**Lidar Object Tracking**

*This Project is a continuation of Gayan's work. Here is his Github repo:* [*https://github.com/gayan68/LiDAR\_Tracking/tree/main*](https://github.com/gayan68/LiDAR_Tracking/tree/main)*.*

**Project Objective:** The purpose was to perform and explore methods of object tracking in extreme weather conditions for the ROADVIEW project, this was done by further developing the existing tracking algorithm created by Gayan, for example by using an extended Kalman filter instead of a regular Kalman filter. The goal was to get as good quantitative and visual results as possible.

**Introduction/Overlook:**

The object tracking has been performed on data from the CADC dataset. This project has not used the objects from Idriss’ object detection algorithm. Instead, the true boxes from the CADC dataset have been used, this data has an ID for objects in all 100 frames which has been used as a ground truth for evaluating the tracking quantitatively. Kalman Filtering and Extended Kalman filtering have been used for tracking. Results were visually assessed by plotting the object bounding boxes and quantitatively by calculating HOTA and IDF1 scores.

The open\_3D visualizations used by Gayan have been removed entirely since getting this to work was a big issue (may have been an OS issue since Gayan used Linux and this project was done on Windows).

**Project Structure**

The project exists in a folder called LiDAR\_Tracking. This folder contains the files with the algorithms:

* *object\_traking.py*: Gayan’s file
* *object\_traking\_modified.py*: the new modified algorithm created in this project based on Gayan’s file.
* *object\_tracking\_EKF.py:* Algorithm that is based on object\_tracking\_modified.py but uses an extended Kalman filter instead of a regular Kalman Filter.

The LiDAR\_Tracking folder also contains other files and subfolders that are not a part of the algorithm itself but used in other purposes:

* *cadc\_devkit:* A folder containing the dataset and the provided files from the CADC dataset to handle the data. The important files and subfolders in this folder are:
  + *run\_demo\_lidar\_bev2.*py: BEV visualization of point cloud and objects. More about the visualization further down in the document.
  + *dataset:* Folder containing the lidar point data, true bounding boxes, and also images but the images are not used in this project.
  + *Images:* Folder containing the images that are created in run\_demo\_lidar\_bev2.py and are used to create the gif output3.gif.
* *3d\_ann.*json: The output json file created in the algorithms and then visualized in run\_demo\_lidar\_bev2.py file.
* *3d\_ann\_objects\_outside\_50m.*json: output of remove\_boxes\_outside\_50m.py that contains the true bounding boxes within 50m of the ego car. These are the objects that are tracked in the created algorithms.
* *EKF\_modified.*py: the modified EKF class from filterpy. More about what is modified in the section about the EKF algorithm.
* *box\_list.pkl:* bounding boxes from Idriss’ object detection algorithm, has similar format as the json files and can be used as input.
* *convert\_novatel\_to\_pose.*py: A try to convert the Novatel data from the dataset to pose matrices in the ENU format. This was created when discussing if adapting the tracking to the speed of the ego car would improve tracking accuracy. One way to calculate this would be to take the difference in positions of the ego car which could be extracted from the novatel data. This process was never finished.
* *evaluation.*py: Code for evaluating the tracking result using HOTA and IDF1 matrices.
* *load\_novatel\_data.*py: function provided by cadc to load novatel data (GPS data)
* *output3.*gif: the output gif of the run\_demo\_lidar\_bev2.py file
* *remove\_boxes\_outside\_50m.py:* takes the true bounding boxes from the CADC dataset and creates the file ‘3d\_ann\_objects\_within\_50m.json’ that has all true boxes inside 50 meters from the car along x-axis and y-axis.
* *requirements.txt:* packages needed to run the algorithms
* *settings.*yaml: setting file that contains the thresholds for the algorithms.

**How to Run the Code**

Download the essential packages stated in the requirements file. You can for example do this by using the following commands:

pip install -r requirements.txt

pip install json

pip install re

When having installed the packages, run one of the two algorithm files created in this project: *objekt\_traking\_modified.py* or *obj\_tracking\_EKF.py.* This will run the object tracking and create the output file 3d\_ann.json which is stored in the same directory as the algorithm file. Note that you must change the path in the “get\_true\_boxes” function, to your own. You can either use the file 3d\_ann\_within\_50m.py (stored in the same directory as the algorithm file) as input or the .json file with all true boxes (stored in a subfolder of the cadc\_devkit folder). If you want to perform tracking for the detected objects (box\_list.pkl file), an example of using this as input is in the “load\_bounding\_boxes” function in the *object\_traking.py* file.

When the algorithm is finished (should not take more than 5 seconds), you can either evaluate the results by running *evaluation.py (*takes ~1sec and prints IDF1 and HOTA scores), or by running *run\_demo\_lidar\_bev2.*py (takes ~2min and creates output3.gif which gets stored in the same directory as the algorithm files and visualizes the result).

