**API List & How to use**

**Agent is the basic class of the vehicle in the simulator.** You may not use it directly.

Constructor:

Agent(int id, std::vector<double> initialState);

Setter:

void setfollowingPlanner(Planner \*p); // set the planner for normal driving

void setlinechangePlanner(Planner \*p); // set the planner for line change action

void setController(Controller \*c); // set the controller

void setModel(Model \*m); // set the model this agent use

void setBehaviour(Behaviour \*b); // set the Behavior model this agent use

void setMapinfo(MapInfo \*m); // set the MapInfo which is the reference manager

Getter:

int getId() // get the id of this agent

std::vector<double> getState() // get the current state

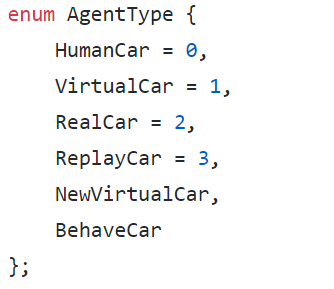
std::vector< std::vector<double>> get\_preState() // get the historical states

AgentType getType() // get the type of this agent

main:

void Run(); // run this car at this time step

Data structure:



BehaveCar **is the class of the ROBOT vehicle in the simulator.**

Constructor:

BehaveCar(int id, std::vector<double> initialState);

Special variable:

bool IDM\_; // if false, the agent will use planner it set to do the planning. If true, it will just use IDM model.

ReplayAgent **is the class of the DATA vehicle in the simulator.**

Constructor:

ReplayAgent(int id, Vector initialState);

Setter: (MUST set Trajectory before update)

void setTrajectory(Trajectory traj)

**Behaviour is the basic class of the behavior model in the simulator.** You may not use it directly.

Constructor:

Behaviour (Agent\* agent\_ibt, BehaviourType t); // ibt means **i**t **b**elongs **t**o

Getter:

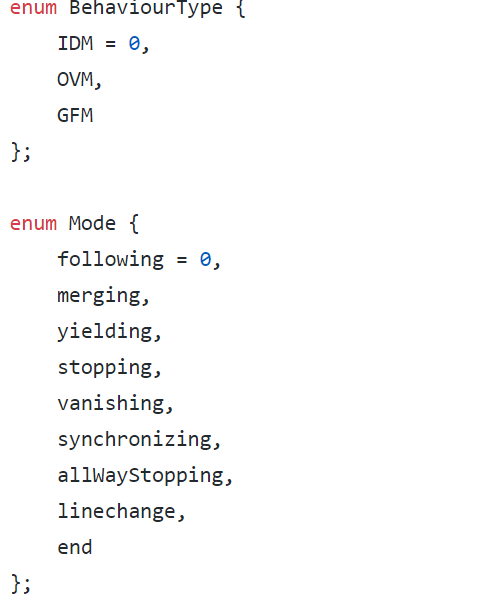
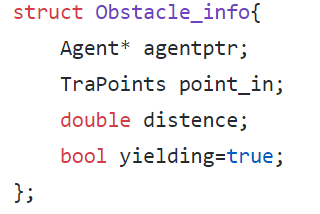
BehaviourType getType() // get the type of naïve action algorithm this behavior use

Mode getMode() // get the current mode of this Planner

main:

std::vector<double> update(std::vector<double> currentState, const std::vector<double> &humanInput, std::vector<Agent\*> agents); // run this behavior at this time step

Data structure:

AoBehaviour **is the class of the rule based behavior model in the simulator.**

Constructor:

AoBehaviour(Agent\* agent\_ibt, BehaviourType t);

**Planner is the basic class of the planner in the simulator.** You may not use it directly.

Constructor:

Planner(Agent\* agent\_ibt, int dimState, int dimInput, MapInfo \*map = nullptr);

Getter:

PlannerType getType() // get the type of this Planner

Setter:

