Spring 2021 ROAR S1 Series First Place Solution

Team: Winning ROAR!!

By James Cheney

Team members: James Cheney, Chufan Guo

With assistance from Michael Wu and Flaviano Christian Reyes

Introduction

The evolution of the winning Roll ContRoll agent for entry into the Spring 2021 ROAR series S1 race began with a curiosity - what if I added a roll input to a PID autonomous vehicle controller as a means of gauging maximum speed in a turn? To satisfy my curiosity, I replaced the entirety of the longitudinal control (throttle control) of the PID agent with a simple algorithm: throttle output = exp (-0.07 * |roll|). This output gives a result of one (maximum throttle) for zero roll and exponentially reduces toward zero (no throttle) as body roll increases (The .07 number was arrived at via a quick calculation of maximum observed body roll at the time and reducing to an arbitrary low value of throttle).

This ultra simple throttle control immediately worked better! With a quick tuning of the roll factor (performed by Flaviano Christian Reyes, a team member on another project who took an interest in this experiment), this agent consistently outperformed the original agent as well as the agent being developed by that team.

The insight gained from this experience was applied to preparing an agent for entry into the ROAR series race, by the Winning ROAR!! Project Team 1 (of two teams), composed of James Cheney and Chufan Guo. Michael Wu graciously offered support as a ROAR platform consultant, and his help was instrumental in the success of our team.

Strategy, Design, and Implementation

During refinement of this agent, and experimentation with other steering control strategies, the decision was taken to change the course for the race to the Berkeley map. This change defined new challenges. While the old course was largely a giant figure-8 shape, the new course was much more rectangular (see Figure 2 below). In the figure-8 shape, managing speed through long sweeping turns was a key component for success, while in the more rectangular Berkeley map, maximizing speed along long straightaways was key. Another significant change was the waypoint map, which went from a fine-grained plot of 6512 points, to a coarse-grained plot with only 263 points for a similar distance. This had the effect of the agent cutting corners too sharply and colliding with the inner barrier in some turns (see Figure 1 below).

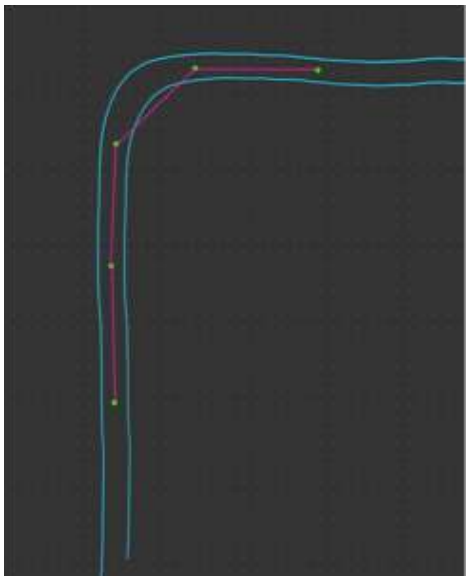

Figure 1
Curve example
(exaggeration of upper left corner for illustration)

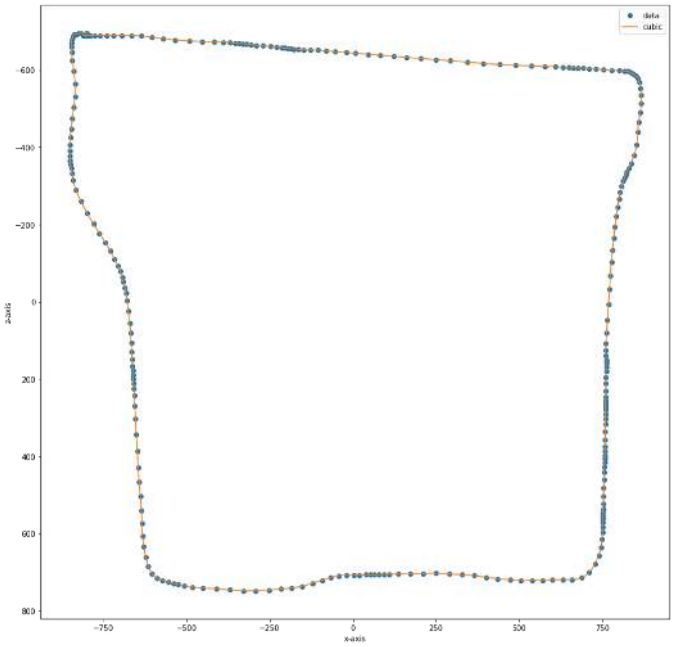

Figure 2
Curve fitting to Berkeley waypoint map

The coarseness of the waypoint map was addressed in two different ways. In the first approach, the way points had curve fitting applied, creating 5000 points of a very smooth course (see Figure 2 above), which was an improvement over the original method (in which the points were generated by manually driving the course via keyboard input, which resulted in a rough plot). This method worked great, but it was determined that an alternate set of waypoints was not allowed. Therefore, we needed to either move the smooth waypoint generation into the live agent or try an alternative approach. The second approach was to simply have a less responsive steering control, which would not respond fast enough to cut through the corner between two waypoints (see Figure 1 above), but would respond fast enough to navigate the curves overall. The second approach worked satisfactorily. Future refinement may re-try the smooth curve approach.

