**from** typing **import** Tuple, Union  
  
**import** numpy **as** np  
  
**from** highway\_env.road.road **import** Road, Route, LaneIndex  
**from** highway\_env.types **import** Vector  
**from** highway\_env.vehicle.controller **import** ControlledVehicle  
**from** highway\_env **import** utils  
**from** highway\_env.vehicle.kinematics **import** Vehicle  
**from** highway\_env.road.objects **import** RoadObject  
  
  
**class** IDMVehicle(ControlledVehicle):  
 *"""  
 A vehicle using both a longitudinal and a lateral decision policies.  
  
 - Longitudinal: the IDM model computes an acceleration given the preceding vehicle's distance and speed.  
 - Lateral: the MOBIL model decides when to change lane by maximizing the acceleration of nearby vehicles.  
 """  
  
 # Longitudinal policy parameters* ACC\_MAX = 6.0 *# [m/s2]* **"""Maximum acceleration."""** COMFORT\_ACC\_MAX = 3.0 *# [m/s2]* **"""Desired maximum acceleration."""** COMFORT\_ACC\_MIN = -7.0 *# [m/s2]* **"""Desired maximum deceleration."""** DISTANCE\_WANTED = 2.0 + ControlledVehicle.LENGTH *# [m]* **"""Desired jam distance to the front vehicle."""** TIME\_WANTED = 1.0 *# [s]* **"""Desired time gap to the front vehicle."""** DELTA = 4.0 *# []* **"""Exponent of the velocity term."""** *# Lateral policy parameters* POLITENESS = 0. *# in [0, 1]* LANE\_CHANGE\_MIN\_ACC\_GAIN = 0.2 *# [m/s2]* LANE\_CHANGE\_MAX\_BRAKING\_IMPOSED = 2.0 *# [m/s2]* LANE\_CHANGE\_DELAY = 1.0 *# [s]* **def** \_\_init\_\_(self,  
 road: Road,  
 position: Vector,  
 heading: float = 0,  
 speed: float = 0,  
 target\_lane\_index: int = **None**,  
 target\_speed: float = **None**,  
 route: Route = **None**,  
 enable\_lane\_change: bool = **True**,  
 timer: float = **None**):  
 super().\_\_init\_\_(road, position, heading, speed, target\_lane\_index, target\_speed, route)  
 self.enable\_lane\_change = enable\_lane\_change  
 self.timer = timer **or** (np.sum(self.position)\*np.pi) % self.LANE\_CHANGE\_DELAY  
  
 **def** randomize\_behavior(self):  
 **pass** @classmethod  
 **def** create\_from(cls, vehicle: ControlledVehicle) -> **"IDMVehicle"**:  
 *"""  
 Create a new vehicle from an existing one.  
  
 The vehicle dynamics and target dynamics are copied, other properties are default.  
  
 :param vehicle: a vehicle  
 :return: a new vehicle at the same dynamical state  
 """* v = cls(vehicle.road, vehicle.position, heading=vehicle.heading, speed=vehicle.speed,  
 target\_lane\_index=vehicle.target\_lane\_index, target\_speed=vehicle.target\_speed,  
 route=vehicle.route, timer=getattr(vehicle, **'timer'**, **None**))  
 **return** v  
  
 **def** act(self, action: Union[dict, str] = **None**):  
 *"""  
 Execute an action.  
  
 For now, no action is supported because the vehicle takes all decisions  
 of acceleration and lane changes on its own, based on the IDM and MOBIL models.  
  
