, referred to as "Mock Minimap," was created. This map was initially generated by clustering the rectangles and drawing their borders. The predefined width and height of the rectangles were then used to calculate the distance between rectangles in each row (ΔX). For the Y-values, the rows were positioned at the average Y-value of each cluster, except for the last row, which was refined using a ΔY adjustment to improve its visual alignment. After setting up the

structure, the occupancy status of each parking space was incorporated, with the colors being updated based on the classification results.



Additionally, an extra method was implemented to overlay the minimap onto the original parking lot image by resizing and placing it in a corner, providing a combined view of both the real scene and the structured parking layout.



Code Explanation

The `Visualizer` class manages the visualization of parking spaces by applying homography transformations, clustering, and drawing rectangles with different colors based on occupancy status. It uses OpenCV for image processing and geometric transformations.

Main Components and Functions:

1. **Constructor (`Visualizer::Visualizer`)**:
    - Initializes the visualizer with parking space information (width, height, and occupancy status).
    - Computes a homography matrix to map parking space centers to a new coordinate system using the `computeHomography` and `applyHomography` methods.
    - Check if the parking space centers and occupancy status vectors have matching sizes.
2. **`drawParkingSpaces`**:
    - Draw the parking spaces on an image as rectangles. The color of each rectangle (red for occupied, green for free) depends on the parking space's occupancy status.
3. **`drawRotatedRect`**:
    - Draws a rotated rectangle on an image, filling it with a specified color and drawing a border around it.
4. **`rectParkingRow`**:
    - Clusters the parking spaces into rows based on their positions and adjusts their placement accordingly. It calculates the average `y`-coordinate of each cluster to organize parking spaces into a row.
5. **`overlaySmallOnLarge`**:
    - Overlays a smaller image (e.g., a 2D visual representation of parking space occupancy) onto a larger image (e.g., the main parking lot image).
6. **`updateMinimap`**:
    - Updates the minimap (a smaller version of the parking lot) that reflects the occupancy status of the parking spaces. It draws the transformed parking space rectangles with color coding based on their occupancy status.
7. **`createMockMinimap`**:
    - Generates a mock minimap, which only shows the outlines of the transformed parking space rectangles.
8. **`findExtremePoints`**:
    - Identifies the extreme points (top-left, top-right, bottom-left, bottom-right) of the parking spaces for use in computing the homography matrix.
9. **`computeHomography`**:
    - Computes a homography matrix that maps the original parking space positions to a new coordinate system, specified by a set of destination points.
10. **`applyHomography`**:

- Applies the homography transformation to the original parking space centers to obtain their new positions.

11. **`clusterParkingSpaces`**:

- Groups the transformed parking space centers into clusters based on their `y`-coordinates using k-means clustering. It sorts the clusters and their elements based on their spatial positions to organize the parking spaces into rows.

Testing and Optimization

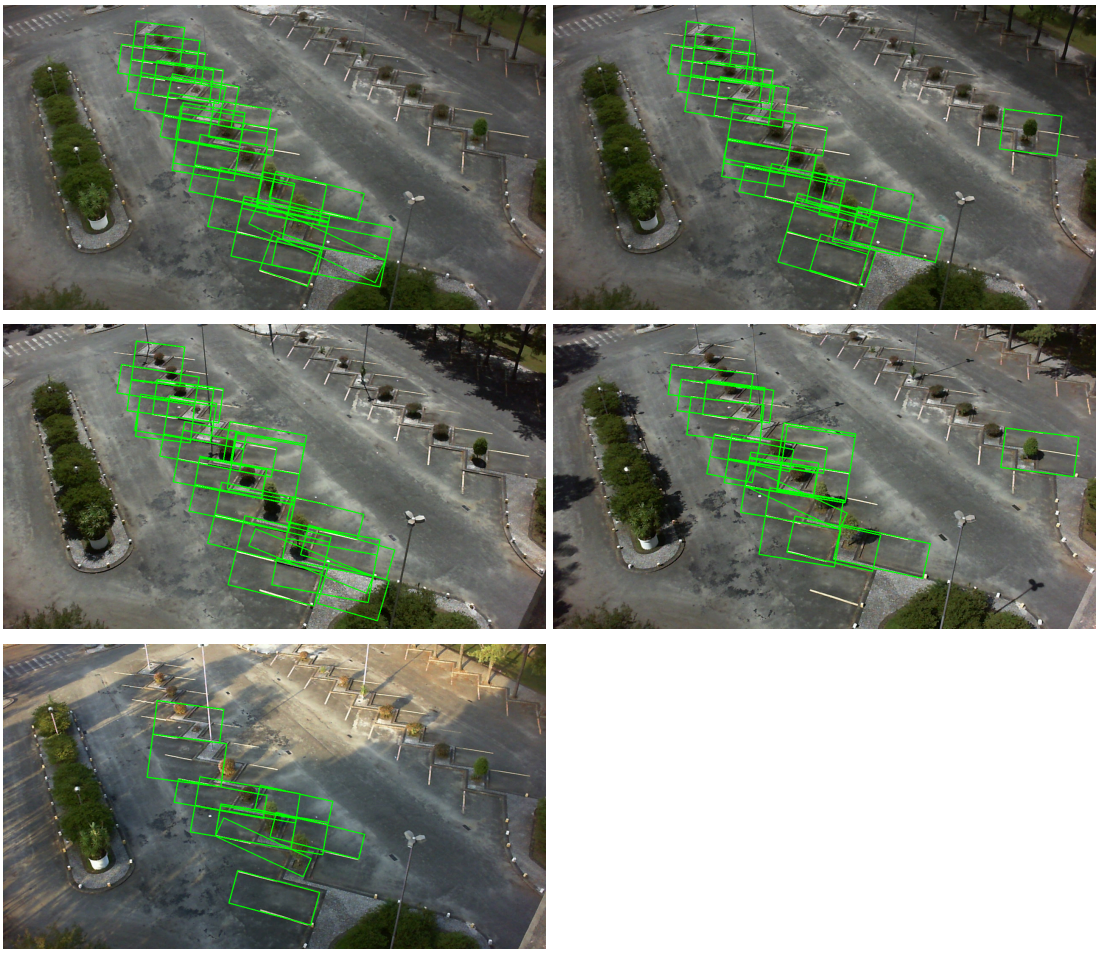
The two metrics we were asked to implement are the mAp and the mIoU, for the reference check the assignment. Due to time limitations, we were able to implement and test only the mIoU, which is used to evaluate the car segmentation.