**Algorithm**

The code first loads the input data using the *get\_true\_boxes* function and the thresholds from the settings.yaml file. Then, an instance of the bb\_traking (holds most functions for the algorithm) class is created which sets the settings from the yaml file. After that, the run\_algorithm function is called, the following explains the algorithm:

* Iterate through frames
  + Set all tracks as unmapped (not used)
  + Iterate through boxes
    - For first frame: create new track for each box
    - For other frames: iterate through tracks
      * check for best match between current box and current track based on distance, velocity and yaw
    - If no match is found, add box to new\_tracks
    - If match is found and track is not mapped or if the
      * Set flag to say that the track is mapped
      * Perform KF or EKF update step for best match track
      * Perform KF or EKF predict step for best match track
      * Set individual distance threshold for best match track based on difference between position and predicted position
      * Create bounding box to be stored in the output json file
    - If it’s not the first frame: count down and remove untracked objects if count is below 0.
    - Create new tracks for all added boxes in new\_tracks
  + Append all created bounding boxes for the current frame
* Return created bounding boxes for all frames

After having run the algorithm, the difficulty scores of each track are calculated (see more about this in the Difficulty Score section). Then, the result is converted to a json file and the frames per second rate for the algorithm is calculated.

**How the Kalman Filter Works**

In this project, the Kalman filter from the filterpy.kalman library is utilized for tracking objects in 3D space using a constant velocity model. The filter operates on a state vector that includes positions (x, y, z) and velocities (vx, vy, vz) of an object.

1. **Initialization**: The BoxTracker class initializes the Kalman Filter with a 6-dimensional state (position and velocity) and a 3-dimensional measurement (position only). The initial position is set from the input bbox\_3d.
2. **State Transition Matrix (F)**: This matrix defines how the state evolves from one time step to the next without control input. For the constant velocity model, it assumes that position is updated by velocity multiplied by the time step (dt) (see state transition matrix in the code).
3. **Measurement Matrix (H)**: This matrix maps the state vector to the measurement vector. It extracts the position from the state vector, ignoring velocities, indicating that only positional data is directly measured.
4. **Covariance Matrices (P, Q, R)**:
   * P is the initial estimate of the state covariance, set to a large value (10), representing initial uncertainty.
   * Q is the process noise covariance matrix, indicating the expected variance in the process (state evolution), set to 0.1, assuming minor prediction errors.
   * R is the measurement noise covariance matrix, set to 0.1, indicating confidence in the position measurements.
5. **Prediction and Update**:
   * The predict() method of the Kalman Filter projects the state forward to the next time step using F and adjusts the state covariance.
   * The update() method incorporates new measurement data (bbox\_3d for position), updating the state estimate and reducing uncertainty using the Kalman Gain computed from P, H, and R.

This setup enables the BoxTracker to estimate and robustly track the position and velocity of objects in 3D space, even in the presence of measurement and prediction uncertainties, using a linear motion model. Each object has a tracker which is an instance of the BoxTracker, in this BoxTracker, the Kalman Filter is stored.

**How the Extended Kalman Filter (CTRV model) Works**

The only difference between the files with Kalman Filter (KF) and Extended Kalman Filter (EKF) is the BoxTracker class which holds the KF/EKF. To implement the EKF the extended Kalman filter library from filterpy was used. The process of updating and predicting are similar between KF and EKF, but EKF is more complicated.

This model uses a constant turn rate and velocity (CTRV) model, implies that the state has five parameters: [x\_pos, y\_pos, velocity, heading, turn rate]. If the model has above a turn rate threshold (set to 0.1), it is considered turning and then a circular motion model is applied, if the turn rate is below that threshold, the object is considered going straight and a linear motion model is applied.

The EKF uses non-linear state transition and measurement functions, which are linearized using Jacobian matrices to update covariance estimates. Each element in the jacobian matrix is extracted as the partial derivative of each objects state transition with respect to each of the five parameters, therefore it becomes a 5x5 matrix, see the state transition function and the jacobian matrix in the code for details.

The ekf.predict step in filterpy was modified, the modified file is EKF\_modified.py. In the original file, the predict step was done with the calculated jacobian of the state transition matrix. Now, the predict step is done with the state transition function and only the uncertainty matrix update in the predict function is done with the jacobian. The model did not seem to work without this change.

**Visualization**

Visualization is done with gifs which are generated by running the *run\_demo\_lidar\_bev2.*py file after having run one of the algorithm files. This is a modified file, it was originally provided with the CADC dataset.

