* Configs:

**Set\_parameters.py:** this function sets different types of parameters for the model in the dictionary ‘paras’:

1. Common Parameters (set\_common\_paras)
   1. MPC parameters: *, ,*  (description presented in the first paper)
   2. IDM model parameters: Acceleration parameters of the IDM model(description available in the second paper Eq. 11)
   3. Simulation parameters: Penetration rate of CAVs, Simulation steps, yellow time, left-right ratio, Communication range, speed limit, vehicle length, number of lanes, distance from upstream intersections, number of phases, low and high volumes, half-period of volume waves.
2. Network Topology Parameters (set\_network\_topology\_paras)

Adds graph of the network:

Position (pos), adjacency of the nodes (adj), distance from upstream intersection (distance\_from\_upstream\_intersections), num lanes (num\_lanes\_each\_road), num phases (num\_phases), communication range (range).

1. Signal Phase Parameters (set\_signal\_phases\_paras)

Paras[‘actions’]: different phases illustration

1. Volume Parameters

Estimates Depart Rate (a range of volumes from lowest to highest with a fixed step) of each bound at each wave interval, based on low and high volumes, half-period of volume waves, num intervals, volume type.

* Environment:

**Single\_intersection.py:** this **class** creates the network (single intersection)

Main functions:

1. \_\_init\_\_:
   1. sumo\_network\_builder(paras): sets the “neighbors” for paras dictionary. Neighbors include the neighboring intersections of the current intersection along with the distance to that node (communication range), and number of lanes connecting the intersections.
   2. SumoRoutesGenerator(paras)
2. start\_sumo
3. is\_active: traci.simulation.getMinExpectedNumber()

returns the number of all active vehicles and persons which are in the net plus the

ones still waiting to start. If the number is 0 however, it is guaranteed that all route files have been parsed completely.

1. get\_state\_cur\_intersection (self, cur\_step): creates the dictionary “network\_state” (for **one slower time step**):

* "num\_vehicles\_max": maximum number of vehicles on the lanes of the intersection (i.e., maximum over the vehicles of each lane)
* "pos\_vehicles": position of vehicles
* "speed\_vehicles": speed of the vehicles
* "wt\_vehicles": waiting time of the vehicles
* "arrival\_times": arrival time step of the future vehicles in communication range
* “signal\_phase”: index of the phase of the signal at the current step
* “num\_lane”
* "lane\_length"
* "lane\_id"
* "vehicle\_id"

at the current step of the simulation:

for **each intersection** in the network:

for **each lane** of the intersection:

it first creates list of lists for above variables. Each of these lists have number of lanes number of list inside and each of the insider lists comprise information on the vehicles on that lane which is in the communication range or will be in the communication range in the prediction horizon. (how many steps? one slower time step)

Signal\_phase -> traci.trafficlight.getPhase(str(inter\_id)): Returns the index of the current phase within the list of all phases of the current program (the phase that is on =1 at the current phase out of all phases (8)).

traci.lane.getLastStepVehicleIDs -> Returns the ids of the vehicles for the last time step on the given lane and stores them in the variable named ‘cars\_lane’

For **each car** in the lane (cars\_lane):

If the vehicle is in the communication range or will be (=if the distance from the communication range/speed limit is less than prediction horizon) fills the information needed for the lists mentioned above:

Position-> traci.vehicle.getLanePosition(cars\_lane[j]),

Speed-> traci.vehicle.getSpeed(cars\_lane[j]),

waiting time-> traci.vehicle.getWaitingTime(cars\_lane[j])

The waiting time of a vehicle is defined as the time (in seconds) spent with a speed below 0.1m/s since the last time it was faster than 0.1m/s.

(basically, the waiting time of a vehicle is reset to 0 every time it moves).

A vehicle that is stopping intentionally with a <stop> does not accumulate waiting time.

arrival time-> cur\_step + int(time to reach the communication range/ delta\_T\_faster)

the arrival time is the step of the simulation (in faster time scale) in which the out-of-range vehicle will enter the range.

1. apply\_control\_commands(should\_update\_signal, next\_signal\_phase, speed\_commands, network\_state, vehicle\_ids): gets the commands from MPC\_agent and implement the commands in SUMO. Speeds are adjusted based on IDM to guarantee safety.

* Agent:

**Mpc\_agent.py:** this class solves the slower and faster scale optimization problem.

1. \_\_init\_\_: set gams working space and initialize lists for slower and faster scale problems’ outputs.
2. Get\_control\_commands (paras, network\_state, cur\_step): main function to run the whole MPC scheme.

**First Part: Solving the MPC problem**

MPC gets solved at slower scale time steps. These steps are stored in the variable ‘next\_global\_step\_to\_re\_solve\_the\_netwok’. This means that the MPC algorithm will be solved at each , if we don’t hit a yellow phase. If the current phase is yellow, the next time step to solve the MPC problem is steps away.

If the current step > next global step to solve the MPC problem:

For **each intersection** of the network graph where there exist vehicles:

2-1- Solve\_single\_intersection:

2-1-1- gather\_static\_paras\_single\_intersection(paras):

Stores the parameters defined in ‘paras’ in set\_parameters.py.

2-1-2- solve\_slower\_scale\_problem\_single\_intersection(intersection\_state)

2-1-2-1- set\_slower\_scale\_constant\_paras\_single\_intersection():

Add the above constant parameters to GAMS workspace.