SEQUENCE	mIoU
1	0.475663
2	0.433067
3	0.363345
4	0.40072
5	0.565971

Result

As stated in the table above, the performances are pretty bad, but these are the results that we were able to achieve considering that we had few times to work on the project and we have dedicated a large amount of the available time for searching and experimenting different approaches that could have been valid for improving the final results. One important fact is that since we were not able to compute the correct bounding boxes from point 1, all the tests have been done concerning the ground truth bounding boxes provided. We know that this was forbidden but since our detections were not good at all, we opted for adopting this solution for going on and not getting stuck.

Parking Space Detection Output



Classification Output

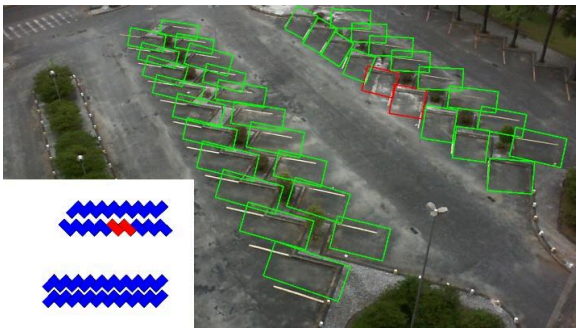
The classification achieved an accuracy of 88% on average on the dataset. The majority of incorrect classifications were false positives, where empty parking spaces were mistakenly detected as occupied. The model remains sensitive to variations in illumination, which affects its reliability under different lighting conditions. As seen in sequence 4 where we have the lowest accuracy while the lowest accuracy excluding this sequence is 83%. The ground pattern also affects the performance as seen in the two parking spaces in the third row usually classified as occupied.

Furthermore, the model struggled to fully differentiate pedestrians from vehicles, as seen in sequence 2, frame 4. This issue likely arose because the final binary image was unable to completely remove pedestrians. One potential solution to address this problem would be to implement a filter based on the size of the contours in the binary bounding boxes. By filtering out smaller contours that correspond to pedestrians, we could improve the system's ability to accurately classify vehicles and parking space occupancy.

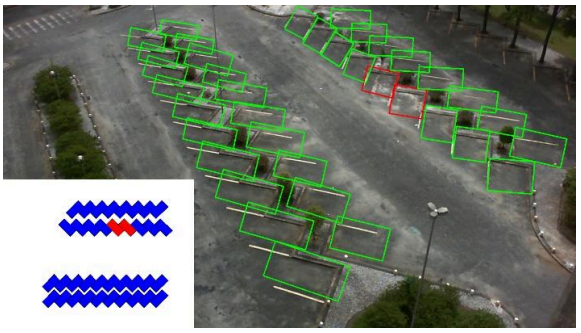
A potential improvement for future iterations would be the implementation of a variable threshold that adapts dynamically based on the original lighting conditions and shadows, rather than relying on a fixed threshold for occupancy detection. This adjustment could significantly enhance the robustness of the system in real-world scenarios.