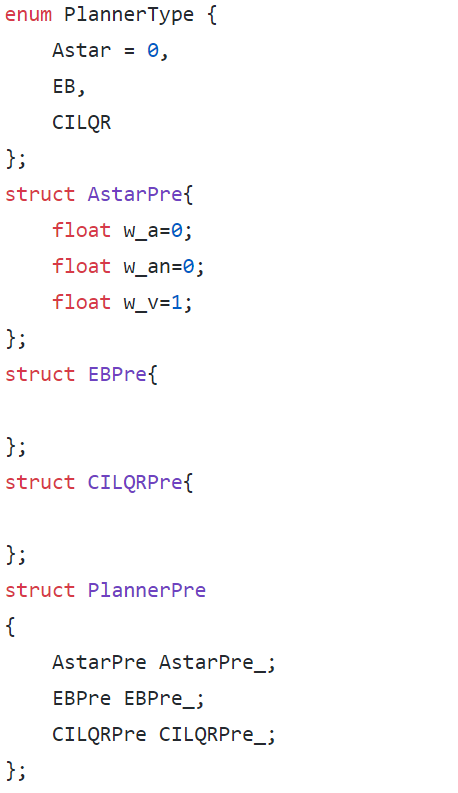
void updatepre(PlannerPre& new\_pre); // update the parameter of this planner.

[not implement now and will fix the typo(updatepre->updatepara; PlannerPre->PlannerPara) in the future]

main:

std::vector<double> update(std::vector<double> currentState, const std::vector<double> &humanInput, std::vector<Agent\*> agents, std::vector<Obstacle\_info> obstacle\_info); // run this planner at this time step

Data structure:

****

AstarPlanner **is the class of the A\* search based planner in the simulator.**

Constructor:

AstarPlanner(Agent\* agent\_ibt, MapInfo\* map);

**Predictor is the basic class of the Predictor in the simulator.** You may not use it directly.

Constructor:

Predictor (Agent\* agent\_ibt, double time\_step, double horizon);

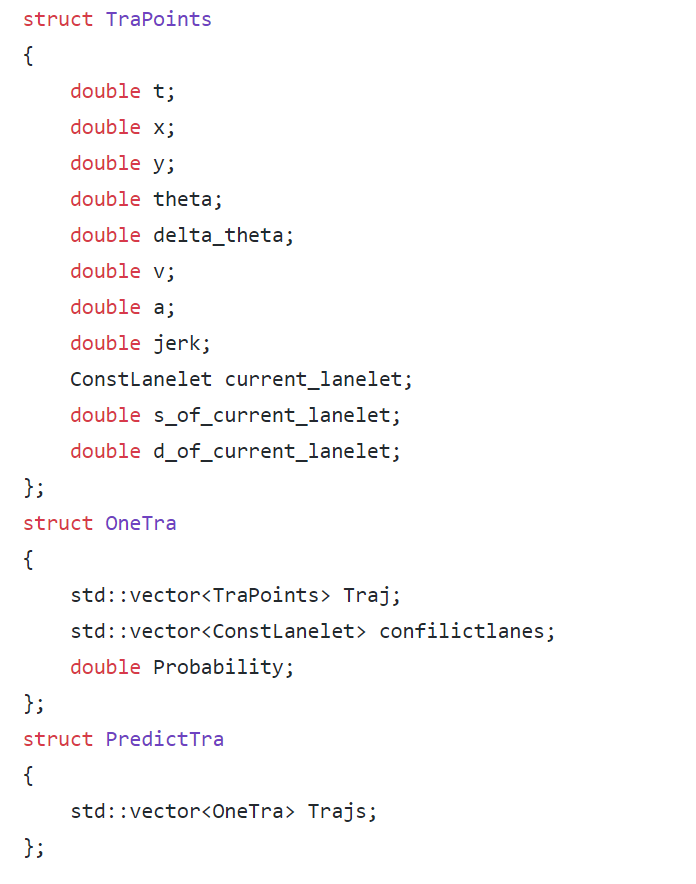
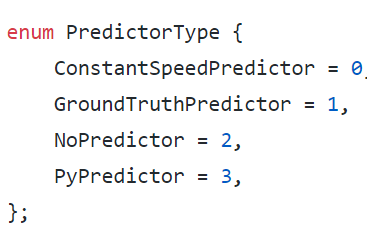
Getter:

