Autonomous Controls System for Global Formula Racing

by
Deven Bishnu

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented May 24, 2021
Commencement June 2021

AN ABSTRACT OF THE THESIS OF

Deven Bishnu for the degree of Honors Baccalaureate of Science in Computer Science presented on May 24, 2021. Title: Autonomous Controls System for Global Formula Racing.

Abstract approved:____

Bob Paasch

This project focuses on implementing a model predictive control (MPC) algorithm as a pathfinding algorithm for navigating tracks with the Oregon State University Global Formula Racing (GFR) team's driverless car. The new algorithm would use an optimization function to calculate the least-cost path for the car to take by assigning costs to actions such as braking and turning. The goal of this project was to improve lap times by increasing the car's average velocity while optimizing the paths taken through turns. This algorithm was chosen because it focuses on generating optimal racing lines rather than staying centered in the track, as well as because of the support for open-source implementations of the algorithm. The new code was added as a Robot Operating System (ROS) node in the existing autonomous controls codebase within the GFR autonomous team. Incorporation of this algorithm demonstrated the capability to match, but not yet substantially exceed the average velocity of the pure pursuit algorithm. This was due to a combination of steering issues at the start of tracks and limited top speeds. However, flexibility in the bounds set by MPC provides the groundwork for fine-tuning to overcome its current limitations.

Key Words: Global Formula Racing, Formula SAE, Model predictive control, Driverless, Autonomous, Controls

Corresponding e-mail address: bishnud@oregonstate.edu

©Copyright by Deven Bishnu
May 24, 2021

Autonomous Controls System for Global Formula Racing

by
Deven Bishnu

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Presented May 24, 2021
Commencement June 2021

Honors Baccalaureate of Science in Computer Science project of Deven Bishnu presented on May 24, 2021.

APPROVED:

Bob Paasch, Mentor, representing Mechanical Engineering

Cindy Grimm, Committee Member, representing Robotics

Connor Kurtz, Committee Member, representing Electrical Engineering

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, Honors College. My signature below authorizes release of my project to any reader upon request.

Deven Bishnu, Author

// 0. Abstract
This project focuses on implementing a model predictive control (MPC) algorithm as a pathfinding algorithm for navigating tracks with the Oregon State University Global Formula Racing (GFR) team's driverless car. The new algorithm would use an optimization function to calculate the least-cost path for the car to take by assigning costs to actions such as braking and turning. The goal of this project was to improve lap times by increasing the car's average velocity while optimizing the paths taken through turns. This algorithm was chosen because it focuses on generating optimal racing lines rather than staying centered in the track, as well as because of the support for open-source implementations of the algorithm. The new code was added as a Robot Operating System (ROS) node in the existing autonomous controls codebase within the GFR autonomous team. Incorporation of this algorithm demonstrated the capability to match, but not yet substantially exceed the average velocity of the pure pursuit algorithm. This was due to a combination of steering issues at the start of tracks and limited top speeds. However, flexibility in the bounds set by MPC provides the groundwork for fine-tuning to overcome its current limitations.

1. Project Description

1.1 Introduction

The Global Formula Racing (GFR) team is the product of collaboration between Oregon State University and Ravensburg University of Cooperative Education (*Duale Hochschule Baden-Württemberg Ravensburg*). Its purpose is to compete in Formula SAE events. Students from these universities design and build cars to fulfill the requirements of these events, which include driverless navigation of race tracks. The team follows a design philosophy of verifying the integrity of its systems through rigorous testing. In the case of driverless navigation, rigorous testing is done through replicating the car's behavior in simulation software. Driverless navigation is handled by the GFR autonomous subteam, of which the autonomous controls node - the focus of this paper - is one element of. The autonomous controls node performs pathfinding on a track, then determines the appropriate inputs to be sent to the car to allow it to follow this path.

This project focuses on implementing a model predictive control (MPC) algorithm as a pathfinding algorithm for navigating tracks with the GFR driverless car. MPC uses an optimization function to find the lowest-cost path around a track, where actions such as braking and steering are considered high-cost. It operates within a given set of bounds - for example, the track boundaries cannot be

exceeded - that limits the horizon of possible solutions to the optimization problem. MPC algorithms have been used in Formula SAE competitions before in cars with a similar software architecture to the GFR car [1], so testing its use in the GFR car is necessary to remain competitive in competitions.

The specific implementation of MPC used in this project is model predictive contouring control (MPCC), an open source project focusing on use of MPC in autonomous racing cars [2]. In addition to the optimization component of the algorithm, MPCC performs contouring within track limits to predict a path for the car to follow. The optimization component is then applied to this path to determine the least-cost route for the car to take. Because contouring is a time-consuming task, the algorithm places a low amount of effort on creating perfect contouring, instead emphasizing high-precision solutions to the optimization problem for following the contouring. The MPCC formulation focuses on following this path by maximizing progress along the contoured path in a minimum number of timesteps while being limited to track constraints, which is equivalent to maximizing progress along the track [3]. Imposing the limitation of track boundaries limits the risk of poor contouring resulting in the car exceeding track boundaries.

The resulting optimization function is quadratic, presenting a nonlinear programming problem. This means that the solution is derived from a system of quadratic equations with polynomials of degrees greater than 1. Liniger [3] presents two possibilities for solving the problem: linear-time-varying approximation and using a nonlinear programming solver. The former is computationally faster, but less precise and limits the number of possible constraints that can be applied to the car model. The latter is the implementation used in the MPCC codebase, and takes longer per timestep but improves driving performance and allows incorporation of additional friction forces in the car's tire model.

The autonomous navigation algorithm is used when the car is operating in driverless mode. The autonomous controls system consists of this algorithm and its input and output interfaces with other projects on the autonomous team. The system reads input information from cone positions on track maps generated by the car's simultaneous localization and mapping (SLAM) node, as well as state estimation data retrieved from the Robot Operating System (ROS) on the car's heading, velocity, and position. With this, the controls system calculates a path for the car to take and the adjustments to throttle, braking, and steering controls needed to follow that path. It then outputs the necessary adjustments to the actuator systems for these controls.

1.2 Requirements

The project is intended to be able to function with only the information other autonomous projects and ROS itself can provide, specifically current controls inputs, odometry information, and cone positions. The current controls inputs affect subsequent inputs to the controls. The project should output valid (numerical) values to the throttle and steering actuators at appropriate times, with steering being a value in degrees, and with throttle being a value representing target velocity in meters per second. The steering actuator attempts to correct the angle of the front tires to the target steering angle value, and the throttle actuator attempts to adjust velocity to the target throttle input value. Within the code, there is no limit to what these values should be at any given point in time, provided they are inside the configurable bounds defined in the algorithm's files. Variation in real cone positions compared to the SLAM map will necessarily decrease the error tolerances for controls inputs, as they are ultimately used to determine the car's relative position and movement within the track.

Code used in the project should be documented and organized. For code styling, file names, and directory names, code conforms to the ROS C++ Style Guide [4]. This guide generally recommends using lowercase, underscored names for files, and combining C-style header files with C++-style source files. It also utilizes camel case for a variety of code elements, including class names and function names, with most other elements being either all lowercase or all uppercase, and underscored. Documentation is done through the tool Doxygen [5], an open-source C++ automatic documentation generating tool. Code comments in the project therefore conform to Doxygen requirements.

The autonomous system, like the entirety of the GFR car, is governed by Formula Student rules [6]. These rules primarily govern the sensors providing information to the autonomous system as well as the driver functions regulating when the autonomous system is used. Sensors are prohibited from using wireless communication and the autonomous system must be regulated by a master switch which can enable or disable the system, and must have accompanying indicator lights. Because the requirements outlined in the Formula Student rules are addressed by other components of the autonomous system, their consideration in the autonomous controls portion of the system (the scope of this project) is not strictly necessary.

It may utilize existing tools developed by the team, as well as those available with ROS, and any code interfacing with ROS should be written in C++. Finally, its runtime should be low enough to not

negatively impact the car's performance. Because the project's runtime affects when commands to control systems are issued, shorter runtimes result in more precise track navigation, and are therefore desirable.