The photos embedded with the 2D minimap are shown below, also an accuracy metric based on **True Positives (TP)**, **False Negatives (FN)**, **False Positives (FP)**, and **True Negatives (TN)** is provided. The recall and precision are available through the C++ code.

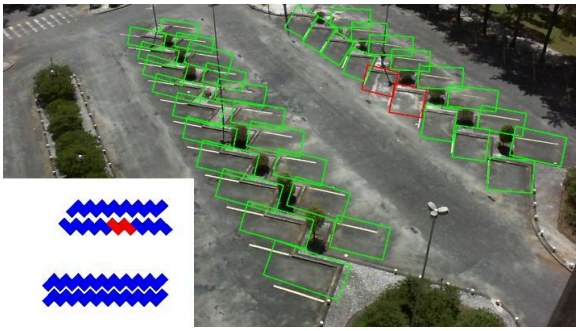
Sequence #0



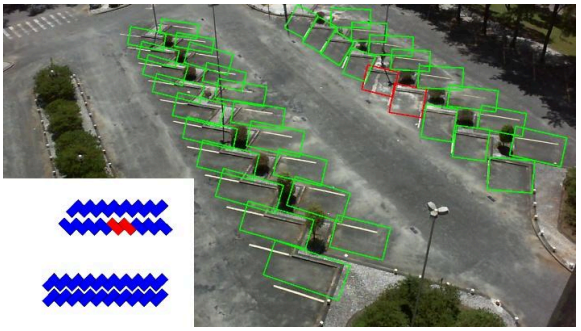
Frame #0



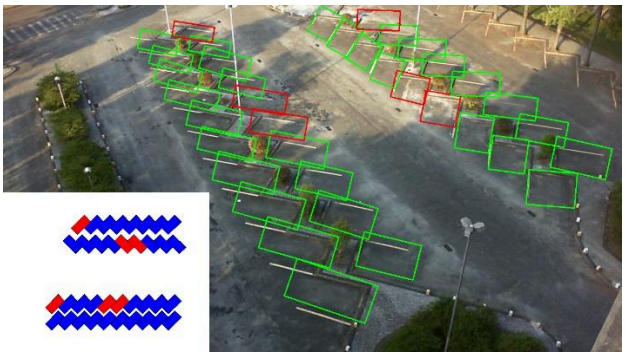
Frame #1



Frame #2



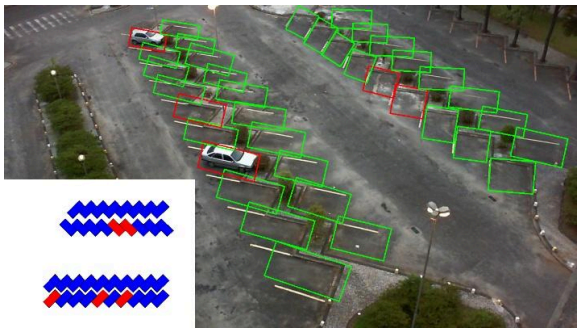
Frame #3



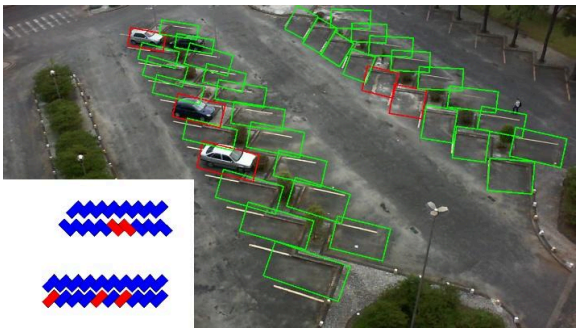
Frame #4

Frame	0	1	2	3	4
Accuracy	0.94	0.94	0.94	0.94	0.83
TP	0	0	0	0	0
FN	0	0	0	0	0
FP	2	2	2	2	6
TN	35	35	35	35	31