PredictorType getType() // get the type of this Predictor

main:

PredictTra update(std::vector<double> currentState, std::vector<Agent\*> agents); // run this Predictor at this time step

Data structure:

GroundTruthPredictor **is the class of predictor that will give you the ground truth of the vehicle directly.**

Constructor:

GroundTruthPredictor (Agent\* agent\_ibt, double time\_step, double horizon);

ConstantSpeedPredictor **is the class of predictor that use constant velocity to predict vehicle stats.**

Constructor:

ConstantSpeedPredictor (Agent\* agent\_ibt, double time\_step, double horizon);

PyPredictor **is the class of predictor that use a external python program to do the prediction through GRPC.**

Constructor:

PyPredictor (Agent\* agent\_ibt, double time\_step, double horizon);

**Mapinfo is the class containing the map and reference information**

**Constructor:**

MapInfo(LaneletMapPtr& mapPtr, routing::RoutingGraphPtr& rgPtr);

**Setter:**

void init(int id, Vector initstate); // set a init states of the agent you may use it after get the routing path

bool setRoutingPath(ConstLanelet& startLanelet, ConstLanelet& destinationLanelet); // get the routing path from lanelet

void setLaneletPath(ConstLanelets& lanelet\_path); // set the routing path by hand

**Getter:**

ConstLanelet getCurrentLanelet() { return currentLanelet\_;}; // get the current lanelet where the agent in.

double getS() { return s\_;} //get the longitude position of the agent in current lanelet.

