Tutorial:

How path planning and control work in an offical exemple automatic_control.py with its simplified version

<u>First: path planning</u>

➔ automatic_control_revised.py:

```
    # generate waypoints_queue and waypoints_buffer for the _local_planner
    # with a global route from global_route_planner(_dao)
    agent.set_destination(agent.vehicle.get_location(), destination.location, clean=True, debug=True)
```

➔ bahvoir_agent.py:

```
def set_destination(self, start_location, end_location, clean=False, debug=True):
    """
    This method creates a list of waypoints from agent's position to destination location
    based on the route returned by the global router.

        :param start_location: initial position
        :param end_location: final position
        :param clean: boolean to clean the waypoint queue
    """
    if clean:
        self._local_planner.waypoints_queue.clear()
    self.start_waypoint = self._map.get_waypoint(start_location)
    self.end_waypoint = self._map.get_waypoint(end_location)

    route_trace = self._trace_route(self.start_waypoint, self.end_waypoint)

    self._local_planner.set_global_plan(route_trace, clean,debug)
```

➔ First set up a global router(aka. Graph representation for a given Carla map) and then obtain route plan(aka. path planning) in _trace_route:

```
def _trace_route(self, start_waypoint, end_waypoint):
    """
    This method sets up a global router and returns the
    optimal route from start_waypoint to end_waypoint.

        :param start_waypoint: initial position
        :param end_waypoint: final position
    """
    # Setting up global router
    if self._grp is None:
        wld = self.vehicle.get_world()
        dao = GlobalRoutePlannerDAO(
            wld.get_map(), sampling_resolution=self._sampling_resolution)
        grp = GlobalRoutePlanner(dao)
        grp.setup()
        self._grp = grp

    # Obtain route plan
    route = self._grp.trace_route(
        start_waypoint.transform.location,
        end_waypoint.transform.location)

    return route
```

1. Setting up a global router

    a. Line151-152: Initialize a route planner for the Carla map.

➔ Global_route_planner.py

```python
class GlobalRoutePlannerDAO(object):
    """
    This class is the data access layer for fetching data
    from the carla server instance for GlobalRoutePlanner
    """

    def __init__(self, wmap, sampling_resolution):
        """
        Constructor method.

            :param wmap: carla.world object
            :param sampling_resolution: sampling distance between waypoints
        """
        self._sampling_resolution = sampling_resolution
        self._wmap = wmap
```

➔ Global_route_planner.py

```python
class GlobalRoutePlanner(object):
    """
    This class provides a very high level route plan.
    Instantiate the class by passing a reference to
    A GlobalRoutePlannerDAO object.
    """


    def __init__(self, dao):
        """
        Constructor
        """
        self._dao = dao
        self._topology = None
        self._graph = None
        self._id_map = None
        self._road_id_to_edge = None
        self._intersection_end_node = -1
        self._previous_decision = RoadOption.VOID
```

b. Get the topology of the Carla map

➔ Global_route_planner.py

```python
    def setup(self):
        """
        Performs initial server data lookup for detailed topology
        and builds graph representation of the world map.
        """

        """
        graph: networkx.Diagraph(); id_map:a dictionary, the key is the waypoint transform, and value is its
        node id in the graph; road_id_to_edge: eg.{road_id:{section_id:{lane_id:(node_id, node_id)}}}
        """

        self._topology = self._dao.get_topology()
        self._graph, self._id_map, self._road_id_to_edge = self._build_graph()
        self._find_loose_ends()
        self._lane_change_link()
```

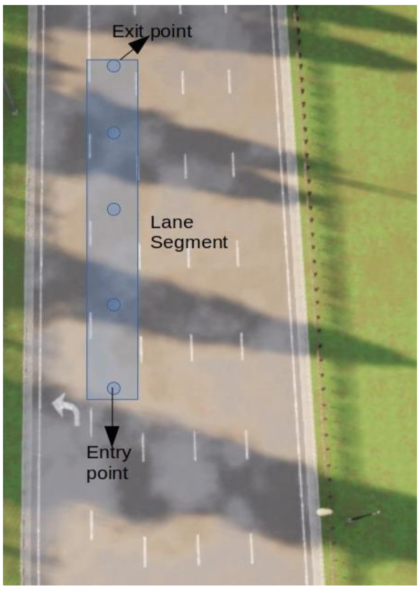
➔ Global_route_planner_dao.py

```python
    def get_topology(self):
        """
        Accessor for topology.
        This function retrieves topology from the server as a list of
        road segments as pairs of waypoint objects, and processes the
        topology into a list of dictionary objects.

            :return topology: list of dictionary objects with the following attributes
                entry    -   waypoint of entry point of road segment
                entryxyz-   (x,y,z) of entry point of road segment
                exit     -   waypoint of exit point of road segment
                exitxyz  -   (x,y,z) of exit point of road segment
                path     -   list of waypoints separated by 1m from entry
                             to exit
```

