Model Predictive Control for F1/10 Race Car

Team Emu Imu: Becky Abramowitz, Vaibhav Arcot, Hunter Lightman, Matt Oslin

Abstract— **This paper explains our approach to a Model Predictive Control based algorithm to compete in head to head racing of F1/10 Autonomous Race Cars. The model relies on adapted bicycle model dynamics and a convex workspace formed by two receding lines from the car to constrain a linear optimization function that minimizes the deviation from predefined waypoints. The method described was successful in racing and was able to operate at speeds of up to 4 m/s and compete at 1.5 m/s in head-to-head race situations.**

I. INTRODUCTION

The goal of the final project was to implement a race technique that would make our car capable of being successful in a head-to-head race. Our team chose to implement Model Predictive Control (MPC) to follow a series of pre-existing waypoints given constraints on feasible dynamics and nearby obstacles.

A. Background on Model Predictive Control

Model Predictive Control is an approach to a receding horizon optimization problem that incorporates constraints [4]. The optimization problem used in MPC is often of a similar form to the Linear Quadratic Regulator (LQR), which is defined by state dynamics and quadratic cost. However, the benefit of LQR is that it has a closed form solution in the Riccati equation, but this solution does not account for system constraints. Therefore, an MPC problem must be solved with an external optimization solver.

Although LQR itself cannot be used, it is reasonable to structure an online MPC problem as having linear dynamics and quadratic cost. The rationale for linear dynamics is two-fold. Linear dynamics are significantly less computationally expensive than non-linear dynamics, and can also be represented in a matrix equation $Ax \leq b$ where A is a constraint matrix, x is the state vector, and b is a constraint vector. Similarly, a quadratic cost means that the cost incurred depends on the magnitude of values rather than their signs. The cost function with a state vector x and control input vector u usually takes the form

$$\mathcal{J} = x^\top Qx + u^\top Ru. \tag{1}$$

where Q is the cost on state and R is the cost on control. The Q and R matrices must be positive definite (or positive semidefinite for some optimization solvers) and are usually diagonal in form, with each value corresponding to the cost weight applied to that element of state or control. A optimization problem that looks to optimize a quadratic cost would be of the form

$$\min_{x,u} x^\top Qx + u^\top Ru. \tag{2}$$

The function detailed above gives the cost for a single state of the system, and we want to plan ahead and compute a trajectory that optimizes progress towards an end goal. For a discrete, infinite time problem, this would yield the optimization problem

$$\min_{x,u} \sum_{i=0}^{\infty} x_i^\top Qx_i + u_i^\top Ru_i. \tag{3}$$

The subscript "$_i$" means that the value is taken for a specific knot point i in time.

However, a system with no clear endpoint cannot be computed infinitely far in advance, and even if it could, the operation would be slow. Therefore, an MPC problem should be dealt with in a receding horizon approach. The system looks some N steps or T timespan ahead and computes the cost function along steps in increments of dt. The state at these times i are included in the state vector x_i, and the control input applied is u_i. The cost from the last point is included in a third matrix Q_N, called the cost to go which incorporates the cost incurred by ending at the final state, as shown in the optimization problem

$$\min_{x_{1:N}, u_{0:N-1}} \sum_{i=0}^{\infty} x_i^\top Qx_t = i + u_i^\top Ru_i + x_N^\top Q_N x_N. \tag{4}$$

This cost to go matrix is usually weighted higher than the the Q matrix as it is imperative for keeping the system on track. This was also found in our tuning, as discussed below.

B. Problem Definition

The first step in defining our MPC problem was to define our system, particularly the constituents of the state vector x and the control vector u, as well as the a model to represent the system dynamics.

Inspired by a bicycle model, as shown in Fig. 1, the system was defined with the state definition shown in (5).

$$x_i = \begin{bmatrix} x_i \\ y_i \\ \theta_i \\ v_i \\ \phi_i \end{bmatrix} \tag{5}$$

In this definition, x_i and y_i are the coordinates of the car (the LiDAR) in global space, θ_i is the heading of the car, v_i is the velocity of the car defined by the forwards linear velocity of the driving rear wheels, and ϕ_i is the steer angle of the front wheels.

