Final Project Solution

JULY 19, 2018

PREPARED BY Ryan De Iaco

UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE & ENGINEERING

Solution Overview

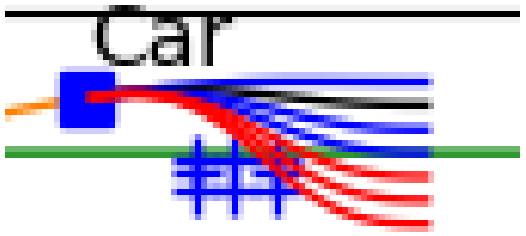
- Path generation
- Path collision checking
- Velocity profile generation
- Behavioural planning

Path Generation

- Spiral optimization was given to you
- Goal points found by finding the point that was ahead of the ego vehicle on the set of waypoints by LOOKAHEAD_DISTANCE
- Goal state set generated by laterally offsetting these points
- Goal points conform to road structure

Collision Avoidance

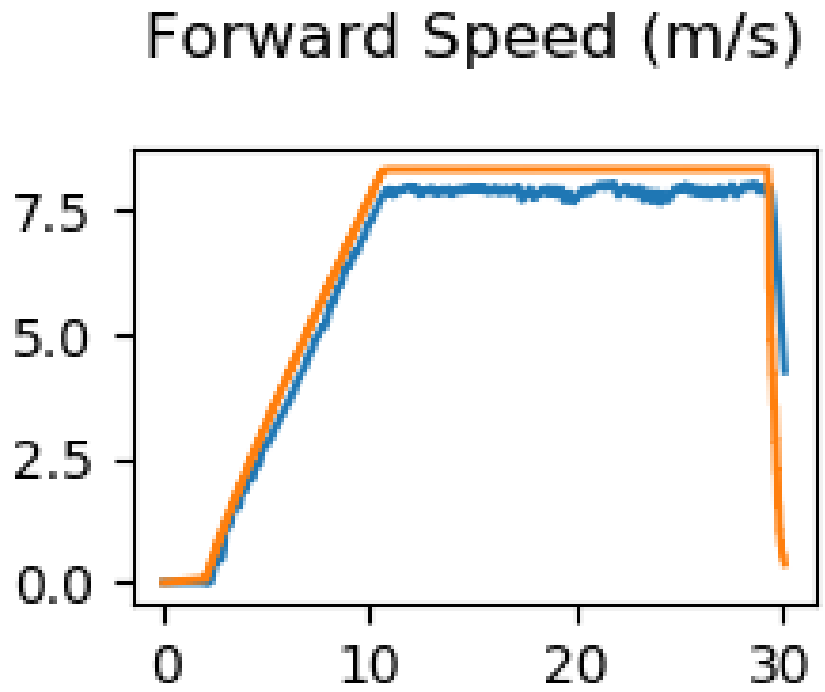
- Can be completed using circle collision checking
- Paths in collision marked in red
- Black path is the optimal path according to our cost function




```
def collision_check(self, paths, obstacles):
        collision_check_array = []
        for i in range(len(paths)):
            collision_free = True
            path = paths[i]
            for j in range(len(path[0])):
                # Compute the circle locations along this point in the path.
                circle_locations = []
                for k in range(len(self._circle_offsets)):
                    circle_x = path[0][j] +
                        self._circle_offsets[k]*cos(path[2][j])
                    circle_y = path[1][j] +
                        self._circle_offsets[k]*sin(path[2][j])
                    circle_locations.append([circle_x, circle_y])

                # Assumes each obstacle is approximated by a collection of points
                # of the form [x, y].
                for k in range(len(obstacles)):
                    for l in range(len(obstacles[k])):
                        for m in range(len(circle_locations)):
                            if np.linalg.norm(np.subtract(circle_locations[m],
                                    obstacles[k][l])) < self._circle_radii[m]:
                                collision_free = False
                                break
                        if not collision_free:
                            break
                    if not collision_free:
                        break

            collision_check_array.append(collision_free)

        return collision_check_array
```

Velocity Profile Generation

- Generate profile to reach goal velocity
- Lead vehicles requires deceleration to avoid collision
- Can construct a ramp velocity profile to reach our goal velocity for a fixed acceleration



Behaviour Planning

- State machine with 3 states, lane following, decelerate to stop, and stay stopped
- Need to stay stopped for a certain number of iterations, before proceeding to next state
- State will inform our outputs to the local planner

```
# Handles state transitions and computes the goal state.
def transition_state(self, waypoints, ego_state, closed_loop_speed):
    # In this state, continue tracking the lane by finding the
    # goal index in the waypoint list that is within the lookahead
    # distance. Then, check to see if the waypoint path intersects
    # with any stop lines. If it does, then ensure that the goal
    # state enforces the car to be stopped before the stop line.
    if self._state == FOLLOW_LANE:
        # First, find the closest index to the ego vehicle.
        closest_len, closest_index = get_closest_index(waypoints, ego_state)
        # Next, find the goal index that lies within the lookahead distance along the
        # waypoints.
        goal_index = self.get_goal_index(waypoints, ego_state, closest_len, closest_index)
        # Finally, check the index set between closest_index and goal_index for stop signs,
        # and compute the goal state accordingly.
        goal_index, stop_sign_found = self.check_for_stop_signs(waypoints, closest_index, goal_index)
        self._goal_index = goal_index
        self._goal_state = waypoints[goal_index]
        if stop_sign_found:
            # Set the goal to zero speed, then transition to the deceleration state.
            self._goal_state[2] = 0.0
            self._state = DECELERATE_TO_STOP

    # In this state, check if we have reached a complete stop. Use the closed loop speed
    # to do so, to ensure we are actually at a complete stop.
    # If so, transition to the next state.
    elif self._state == DECELERATE_TO_STOP:
        if closed_loop_speed < STOP_THRESHOLD:
            self._state = STAY_STOPPED

    # In this state, check to see if we have stayed stopped for at
    # least STOP_COUNTS number of cycles. If so, we can now leave
    # the stop sign and transition to the next state.
    elif self._state == STAY_STOPPED:
        # We have stayed stopped for the required number of cycles.
        # Allow the ego vehicle to leave the stop sign. Once it has
        # passed the stop sign, return to lane following.
        if self._stop_count == STOP_COUNTS:
            closest_len, closest_index = get_closest_index(waypoints, ego_state)
            goal_index = self.get_goal_index(waypoints, ego_state, closest_len, closest_index)
            # We've stopped for the required amount of time, so the new goal index for the stop
            # line is not relevant. Use the goal index that is the lookahead distance away.
            stop_sign_found = self.check_for_stop_signs(waypoints, closest_index, goal_index)[1]
            self._goal_index = goal_index
            self._goal_state = waypoints[goal_index]
            if not stop_sign_found:
                self._stop_count = 0
                self._state = FOLLOW_LANE
        else:
            self._stop_count += 1
    else:
        raise ValueError('Invalid state value.')
```

Complete Motion Planner

- Bringing these together gives a motion planner with the full behaviour required for this scenario
- This is only one solution, many solutions possible