1.3 Black Box Description

The autonomous controls system accepts a track map generated by SLAM, and outputs instructions to the throttle and steering systems in the form of target velocity and steering angle to navigate the track. The locations of the cones on the track map are determined by a combination of camera and lidar, as well as the SLAM node's interpretation of camera and lidar outputs, so the accuracy of the controls algorithm is dependent on the accuracy of those systems as well. Prior to providing input to controls, the track map should be formatted in YAML format.

1.4 System Description

The autonomous controls system is part of the larger autonomous system. It interfaces with the SLAM node of the autonomous system to obtain track maps and with the throttle and steering actuators for outputting controls. The overall system should provide input to the car's controls when it is operating driverless using the known positions of cones around the track by interpolating track boundaries from the cones and calculating an ideal racing line around the track. State estimation data, such as position, heading, and velocity, are obtained from ROS, which is integral to the autonomous system as a whole. The physics of the car (including weight, dimensions, tires, velocity, and position) should be combined with the ideal racing line, state estimation information, and prior control values to determine the new values of inputs provided to the controls.

2. Current State Analysis and Benchmarking

2.1 Current State Analysis

The ending state of the prior year's vehicle used a C++ implementation of a pure pursuit algorithm. It contains two main files, named "inspection" and "pure_pursuit." The pure_pursuit file, which is the file executed by default, contains eight functions, including separate functions for calculating a target point, calculating steering angle, and calculating acceleration. The inspection file serves as a test file, providing alternating steering angles at low speeds.

The pure pursuit file (pure_pursuit) runs primarily by repeatedly calling its update() method. This method calls transform_path(), which updates the traveled path with the most recent line traveled by the car based on prior trajectories. This is followed by findTarget(), which determines the next point to travel to based on the car's current position and visible cones. update() then calls calculateSteering() and calculateAcceleration() which determine the commands to be sent to the physical controls, and finishes by publishing the steering and acceleration commands to the controls and the new target as a marker. A flowchart of the code is given below.

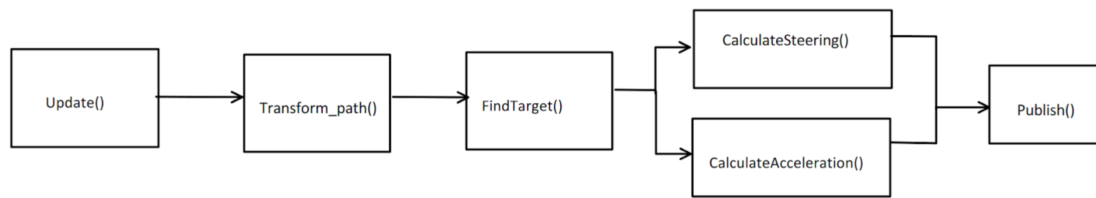

Figure 1: A flowchart of the prior year's pure pursuit code.

Because the prior year was the first year of GFR using an autonomous controls node, no other iterations of the code exist.

In the case of the MPCC library, by default the car model is a 1:43 scale electric vehicle [2] with significantly different physical properties from the GFR car. It also includes many functions unnecessary to the functionality of the autonomous controls node, which are present for the purpose of demonstration and visualization of the code's output. The cycle used by MPCC begins by loading its own default track, then repeatedly solving MPC problems generated by the code's own internal simulator (referred to as its "integrator"), and finally plotting the results of each solution. The MPCC integrator is similar to the GFR simulator in its physics model, but uses fewer parameters for modeling motor strength and tires, and therefore is expected to produce different results.

Strengths-Weaknesses-Opportunities-Threats (SWOT) Analysis

A SWOT analysis for replacing pure pursuit with MPC is given below.

Strengths	**Weaknesses**
● MPC is focused on producing an ideal racing line with variable throttle inputs, leading to potentially better lap times ● Both the MPC software and the Global Formula Racing autonomous code are written in C++, minimizing risk of compatibility issues	● Pure pursuit is already implemented and known to work ● Pure pursuit has a fast runtime ● MPC is significantly more complex and difficult to debug ● Pure pursuit can be used on courses where not all the cones are known, whereas MPC requires the entire track to be known in advance
Opportunities ● Open-source implementations of MPC are readily available ● Potential to catch up to teams already using MPC	**Threats** ● External MPC software is not plug-and-play and will require the creation of intermediate code ● MPC software will need to be modified to fit the physics and controls format of the GFR car

Overall, MPC presents advantages in its optimized racing line construction, its availability in compatible open-source implementations, and possible lap time reduction. However, it is competing with a fully-implemented and effective pure pursuit implementation, while also suffering drawbacks in its complexity, course limitations, and changes needed to make it compatible with the current GFR autonomous software pipeline.

2.2 Benchmarking

The previously-implemented pure pursuit algorithm performs its calculations by finding a new target point to drive to, then calculating the steering and acceleration needed to reach that target. These instructions are then published to the controls as commands. Therefore, useful information that can be measured from pure pursuit in the simulator are the throttle and steering commands, velocity, execution time, and a visual representation of the real path followed along the track. To benchmark this information, the pure pursuit algorithm was run for one lap around the GFR simulator's default track. Graphs of the information obtained from this are given below.

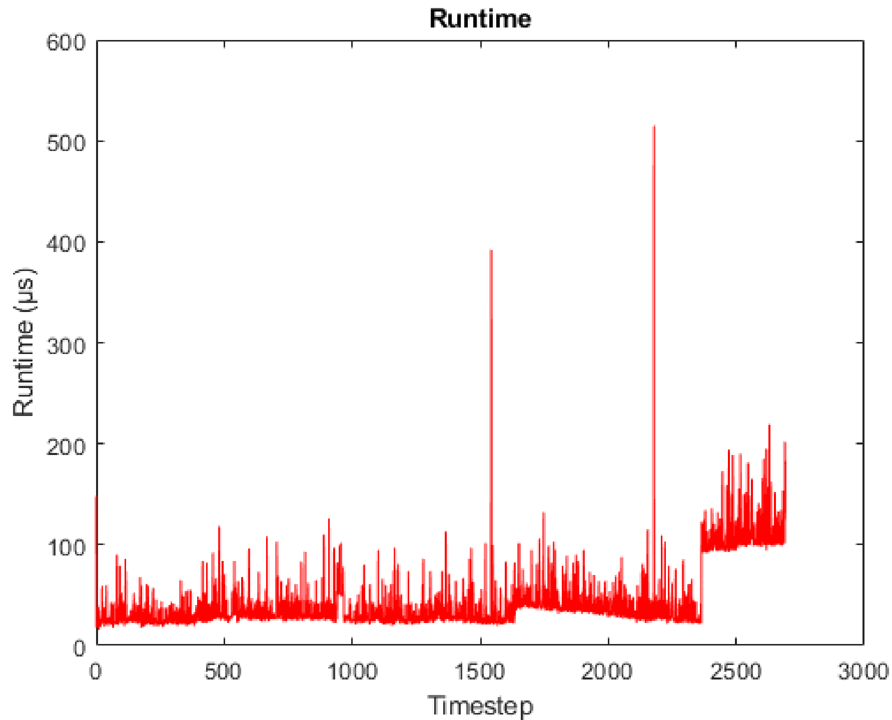

Figure 2: Plot of pure pursuit execution time along different points in the default autonomous track.

The above graph shows the time taken to calculate and issue throttle and steering commands using pure pursuit. Currently there is no explanation for the marginal yet consistent increase in computational time beyond 90% track completion. However, the dataset overall showed that the group of functions used for pure pursuit calculations averaged 44 µs.

A well-implemented MPC algorithm does not need to beat these times, and given the complexity of its algorithm it likely cannot. However, it does provide the foundation for expectations on what a reasonable runtime should be when the MPC functions are called.