Rather than driving the car with the velocity and steer angle, as done in the Pure Pursuit assignment earlier in the

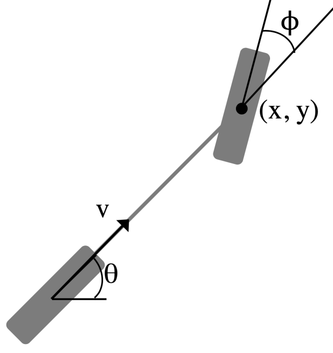

Fig. 1. Bicycle Model of Car

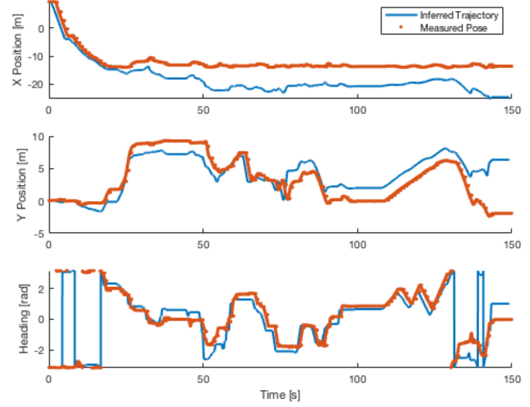

Fig. 2. Dynamic Model Verification

course, we followed a more traditional approach and used the control vector shown in (6) which uses the linear acceleration of the rear wheels and the rotational velocity of the front wheels.

$$u_i = \begin{bmatrix} \dot{v}_i \\ \dot{\phi}_i \end{bmatrix} \tag{6}$$

We can combine the state and control vectors into a single state vector X which contains the state of the car at the next N projected knot points. This resulting vector is of the form shown in (7).

$$X = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ \dots \\ x_{N-1} \\ u_{N-1} \\ x_N \end{bmatrix} \tag{7}$$

C. Dynamics Constraints

There are two challenges to determining the dynamics constraints of the system: fitting a linear model and choosing what to linearize about. Both will be discussed in this section.

1) Bicycle Dynamics: Using our bicycle model in Fig. 1, our system dynamics are of the form shown in (8) where $l = 0.3\text{m}$ is the distance from the front to the rear wheel.

$$\begin{aligned} x_{i+1} &= x_i + dt \cdot v_i \cdot \cos(\theta_i + \phi_i) \\ y_{i+1} &= y_i + dt \cdot v_i \cdot \sin(\theta_i + \phi_i) \\ \theta_{i+1} &= \theta_i + dt \cdot v_i \cdot \sin(\phi_i)/l \\ v_{i+1} &= v_i + dt \cdot \dot{v}_i \\ \phi_{i+1} &= \phi_i + dt \cdot \dot{\phi}_i \end{aligned} \tag{8}$$

We compared the predicted trajectory given this model and a series of control inputs to the actual measured pose of the car in Fig. 2. We were particularly pleased with the drift rejection over two and a half minutes of driving.

2) Linear Model: To get a linear model from (8), we chose to linearize around error coordinates with $v_i^*, \theta_i^*, \phi_i^*$ as the desired states, meaning that errors in θ and ϕ would be small and small angle approximations would be valid. This led to the dynamical model shown in (9).

$$\begin{aligned} x_{i+1} = x_i &+ dt \cdot (v_i \cdot \cos(\phi_i^* + \theta_i^*) - \\ &\theta_i \cdot v_i^* \cdot \sin(\phi_i^* + \theta_i^*) + v_i^* \cdot \sin(\phi_i^* + \theta_i^*) \cdot \\ &(\phi_i^* + \theta_i^*) - \phi_i \cdot v_i^* \cdot \sin(\phi_i^* + \theta_i^*)) \\ y_{i+1} = y_i &+ dt \cdot (v_i \cdot \sin(\phi_i^* + \theta_i^*) + \\ &\theta_i \cdot v_i^* \cdot \cos(\phi_i^* + \theta_i^*)) - v_i^* \cdot \cos(\phi_i^* + \theta_i^*) \cdot \\ &(\phi_i^* + \theta_i^*) + \phi_i \cdot v_i^* \cdot \cos(\phi_i^* + \theta_i^*) \\ \theta_{i+1} = \theta_i &+ (dt \cdot (v_i \cdot \sin(\phi_i^*) - \phi_i^* \cdot v_i^* \cdot \cos(\phi_i^*) + \\ &\phi_i \cdot v_i^* \cdot \cos(\phi_i^*)))/l \\ v_{i+1} = v_i &+ dt \cdot \dot{v}_i \\ \phi_{i+1} = \phi_i &+ dt \cdot \dot{\phi}_i \end{aligned} \tag{9}$$