2-1-2-2- set\_slower\_scale\_dynamic\_paras\_single\_intersection(intersection\_state):

Adds the dynamic parameters of ‘intersection\_state’ to the GAMS workspace. These parameters include:

* ‘pos\_vehicles’: Positions of each vehicle, 1 \* m list where m is the number of lanes, each element is a 1\*n list where each element is the position of each vehicle.
* ‘wt\_vehicles’= Waiting time, dimensions are the same as pos\_vehicles.
* ‘cur\_phase’: Current signal phase, int.
* ‘num\_vehicles\_max’: The maximum number of existing and incoming vehicles on each lane, int.

2-1-2-3- run\_gams\_to\_solve\_slower\_scale\_problem\_single\_intersection()

2-1-2-4- collect\_solution\_from\_slower\_scale\_problem\_single\_intersection (intersection\_state):

* ‘p\_gams’: whether a phase is on =1 or off=0. 1\*m list where m is the total number of phases (8), and each element of these phases is a 1\*n list where n is the number of prediction steps+1 (current step + 6 =7). Yellow phase is identified by not finding any p\_gams[:][k]=1.
* ‘s\_vehicles\_slower’: is a 1\*m list where m is the number of lanes (12), each of its elements is a 1\*n list where n is the num\_vehicles\_max, where each of its elements is a 1\*k list of the position of the vehicle where k is the number of prediction steps +1 (7).
* ‘following\_phases’: is a list with number of prediction steps + 1 (7) elements representing the phase numbers that are going to be active at the current and following steps according to the MPC results.

Phase map:

0: <phase state="GGGGrrrrrrrrGGGGrrrrrrrr" duration="29"/>

1: <phase state="yyyyrrrrrrrryyyyrrrrrrrr" duration="5"/>

2: <phase state="rrrrGGrrrrrrrrrrGGrrrrrr" duration="6"/>

3: <phase state="rrrryyrrrrrrrrrryyrrrrrr" duration="5"/>

4: <phase state="rrrrrrGGGGrrrrrrrrGGGGrr" duration="29"/>

5: <phase state="rrrrrryyyyrrrrrrrryyyyrr" duration="5"/>

6: <phase state="rrrrrrrrrrGGrrrrrrrrrrGG" duration="6"/>

7: <phase state="rrrrrrrrrryyrrrrrrrrrryy" duration="5"/>

If none of the phases in the current step are 1 then the phase should be yellow and the next global step to solve the MPC problem will be updated.

2-1-3- prepare\_faster\_scale\_input\_from\_slower\_scale\_solution (intersection\_state):

Returns:

* critical\_points: The critical points in the faster scale (h=3 time points in faster scale).
* pos\_vehicles\_point: Position of vehicles at critical points. is a 1\*m list where m is the number of lanes (12), each of its elements is a 1\*n list where n is the num\_vehicles\_max, where each of its elements is a 1\*k list of the position of the vehicle where k is the number of critical points (h=3).
* steps\_faster: Total number of faster scale steps corresponding to the entire slower scale critical steps (3\*5s/0.5s=30 steps).

2-1-4- solve\_faster\_scale\_problem\_single\_intersection (critical\_points, s\_vehicles\_point, steps\_faster, intersection\_state):

For **each lane**:

2-1-4-1- set\_faster\_scale\_dynamic\_paras\_single\_lane(critical\_points, pos\_vehicles\_point, pos\_vehicles\_init, speed\_vehicles\_init, steps\_faster):

Adds the dynamic parameters of initial speed of the vehicles, initial positions of the vehicles, critical positions of the vehicles, and critical points of time for one lane to the GAMS workspace.

2-1-4-2- run\_gams\_to\_solve\_faster\_scale\_problem\_single\_lane()

2-1-4-3- self.collect\_solution\_from\_faster\_scale\_problem\_single\_lane(lane, a\_vehicles\_faster, v\_vehicles\_faster, s\_vehicles\_faster):

Returns:

* ‘a\_vehicles\_faster’: a 1\*n list where n is the number of faster scale steps covering critical points (31), each element of which is a list of 1\*m dimension where m is the number of lanes (12) in each element of which a list of 1\*k dimensions exists where the k is the number of vehicles on the lane.
* ‘v\_vehicles\_faster’
* ‘s\_vehicles\_faster’

**Second Part: Giving the Control Commands**

Whether or not the above ‘solve\_single\_intersection’ function runs in the current step (MPC problem is solved or not), the get\_control\_command will return the control commands in faster time scales (i.e., every 0.5 seconds):

Returns:

• should\_update\_signal: Whether we should update the signal phase. Set to be true each time we solve new MPC problems.

• next\_signal\_phase: The next signal phase that we can use if should\_update\_signal = True. {0, 1, 2, 3, -1} (normal phases + yellow)

• speed\_commands: Speeds that need to be applied to all the vehicles in the network. HDVs will be skipped in the control part. But we still generate solutions for them. 1 \* m list where m is the number of lanes, and each element is a 1 \* n list where n is the number of vehicles. The time dimension is the next critical point horizon (3\*5s/0.5s=30 steps).

• vehicle\_ids: IDs of vehicles that are solved in the current or previous round's MPC problems.

A screenshot of a diagram

Description automatically generated