Adam Veraszto, Daniel Balint,
Benedek Kornyei

CodeLikeABosch Software Challange

Project Documentation

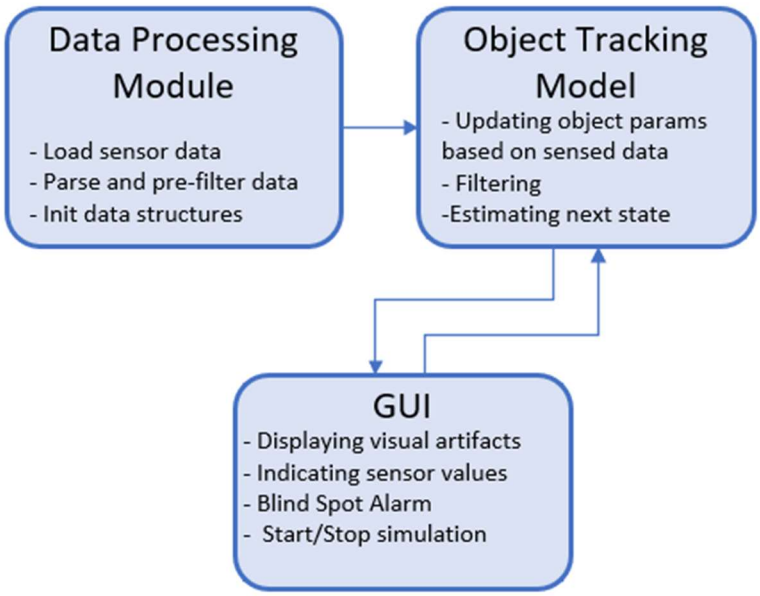
10/02/2022

Budapest

1 Introduction

Our first task was to build up an object tracking system all around the vehicle and show how our solution can track the objects even in case they are in sensor blind spots. We started our work by assessing our input datasets and laying out the mathematical concepts for object tracking. Afterwards, we designed the basic architecture of our software solution. We used Python 3.9 and the PyGame environment to implement our solution, and we created an interactive GUI with control buttons, and built-in video features and value indicators.

2 Software Architecture



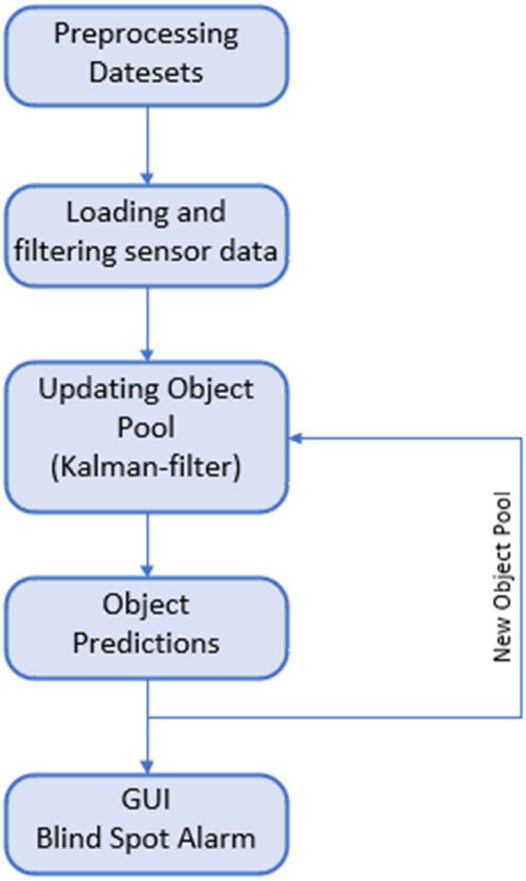
The architecture of our solution can be segmented into three main modules. The Data Processing Module parses the bulk input dataset we recieved containing all the sensor measurments. It sorts data columns by sensors and sensed objects and pre-filters data for the tracking model (sanity checks, normalizations). It also structures the input data into sensor object instances (object-oriented Python classes) that can be easily passed to the Object Tracking Modul.

The Object Tracking Module creates an internal representation of the objects (an object pool) in our vehicle's environment. Based on the recieved sensor data, and through a match-making algorithm between our internal representation and the sensed objects, we update object parameters (position, velocity, acceleration, type) in our internal model and add or remove objects if neccesary. For updating the object pool, we also use several filters, such as a low-pass filter and distance limits, to reduce noise and flickering in our object tracking. With our model, we are able to estimate the location, velocity and acceleration of the detected objects in the next timestamp by applying the basic kinematic equations, and thus, continue tracking objects in blind spots as well.

The Graphical User Interface displays the vehicle and the detected objects in its environment in real time, along with the synchronized real-world video of the front camera that provided the sensor values for the represented test drive. The GUI also provides a blind

spot alarm that activates when our tracking model indicates that there is a vehicle in our abovementioned blind spot.