 :param action: the action  
 """* **if** self.crashed:  
 **return** action = {}  
 front\_vehicle, rear\_vehicle = self.road.neighbour\_vehicles(self)  
 *# Lateral: MOBIL* self.follow\_road()  
 **if** self.enable\_lane\_change:  
 self.change\_lane\_policy()  
 action[**'steering'**] = self.steering\_control(self.target\_lane\_index)  
 action[**'steering'**] = np.clip(action[**'steering'**], -self.MAX\_STEERING\_ANGLE, self.MAX\_STEERING\_ANGLE)  
  
 *# Longitudinal: IDM* action[**'acceleration'**] = self.acceleration(ego\_vehicle=self,  
 front\_vehicle=front\_vehicle,  
 rear\_vehicle=rear\_vehicle)  
 *# action['acceleration'] = self.recover\_from\_stop(action['acceleration'])* action[**'acceleration'**] = np.clip(action[**'acceleration'**], -self.ACC\_MAX, self.ACC\_MAX)  
 Vehicle.act(self, action) *# Skip ControlledVehicle.act(), or the command will be overriden.* **def** step(self, dt: float):  
 *"""  
 Step the simulation.  
  
 Increases a timer used for decision policies, and step the vehicle dynamics.  
  
 :param dt: timestep  
 """* self.timer += dt  
 super().step(dt)  
  
 **def** acceleration(self,  
 ego\_vehicle: ControlledVehicle,  
 front\_vehicle: Vehicle = **None**,  
 rear\_vehicle: Vehicle = **None**) -> float:  
 *"""  
 Compute an acceleration command with the Intelligent Driver Model.  
  
 The acceleration is chosen so as to:  
 - reach a target speed;  
 - maintain a minimum safety distance (and safety time) w.r.t the front vehicle.  
  
 :param ego\_vehicle: the vehicle whose desired acceleration is to be computed. It does not have to be an  
 IDM vehicle, which is why this method is a class method. This allows an IDM vehicle to  
 reason about other vehicles behaviors even though they may not IDMs.  
 :param front\_vehicle: the vehicle preceding the ego-vehicle  
 :param rear\_vehicle: the vehicle following the ego-vehicle  
 :return: the acceleration command for the ego-vehicle [m/s2]  
 """* **if not** ego\_vehicle **or** isinstance(ego\_vehicle, RoadObject):  
 **return** 0  
 ego\_target\_speed = utils.not\_zero(getattr(ego\_vehicle, **"target\_speed"**, 0))  
 acceleration = self.COMFORT\_ACC\_MAX \* (  
 1 - np.power(max(ego\_vehicle.speed, 0) / ego\_target\_speed, self.DELTA))  
  
 **if** front\_vehicle:  
 d = ego\_vehicle.lane\_distance\_to(front\_vehicle)  
 acceleration -= self.COMFORT\_ACC\_MAX \* \  
 np.power(self.desired\_gap(ego\_vehicle, front\_vehicle) / utils.not\_zero(d), 2)  
 **return** acceleration  
  
 **def** desired\_gap(self, ego\_vehicle: Vehicle, front\_vehicle: Vehicle = **None**, projected: bool = **True**) -> float:  
 *"""  
 Compute the desired distance between a vehicle and its leading vehicle.  
  
 :param ego\_vehicle: the vehicle being controlled  
 :param front\_vehicle: its leading vehicle  
 :param projected: project 2D velocities in 1D space  
 :return: the desired distance between the two [m]  
 """* d0 = self.DISTANCE\_WANTED  
 tau = self.TIME\_WANTED  
 ab = -self.COMFORT\_ACC\_MAX \* self.COMFORT\_ACC\_MIN  
 dv = np.dot(ego\_vehicle.velocity - front\_vehicle.velocity, ego\_vehicle.direction) **if** projected \  
 **else** ego\_vehicle.speed - front\_vehicle.speed  
 d\_star = d0 + ego\_vehicle.speed \* tau + ego\_vehicle.speed \* dv / (2 \* np.sqrt(ab))  
 **return** d\_star  
  
 **def** maximum\_speed(self, front\_vehicle: Vehicle = **None**) -> Tuple[float, float]:  
 *"""  
 Compute the maximum allowed speed to avoid Inevitable Collision States.  
  
 Assume the front vehicle is going to brake at full deceleration and that  
 it will be noticed after a given delay, and compute the maximum speed  
 which allows the ego-vehicle to brake enough to avoid the collision.  
  