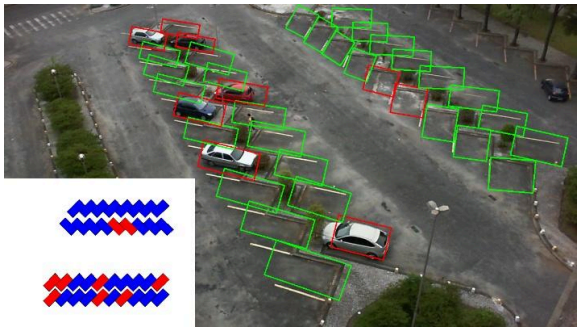
Sequence #1



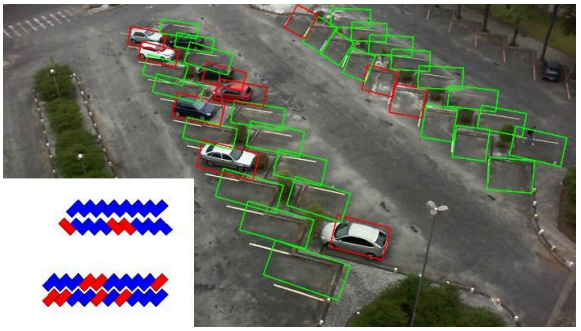
Frame #0



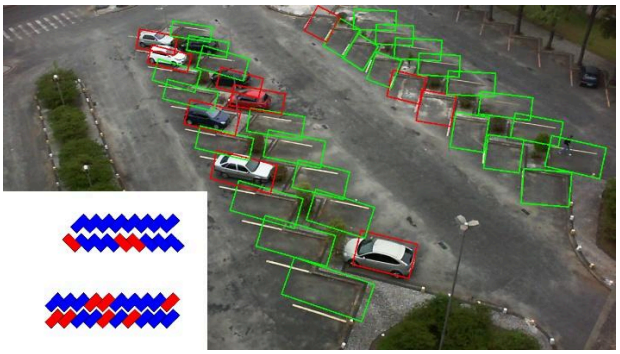
Frame #1



Frame #2



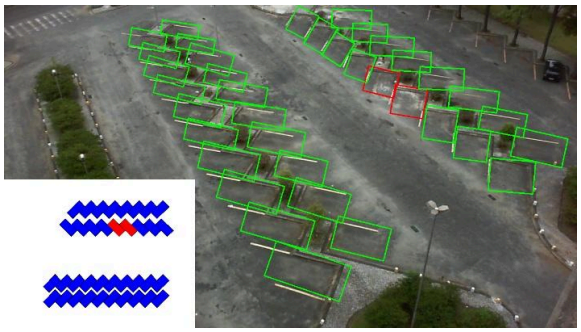
Frame #3



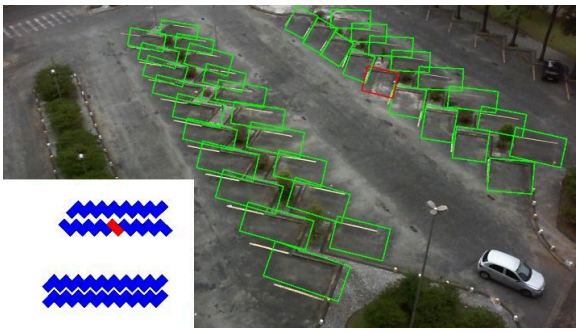
Frame #4

Frame	0	1	2	3	4
Accuracy	0.92	0.92	0.92	0.89	0.94
TP	2	3	6	7	11
FN	0	1	0	1	0
FP	3	2	3	3	2
TN	32	31	28	26	24