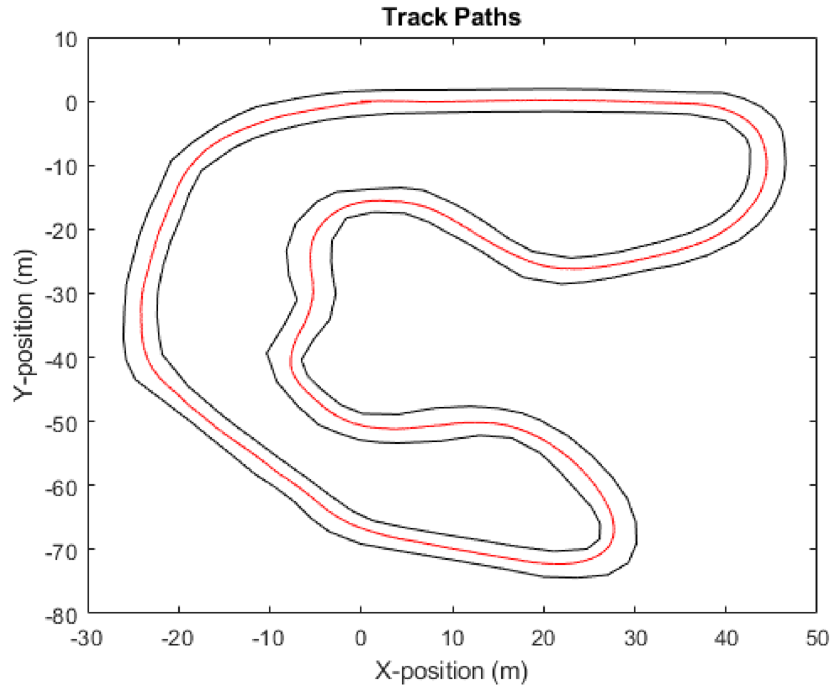

Figure 3: Plot of pure pursuit track path along the default autonomous track.

The above graph shows the path taken using pure pursuit over one lap of the GFR simulator's default track. The pure pursuit approach results in the car staying in the approximate center of the track throughout the entire lap. Importantly, the car does not at any point exceed track boundaries.

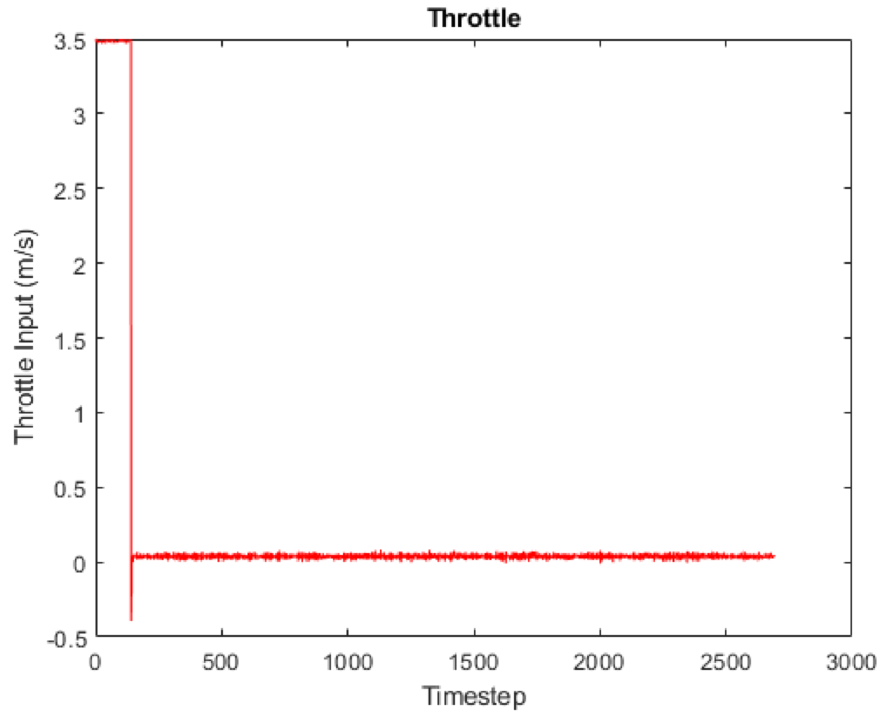

Figure 4: Plot of pure pursuit throttle command along different points in the default autonomous track.

This graph provides visualization of throttle input over the course of one lap. Pure pursuit provides throttle input equal to the target velocity immediately, and continues to do so until the target velocity is reached. After this, it attempts to maintain the target velocity as closely as possible. In this case, the target velocity is approximately 3.5 m/s, or 12.6 km/h.

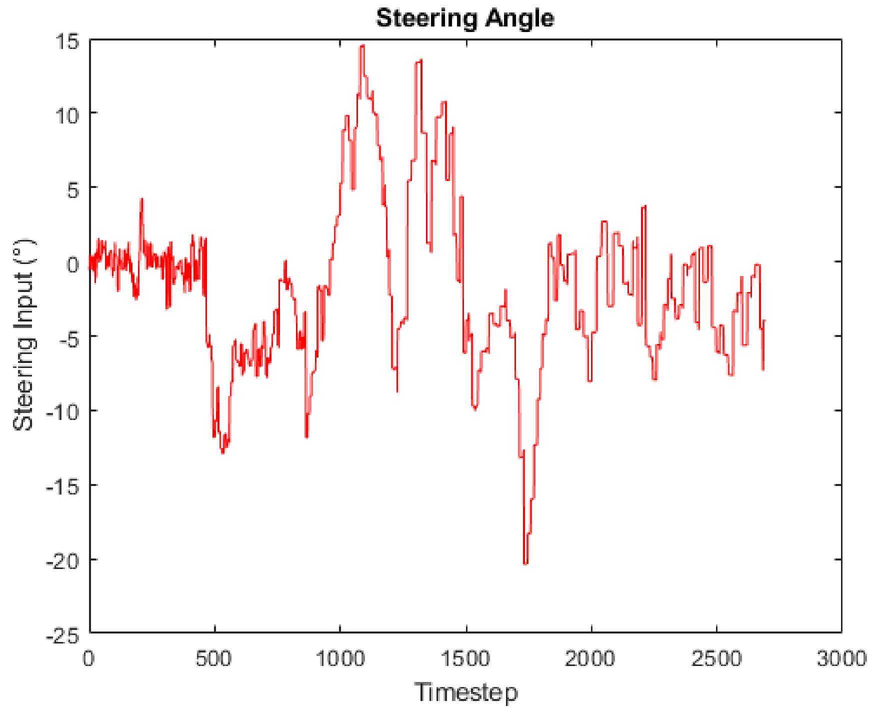

Figure 5: Plot of pure pursuit steering command along different points in the default autonomous track.

This graph shows the target steering angle over the course of one lap. Steering angle is measured as degrees of rotation of the front tires. Although overall the graph does not show any unusual steering behavior, it does demonstrate an imprecise approach to steering correction, as the car appears to remain at a target steering angle for some number of time steps before abruptly adjusting. Smoother and less discrete transitions between steering angles may provide more flexibility in the steering angles the car can expect to achieve.