However, these dynamics did not work well on the car, so we instead linearize around the starting heading by introducing coordinates $\bar{\theta}_i = \theta_i - \theta_0, \bar{\phi}_i = \phi_i - \phi_0$. Since our horizon is relatively short, we can assume $\bar{\theta}_i \approx 0, \bar{\phi}_i \approx 0$. To make our dynamics linear, we also assume $v_i \approx \bar{v}_i$, given by (10).

$$\begin{aligned} \bar{v}_i &= \begin{cases} \min(\bar{v}_{i-1} + dt \cdot \dot{v}_{max}, v_i^*) & \bar{v}_{i-1} \leq v_i^* \\ \max(\bar{v}_{i-1} - dt \cdot \dot{v}_{max}, v_i^*) & \bar{v}_{i-1} > v_i^* \end{cases} \\ \bar{v}_0 &= v_0 \end{aligned} \tag{10}$$

Now we can rewrite our system dynamics in the form (11).

$$\begin{aligned} x_{i+1} &\approx x_i + dt \cdot \bar{v}_i \cdot \cos(\phi_0 + \theta_0 + \bar{\theta}_i + \bar{\phi}_i) \\ &= x_i + dt \cdot \bar{v}_i \cdot (\cos(\theta_0 + \phi_0) \cos(\bar{\theta}_i + \bar{\phi}_i) - \\ &\quad \sin(\theta_0 + \phi_0) \sin(\bar{\theta}_i + \bar{\phi}_i)) \\ y_{i+1} &\approx y_i + dt \cdot \bar{v}_i \cdot \sin(\phi_0 + \theta_0 + \bar{\theta}_i + \bar{\phi}_i) \\ &= y_i + dt \cdot \bar{v}_i \cdot (\sin(\theta_0 + \phi_0) \cos(\bar{\theta}_i + \bar{\phi}_i) + \\ &\quad \cos(\theta_0 + \phi_0) \sin(\bar{\theta}_i + \bar{\phi}_i)) \\ \theta_{i+1} &\approx \theta_i + dt \cdot \bar{v}_i \cdot \sin(\phi_0 + \bar{\phi}_i)/l_r \\ &= \theta_i + dt \cdot \bar{v}_i \cdot (\sin(\phi_0) \cos(\bar{\phi}_i) + \\ &\quad \cos(\phi_0) \sin(\bar{\phi}_i)) \end{aligned} \tag{11}$$

If we apply small angle approximations we get the model shown in (12).

$$\begin{aligned}
x_{i+1} &\approx x_i + dt \cdot \bar{v}_i \cdot (\cos(\theta_0 + \phi_0) + \\
&\quad \sin(\theta_0 + \phi_0) \cdot (\theta_0 + \phi_0) - \\
&\quad \sin(\theta_0 + \phi_0) \cdot (\theta_i + \phi_i)) \\
y_{i+1} &\approx y_i + dt \cdot \bar{v}_i \cdot (\sin(\theta_0 + \phi_0) - \\
&\quad \cos(\theta_0 + \phi_0) \cdot (\theta_0 + \phi_0) + \\
&\quad \cos(\theta_0 + \phi_0) \cdot (\theta_i + \phi_i)) \\
\theta_{i+1} &\approx \theta_i + dt \cdot \bar{v}_i \cdot (\sin(\phi_0) + \cos(\phi_0) \cdot \bar{\phi}_i)/l \\
v_{i+1} &= v_i + dt \cdot \dot{v}_i \\
\phi_{i+1} &= \phi_i + dt \cdot \dot{\phi}_i
\end{aligned} \tag{12}$$

These new system dynamics are linear and can be expressed in the form $A(x_0) \cdot X = b(x_0)$.