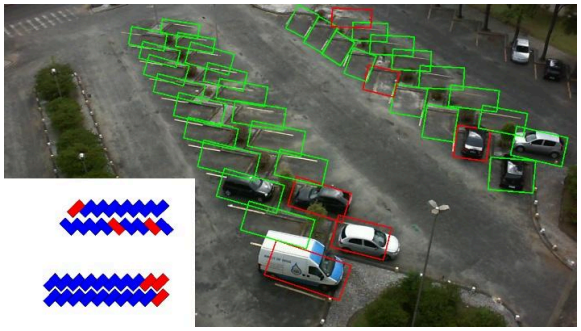
Sequence #2



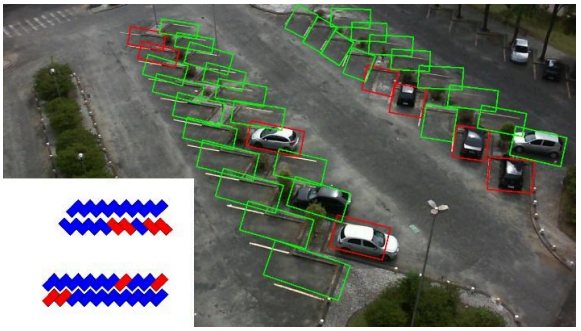
Frame #0



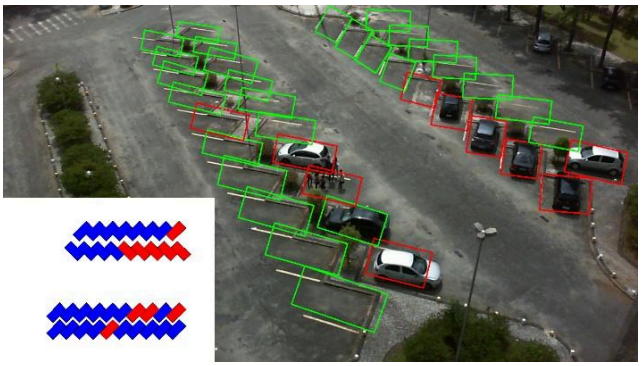
Frame #1



Frame #2



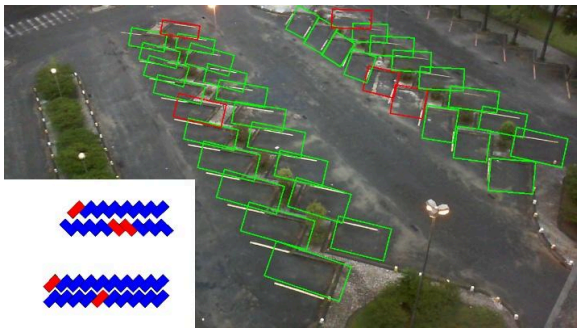
Frame #3



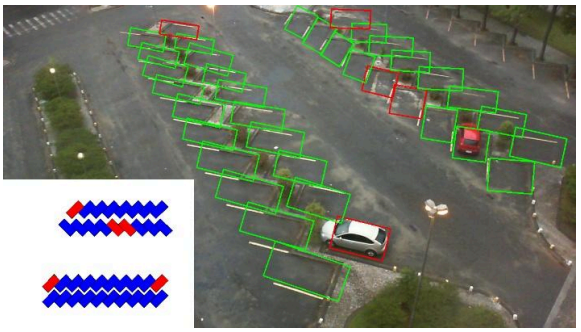
Frame #4

Frame	0	1	2	3	4
Accuracy	0.94	0.97	0.86	0.86	0.89
TP	0	0	4	5	7
FN	0	0	3	2	1
FP	2	1	2	3	3
TN	35	36	28	27	26

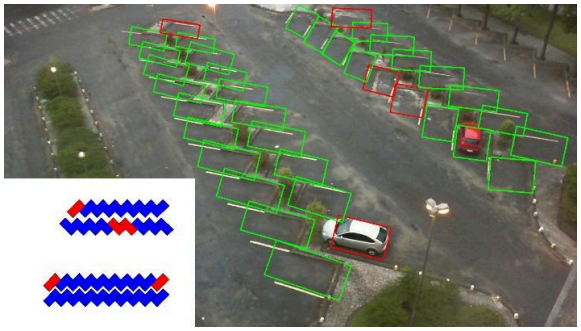
Sequence #3



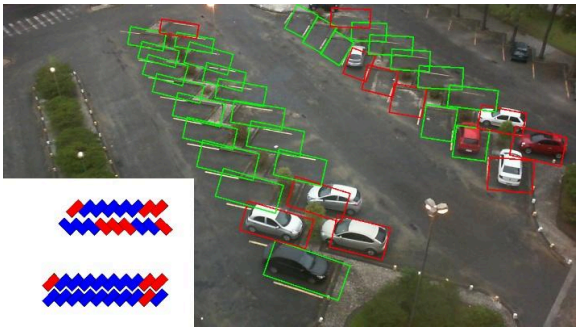
Frame #0



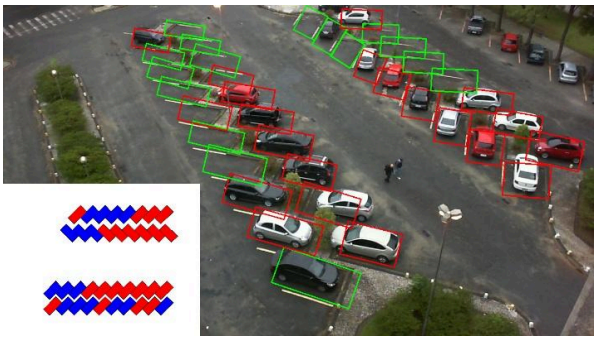
Frame #1



Frame #2



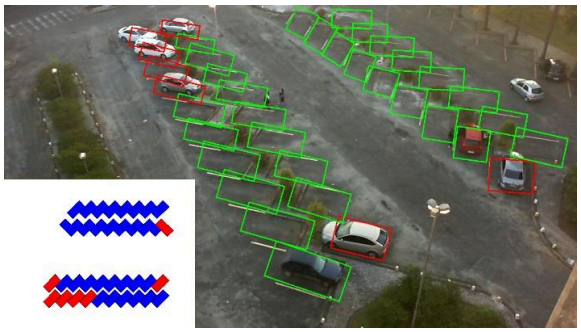
Frame #3



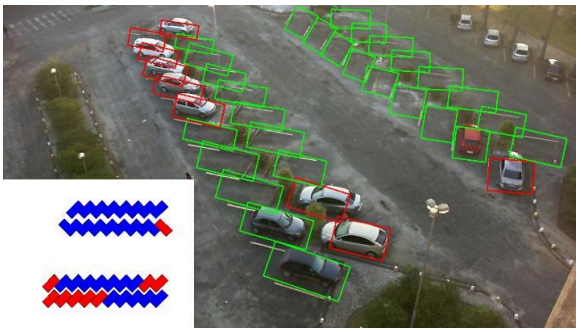
Frame #4

Frame	0	1	2	3	4
Accuracy	0.86	0.86	0.89	0.83	0.84
TP	0	1	4	7	18
FN	0	1	2	2	3
FP	5	4	2	4	3
TN	32	31	29	24	13