 :param front\_vehicle: the preceding vehicle  
 :return: the maximum allowed speed, and suggested acceleration  
 """* **if not** front\_vehicle:  
 **return** self.target\_speed  
 d0 = self.DISTANCE\_WANTED  
 a0 = self.COMFORT\_ACC\_MIN  
 a1 = self.COMFORT\_ACC\_MIN  
 tau = self.TIME\_WANTED  
 d = max(self.lane\_distance\_to(front\_vehicle) - self.LENGTH / 2 - front\_vehicle.LENGTH / 2 - d0, 0)  
 v1\_0 = front\_vehicle.speed  
 delta = 4 \* (a0 \* a1 \* tau) \*\* 2 + 8 \* a0 \* (a1 \*\* 2) \* d + 4 \* a0 \* a1 \* v1\_0 \*\* 2  
 v\_max = -a0 \* tau + np.sqrt(delta) / (2 \* a1)  
  
 *# Speed control* self.target\_speed = min(self.maximum\_speed(front\_vehicle), self.target\_speed)  
 acceleration = self.speed\_control(self.target\_speed)  
  
 **return** v\_max, acceleration  
  
 **def** change\_lane\_policy(self) -> **None**:  
 *"""  
 Decide when to change lane.  
  
 Based on:  
 - frequency;  
 - closeness of the target lane;  
 - MOBIL model.  
 """  
 # If a lane change already ongoing* **if** self.lane\_index != self.target\_lane\_index:  
 *# If we are on correct route but bad lane: abort it if someone else is already changing into the same lane* **if** self.lane\_index[:2] == self.target\_lane\_index[:2]:  
 **for** v **in** self.road.vehicles:  
 **if** v **is not** self \  
 **and** v.lane\_index != self.target\_lane\_index \  
 **and** isinstance(v, ControlledVehicle) \  
 **and** v.target\_lane\_index == self.target\_lane\_index:  
 d = self.lane\_distance\_to(v)  
 d\_star = self.desired\_gap(self, v)  
 **if** 0 < d < d\_star:  
 self.target\_lane\_index = self.lane\_index  
 **break  
 return** *# else, at a given frequency,* **if not** utils.do\_every(self.LANE\_CHANGE\_DELAY, self.timer):  
 **return** self.timer = 0  
  
 *# decide to make a lane change* **for** lane\_index **in** self.road.network.side\_lanes(self.lane\_index):  
 *# Is the candidate lane close enough?* **if not** self.road.network.get\_lane(lane\_index).is\_reachable\_from(self.position):  
 **continue** *# Does the MOBIL model recommend a lane change?* **if** self.mobil(lane\_index):  
 self.target\_lane\_index = lane\_index  
  
 **def** mobil(self, lane\_index: LaneIndex) -> bool:  
 *"""  
 MOBIL lane change model: Minimizing Overall Braking Induced by a Lane change  
  
 The vehicle should change lane only if:  
 - after changing it (and/or following vehicles) can accelerate more;  
 - it doesn't impose an unsafe braking on its new following vehicle.  
  