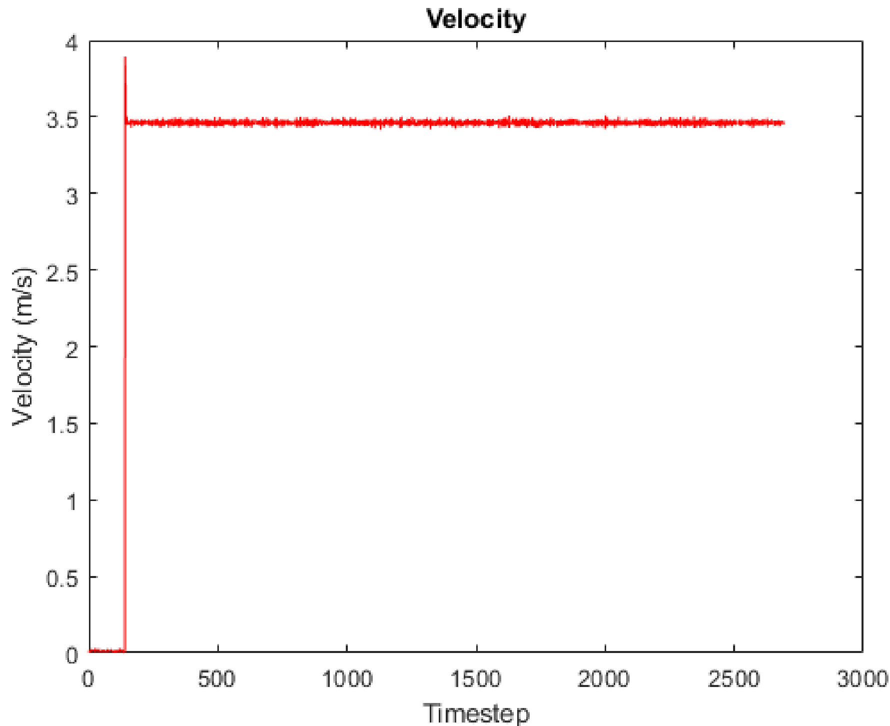

Figure 6: Plot of pure pursuit velocity along different points in the default autonomous track.

This graph shows the actual velocity of the car over the course of one lap. Much like the throttle graph, it demonstrates that the car rapidly accelerates to 3.5 m/s, and attempts to remain at this velocity over the course of the lap, making corrections to this end when needed.

This data do not indicate any problems that fundamentally prevent the car from being competitive, but they do provide a benchmark to compare MPC results with.

3. Design Analysis

3.1 Design Concepts

The autonomous controls system aims to calculate the fastest path around a track. To do so, the position of the cones forming track boundaries must be known, as well as both transitory and inherent aspects of the car including current velocity, current steering angle, car weight, car length, and car mass. The existing implementation of this system is in the form of a pure pursuit algorithm; the primary alternative to this is an MPC algorithm. Because MPC is an optimization algorithm, it is possible to weigh some objectives over others; for example, it is possible to prioritize avoiding the

track boundaries more, which would necessitate more cautious driving - and therefore possibly slower lap times - but reduce the risk of exceeding the track limits. The MPCC implementation prioritizes maximizing average velocity over the course of one lap with the condition that track boundaries are never exceeded.

3.1.1 Concept 1 - Pure Pursuit

A pure pursuit algorithm keeps track of all visible cones in front of the car, as well as all previously-seen cones within a specific radius of the car. The algorithm works in a cycle, finding a point that is both within the specified radius (limited by previously-seen cones) as well as between the inner and outer cones seen ahead. The car then drives straight to this point, adjusting steering angle as necessary and braking as the point approaches. Once the point is reached, the car repeats the cycle. This algorithm is dynamic and does not require knowing the course ahead of time. However, there is no attempt to find the quickest path around a circuit, only an accurate path.

The algorithm works in the context of a car by adjusting the steering angle to allow a car in motion to reach a given point. This is necessary because a car in motion cannot turn on the spot to face a new target point, but rather must follow an arc to reach it. A visual representation of this is shown below.

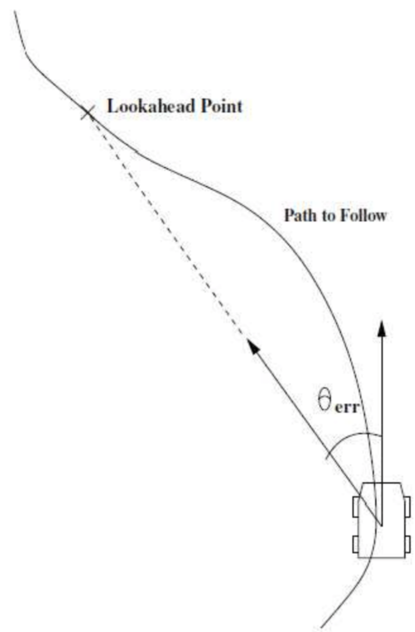

Figure 7: Visual representation of pure pursuit [7].

3.1.2 Concept 2 - MPC

An MPC algorithm requires existing knowledge of cone positions. With this information, the algorithm calculates a path through the course using an optimization algorithm maximizing rate of change in position while minimizing necessary changes to controls such as steering angle and braking. It is bound by pre-defined constraints that it cannot exceed, such as the physical attributes of the car and the boundaries of the course. Open-source implementations of MPC algorithms exist with pre-defined optimization parameters while car and course attributes can be modified in configuration files.

This algorithm has both "hard" and "soft" limitations hardcoded into whichever open-source implementation is chosen. Hard limitations - those that cannot ever be exceeded - may include track boundaries and the physics of the car. Soft limitations - those that are given priority in the optimization algorithm - may include minimum speed or maximum steering angle at a given speed. These, along with the weights of different components of the optimization algorithm, would not be changed if an unmodified MPC algorithm were implemented. An example of a path followed by an MPC algorithm around a track is given below. This example is included in the MPCC library as a demonstration of the code, and is not using the GFR car model.

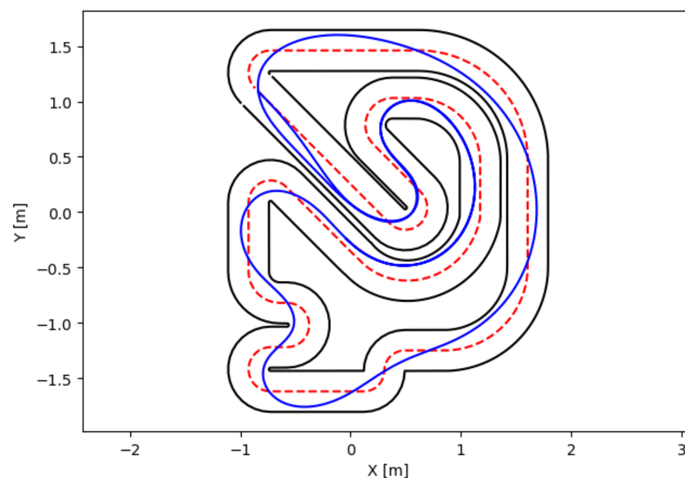

Figure 8: A path followed around a track calculating using MPCC; the red dashed line is the centerline, and the blue line is the path taken.

Regardless of how an MPC algorithm is implemented, the model used would have to be adjusted to accommodate the GFR car's physical attributes, such as weight and dimensions.

3.1.3 Concept 3 - MPC with Changes to Optimization

Existing MPC algorithms can be modified not only in the car and track parameters, but also in the relative priority placed on minimizing changes to controls while avoiding the track boundaries. If current open-source solutions are excessively cautious in avoiding cones, it may be advantageous to edit the code to place additional emphasis on speed. On the other hand, if current implementations are too risky with their behavior, it is possible to prioritize avoiding cones more instead. Whether these changes are necessary can only be determined through trial and error in testing. As with Concept 2, changes to the model to match the GFR car's physical attributes would be necessary.

3.2 Design Iterations

The three implementations listed above are described below.

3.2.1 Design Iteration 1 - Pure Pursuit

Advantages	Disadvantages
● Can always be used for events where the course is not known beforehand ● Already implemented	● Does not attempt to find optimal racing line ● Does not vary throttle input to account for turns or straights
Risks	Potential Failure Modes
● Slowness from non-ideal racing line could compromise lap times ● Relies on consistently accurate input about cone positions	● Inaccurate inputs could result in an incorrect line if camera or LIDAR miss or erroneously detect a cone

The pure pursuit concept has advantages in its versatility and existing implementation. However, it is not a competitive algorithm in terms of expected results. These jeopardize the chances of posting fast lap times when used on courses where use of an MPC algorithm is possible.

3.2.2 Design Iteration 2 - MPC

Advantages	Disadvantages
• Finds an ideal racing line • Open-source implementations exist, limiting necessary work	• Only works on courses where cone positions are known in advance • Requires integration with existing software
Risks	**Potential Failure Modes**
• Car comes much closer to track boundaries than pure pursuit • Untested, more risks may arise spontaneously	• Existing open-source models may not handle existing car model well • Varying levels of precision in optimization may push car too close to track boundaries

The MPC concept showcases the main advantage of the algorithm, in that it attempts to find an ideal racing line in advance, producing a faster theoretical lap without needing to redo these calculations while in motion. However, while implementing unmodified open-source software is both easier and more stable, it compromises the accuracy and effectiveness of the output, with racing lines possibly coming too close to track boundaries or the car physics being improperly modeled, with unpredictable results.