With this issue resolved, focus shifted to addressing the key component to a fast time on the Berkeley track - maximizing speed in the long straightaways. First, in order to maximize speed in the straightaways, we had to find a way to go as fast as possible, yet reduce speed in time to make it safely through every turn. Second, we need to lose as little speed in turns as possible, as it takes time to accelerate back to high speed, especially on the uphill portion of the course.

The strategy for going fast was to always apply maximum throttle on straight portions - starting from the exit of a turn. To slow for turns, a look-ahead algorithm was applied to reduce throttle (there were no brakes - only throttle variations) in time to safely yet minimally reduce speed for curves. This look-ahead used the waypoints and trigonometry to calculate the heading change

in the road ahead, by comparing the current vehicle heading (provided by CARLA) to the angle of the line segment between two waypoints on the path ahead in reference to the environment's coordinate system (heading error). The throttle was reduced by a factor of the magnitude of the heading error, and the distance ahead of the points used for comparison was determined by a factor of speed. This resulted in reducing the speed further in advance when going faster, and reducing it more aggressively in proportion to the increase in curvature. Once this process was working, it was a matter of tuning these factors to achieve the best performance on the track. This approach was the most critical aspect of strong and safe performance in the race.

The second part of maximizing speed on the straightaways was maximizing speed through the turns, so a higher speed could be reached sooner for the straight sections. This was accomplished by using the Roll ContRoll to maximize speed through the corners by accelerating to the limits of controlled trajectory with a little sliding in the turns.

The key issues addressed in the Roll ContRoll approach to fast lap times were discussed above. The basis of overall steering control was adapted from the basic PID controller provided in the ROAR platform. One issue encountered was that aggressive tuning (high gain) in this controller resulted - as expected from control theory - in instability, resulting physically in the car oscillating back and forth across the desired path of travel until it crashes, turns around, or a new input reset the oscillation. The gain had to be tuned to keep the car tracking the desired path satisfactorily, while not introducing detrimental oscillations.

Conclusion

In summary, the Roll ContRoller agent implemented a PID steering control (essentially PD as the integral gain was not really needed), with throttle control being implemented by the Roll ContRoller algorithm combined with a speed-sensitive look ahead for turns. Tuning of the PD gains, look-ahead speed and throttle factors, and the roll factor were accomplished experimentally to satisfactory performance in terms of speed and safety.

There is further room for improvement, refinement, and experimentation. Machine learning could be applied to tune all of the control factors, which would optimize performance. There are other steering control solutions that may offer superior integration possibilities - for example a Stanley inspired control which implemented the look ahead calculation to widen the vehicle's approach to turns was attempted, but not adequately finalized to enter the competition. If finalized, it is hoped this will allow even higher speed through corners, significantly reducing lap times. The waypoint refinement could also be implemented in the agent, and then this agent version tuned with machine learning to see if it resulted in improvement. Finally, using an accelerometer to gauge cornering limits would likely prove superior to using roll, especially for vehicles with aggressive suspension tuning (this race used a passenger car tuning, that offered plenty of body roll for application in the Roll ContRoller algorithm).