 :param lane\_index: the candidate lane for the change  
 :return: whether the lane change should be performed  
 """  
 # Is the maneuver unsafe for the new following vehicle?* new\_preceding, new\_following = self.road.neighbour\_vehicles(self, lane\_index)  
 new\_following\_a = self.acceleration(ego\_vehicle=new\_following, front\_vehicle=new\_preceding)  
 new\_following\_pred\_a = self.acceleration(ego\_vehicle=new\_following, front\_vehicle=self)  
 **if** new\_following\_pred\_a < -self.LANE\_CHANGE\_MAX\_BRAKING\_IMPOSED:  
 **return False** *# Do I have a planned route for a specific lane which is safe for me to access?* old\_preceding, old\_following = self.road.neighbour\_vehicles(self)  
 self\_pred\_a = self.acceleration(ego\_vehicle=self, front\_vehicle=new\_preceding)  
 **if** self.route **and** self.route[0][2]:  
 *# Wrong direction* **if** np.sign(lane\_index[2] - self.target\_lane\_index[2]) != np.sign(self.route[0][2] - self.target\_lane\_index[2]):  
 **return False** *# Unsafe braking required* **elif** self\_pred\_a < -self.LANE\_CHANGE\_MAX\_BRAKING\_IMPOSED:  
 **return False** *# Is there an acceleration advantage for me and/or my followers to change lane?* **else**:  
 self\_a = self.acceleration(ego\_vehicle=self, front\_vehicle=old\_preceding)  
 old\_following\_a = self.acceleration(ego\_vehicle=old\_following, front\_vehicle=self)  
 old\_following\_pred\_a = self.acceleration(ego\_vehicle=old\_following, front\_vehicle=old\_preceding)  
 jerk = self\_pred\_a - self\_a + self.POLITENESS \* (new\_following\_pred\_a - new\_following\_a  
 + old\_following\_pred\_a - old\_following\_a)  
 **if** jerk < self.LANE\_CHANGE\_MIN\_ACC\_GAIN:  
 **return False** *# All clear, let's go!* **return True  
  
 def** recover\_from\_stop(self, acceleration: float) -> float:  
 *"""  
 If stopped on the wrong lane, try a reversing maneuver.  
  
 :param acceleration: desired acceleration from IDM  
 :return: suggested acceleration to recover from being stuck  
 """* stopped\_speed = 5  
 safe\_distance = 200  
 *# Is the vehicle stopped on the wrong lane?* **if** self.target\_lane\_index != self.lane\_index **and** self.speed < stopped\_speed:  
 \_, rear = self.road.neighbour\_vehicles(self)  
 \_, new\_rear = self.road.neighbour\_vehicles(self, self.road.network.get\_lane(self.target\_lane\_index))  
 *# Check for free room behind on both lanes* **if** (**not** rear **or** rear.lane\_distance\_to(self) > safe\_distance) **and** \  
 (**not** new\_rear **or** new\_rear.lane\_distance\_to(self) > safe\_distance):  
 *# Reverse* **return** -self.COMFORT\_ACC\_MAX / 2  
 **return** acceleration  
  
  
**class** LinearVehicle(IDMVehicle):  
  
 *"""A Vehicle whose longitudinal and lateral controllers are linear with respect to parameters."""* ACCELERATION\_PARAMETERS = [0.3, 0.3, 2.0]  
 STEERING\_PARAMETERS = [ControlledVehicle.KP\_HEADING, ControlledVehicle.KP\_HEADING \* ControlledVehicle.KP\_LATERAL]  
  
 ACCELERATION\_RANGE = np.array([0.5\*np.array(ACCELERATION\_PARAMETERS), 1.5\*np.array(ACCELERATION\_PARAMETERS)])  
 STEERING\_RANGE = np.array([np.array(STEERING\_PARAMETERS) - np.array([0.07, 1.5]),  
 np.array(STEERING\_PARAMETERS) + np.array([0.07, 1.5])])  
  
 TIME\_WANTED = 2.0  
  
 **def** \_\_init\_\_(self,  
 road: Road,  
 position: Vector,  
 heading: float = 0,  
 speed: float = 0,  
 target\_lane\_index: int = **None**,  
 target\_speed: float = **None**,  
 route: Route = **None**,  
 enable\_lane\_change: bool = **True**,  
 timer: float = **None**,  
 data: dict = **None**):  
 super().\_\_init\_\_(road, position, heading, speed, target\_lane\_index, target\_speed, route,  
 enable\_lane\_change, timer)  
 self.data = data **if** data **is not None else** {}  
 self.collecting\_data = **True  
  
 def** act(self, action: Union[dict, str] = **None**):  
 **if** self.collecting\_data:  
 self.collect\_data()  
 super().act(action)  
  