Data format of its return:

```
[{entry:Waypoint1, exit:Waypoint2, path:[Waypoint1_1, Waypoint1_2,......]},
 {entry:Waypoint1, exit:Waypoint2, path:[Waypoint1_1, Waypoint1_2,......]},
 ......]
```

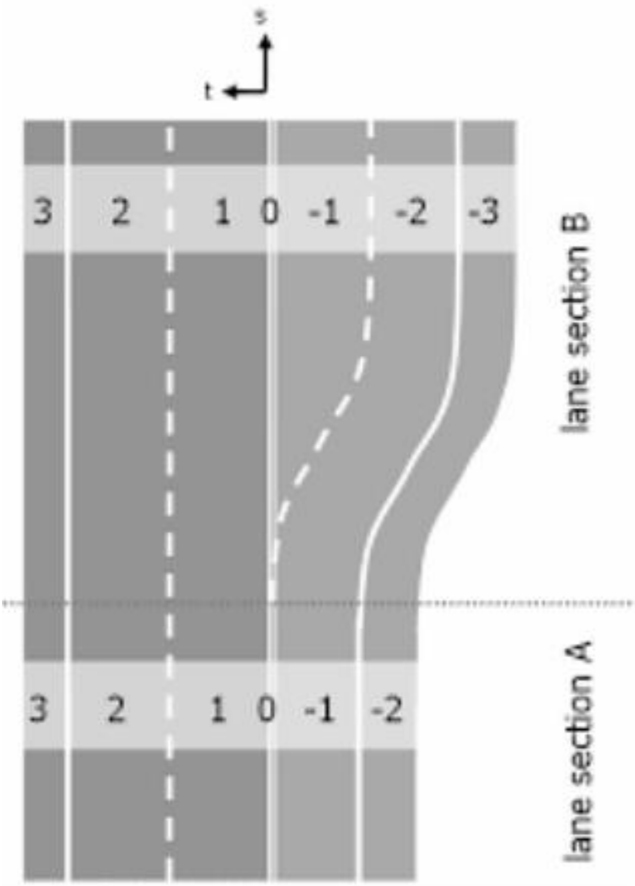


c. build a graph based on the topology

➔ Global_route_planner.py

```
    def _build_graph(self):
        """
        This function builds a networkx graph representation of topology.
        The topology is read from self._topology.
        graph node properties:
            vertex   -   (x,y,z) position in world map
        graph edge properties:
            entry_vector    -   unit vector along tangent at entry point
            exit_vector     -   unit vector along tangent at exit point
            net_vector      -   unit vector of the chord from entry to exit
            intersection    -   boolean indicating if the edge belongs to an
                                intersection
        return      :   graph -> networkx graph representing the world map,
                        id_map-> mapping from (x,y,z) to node id
                        road_id_to_edge-> map from road id to edge in the graph
```

What it does: connect the land segments together



d. self._find_loose_ends(), self.lane_change_link():
   Complete the lane map, for example: connect changable lanes.

Now back to here: global router is set up(= a graph representation for the Carla map is set up).

```python
    def _trace_route(self, start_waypoint, end_waypoint):
        """
        This method sets up a global router and returns the
        optimal route from start_waypoint to end_waypoint.

            :param start_waypoint: initial position
            :param end_waypoint: final position
        """
        # Setting up global router
        if self._grp is None:
            wld = self.vehicle.get_world()
            dao = GlobalRoutePlannerDAO(
                wld.get_map(), sampling_resolution=self._sampling_resolution)
            grp = GlobalRoutePlanner(dao)
            grp.setup()
            self._grp = grp

        # Obtain route plan
        route = self._grp.trace_route(
            start_waypoint.transform.location,
            end_waypoint.transform.location)

        return route
```