3.2.3 Design Iteration 3- MPC with Changes to Optimization

Advantages	Disadvantages
• Finds an ideal racing line • Cone positions are known to be accurate, so no room for input error from sensors • Allows balancing speed and risk-taking to levels acceptable to the team	• Only works when cone positions are known in advance • Requires integration with existing software • Requires modification of existing third-party code, which may not be well-documented or stable in when modified
Risks	**Potential Failure Modes**
• Totally untested, more risks may arise much more spontaneously	• Misjudging risk-taking levels may result in higher rates of exceeding track limits or slower than optimal speed

The third option, modifying existing MPC software, poses a greater risk of failure due to being completely untested and requiring changes to unfamiliar code. This could have the consequences of unpredictable behavior during races. However, it allows fine-tuning the optimization algorithm to what is reasonable for the car used by GFR, potentially producing both faster and safer laps.

4. Design Selected

The design selected for implementation was Concept 2 with some elements of Concept 3. This design uses MPCC, a largely third-party implementation of an MPC algorithm. The selected implementation does not model cars of the specification used by GFR. Therefore, minor changes to the model itself (as opposed to simply the track and car physics inputs) were necessary. While changes to the optimization algorithm itself were not made, the relative weights assigned to some components of the algorithm - primarily the soft limitations on maximum and minimum speed - were changed. Initial testing indicated the car did not vary its speed to an acceptable degree using the default settings, and so the minimum speed was lowered and the maximum speed raised. However, these are predefined variables that can be configured in external, non-code configuration files, and therefore do not affect the method in which the MPC solution is derived.

The algorithm selected for this implementation was Liniger's [2] MPCC C++ algorithm, which allows an input track and car model, and produces output plots of the ideal racing line, controls inputs, and expected physical results such as position and velocity.

4.1 Rationale for Selection

The algorithm selected met the requirement of forecasting a racing line around an arbitrary course while providing the inputs for a car with arbitrary specifications needed to navigate the course. This is needed for the car to move while not exceeding track boundaries. MPCC is implemented in C++, making it easily compatible with GFR's ROS architecture. While not a strict requirement, this does provide it an advantage over other MPC implementations that provide similar behavior but may require programming language translation to implement. The software is compiled from a large set of C++ files handling different parts of the MPC algorithm, and outputs its instructions and predictions in the form of plots. When implemented into ROS, these instructions can instead be sent directly to the GFR car's controls rather than plotting, which would reduce computation time, as plots of the controls are unnecessary while the car is running.

Modifications to the configuration of MPCC were deemed necessary based on a lack of variation in the car's velocity in simulator testing. Because this was the only area of concern in testing, no changes were made to the algorithm itself, but rather to the predefined weights used by the algorithm assigned to increasing and decreasing velocity.

4.2 Technical Specification

The MPC algorithm interfaces with a pre-generated track map and ROS odometry in order to verify its position along the course at any given point. Once the position is verified and compared to the expected point and physical state predicted by the MPC algorithm, the appropriate instructions to correct to or maintain the forecasted path are relayed to the throttle and steering actuators. These interfaces - odometry to MPC and MPC to actuators - are the components of the code implemented by this project.

4.2.1 Specification Sheet

The following equations are specified by Liniger [8] as the basis for the MPCC codebase's optimization problem generation:

$$\begin{aligned}
\min \quad & \sum_{k=1}^{N} \begin{bmatrix} \hat{e}^c_k \\ \hat{e}^l_k \end{bmatrix}^T \begin{bmatrix} q_c & 0 \\ 0 & q_l \end{bmatrix} \begin{bmatrix} \hat{e}^c_k \\ \hat{e}^l_k \end{bmatrix} - q_v v_{\theta,k} + \Delta u_k^T R_\Delta \Delta u_k \\
\text{s.t.} \quad & x_0 = x(0) \\
& x_{k+1} = f(x_k, u_k) \\
& \hat{e}^c(x_k) = \quad \sin\left(\Phi^{\text{ref}}(\theta_k)\right)\left(X_k - X^{\text{ref}}(\theta_k)\right) - \cos\left(\Phi^{\text{ref}}(\theta_k)\right)\left(Y_k - Y^{\text{ref}}(\theta_k)\right) \\
& \hat{e}^l(x_k) = -\cos\left(\Phi^{\text{ref}}(\theta_k)\right)\left(X_k - X^{\text{ref}}(\theta_k)\right) - \sin\left(\Phi^{\text{ref}}(\theta_k)\right)\left(Y_k - Y^{\text{ref}}(\theta_k)\right) \\
& \Delta u_k = u_k - u_{k-1} \\
& x_k \in \mathcal{X}_{\text{Track}} \\
& \underline{x} \leq x_k \leq \bar{x} \\
& \underline{u} \leq u_k \leq \bar{u} \\
& \underline{\Delta u} \leq \Delta u_k \leq \overline{\Delta u}
\end{aligned}$$

Figure 9: MPC model equation [8].

This in turn relies on the following model for vehicle dynamics (a bicycle model) to provide X and Y:

$$
\begin{aligned}
\dot{X} &= v_x \cos(\varphi) - v_y \sin(\varphi) \\
\dot{Y} &= v_x \sin(\varphi) + v_y \cos(\varphi) \\
\dot{\varphi} &= \omega \\
\dot{v}_x &= \frac{1}{m}(F_{r,x} - F_{f,y} \sin\delta + m v_y \omega) \\
\dot{v}_y &= \frac{1}{m}(F_{r,y} - F_{f,y} \cos\delta - m v_x \omega) \\
\dot{\omega} &= \frac{1}{I_z}(F_{f,y} l_f \cos\delta - F_{r,y} l_r) \\
\dot{\theta} &= v_\theta
\end{aligned}
$$

Figure 10: Bicycle model equations [8].

A bicycle model is not as computationally demanding as a double-track model, which is a more generally applicable model for car applications. However, Vazquez [9] argues that the bicycle model is applicable in the specific case of Formula SAE cars, as the cars generally follow the basic assumptions of the bicycle model that the track is level, load changes and combined slip are ignored, drivetrain forces act on the vehicle's center of gravity, and the vehicle is a rigid body. The tire and drivetrain model listed in the above tire model and the constraints listed on the above MPC model are defined by the following equations:

$$
\begin{aligned}
F_{f,y} &= D_f \sin(C_f \arctan(B_f \alpha_f)) \quad \text{where} \quad \alpha_f = -\arctan\left(\frac{\dot{\varphi} l_f + v_y}{v_x}\right) + \delta \\
F_{r,y} &= D_r \sin(C_r \arctan(B_r \alpha_r)) \quad \text{where} \quad \alpha_r = \arctan\left(\frac{\dot{\varphi} l_r - v_y}{v_x}\right) \\
F_{r,x} &= (C_{m1} - C_{m2} v_x) d - C_r - C_d v_x^2
\end{aligned}
$$

Figure 11: Tire and drivetrain model equations [8].

Finally, the variables in these equations are defined with the state and inputs:

$$
\begin{aligned}
x &= [X, Y, \varphi, v_x, v_y, \omega, \theta] \\
u &= [d, \delta, v_\theta]
\end{aligned}
$$

Figure 12: State inputs [8].