D. State and Control Bounds

To prevent our optimizer from choosing infeasible control inputs, we limited the linear acceleration and turning rate as well as the the velocity and steer angle states, shown in (13).

$$\begin{aligned}
-\dot{v}_{max} &\le \dot{v}_i \le \dot{v}_{max} \\
-\dot{\phi}_{max} &\le \dot{\phi}_i \le \dot{\phi}_{max} \\
-v_{max} &\le v_i \le v_{max} \\
-\phi_{max} &\le \phi_i \le \phi_{max}
\end{aligned} \tag{13}$$

These constraints can be combined into the form $A \cdot X \le b$.

E. Target Point Selection

To choose our desired state vector X^*, we assume our desired control input is zero. To find our desired x_i, y_i, θ_i, we select waypoints that our feasibly reachable by assuming we accelerate to the desired maxVelocity along the given waypoints. This is shown in Alg. 1.

Algorithm 1 GetTargetWaypoints

```
Input: Car, Waypoints
Output: Target Points
  targetPoints = []
  closestIdx ← None
  closestDist ← inf
  for waypoint in Waypoints do
    if dist(waypoint, car) < closestDist then
      closestIdx ← waypoint index
      closestDist ← dist(waypoint, car)
    end if
  end for
  travelledDist ← 0
  vel ← car.v
  targetDist ← 0
  targetPoints[0] ← waypoints[closestIdx]
  for i in 1 to N do
    targetDist ← targetDist + vel × dt
    vel ← min(vel + maxVDot × dt, maxVelocity)
    while travelledDist < targetDist do
      travelledDist += dist(waypoints[closestIdx], way-
      points[closestIdx + 1] )
      increment closestIdx
    end while
    targetPoints[i] ← waypoints[closestIdx]
  end for
  return targetPoints
```

F. Workspace Constraints

With a working set of dynamics constraints, we were able to follow a predefined series of waypoints. However, this was effectively another implementation of a blind pure pursuit; it did not account for any of the LiDAR data or any obstacles that may appear along the car's path. It is thus important to define workspace constraints, which are derived from the LiDAR data and which enter the optimization solver along with the dynamics constraints to limit the car's path to a physically feasible area.

Due to the nature of the optimization solver, the working area needs to be defined by linear bounds so as to form linear constraints, and since all constraints must be active, the limited area must be defined to be convex. It is important that this area is sufficiently large that the car can create a path within it, but also be sufficiently small that the car does not intersect any obstacles.

The first approach that we tried was to grow the kernel found with [3] from the star-shaped polygon created by the LiDAR data and use that to get an approximation of the maximum sized convex shape as done in [1]. This is a faster approach, with a runtime of $\mathcal{O}(n + k \log k)$, where n is the total number of points in the scan and k is the number of reflex points, compared to the standard $\mathcal{O}(n^7)$ solution to the "Potato Peeling" Problem. However, this approach was complex and the largest sized polygon would not always yield the best constraints for a race situation. Therefore, we decided to pick a simpler approach.

Our simplified approach in Alg. 2 resembled the gap finding method that we solved earlier in the semester and produces two linear constraints that define the workspace. These constraints are found given a lookahead distance by scanning the LiDAR data to find two consecutive points that are the greatest lateral distance apart with one point being on either side of the lookahead distance away longitudinally. The search is started at a nominal lookahead distance, and if no viable gaps are found the distance is shortened. This method tends to find the widest gaps that are far enough ahead to act on.

Hysteresis is introduced by taking the gap closest to the previous best gap when more than one gap are ε similar in size. The gap is also shrunk by δ laterally on each point to ensure buffer between the car and the obstacle, and gaps are transformed into the map frame.