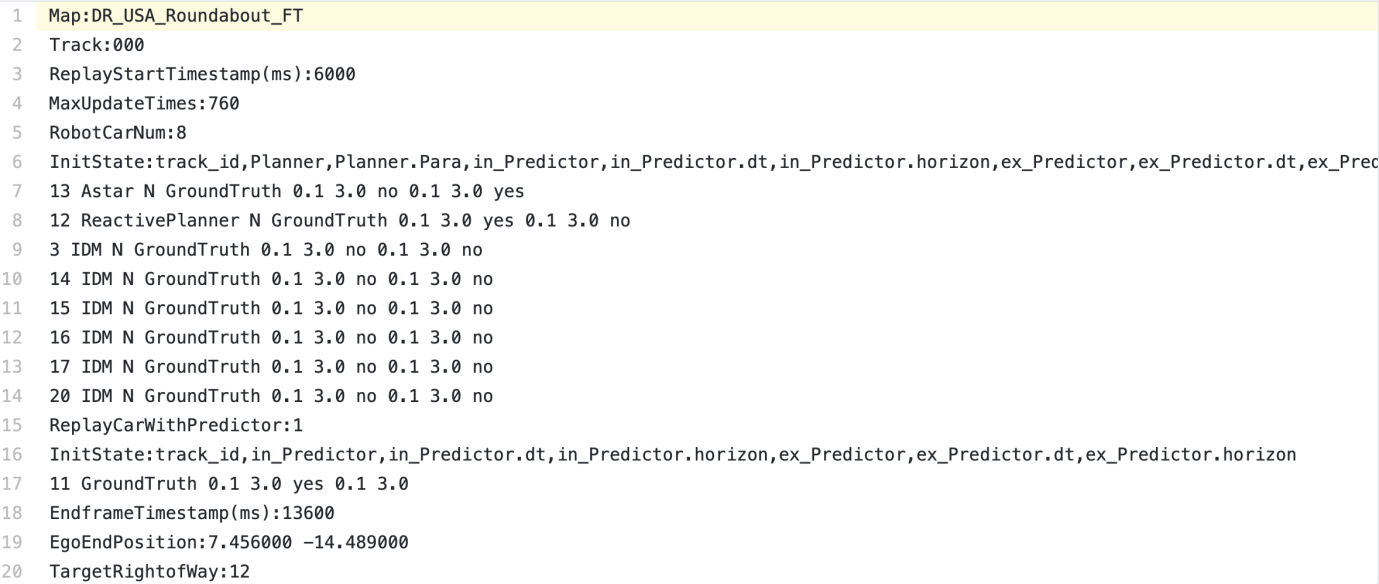
**Main:**

void update(Vector nextstate); //use the next state to update

**How to initialize the simulator**

**-** Write a configuration file, the simulator will generate behavior cars (the cars controlled by the algorithm) and replay cars (the cars following the routing in the .csv) according to that config file.

**A sample config([anyname].txt)**

****

**Explanation**

**- Line 1 specifies the map**

**- Line 2 specifies the dataset, e.g. DR\_USA\_Roundabout\_FT/000.csv**

**- Line 3 & 4 specify the start time of the csv and the update times (each update takes 10ms)**

**- Line 5 specify the number of robot car, namely the behavior car**

**- Line 6 - 14 specify each behavior car (id, planner, internal\_predictor, external\_predictor)**

**- Line 15 specify the number of replay cars which have a predictor (the remaining replay cars don’t have predictors)**

**- Line 16 - 17 specifies the replay car (id, internal\_predictor, external\_predictor)**

**- Line 18 End frame which satisfies EndframeTimestamp\_ms = ReplayStartTimesampe\_ms + Updatetimes \* 10**

**- Line 19 - 20 nevermind. This is used for calculating metrics.**

**If you want to generate cars after the initializing config**

**- we have two functions in the simulator:**

**void generateReplayCar(ReplayCarInfo replay\_info);**

**void generateBehaveCar(ReplayCarInfo behave\_info);**

**- typedef std::tuple<int, int, int, std::string> ReplayCarInfo; // (track\_id, start\_ms, end\_ms, others)**

**- `others` is a string like `Astar N GroundTruth 0.1 3.0 no 0.1 3.0 yes`**

**If you want to add your new DIY planner or predictor**

1. **specify the planner or predictor type in the config file**
2. **Add some lines in the generateReplayCar or generateBehaveCar function. E.g. `if planner\_type == ‘....’`**

****

**Simulator is the class doing the main loop**

**Constructor**

Simulator(int rviz\_port); // You can set rviz point for real time visualization, or set rviz\_port == -1 to avoid that visualization

**Main Body**

void InitSimulation(std::string scenario\_id, std::string Config\_Path, std::string log\_folder, const bool verbose); //scenario\_id is used to name the log file. Config\_path specified the config file. log\_folder is the path storing the log. Verbose==True means we output the details during the simulation.

void run(); // start simulation

**Setter**

void generateReplayCar(ReplayCarInfo replay\_info);

void generateBehaveCar(ReplayCarInfo behave\_info);

bool removeAgentIfNeeded(); // remove the cars that have arrived at the terminal point.