Where, according to Liniger, "(X,Y) is the global position phi the heading of the car, v_x, and v_y the longitudinal respectively the lateral velocity and omega the yaw rate." This formula determines how a car will navigate a given stretch of track given the car's physics and bound by the constraints of the

track boundaries. The tire model provided by Liniger was revised in order to enable compatibility with the GFR model. In the above equations, it is listed as:

$$F_y = D\, sin(C\, arctan(B\, \alpha))$$

This is the MPCC solution for lateral force applied to a tire at any given point. B, C, and D are predefined physical properties of the car (in the case of the GFR car, these are B = 12.56, C = 1.38, and D = 1.60). However, The model GFR uses for F_y is:

$$F_y = D\, sin(C\, arctan\,(B\,(1 - E)\,\alpha + E\, arctan(B\,\alpha)))\, F_z$$

This includes both an E component for the tires, as well as an F_z component (in the case of the GFR car, E = 0.58). This difference results from the MPCC software being designed around small, radio-controlled cars rather than human-driven cars, as the smaller car deals with less substantial physical forces and as such requires fewer variables to model accurately. All the other input parameters (given in x) are directly compatible with what is used in the GFR simulator and only require a function, which transfers them to the MPC software at runtime by reading the values from ROS.

The remaining variables are not identified in the paper published by Liniger [8] pertaining to the MPCC code. Inspection of the code suggests φ to be the heading angle of the car, θ to be the progress along the track, d to be the throttle input as a target velocity in meters per second, δ to be the steering angle in radians, and v_θ to be the rate of change of θ. Components of x (states) are values that are measured prior to each solution, whereas components of u (inputs) are numbers taken directly from the prior solution (being 0 initially). The identity of all other listed variables remains uncertain, as they are documented neither in the Liniger [8] paper nor in the MPCC code. Moreover, because the C++ implementation of the MPCC code uses different variable names altogether from those presented by Liniger, mapping the variables used from the code to the equations is not possible without completely reverse-engineering MPCC, which is outside the scope of this project.

4.2.2 Performance Specification Estimation

Performance estimations for the MPC algorithm can be obtained by recording identical data to pure pursuit, as the same data is available across both, and provides a point of comparison for the two algorithms. This means the execution time of the MPCC function, coordinates of the car across one lap to generate a track path, velocity across one lap, and the steering and throttle control outputs across one lap can be recorded at time step intervals. While the two algorithms will require a different

quantity of timesteps to navigate one lap, converting the vectors containing each timestep to linearly spaced vectors of equivalent length allows for simpler comparison and visualization.

5. Implementation

Ultimately, the MPC algorithm was implemented by adding the existing MPCC codebase as a submodule of the controls codebase. Extraneous files such as the plotting function and MATLAB implementation of the program were dropped, and the software's configuration files were relocated to the controls directory. Within the controls directory, a driver C++ file was added to handle fetching odometry information, converting the track file to a format readable by MPCC, executing the MPC function, and issuing controls to the throttle and braking actuators. Within the MPCC configuration files, the costs for changing throttle input were decreased, the target velocity range increased, and the physical parameters of the car updated in order to account for the demands of the GFR car.

5.1 Interface Definitions

The MPCC driver file interfaces with the controls actuators by publishing steering and throttle commands. The method in which it does this is identical to the existing pure pursuit implementation.

5.2 Code Architecture

The MPC implementation uses the predominantly-external MPCC codebase for solving MPC problems. Therefore, the driver file focuses primarily on generating the problems and serving as an interface between the track maps, ROS, MPC, and actuators. It does so by initializing the track map by generating an MPCC-compatible configuration file based on information read from the simulator's track directory. Which track it selects from there can itself be set in a configuration file. Once the track is set, the software enters a cycle where an MPC structure is generated containing the car's current position, velocity, heading, and commands. The track is assigned as an attribute of this structure, and the structure sent to the MPC solving function. The MPC solving function returns throttle and steering commands, which are forwarded to the command publisher. After this, the cycle repeats by generating a new MPC structure with the previously-returned throttle and steering outputs included as inputs.

5.3 Parameters

The MPC implementation has a variety of configuration files it uses as parameters. However, as most of the definitions in these files never need modification, they can be treated as constants. Therefore, only the following parameters are fetched from configuration files located in the controls configuration directory: a track file name to use, the target average velocity, and range of allowable deviation from the target average velocity. While the target average velocity is based on the physical parameters of the car, increasing it can result in faster but risker track navigation, whereas decreasing it results in slower but safer track navigation. Similarly, increasing the range for average velocity increases speed and risk while decreasing it decreases speed and risk. This justifies customization of these values, and therefore they must be treated as parameters.

Also used as parameters are the car's current velocity in both the x and y directions, current x and y position, and heading. These are fetched from basic ROS operations, and are updated by gfr_common, which handles processing commands published to actuators. While these parameters are not customizable, their variation does affect the operation of each cycle of MPC solving.

6. Testing & Support

Because the MPCC library is treated as a black box within this project due to the selection of Concept 2, the only tests for which the expected output were verified are navigation attempts of pre-existing tracks. Individual sections of the MPCC code were not tested. Comparison to benchmarking of pure pursuit was the most effective way to do this, as navigating a track requires similar commands and similar points in time; for example, no matter which algorithm is used, a steer left command must be issued at a left turn. Therefore, a successful test can be viewed as navigation of a track with similar velocity and commands over one lap when compared with pure pursuit while also not exceeding track boundaries (which can be quantified by visual inspection of the track path map). Runtime should also be tested - although it is not a metric of success for the autonomous controls node, it does provide an indication of potential sources of error, as higher runtimes result in greater time intervals between commands (and thus less precise commands).

6.1 Tests complete to Date

The following data were obtained when executing MPC on the default simulator track, with the same methodology as was done with pure pursuit in benchmarking. Pure pursuit outputs are included to provide easy comparison.

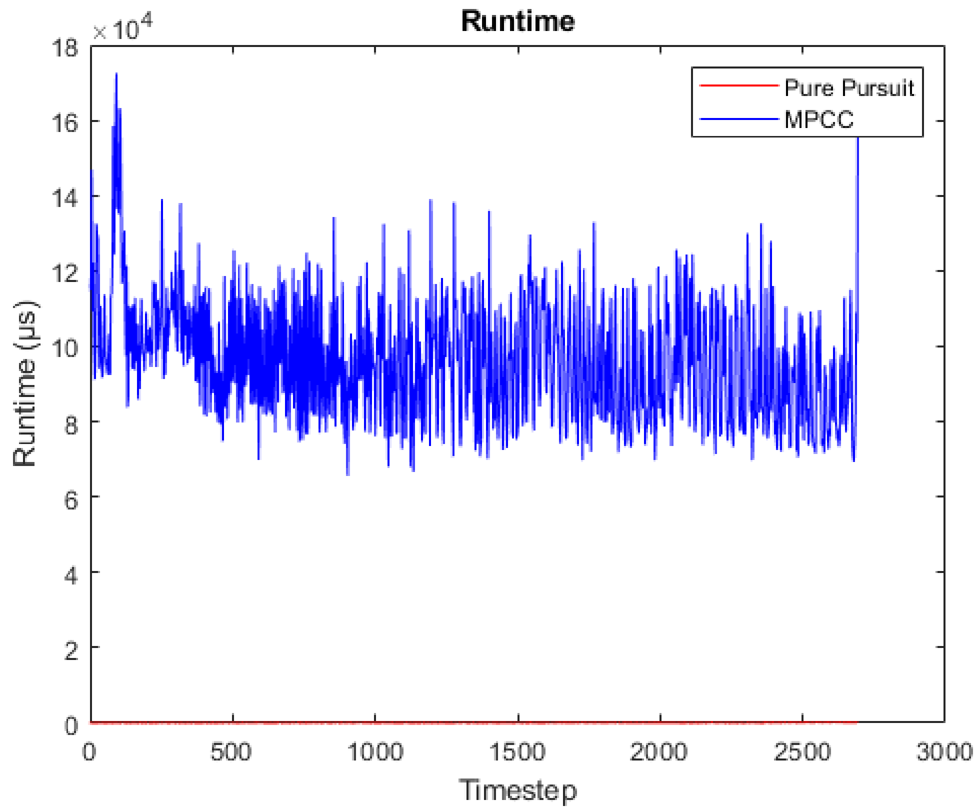

Figure 13: Plot of MPC pure pursuit execution time along different points in the default autonomous track.