Given the two points from the gap $g = [x_{p1}, y_{p1}, x_{p2}, y_{p2}]$ and the car's current location, the next step was to convert these to linear constraints. We modeled these two points and the car as shown in Fig. 3. These constraints then followed the linear equations shown in (14) in global space relative to

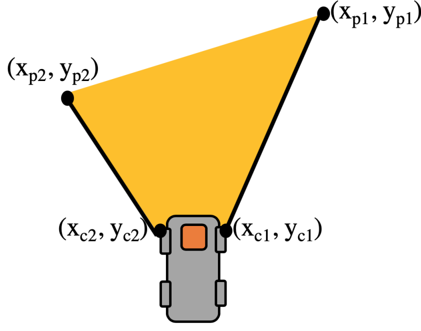

Fig. 3. Drivable Workspace Diagram

the map frame.

$$\begin{aligned} x(y_{c1}-y_{p1})+y(x_{p1}-x_{c1}) &\leq y_{c1}x_{p1}-y_{p1}x_{c1} \\ x(y_{p2}-y_{c2})+y(x_{c2}-x_{p2}) &\leq y_{p2}x_{c2}-y_{c2}x_{p2} \end{aligned} \tag{14}$$

where $c1, c2$ are always a fixed distance r laterally to the car. The constraints in (14) must be true for all x_i and y_i values rolled forward in time in the MPC and can be written in the form $A(x_0, g) \cdot X \leq b(x_0, g)$.

II. Our MPC Algorithm

Our Model Predictive Control algorithm was designed with the above definition of state and constraints. Our methodology is shown in the high-level psuedo-code shown in Alg. 4 and the MPC problem is solved in in Alg. 3. It is useful to note that our optimization approach uses the `quadprog` quadratic optimization solver (instructions to download in *Necessary Packages* section below) which solves the problem defined in (15) using the algorithm in [2].

$$\begin{aligned} &\min_X \frac{1}{2} X^\top G X - a^\top X \\ &\text{subject to } C^\top X \geq b \end{aligned} \tag{15}$$

where the first m columns of C are equality constraints. The definition used in `quadprog` is syntactically different than the minimization problem detailed above because it minimizes in error coordinates, therefore X^*, which represents the desired states from our waypoints, is included directly into the formulation (16).

$$\begin{gathered} (X-X^*)^\top G(X-X^*) \\ X^\top GX-(X^*)^\top GX-X^\top GX^*-(X^*)^\top GX^* \\ X^\top GX-2(X^*)^\top GX-(X^*)^\top GX^* \\ X^\top GX-2(X^*)^\top GX \\ X^\top GX-2aX \text{ where } a=(X^*)^\top G \end{gathered} \tag{16}$$

Since the problem is a minimization, a scalar multiple of 2 and removing constants does not change the resulting solution.

Asynchronously we are constantly maintaining the car's current state by listening to messages published by the vesc and the particle filter.

Our MPC algorithm runs an optimization to minimize the distance of the car from a series of pre-set waypoints. These

Algorithm 2 Find Gap

Input: Car (with current v, ϕ, x, y, θ, transformation from `laser` to `map` frames), LaserScan (with longitudinal, lateral distance x, y and heading theta for each point), PrevGapHeading
Output: Gap

```
lookahead ← lookaheadStart
bestGaps = []
bestSize ← 0
while lookahead > 0 do
  for consecutive points a,b in LaserScan do
    if a.x < lookahead and b.x > lookahead then
      if abs(a.y − b.y) > minGapSize then
        bestGaps[end] ← (a,b)
        if abs(a.y − b.y) > bestSize then
          bestSize ← abs(a.y − b.y)
        end if
      end if
    end if
  end for
  if bestSize > 0 then
    bestGap ← None
    closestDist ← inf
    for gap in bestGaps do
      dist ← abs(mean(gap.a.theta, gap.b.theta) − PrevGapHeading)
      if abs(gap.a.y − gap.b.y) > bestSize − epsilon and
      dist < closestDist then
        closestDist ← dist
        bestGap ← gap
      end if
    end for
    return bestGap
  end if
  lookahead ← lookahead − increment
end while
```

Algorithm 3 MPC Approach

Input: Car (with current v, ϕ, x, y, θ, transformation from `laser` to `map` frames), Waypoints, G
Output: ResultPolicy

```
targetPoints ← GetTargetWaypoints(Car, Waypoints) [1]
Xdes ← targetPoints in state space
a ← calculated from G and Xdes (16)
A1, B1 ← from dynamics constraints (12)
A2, B2 ← from state and control bounds (13)
A3, B3 ← from workspace constraints (14)
C ← concatenation of A1, A2, A3
b ← concatenation of B1, B2, B3
solve with quadprog and return resultPolicy
```