So:

2. Obtain route plan: path planning with A*

➔ Global_route_planner.py

```python
    def trace_route(self, origin, destination):
        """
        This method returns list of (carla.Waypoint, RoadOption)
        from origin to destination
        """

        route_trace = []
        route = self._path_search(origin, destination)
```

```python
def _path_search(self, origin, destination):
    """
    This function finds the shortest path connecting origin and destination
    using A* search with distance heuristic.
    origin      :   carla.Location object of start position
    destination :   carla.Location object of of end position
    return      :   path as list of node ids (as int) of the graph self._graph
    connecting origin and destination
    """

    start, end = self._localize(origin), self._localize(destination)

    route = nx.astar_path(
        self._graph, source=start[0], target=end[0],
        heuristic=self._distance_heuristic, weight='length')
    route.append(end[1])
    return route
```

Return: a optimal path from a pair of start point and end point w.r.t A*.

Now the path planning is done.

Second: control

➔ automatic_control_revised.py:

```python
            control = agent.run_step(debug=True) # if debug is on: target waypoints will be shown
            vehicle.apply_control(control)
```

➔ behavior_agent.py:

def run_step()

```python
        # Calculate controller based on no turn, traffic light or vehicle in front
        else:
            control = self._local_planner.run_step(
                target_speed=min(self.behavior.max_speed, self.speed_limit - self.behavior.speed_lim_dist), debug=debug)

        return control
```

➔ local_planner_behavior.py:

def run_step()

```python
        control = self._pid_controller.run_step(self._target_speed, self.target_waypoint)

        # Purge the queue of obsolete waypoints
        vehicle_transform = self._vehicle.get_transform()
        max_index = -1

        for i, (waypoint, _) in enumerate(self._waypoint_buffer):
            if distance_vehicle(
                    waypoint, vehicle_transform) < self._min_distance:
                max_index = i
        if max_index >= 0:
            for i in range(max_index + 1):
                self._waypoint_buffer.popleft()

        if debug:
            draw_waypoints(self._vehicle.get_world(),
                           [self.target_waypoint], 1.0)
            print("current config:",self.target_waypoint.transform.location,self.target_way

        return control
```

➔ control.py

```python
    def run_step(self, target_speed, waypoint):
        """
        Execute one step of control invoking both lateral and longitudinal
        PID controllers to reach a target waypoint
        at a given target_speed.

            :param target_speed: desired vehicle speed
            :param waypoint: target location encoded as a waypoint
            :return: distance (in meters) to the waypoint
        """

        acceleration = self._lon_controller.run_step(target_speed,debug=False)
        current_steering = self._lat_controller.run_step(waypoint)
```

For lon_controller:

```python
    def run_step(self, target_speed, debug=False):
        """
        Execute one step of longitudinal control to reach a given target speed.

            :param target_speed: target speed in Km/h
            :param debug: boolean for debugging
            :return: throttle control
        """
        current_speed = get_speed(self._vehicle)

        if debug:
            print('Current speed = {}'.format(current_speed))

        return self._pid_control(target_speed, current_speed)
```

Send data to Matlab to get the control value:

```python
    def _pid_control(self, target_speed, current_speed):
        """
        Estimate the throttle/brake of the vehicle based on the PID equations

            :param target_speed:  target speed in Km/h
            :param current_speed: current speed of the vehicle in Km/h
            :return: throttle/brake control
        """

        error = target_speed - current_speed
        self._error_buffer.append(error)

        if len(self._error_buffer) >= 2:
            _de = (self._error_buffer[-1] - self._error_buffer[-2]) / self._dt
            _ie = sum(self._error_buffer) * self._dt
        else:
            _de = 0.0
            _ie = 0.0

        ## 1. option: direct computation for throttle
        # throttle = (self._k_p * error) + (self._k_d * _de) + (self._k_i * _ie)

        ## 2. option: connect matlab, compute the throttle in matlab and get it back through matlab.engine
        ## Ps: make sure the following command is executed in matlab command window, before running python scripts:
        ## matlab.engine.shareEngine
        eng = matlab.engine.connect_matlab()
        throttle = eng.LongPID(self._k_p,error,self._k_d,_de,self._k_i,_ie)

        return np.clip(throttle, -1.0, 1.0)
```