MPCC demonstrates a far higher runtime than pure pursuit, exceeding it by at least 70,000 µs (70 ms), three orders of magnitude, at any given timestep. However, this does not necessarily have serious consequences for the functionality of the MPC algorithm. As MPCC is a far more complex calculation than pure pursuit, longer computation times are expected, and adjusting the prediction timestep in the MPCC portion of the code allows for farther "lookahead" distances (at the cost of less precise inputs) that accommodate the computation time taken when determining what commands to issue.

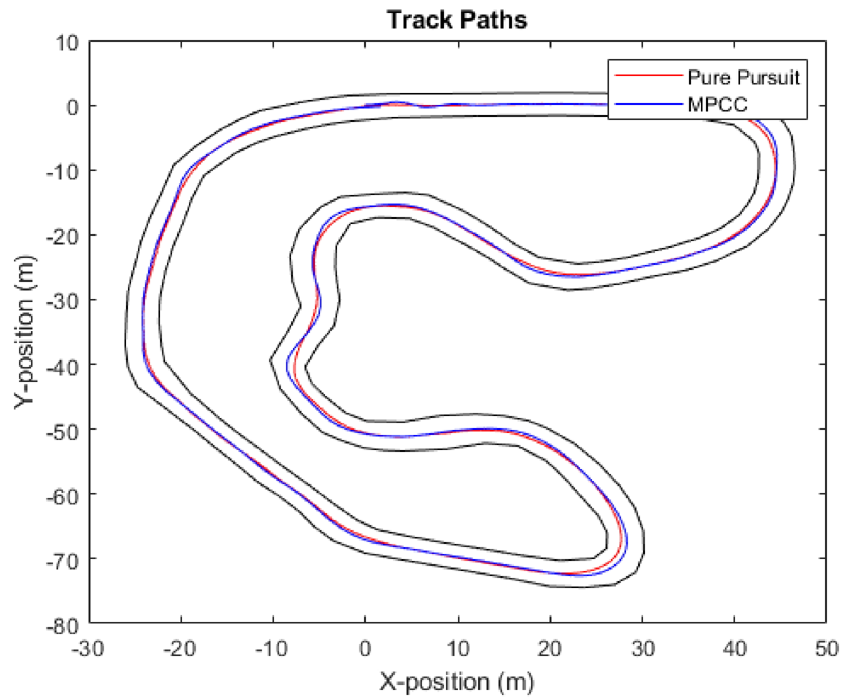

Figure 14: Plot of MPC and pure pursuit track path along the default autonomous track.

This graph shows that generally, MPCC and pure pursuit follow identical paths around the track, with neither exceeding track boundaries. Interestingly, a qualitative visual inspection suggests that MPCC actually takes wider turns than pure pursuit at several corners, a trait expected of sub-optimal racing lines.

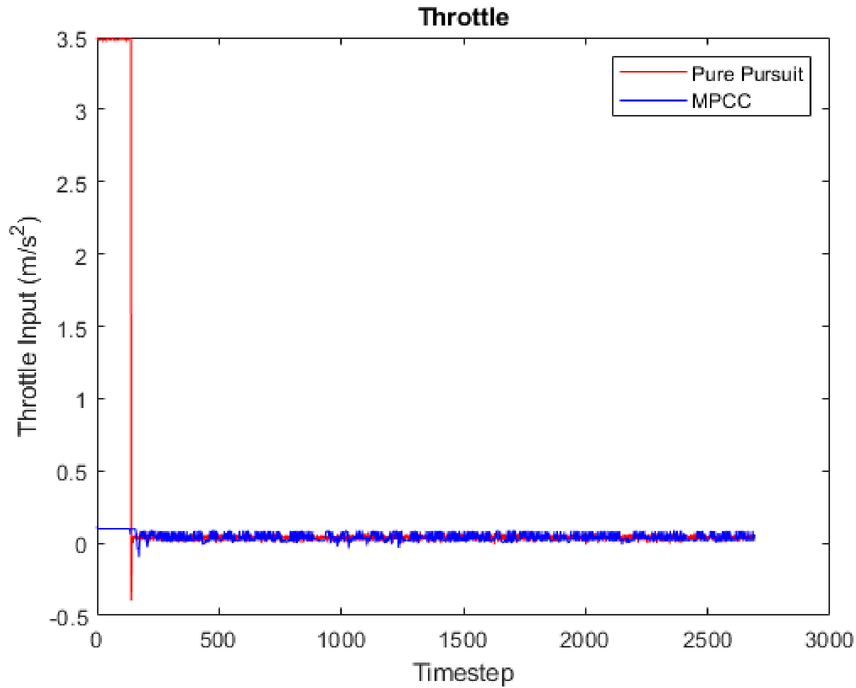

Figure 15: Plot of MPC and pure pursuit throttle command along different points in the default autonomous track.

This graph compares the throttle input of MPCC and pure pursuit over one lap. MPCC maintains a higher average throttle input beyond the first 200 timesteps, resulting in a more gradual approach to its target velocity. Wider fluctuations in the throttle input are also indicative of speeding up coming out of turns and slowing down going into turns, an expected result that contrasts with the constant speed maintained by pure pursuit.

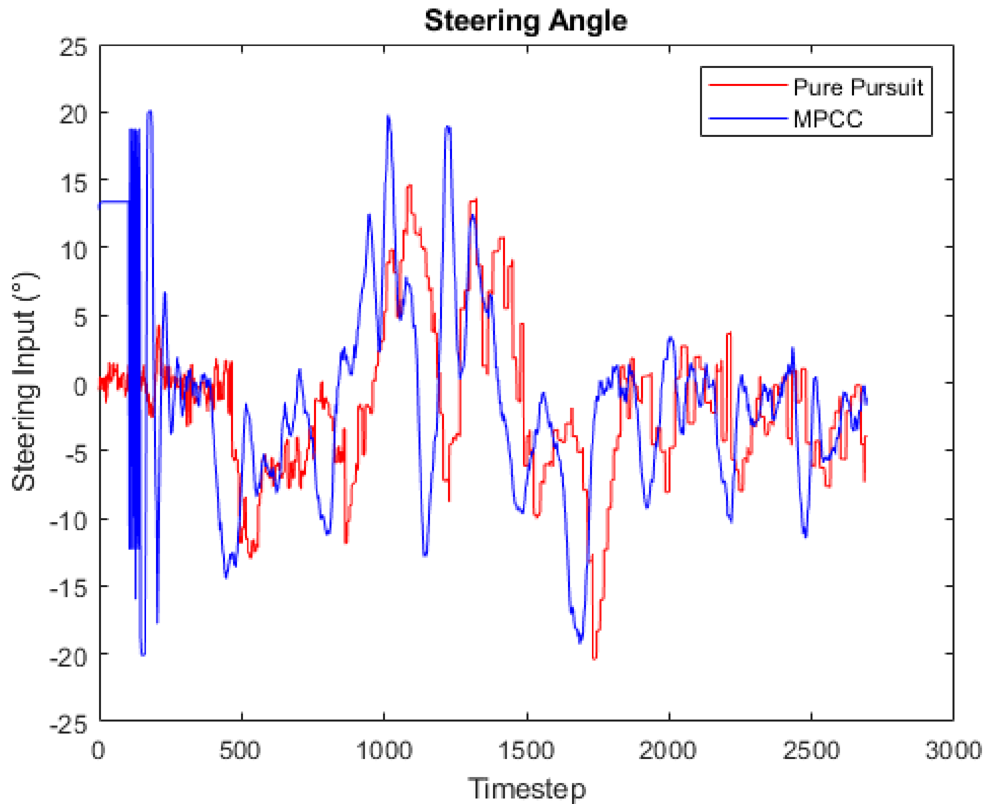

Figure 16: Plot of MPC and pure pursuit steering command along different points in the default autonomous track.