2.1 Control Flow:



During processing datasets, we structure the sensor data into Vehicle, Camera and Corner Radar Data classes and instantiate these classes for every detected object. Kinematic parameters are normalized, and NONE object types and low-probability obstacles are filtered out.

After preprocessing, and initializing our model, the data structures are passed onto the tracking module, which starts iterating through the sensor data timestamp to timestamp. To update our internal representation every T_k step, we use the sensor data corresponding to that step and our object predictions for the T_k step generated at T_{k-1}. With nested loops around the sensor data and our object pool, we try to match the sensed objects with the objects in our estimated internal representation by optimizing for minimal distance. After finding matches, we update the parameters of the objects in our pool with the corresponding sensor

measurments through a low-pass filter. We also apply a distance and a lifetime limit, meaning objects that haven't been detected for a certain time and predicted to be out of a specified range will be removed from our model.

After calculatin our internal object pool for T_k step, we can predict the model state for T_{k+1} step by applying the basic kinematic equations for the system.

First off, since the vehicle has a yaw rate around the z axis and our coordinate system is fixed to the vehicle, we apply a roational coordinate transformation for our system with an angle of $\varphi = yawrate * delta\_T$ as follows.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

In the rotated coordinate system, we can now predict the next $(x, y)_{k+1}$ positions with the velocities at T_k moment, and the next velocities from the accelerations respectively.

The predicted object pools are then used for the next iteration and fed to the GUI for visual representation.

3 Graphical User Interface

In this section the graphical user interface (GUI) of the project will be presented. On the screen we display the detected objects timestamp to timestamp with color coded dots. Our vehicle is fixed in the centre and the objects are always represented in the vehicle's local coordinate system. Blind spots are indicated by colored rectangles next to the vehicle, and when another detected object enters this area, the corresponding alarm (left, right) will light up at the right botttom corner of the screen.. At the left bottom of the GUI, we can find control buttons and position and velocity indicators for the ADMA tracked vehicle and its corresponding estimations. These ADMA values serve as a validation method for our tracking system.

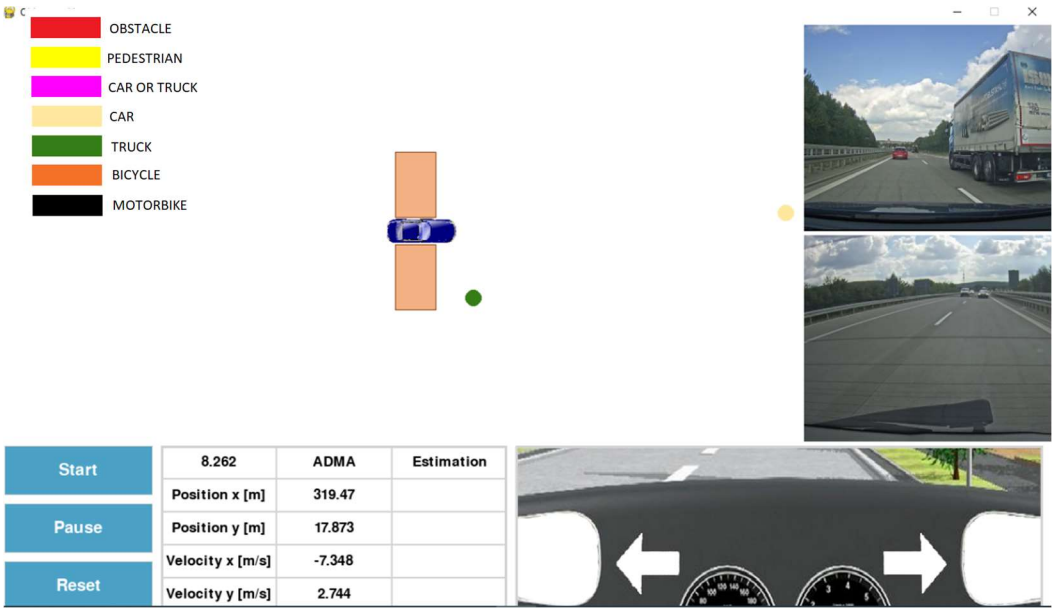


Figure 4: Screenshot of the graphical user interface

On the right side of the display, the recorded video of the test drive is played concurrently with the representation of our tracking system.

4 Install

The application was written in Python language. It was tested with Python 3.9. The dependencies of the project can be found at the „requirements.txt” file of the project and they can be installed with the „pip install requirements.txt” command. To load the appropriate datasets, the csv files and corresponding videos should be placed in the source folder in a dataset folder (similiarly to the provided test data). The exact relative path shall be defined at the variable DATASET_RELATIVE_PATH in *main.py* line 5.