Control Module Code

```
class PIDRollController(Controller):
    def __init__(self, agent, steering_boundary: Tuple[float, float],
                 throttle_boundary: Tuple[float, float], **kwargs):
        super().__init__(agent, **kwargs)
        self.max_speed = math.ceil(2*self.agent.agent_settings.max_speed)
        self.throttle_boundary = throttle_boundary
        self.steering_boundary = steering_boundary
        self.config = json.load(Path(agent.agent_settings.pid_config_file_path).open(mode='r'))
        self.long_pid_controller = LongPIDController(agent=agent,
                                                     throttle_boundary=throttle_boundary,
                                                     max_speed=self.max_speed,
                                                     config=self.config["longitudinal_controller"])
        self.lat_pid_controller = LatPIDController(
            agent=agent,
            config=self.config["latitudinal_controller"],
            steering_boundary=steering_boundary
        )
        self.logger = logging.getLogger(__name__)

    def run_in_series(self, next_waypoint: Transform, **kwargs) -> VehicleControl:
        throttle = self.long_pid_controller.run_in_series(next_waypoint=next_waypoint,
                                                          target_speed=kwargs.get("target_speed",
self.max_speed))
        steering = self.lat_pid_controller.run_in_series(next_waypoint=next_waypoint)
        return VehicleControl(throttle=throttle, steering=steering)

    @staticmethod
    def find_k_values(vehicle: Vehicle, config: dict) -> np.array:
        current_speed = Vehicle.get_speed(vehicle=vehicle)
        k_p, k_d, k_i = .5, 0.1, 0
        for speed_upper_bound, kvalues in config.items():
            speed_upper_bound = float(speed_upper_bound)
            if current_speed < speed_upper_bound:
                k_p, k_d, k_i = kvalues["Kp"]*.4, kvalues["Kd"]*.3, kvalues["Ki"]*.05 #******* lowered gain for
smoothness
                break
        return np.clip([k_p, k_d, k_i], a_min=0, a_max=1)


# *** original Roll ContRoller + v2 ***
class LongPIDController(Controller):
    def __init__(self, agent, config: dict, throttle_boundary: Tuple[float, float], max_speed: float,
                 dt: float = 0.03, **kwargs):
        super().__init__(agent, **kwargs)
        self.config = config
        self.max_speed = max_speed
        self.throttle_boundary = throttle_boundary
        self._error_buffer = deque(maxlen=10)

        self._dt = dt

    def run_in_series(self, next_waypoint: Transform, **kwargs) -> float:
        target_speed = min(self.max_speed, kwargs.get("target_speed", self.max_speed))
        # self.logger.debug(f"Target_Speed: {target_speed} | max_speed = {self.max_speed}")
        current_speed = Vehicle.get_speed(self.agent.vehicle)

        print('max speed: ',self.max_speed)

        k_p, k_d, k_i = PIDRollController.find_k_values(vehicle=self.agent.vehicle, config=self.config)
        error = target_speed - current_speed

        self._error_buffer.append(error)


        #****************** implement look ahead ********************
        la_err = self.la_calcs(next_waypoint)
        # kla = .09
        #kla = 1/11000 # *** calculated ***
        kla = 1/10000 # *** tuned ***

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

```
            _de = 0.0
            _ie = 0.0
        # output = float(np.clip((k_p * error) + (k_d * _de) + (k_i * _ie), self.throttle_boundary[0],
        #                        self.throttle_boundary[1]))
        print(self.agent.vehicle.transform.rotation.roll)
        vehroll = self.agent.vehicle.transform.rotation.roll
        if current_speed >= (target_speed + 2):  # *** reduces speed at max limit more smoothly
            out = 1 - .08 * (current_speed - target_speed)
        # *** old guesses ***
        # else:
        #     if abs(self.agent.vehicle.transform.rotation.roll) <= .35:
        #         out = 6 * np.exp(-0.05 * np.abs(vehroll))-(la_err/180)*current_speed*kla
        #     else:
        #         out = 2 * np.exp(-0.05 * np.abs(vehroll))-(la_err/180)*current_speed*kla #
*****ALGORITHM*****
       # *** calculated formula ***
        else:
            if abs(self.agent.vehicle.transform.rotation.roll) <= 1.2:
                out = 2 * np.exp(-.03 * np.abs(vehroll))-la_err*current_speed*kla
            else:
                out = np.exp(-.06 * np.abs(vehroll))-la_err*current_speed*kla # *****ALGORITHM*****

        output = np.clip(out, a_min=0, a_max=1)
        print('*************')
        print('vehroll:',vehroll)
        print('unclipped throttle = ',out)
        print('throttle = ', output)
        print('*************')


        return output

    def la_calcs(self, next_waypoint: Transform, **kwargs):

        current_speed = int(Vehicle.get_speed(self.agent.vehicle))
        cs = np.clip(current_speed, 70, 200)

        la_indx = 43 #coarse points


        lf1 = math.ceil(2*cs/la_indx)
        lf2 = math.ceil(3*cs/la_indx)
        print ('++++++++++ curr wp indx: ',self.agent.local_planner.get_curr_waypoint_index()+lf2+4)
        print ('length wp queue',len(self.agent.local_planner.way_points_queue) )
        if self.agent.local_planner.get_curr_waypoint_index()+lf2+4<=\
            len(self.agent.local_planner.way_points_queue):

            next_pathpoint1 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf1])
            next_pathpoint2 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf1+1])
            next_pathpoint3 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf1+2])
            next_pathpoint4 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf2+1])
            next_pathpoint5 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf2+2])
            next_pathpoint6 = (self.agent.local_planner.way_points_queue\
                [self.agent.local_planner.get_curr_waypoint_index()+lf2+3])