Some features are added to the visualization. For example, by changing row 116 so that cuboid[‘stationary’] = False, you can choose to plot the last 12 positions of the moving objects as a line that starts at the current position. This looks good when using the tracking ground truth because this data has a ground truth for the stationary parameter. However, since the ego-vehicle moves at different speeds and the tracking algorithm does not take ego-velocity into account, it is difficult to set the stationary parameter to each object. Right now, the stationary parameter is set based on velocities observed from the visualizations: stationary objects move at ~3.5 m/s through the frame, objects moving in the same direction move at ~0 m/s through the frame, and objects moving in the opposite direction move at ~7m/s through the frame. But, since the ego velocity changes, these velocities change, which makes it difficult to set the stationary parameter. Right now it looks ok but not good when plotting the past positions on the output of the algorithm.

By uncommenting row 85 you can also plot only one uuid, when doing this you have to find the wanted uuid in the json file used for visualization.

The code now also visualizes how difficult it is to track each object. This is calculated by adding an occupancy score (how many objects are nearby) and the objects’ velocity. More about this in the “Difficulty score” section.

It is also possible to choose the number of frames to plot on row 210 in *run\_demo\_lidar\_bev2.*py.

The duration of the output gif can be changed with the duration parameter in *run\_demo\_lidar\_bev2.*py.

**Difficulty Score**

A difficulty score was calculated for each object in each frame, based on the velocity of the object and occupancy (how many other objects are close by). This was done in the ‘*add\_difficulty\_score’* function in the algorithm files. The occupancy score formula was set to:

)

Where distance is the distance between the object and each neighbouring object. The formula can be changed if needed.

The velocity score for each object in each frame was set to its velocity. Adding the velocity score and occupancy score made up the total difficulty score for that object.

In *run\_demo\_lidar\_bev2.*py the difficulty was printed as easy if the difficulty score was less than 7, medium if it was between 7 and 15, and hard if it was above 15.

**Quantitative Score Calculation**

The quantitative score calculation of HOTA and IDF1 is possible because the UUID of each object and each frame is in both the output json file of the algorithms and the ground truth json file. HOTA combines detection and association accuracies, while IDF1 only considers correct identity matches. The metrics were taken from this <paper>. Metrics that were only based on detection accuracy were not used since the importance lied in association accuracy for this project.

The HOTA score is calculated with the following formula:

​

Where,

​

Where, TP = True Positives: Correctly matched detections, FP = False Positives: Detections with no corresponding ground truth, FN = False Negatives: Ground truth objects not detected. See the definitions of AssA parameters under the IDF1 score. The HOTA score is calculated for 10 different thresholds linearly spaced from 0.1 to 1 meter, saying if the detections are considered correct or not, and then taking the average of those scores to calculate the final HOTA score.

The IDF1 score is calculated with the following formula:

Where, IDTP = Identity True Positives: Correctly matched identities across frames, IDFP = Identity False Positives: Mismatches where a detection is assigned a wrong identity, IDFN = Identity False Negatives: Ground truth identities not matched in detections.

**General Changes of the Algorithm**

-Corrected velocity error

-Moved the creation of new tracks outside the object loop because if this is done when looping through objects, two new objects that are close would match with each other's tracks instead of creating to new tracks.

-Changed parameters in the settings file.

-Counted how many times each uuid occured across the frames and found that there is a tracking ground truth in the dataset.

-Now, each object has its own distance threshold based on how fast the object is moving between frames. This won’t have to be adaptive once the algorithm is adapted to the ego-movement.

-A weighted score for matching was implemented based on the scale of the distande\_diff, vel\_diff and yaw\_diff values, and which one is most important (distance most important).

-Yaw is taken into account when matching. First, it was calculated in a way that generated many errors when yaw went from for example 3.12 to -3.12. When changing this, the number of faulty generations of new tracks was reduced from 12 to 1 for the traffic in the opposite direction. See how it is calculated on row 206 in the *obj\_tracking\_EKF.py* file.

-A file was created that generate a new json with all objects inside 50m in x and y.

- The algorithm was earlier comparing and calculating the velocity between the predicted pos and measurement, which should be between the previous pos and measurement. This was corrected and the score went from 94 to 95 for idf1 and Hota.

**Results**

The final quantitative result was 95.4 for HOTA and 95.3 for IDF1 for the regular Kalman Filter, compared to 97.4 (HOTA) and 97.3 (IDF1) for the Extended Kalman Filter. The HOTA score was consistently slightly higher which is logical since it takes the positions of the tracks into account. Since the algorithm is based on the true bounding box positions from the dataset this score should be slightly higher than the IDF1 score.

For visual results see the two GIF files represented in the git repo.

**Suggestions for Further Development**

* Make it device-independent (cpu/gpu)
* Adapt it to ego-velocity because the Kalman filter and extended Kalman filter are more robust the slower the vehicles move. In real life, this can be done by using the real-time velocity data from the ECU of the car.
* A way to remove ghost objects: don’t create new bounding boxes until the tracker is matched once.
* Convert to torch.
* Originally, the tracker was only updated if the track had not been mapped or if it had been mapped and this match was even better than the last match, for some reason the result got better if the tracker was always updated, the only condition being that it found a best match. Maybe look into this?

add read me and gifs in the git repo.