Project 1 – Lane Detection

Eleyah Melamed (ID 313905945) & Shaked Oren (ID 316433937)

LINK for all the videos used in the project.

General code architecture:

The code runs as a loop on three videos, one is a driving video at daytime with lane switching, one is a driving video at night with lane switching, and one is for car detection and danger warning.

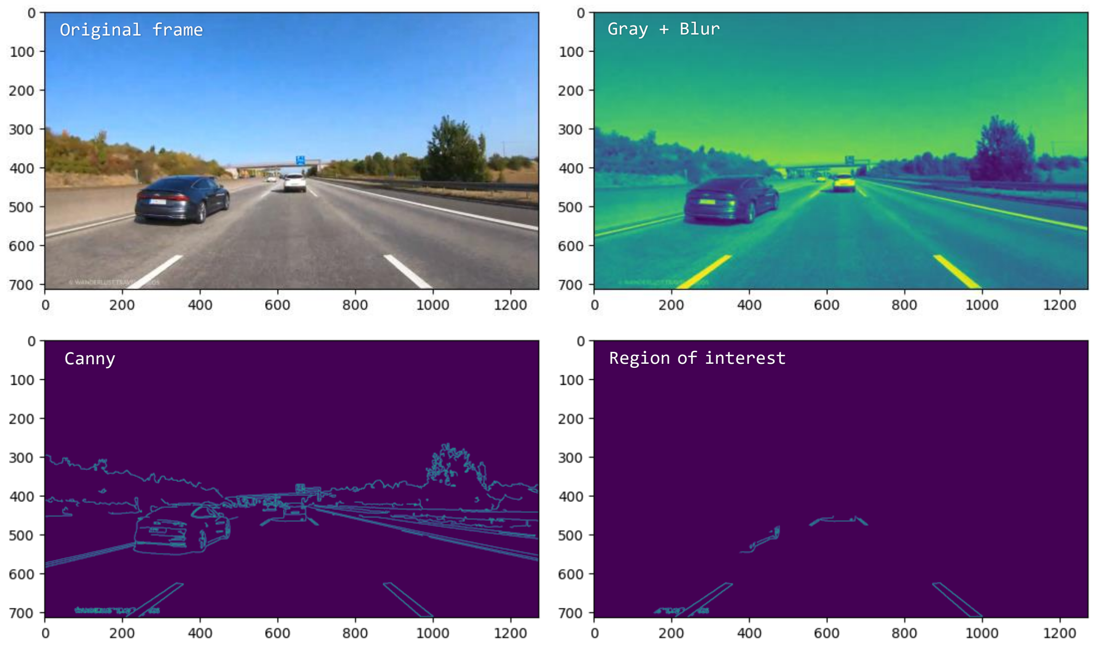
For each video, our code processes each frame, marks the lanes inside the frame, checks for lane switching, and in the relevant video also marks the cars around and checks for dangers. The modified frame is written into the output file.

Lane Detection and lane switching process:

We begin by setting the global variables in the first frame of each new video, these parameters were chosen based on the characteristics of each video.

After that, for each frame of the video, the algorithm calls the function ProcessFrame(). This function processes the frame and returns a new frame with the lanes marked and a lane switching message if necessary. It does this by converting the frame to grayscale, applying Gaussian blur to reduce noise, and then using the Canny edge detector to identify edges. The edges image is sent to the function RegionOfInterest() which returns only the edges that are inside the lane polygon. We set the lane polygon as a global parameter for each of the videos, to be the part of the image where lane markings are expected.

These are the steps in an example frame process:

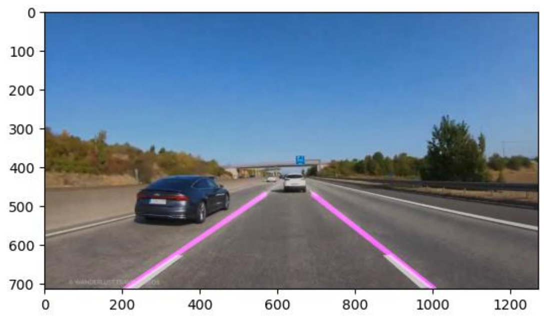


After these image-processing techniques, the Hough Transform is applied to these masked edges to detect linear segments that could represent lane marking. The detected lines are sent to the function DisplayLines(). DisplayLines() function uses the helper function FilterLines(), which goes over all the lines detected and first filters out irrelevant segments based on their slope. Then, it uses BestLeftLine() and BestRightLine() to select at most one line each, that is very likely to represent the left lane and right lane accordingly, based on their length, slope and position in the frame. The function stores their values inside global lane variables and sets counters for them. This is done so later frames that do not have a clear lane marking (no left/right line with high enough confidence to represent lanes was found) can use the previous lanes detected to maintain the consistent output of lanes over the next several frames (limited by their global counter). DisplayLines() then takes the best lines chosen and stretches them using the function CalculateLaneBoundaries() that calculates their linear equation and places the two edge points, according to the length of the lanes in the video (the y axis length of the lanes is also a global parameter defined differently for each of the videos).

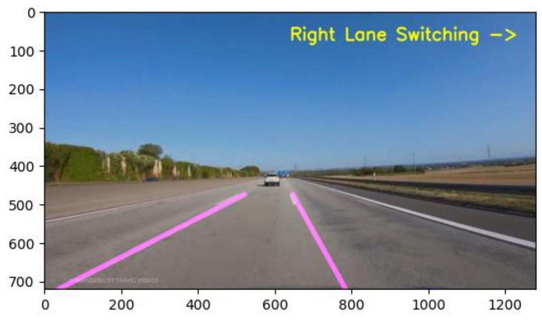
The algorithm then checks if there is a lane switch occurring with the function CheckForLaneSwitching(). It does this by checking if one of the lanes detected is too close to the relative middle of the frame. If a lane change is detected, a global variable is changed according to the direction of the switch. A timer is set for the message so that in the next frames will know how long ago the switching started, to display the message accordingly.

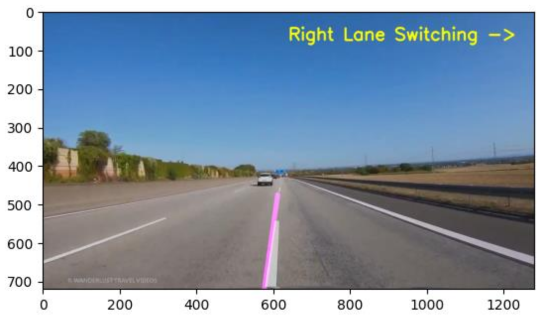
At the end of all these processes, the lanes found are marked on the original video frame and if necessary, a right or left switching message is added to the frame based on the direction of the change.

This is the result frame for an example frame process:



And these are the result frames for frames with a lane switch:





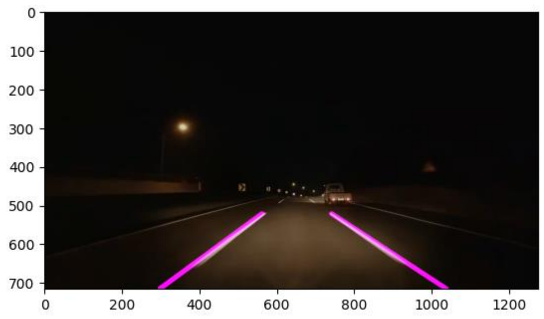
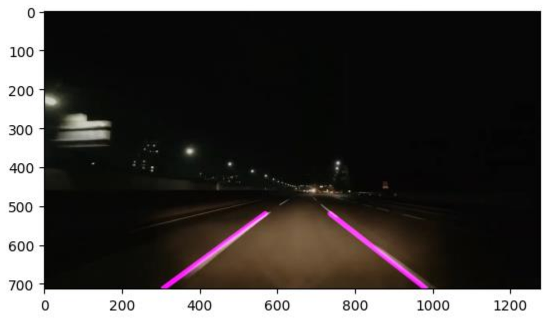
The algorithm operates under assumptions about the geometry of lanes, such as their parallel nature and positioning relative to the vehicle. It also uses specific parameters chosen for each video in advance. For example, in each video the length of the lanes marking on the road and the distance between them is different (we set the global lanes delay counter to fit them accordingly), also the lanes polygon is different for each video (depends on the relative camera's position). Another disadvantage of the algorithm is that we see in the output video a bit of flickering between the inside of the lane line and the outside of the lane line.

The algorithm has been successful in detecting and marking the lanes, even when the markings in the frame are not clearly visible. It accurately identifies the lanes without confusing them with other lines in the frame and is not disrupted when a car passes by. Additionally, it generates a message indicating when the car is switching lanes (left or right) based on the actual direction of the car's movement.