 **def** randomize\_behavior(self):  
 ua = self.road.np\_random.uniform(size=np.shape(self.ACCELERATION\_PARAMETERS))  
 self.ACCELERATION\_PARAMETERS = self.ACCELERATION\_RANGE[0] + ua\*(self.ACCELERATION\_RANGE[1] -  
 self.ACCELERATION\_RANGE[0])  
 ub = self.road.np\_random.uniform(size=np.shape(self.STEERING\_PARAMETERS))  
 self.STEERING\_PARAMETERS = self.STEERING\_RANGE[0] + ub\*(self.STEERING\_RANGE[1] - self.STEERING\_RANGE[0])  
  
 **def** acceleration(self,  
 ego\_vehicle: ControlledVehicle,  
 front\_vehicle: Vehicle = **None**,  
 rear\_vehicle: Vehicle = **None**) -> float:  
 *"""  
 Compute an acceleration command with a Linear Model.  
  
 The acceleration is chosen so as to:  
 - reach a target speed;  
 - reach the speed of the leading (resp following) vehicle, if it is lower (resp higher) than ego's;  
 - maintain a minimum safety distance w.r.t the leading vehicle.  
  
 :param ego\_vehicle: the vehicle whose desired acceleration is to be computed. It does not have to be an  
 Linear vehicle, which is why this method is a class method. This allows a Linear vehicle to  
 reason about other vehicles behaviors even though they may not Linear.  
 :param front\_vehicle: the vehicle preceding the ego-vehicle  
 :param rear\_vehicle: the vehicle following the ego-vehicle  
 :return: the acceleration command for the ego-vehicle [m/s2]  
 """* **return** float(np.dot(self.ACCELERATION\_PARAMETERS,  
 self.acceleration\_features(ego\_vehicle, front\_vehicle, rear\_vehicle)))  
  
 **def** acceleration\_features(self, ego\_vehicle: ControlledVehicle,  
 front\_vehicle: Vehicle = **None**,  
 rear\_vehicle: Vehicle = **None**) -> np.ndarray:  
 vt, dv, dp = 0, 0, 0  
 **if** ego\_vehicle:  
 vt = ego\_vehicle.target\_speed - ego\_vehicle.speed  
 d\_safe = self.DISTANCE\_WANTED + np.maximum(ego\_vehicle.speed, 0) \* self.TIME\_WANTED  
 **if** front\_vehicle:  
 d = ego\_vehicle.lane\_distance\_to(front\_vehicle)  
 dv = min(front\_vehicle.speed - ego\_vehicle.speed, 0)  
 dp = min(d - d\_safe, 0)  
 **return** np.array([vt, dv, dp])  
  
 **def** steering\_control(self, target\_lane\_index: LaneIndex) -> float:  
 *"""  
 Linear controller with respect to parameters.  
  
 Overrides the non-linear controller ControlledVehicle.steering\_control()  
  
 :param target\_lane\_index: index of the lane to follow  
 :return: a steering wheel angle command [rad]  
 """* **return** float(np.dot(np.array(self.STEERING\_PARAMETERS), self.steering\_features(target\_lane\_index)))  
  
 **def** steering\_features(self, target\_lane\_index: LaneIndex) -> np.ndarray:  
 *"""  
 A collection of features used to follow a lane  
  
 :param target\_lane\_index: index of the lane to follow  
 :return: a array of features  
 """* lane = self.road.network.get\_lane(target\_lane\_index)  
 lane\_coords = lane.local\_coordinates(self.position)  
 lane\_next\_coords = lane\_coords[0] + self.speed \* self.PURSUIT\_TAU  
 lane\_future\_heading = lane.heading\_at(lane\_next\_coords)  
 features = np.array([utils.wrap\_to\_pi(lane\_future\_heading - self.heading) \*  
 self.LENGTH / utils.not\_zero(self.speed),  
 -lane\_coords[1] \* self.LENGTH / (utils.not\_zero(self.speed) \*\* 2)])  
 **return** features  
  