            print('next waypoint: ',
self.agent.local_planner.way_points_queue[self.agent.local_planner.get_curr_waypoint_index()])
            print('$$$$$$$$$$$$$way points length:
',self.agent.local_planner.get_curr_waypoint_index(),'/',len(self.agent.local_planner.way_points_queue))

            nx0 = next_pathpoint1.location.x
            nz0 = next_pathpoint1.location.z
            nx = (
                            next_pathpoint1.location.x + next_pathpoint2.location.x +
next_pathpoint3.location.x + next_pathpoint4.location.x + next_pathpoint5.location.x +
next_pathpoint6.location.x) / 6
            nz = (
                            next_pathpoint1.location.z + next_pathpoint2.location.z +
next_pathpoint3.location.z + next_pathpoint4.location.z + next_pathpoint5.location.z +
next_pathpoint6.location.z) / 6
            nx1 = (next_pathpoint1.location.x + next_pathpoint2.location.x + next_pathpoint3.location.x) / 3
            nz1 = (next_pathpoint1.location.z + next_pathpoint2.location.z + next_pathpoint3.location.z) / 3
            nx2 = (next_pathpoint4.location.x + next_pathpoint5.location.x + next_pathpoint6.location.x) / 3
            nz2 = (next_pathpoint4.location.z + next_pathpoint5.location.z + next_pathpoint6.location.z) / 3
```

```
            npath0 = np.transpose(np.array([nx0, nz0, 1]))
            npath = np.transpose(np.array([nx, nz, 1]))
            npath1 = np.transpose(np.array([nx1, nz1, 1]))
            npath2 = np.transpose(np.array([nx2, nz2, 1]))

            path_yaw_rad = -(math.atan2((nx2 - nx1), -(nz2 - nz1)))

            path_yaw = path_yaw_rad * 180 / np.pi
            print(' !!! path yaw !!! ', path_yaw)

            veh_yaw = self.agent.vehicle.transform.rotation.yaw
            print(' !!! veh yaw  !!! ', veh_yaw)
            ahead_err = abs(abs(path_yaw)-abs(veh_yaw))

        else:
            ahead_err = 105

        if ahead_err < 60:
            la_err = 0
        else:
            la_err =(.05 * ahead_err)**3

        print('-------------------------------------')


        print('** la err **', la_err)
        print('-------------------------------------')

        return la_err

        #*************************************************************

# ***** end original version Roll ContRoller *****


class LatPIDController(Controller):
    def __init__(self, agent, config: dict, steering_boundary: Tuple[float, float],
                 dt: float = 0.03, **kwargs):
        super().__init__(agent, **kwargs)
        self.config = config
        self.steering_boundary = steering_boundary
        self._error_buffer = deque(maxlen=10)
        self._dt = dt

    def run_in_series(self, next_waypoint: Transform, **kwargs) -> float:
        """
        Calculates a vector that represent where you are going.
        Args:
            next_waypoint ():
            **kwargs ():
        Returns:
            lat_control
        """
        # calculate a vector that represent where you are going
        v_begin = self.agent.vehicle.transform.location.to_array()

        print(v_begin)
        print('next wp x: ', next_waypoint.location.x)
        print('next wp z: ', next_waypoint.location.z)
        print('next wp y: ', next_waypoint.location.y)

        direction_vector = np.array([-np.sin(np.deg2rad(self.agent.vehicle.transform.rotation.yaw)),
                                     0,
                                     -np.cos(np.deg2rad(self.agent.vehicle.transform.rotation.yaw))])

        v_end = v_begin + direction_vector

        v_vec = np.array([(v_end[0] - v_begin[0]), 0, (v_end[2] - v_begin[2])])
        # calculate error projection
        w_vec = np.array(
            [
                next_waypoint.location.x - v_begin[0],
                0,
                next_waypoint.location.z - v_begin[2],
            ]
        )

        v_vec_normed = v_vec / np.linalg.norm(v_vec)
        w_vec_normed = w_vec / np.linalg.norm(w_vec)
        error = np.arccos(v_vec_normed @ w_vec_normed.T)
        _cross = np.cross(v_vec_normed, w_vec_normed)
```

```
        if _cross[1] > 0:
            error *= -1
        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

        k_p, k_d, k_i = PIDRollController.find_k_values(config=self.config, vehicle=self.agent.vehicle)
        print ('kp, kd, ki: ', k_p, k_d, k_i)
        lat_control = float(
            np.clip((k_p * error) + (k_d * _de) + (k_i * _ie), self.steering_boundary[0],
self.steering_boundary[1])
        )
        # lat_control = float(
        #     np.clip((k_p * error) + (k_d * _de) + (k_i * _ie), -.9, .9)
        # )

        print('lateral control:',lat_control)
        return lat_control
```