Algorithm 4 MPC Loop

Input: CurrentControlPolicy
  while true **do**
    if mpc solver thread is finished **then**
      currentControlPolicy ← resultPolicy
      spawn a new mpc solver thread
    else
      apply next control from currentControlPolicy
    end if
    wait dt time
  end while

waypoints are brought in from a CSV file which contains the global x, y, and θ values for the car at these waypoints along with an ”aggro scale” telling the car how quickly it can relatively move at that locale. The target waypoints are selected as shown in Alg 1. The algorithm first looks for the closest waypoint to the current location of the car (lines 4-9 of Alg 1). It then rolls the dynamics forward to find the distance that the car should be from its current location at each of the lookahead knot points and finds the waypoint one step back from that distance. A more comprehensive implementation could interpolate between waypoints, but we defined a smoothed and very dense waypoint path so as to mitigate the effects of this assumption.

A. Parameters

There were several sets of parameters that needed to be tuned for a working MPC algorithm, which are detailed in sections below.

The first subset of parameters were car-specific and were used in the car dynamics calculations. These came from experimentation on the car and are defined in Table I.

TABLE I
CAR PARAMETERS

Param	Value	Meaning
v_{max}	5	Max Linear Velocity, Rear Wheels $(\mathrm{m/s})$
ϕ_{max}	0.34	Max Angular Position, Steering (rad)
$\dot{v}_{max}$	100	Max Linear Acceleration, Rear Wheels $(\mathrm{m/s^2})$
$\dot{\phi}_{max}$	3.2	Max Angular Velocity, Steering $(\mathrm{rad/s})$

As with any optimization problem, the weight matrix G also required tuning. The G matrix is diagonal of size n by n where n is the size of the total state vector X and each element indexed at (i, i) is the weight of that element in the state vector. We kept the weight of each parameter the same at all knot points spare the last, and therefore we only had to tune one weight for each element of the state in addition to the final cost-to-go values. The cost-to-go is included in the final rows of the G matrix which weight the values of x_N. The resulting values for each of these weights, as well as their meanings are defined in Table II.

There were some other, more system-level parameters that were also tuned for the MPC approach. These are detailed in Table III.

TABLE II
STATE PARAMETERS

Param	Value	Meaning
W_x	20	Weight of x error at knot point
W_y	20	Weight of y error at knot point
W_θ	50	Weight of θ error at knot point
W_v	2	Weight of v error at knot point
W_ϕ		Weight of ϕ error at knot point
$W_{\dot{v}}$	0.01	Weight of $\dot{v}$ at knot point
$W_{\dot{\phi}}$	0.01	Weight of $\dot{\phi}$ at knot point
W_{x_N}	5	Weight of final x error
W_{y_N}	5	Weight of final y error
W_{θ_N}	1000	Weight of final θ error
W_{v_N}	2	Weight of final v error
W_{ϕ_N}		Weight of final ϕ error

TABLE III
STATE PARAMETERS

Param	Value	Meaning
dt	0.1	Timestep (s)
T	0.5	Number of seconds to look ahead (s)
maxVelocity	2	Desired max speed when going straight
aggroScale	1	Scale factor for turn slowdown
lookahead	2	Obstacle Look Ahead Distance (m)

All of the aforementioned parameters can be found and set in the `optimizer.py` file.

B. Speed

We designed our algorithm with speed in mind by using a simple quadratic solver and linearizing the dynamics so as to minimize computation time. To our surprise, the optimization solver ran faster than the code used to set the constraints. The optimization solver itself was able to consistently run in under 10 milliseconds, and the whole MPC stack runs at around 20 Hz on the TX2 processor. Since our final time step size was 100 milliseconds, this gave us a nice buffer as our actual controls were applied with a 10 Hz frequency.

We did use multi-threading on the MPC solver to ensure that the computations were done as quickly as possible. This also let us avoid blocking the main MPC loop with our solver—if a solve took especially long, we could keep applying the old control policy until the solution was ready. Additional improvements could be made by switching our code from Python to C++ but our speed was sufficient for the problem we were trying to solve. Additional gains could have been achieved by offloading matrix computation to the GPU, but since most of the heavy computation occurred within numpy (for our calculations) or quadprog (for the optimizer specific calculations), and because we were achieving sufficient speeds, it did not seem necessary to add in the extra complexity of finding/writing GPU accelerated code.