 **def** longitudinal\_structure(self):  
 *# Nominal dynamics: integrate speed* A = np.array([  
 [0, 0, 1, 0],  
 [0, 0, 0, 1],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0]  
 ])  
 *# Target speed dynamics* phi0 = np.array([  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, -1, 0],  
 [0, 0, 0, -1]  
 ])  
 *# Front speed control* phi1 = np.array([  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [0, 0, -1, 1],  
 [0, 0, 0, 0]  
 ])  
 *# Front position control* phi2 = np.array([  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [-1, 1, -self.TIME\_WANTED, 0],  
 [0, 0, 0, 0]  
 ])  
 *# Disable speed control* front\_vehicle, \_ = self.road.neighbour\_vehicles(self)  
 **if not** front\_vehicle **or** self.speed < front\_vehicle.speed:  
 phi1 \*= 0  
  
 *# Disable front position control* **if** front\_vehicle:  
 d = self.lane\_distance\_to(front\_vehicle)  
 **if** d != self.DISTANCE\_WANTED + self.TIME\_WANTED \* self.speed:  
 phi2 \*= 0  
 **else**:  
 phi2 \*= 0  
  
 phi = np.array([phi0, phi1, phi2])  
 **return** A, phi  
  
 **def** lateral\_structure(self):  
 A = np.array([  
 [0, 1],  
 [0, 0]  
 ])  
 phi0 = np.array([  
 [0, 0],  
 [0, -1]  
 ])  
 phi1 = np.array([  
 [0, 0],  
 [-1, 0]  
 ])  
 phi = np.array([phi0, phi1])  
 **return** A, phi  
  
 **def** collect\_data(self):  
 *"""Store features and outputs for parameter regression."""* self.add\_features(self.data, self.target\_lane\_index)  
  
 **def** add\_features(self, data, lane\_index, output\_lane=**None**):  
  
 front\_vehicle, rear\_vehicle = self.road.neighbour\_vehicles(self)  
 features = self.acceleration\_features(self, front\_vehicle, rear\_vehicle)  
 output = np.dot(self.ACCELERATION\_PARAMETERS, features)  
 **if "longitudinal" not in** data:  
 data[**"longitudinal"**] = {**"features"**: [], **"outputs"**: []}  
 data[**"longitudinal"**][**"features"**].append(features)  
 data[**"longitudinal"**][**"outputs"**].append(output)  
  
 **if** output\_lane **is None**:  
 output\_lane = lane\_index  
 features = self.steering\_features(lane\_index)  
 out\_features = self.steering\_features(output\_lane)  
 output = np.dot(self.STEERING\_PARAMETERS, out\_features)  
 **if "lateral" not in** data:  
 data[**"lateral"**] = {**"features"**: [], **"outputs"**: []}  
 data[**"lateral"**][**"features"**].append(features)  
 data[**"lateral"**][**"outputs"**].append(output)  
  
  
**class** AggressiveVehicle(LinearVehicle):  
 LANE\_CHANGE\_MIN\_ACC\_GAIN = 1.0 *# [m/s2]* MERGE\_ACC\_GAIN = 0.8  
 MERGE\_VEL\_RATIO = 0.75  
 MERGE\_TARGET\_VEL = 10  
 ACCELERATION\_PARAMETERS = [MERGE\_ACC\_GAIN / ((1 - MERGE\_VEL\_RATIO) \* MERGE\_TARGET\_VEL),  
 MERGE\_ACC\_GAIN / (MERGE\_VEL\_RATIO \* MERGE\_TARGET\_VEL),  
 0.5]  
  
  
**class** DefensiveVehicle(LinearVehicle):  
 LANE\_CHANGE\_MIN\_ACC\_GAIN = 1.0 *# [m/s2]* MERGE\_ACC\_GAIN = 1.2  
 MERGE\_VEL\_RATIO = 0.75  
 MERGE\_TARGET\_VEL = 10  
 ACCELERATION\_PARAMETERS = [MERGE\_ACC\_GAIN / ((1 - MERGE\_VEL\_RATIO) \* MERGE\_TARGET\_VEL),  
 MERGE\_ACC\_GAIN / (MERGE\_VEL\_RATIO \* MERGE\_TARGET\_VEL),  
 2.0]