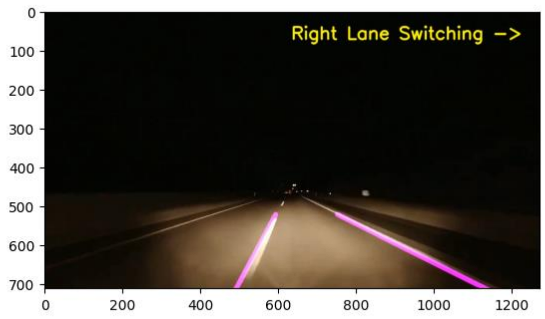
Night-time Lane Detection:

We chose to include in our project night-time lane detection, and a big advantage of our algorithm is that we made sure that aside from changing the parameters for the chosen night video proportions, the same code gets a good output for both lighting conditions.

These are examples of result frames for night conditions:

And this is an example of a result frame during lane switching from our night video:



Proximity-Based Vehicle Detection for Collision Avoidance:

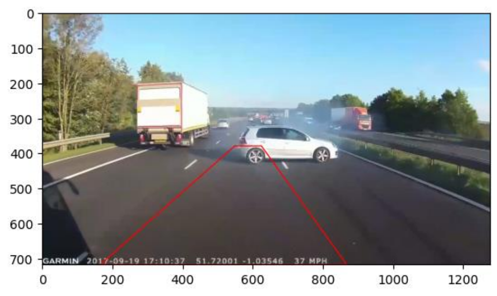
The second enhancement we chose is detecting cars around the vehicle and alerting for too close danger. We used template matching to identify vehicles based on visual similarity and assessed danger risks based on proximity.

We chose a few car templates from the video. These are some examples:

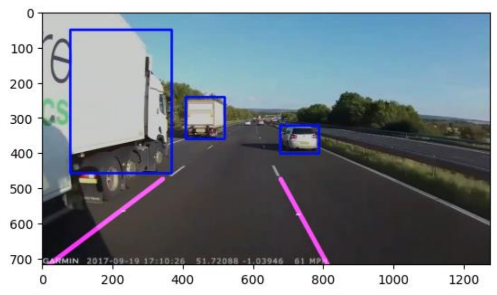
Then in the code, we generated duplicates of the templates in different sizes to match a moving car changing size. When the relevant video is processed, each frame is scanned using the function FindMatches(). The function cv2.matchTemplate() is used with all the templates, and for every match with a value above a predefined threshold, a rectangle around the car found is added to a rectangles list. The function cv2.matchTemplate() slows down the code, especially when used with a lot of templates. We also make sure there are no overlaps between the templates found, using our function called UniqueRectangle().

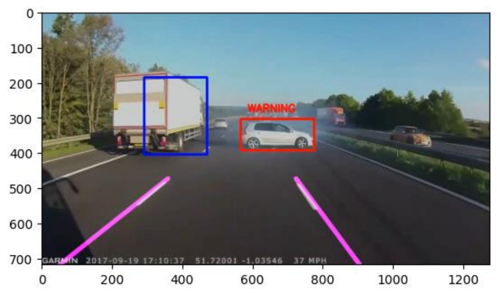
The next step is to draw the rectangle boxes we found onto the frame with the function DrawRectangles(). While drawing the rectangles, each rectangle is checked with the function IsInsidePolygon(). It checks whether one of the rectangle's corners is inside the predefined danger zone for our vehicle. If the car's template rectangle overlaps with the danger zone, it is colored in red and a warning message is added to the frame. Otherwise, the car's marking is in blue.

This is a visualization of the danger zone drawn on a frame from our video:



And these are examples of result frames for the car detection video:





For this part, we used templates of cars taken from the video, so to generalize the code a lot more templates will be needed. Here also we used specific parameters of the danger zone fit for the video we chose. The danger zone needs to be predefined for different videos proportions. Another disadvantage of the car detection algorithm is that it takes a long time to process each frame and it makes the code very slow.

The algorithm successfully detects and marks the lanes, and the cars around the vehicle. It changes the color of the cars marking if they are too close to our vehicle and adds a warning message to alert before a collision.

Inside our zip folder, you can find the Python code, and a Templates folder which contains the templates used for the car detection. The original videos used, and the output videos are found inside a YouTube playlist in this [LINK](#).

All the materials are also available in our GitHub repository in this [LINK](#).