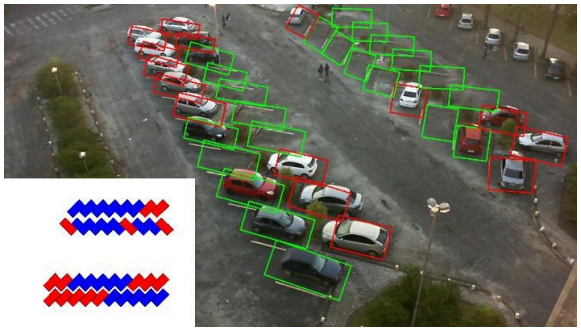
Sequence #4



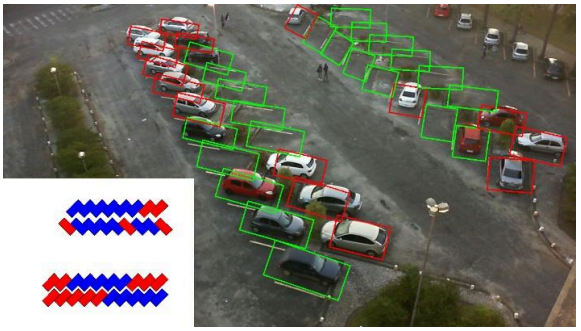
Frame #0



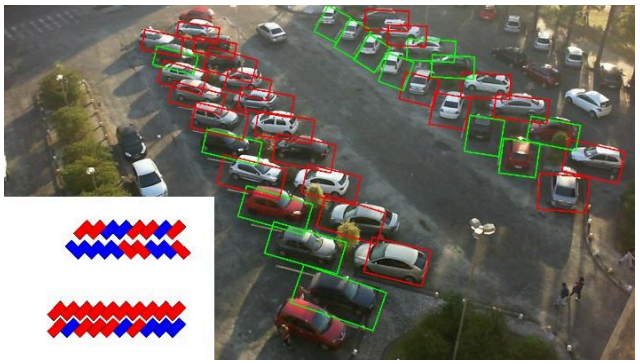
Frame #1



Frame #2



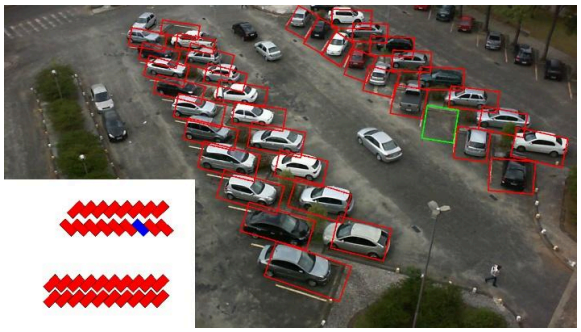
Frame #3



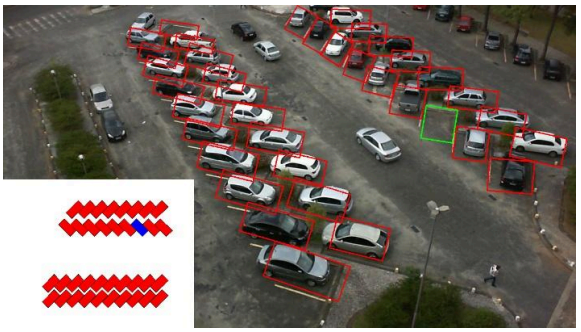
Frame #4

Frame	0	1	2	3	4
Accuracy	0.92	0.92	0.84	0.67	0.62
TP	6	9	15	24	23
FN	2	3	6	12	14
FP	1	0	0	0	0
TN	28	25	16	1	0