void Agentmanager(); // generate cars at each timestep

void isThereCollision(); // check whether there is any collision

void LogTick(); // output to log file

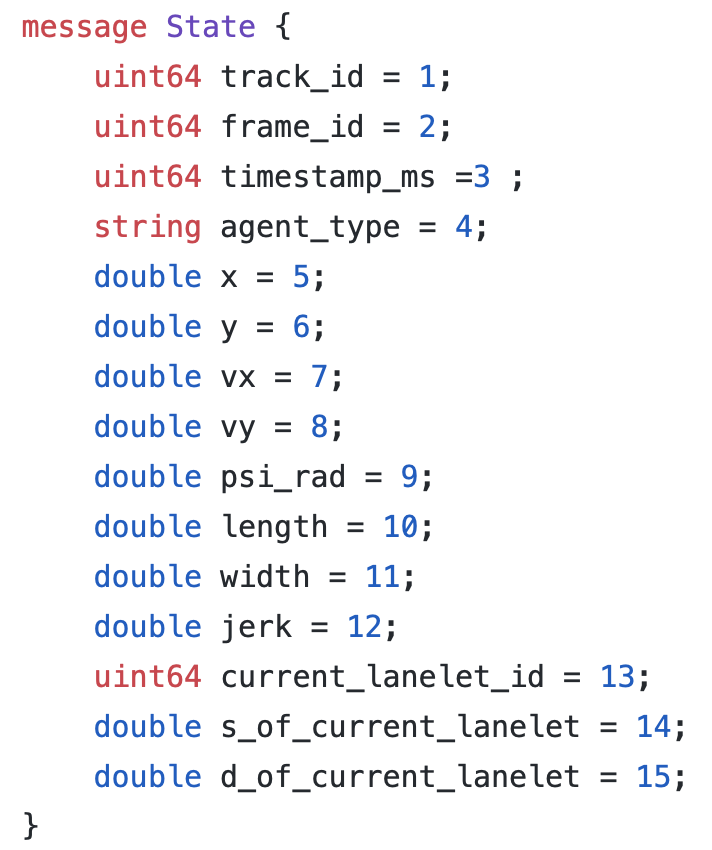
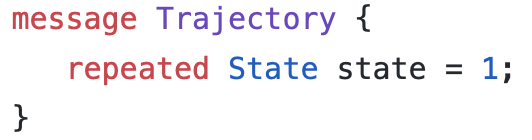
void updateTick(); // update the states for all the cars

**For gRPC**

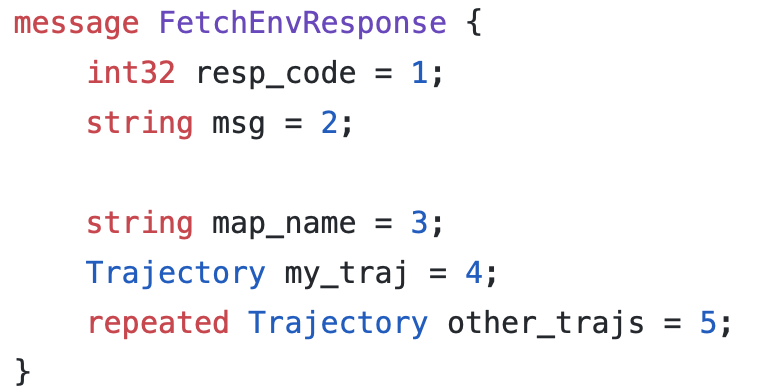
core::SimulationEnv fetch\_history(); // find a car that is waiting for python predictor.

void upload\_traj(int car\_id, std::vector<core::Trajectory> pred\_trajs, std::vector<double> probability); // upload the results from the python side.

**What information needed to be transmitted between the simulator and the predictor**

**fetch\_history will return:**

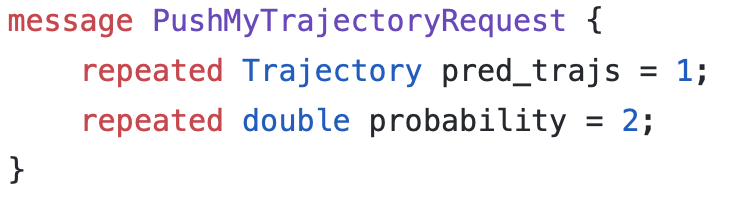


**- map\_name: string, like FT, OF, MA**

**- my\_traj: the historical trajectory of the ego car**

**- other\_traj: the historical trajectories of the surrounding cars.**

**upload\_traj will return:**



**- pred\_trajs: each predicted trajectories in the future**

**- probability: the probability for each trajectory. Sum(probability) = 1.0**

**An example to build a simulator class**

**1. simulator(r\_viz) //constructor**

**2.** simulator.InitSimulation(scenario.id, config\_file, log\_folder, verbose);

**3.** Simulator::run()