C. Necessary Packages

We used the `quadprog` optimization package for Python. The package can be downloaded with the command `pip install quadprog` and further information on this package can be found at https://pypi.org/project/quadprog/.

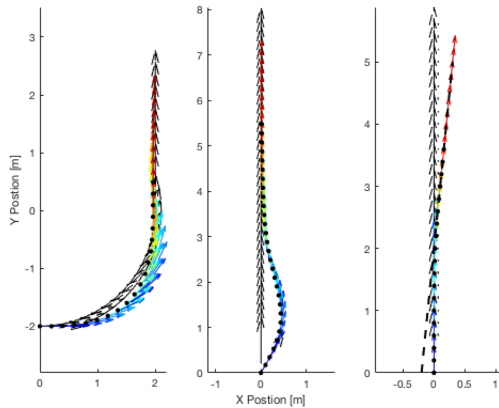

Fig. 4. MPC Problem Verification

D. Testing

Due to the unrealistic nature of our Gazebo environment (which often has the car doing somersaults and other infeasible 3D maneuvers), most of the testing on the car was done in the real world. The testing was done in various levels in order to ensure that the foundations were working well before adding a new layer and are detailed in the following paragraphs.

1) Matlab Verification: To esnure we had properly defined our optimization problem, we implemented our nonlinear system dynamics and MPC solution in Matlab. Various waypoints were tested, shown in Fig. 4, including going around a turn, handling an initial heading error, and avoiding a linear constraint. The desired trajectory is outlined with black arrows, the MPC planned trajectories are plotted in rainbow with once color for each control policy, and the actual trajectory is marked with black dots.

2) Localization: The first tests were performed on the car's localization within the track. This proved at times problematic, since the walls of the track did not provide absolute barriers to the LiDAR and the scans sometimes returned values outside of the track walls. This phase of testing involved running Google Cartographer and manually editing the resulting map in Adobe Photoshop to get a map that meshed well with the surroundings. We also developed some intuition on the locations in which the car loses its localization which was useful in debugging future cases.

3) Optimization Approach: Once we had developed the MPC algorithm, we ran tests without the regional constraints. This was useful for tuning the cost matrix for the car to best follow the desired waypoints. Perhaps our biggest realization in this phase was the need to put a large weight on the final heading of the car; otherwise the weight on control authority needed for the car to turn kept the car straight and too little weight on control left the car fishtailing. These tests were done on the racetrack, looking at our visualization in RVIZ.

4) Regional Constraints: The final set of testing was performed with obstacles in the car's workspace and the regional constraints added to the MPC solver. These obstacles were originally static boxes, and as we further tuned our system, we were able to make the obstacles quasi-dynamic by moving the boxes around and walking in line with the car.

5) Speed: We also tested the car (albeit briefly) for speed by running quickly around the racetrack and determining what aggro_scale values were necessary for the car to drive (sans obstacles) at varying velocities.

III. Raceday Results

On race day, our car successfully navigated the track autonomously using the MPC algorithm, both with and without obstacles. Without obstacles, we were able to run the car at a max speed of 4 m/s on the straight-aways for lap times a little over 6 seconds. The car was certified for head to head racing and competed against the Ninja Turtles in both autonomous and manual driving modes.

A feature of the MPC algorithm approach that we used is that our car hesitates when it does not immediately see a gap. From a computational standpoint, this situation occurs when the optimization problem is unable to be solved as the workspace of the car – as defined by the regional constraints – is too small. Although this would likely be a good feature from a safety standpoint, it makes our car slower in a race-track situation.

References

[1] D Coeurjolly and J-M Chassery. Fast approximation of the maximum area convex subset for star-shaped polygons, 2004.

[2] D Goldfarb and A Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, 27:1–33, Sept. 1983.

[3] D Lee and F Preparata. An optimal algorithm for finding the kernel of a polygon. *Journal of the ACM (JACM)*, 26:415–421, July 1979.

[4] D Mayne, J Rawlings, C Rao, and P Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789 – 814, 2000.