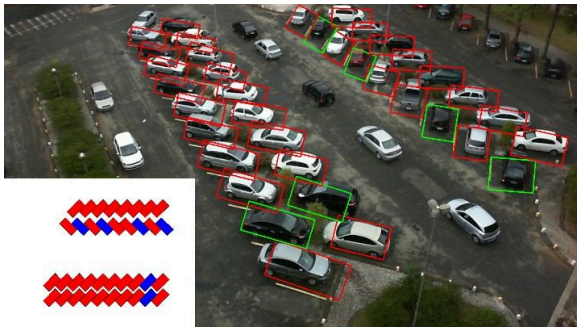
Sequence #5



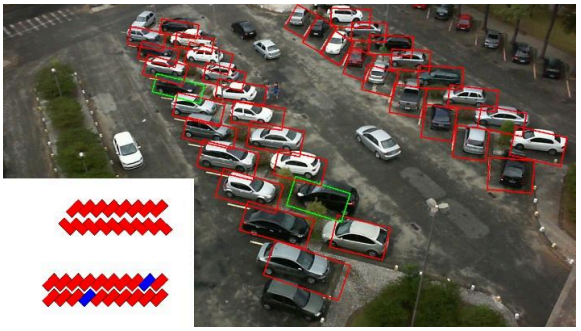
Frame #0



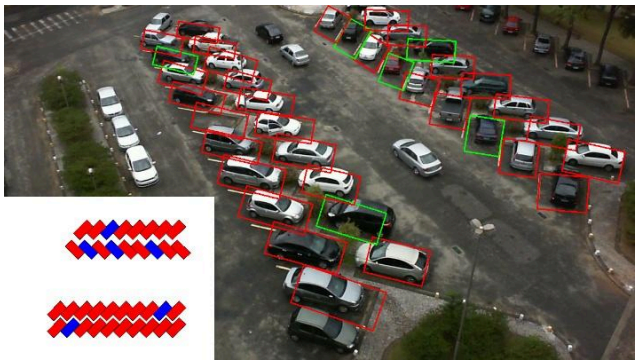
Frame #1



Frame #2

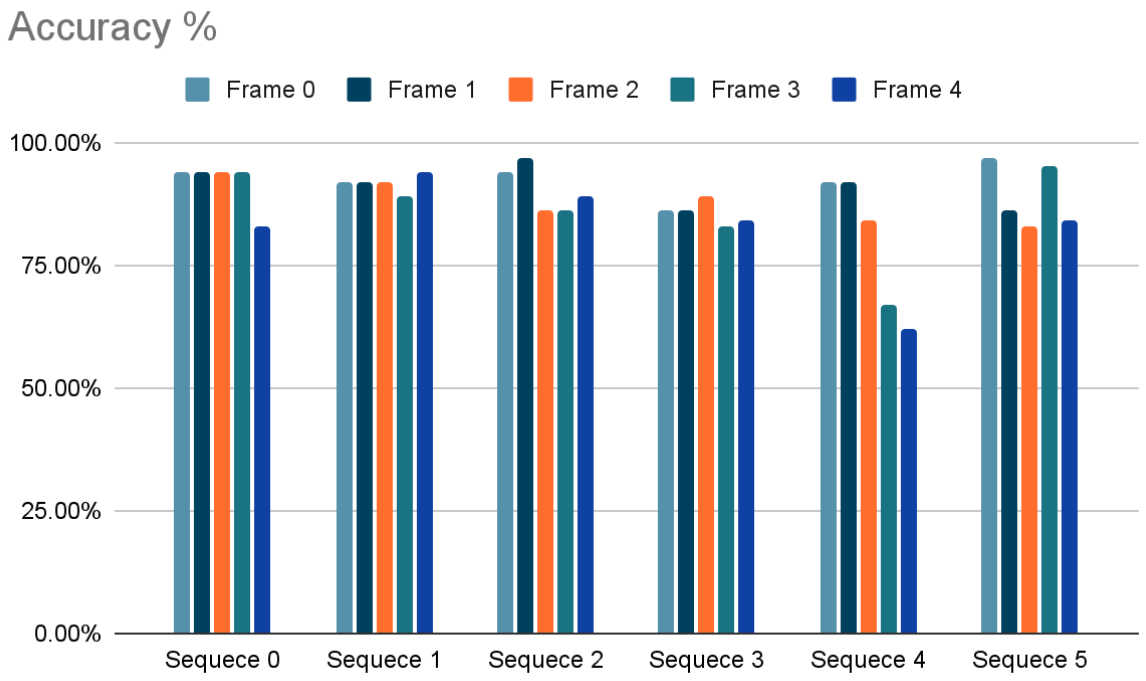


Frame #3



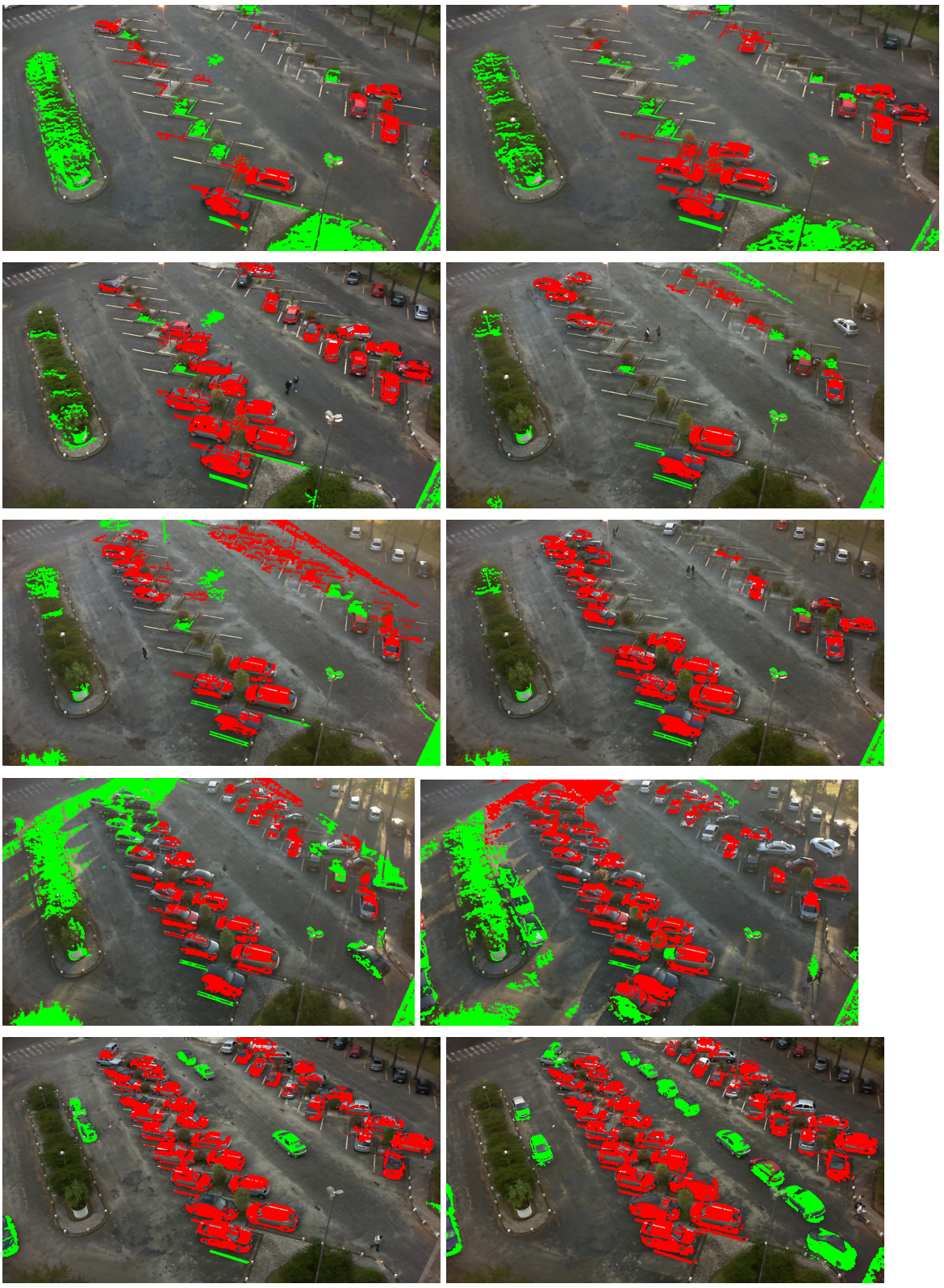
Frame #4

Frame	0	1	2	3	4
Accuracy	0.97	0.86	0.83	0.95	0.84
TP	35	32	31	35	31
FN	0	5	6	2	6
FP	0	0	0	0	0
TN	1	0	0	0	0



Accuracy per Sequence		
Sequence	Average	Total Avg
0	0.918	0.88
1	0.918	
2	0.904	
3	0.856	
4	0.794	
5	0.89	

Segmentation Output





Contributions

Pietro Volpato:

- Implementation of the parking space detector: searching of different approaches, testing and experimenting different techniques
- Implementation of the car segmenter: same as before
- Implementation of the metrics and part of the Loader
- Total hours spent: I do not know an exact amount of hours but at least 100 hours, which were mostly spent searching for different techniques in particular for point 1, which is the most difficult and required a lot of effort; the second point required less effort but the results are poor.

Ali Esmaeili nasab:

- Implementation of the parking space classifier: implementing different approaches and methods and testing them on different images to reach an acceptable result
- Implementing the Visualizer: same, as mentioned in the Visualization
- Part of the Reading the ground truth
- Total hours spent: Considering the time I dedicated to the project, I would estimate around 100 hours. A large portion of this time was spent on the classification process, as finding the right approach through image processing was quite challenging, while the visualization aspect was more straightforward to manage.