This graph compares the steering angles used by the two algorithms over the course of a lap. Because MPCC is slightly faster to complete the lap, its plot is shifted left of pure pursuit at several points on the graph, but generally the same commands are issued at the same time. The largest source of error in the MPCC implementation becomes apparent at the start of the graph, however: MPCC results in rapid back-and-forth steering in excess of 10 degrees left and right during the first 200 timesteps. The cause of this is unknown, and no solution was found during the duration of this project. The impact of this was severe enough that the top speed of the MPCC implementation was limited such that the car would not exceed track boundaries during this period.

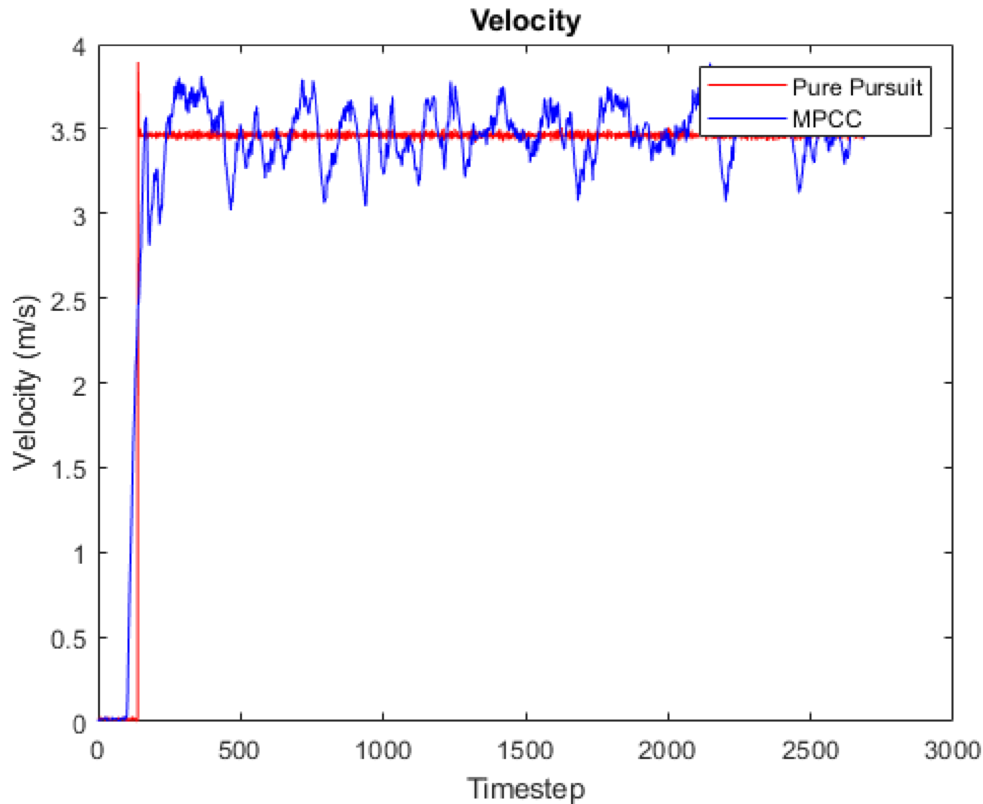

Figure 17: Plot of MPC pure pursuit velocity along different points in the default autonomous track.

The testing indicates that the MPC implementation produces behavior consistent with pure pursuit in velocity, track path, and commands. Its significantly higher runtime does not seem to affect this. While it does not correlate directly with lap times, average velocity can nevertheless be a good predictor of how quickly a car can navigate a track. MPC, with its variable velocity, maintained an average velocity of 3.3096 m/s, whereas pure pursuit maintained an average of 3.2830 m/s. The average velocity of MPCC can be configured, however as mentioned in the discussion of Figure 16, exceeding this average velocity resulted in track boundaries being exceeded as well within the first 200 timesteps. Should this problem be corrected, it is expected that the minimum velocity set by MPCC would be equal to or greater than the average velocity of pure pursuit. This is because pure pursuit maintains a constant velocity over the course of the entire track, and therefore its average velocity is equivalent to the maximum velocity needed to navigate the slowest turn on the track without losing grip.

7. Conclusion

This project resulted in successful implementation of an MPC algorithm using the open-source MPCC codebase, with the GFR car being able to navigate the default track of the GFR simulator in comparable time to the preexisting pure pursuit algorithm. MPCC requires the track to be known in advance, providing a limitation that does not exist with pure pursuit, meaning it is not suitable for every application.

The MPC algorithm demonstrated the ability to maintain an average velocity approximately equal, but marginally greater than, pure pursuit. Critically, its top speed is limited due to severe overcorrections in steering in the first 200 timesteps of its first lap around a track. The cause of this is unknown, and the issue was unable to be resolved in the timeframe of the project. It happens to be the case that the top speed limit needed to prevent a track boundary violation in this timeframe results in near-equal performance with pure pursuit. Additionally, MPCC showed possible sub-optimal racing lines upon visual inspection of its path around the track, suggesting a problem in the calibration of its steering solution. Further testing is needed to determine the cause and solution for this. MPCC necessitated longer runtimes between issuing commands, an expected and unavoidable drawback that can be accommodated by using larger timesteps in its optimization function.

It is possible the cause of the chaotic navigation can be attributed to MPCC's large runtimes. In this scenario, the odometry data is taken from the ROS measurements of the car at the start of a timestep, but by the time the MPC solution is complete, the car's position has changed and the controls output does not have the expected effect. Such an issue would also explain the seemingly sub-optimal racing lines taken when MPCC is used. The solution to this would likely involve adjusting the timestep provided to MPCC to be larger. Additional testing is needed to determine whether this is in fact the issue and whether timestep modification is a substantial enough change to resolve the problem.

While the current implementation of MPCC is functional, the greater risk associated with its more abrupt steering at the beginning, as well as its less-than-ideal racing lines and relatively comparable time compared to pure pursuit, means that there is not yet a compelling reason to fully transition to using MPCC over pure pursuit. Should the problems with its steering be rectified, it is likely a far higher top speed can be set in its configuration, thus allowing it to reach speeds far beyond pure pursuit on portions of the track where minimal steering is necessary. This is because pure pursuit's average speed is limited by its top speed on the portion of the track where steering is at a maximum.

Addressing the overcorrection issue will therefore result in MPC becoming a competitive alternative to the current implementation.

7.1 Future Development

Future development should primarily focus on solving the issue with overcorrection in the first 200 timesteps. One possible solution to this issue may be "hard-coding" the car to drive straight during this period, though this runs the risk of exceeding track boundaries should the starting line be followed immediately by a sharp turn. A more sustainable solution should focus on exploring why the issue is occuring to begin with. Testing larger timesteps may be the preferred way to start this process. However, GFR is a time-sensitive operation with strict deadlines in order to be able to attend competitions, and sustainable solutions may not be the best course of action in scenarios where it would run the risk of exceeding such a deadline. Any future development should therefore be done under good judgment, taking into account how much work is realistically achievable that would allow the car to be functional at the soonest competition. While solving the overcorrection issue is the ideal long-term goal, working towards this goal in steps that gradually improve the performance with the car in testing is more consistent with the test-centric approach of GFR team philosophy.

Future development may also explore other algorithms as well as improvements on the MPC algorithm. MPCC documentation is limited and the code is a "black box" in many cases. However, this presents a good opportunity for future projects to explore the code in greater detail, and also opens up the possibility of further improvements to the physics model used by MPCC and the weights in the optimization algorithm itself. In particular, once the MPCC codebase is well understood, future developers can improve on the physics model used by the software as well as the optimization weights given in the parameters.

8. Works Cited

[1] J. Kabzan, M. Valls, V. Reijgwart, H. Hendrikx, C. Ehmke, M. Prajapat, A. Buhler, N. Gosala, M. Gupta, R. Sivanesan, A. Dhall, E. Chisari, N. Karnchanachari, S. Brits, M. Dangel, I. Sa, R. Dube, A. Gawel, M. Pfeiffer, A. Liniger, J. Lygeros, and R. Siegwart, "AMZ Driverless: The Full Autonomous Racing System," *Journal of Field Robotics*, vol. 37, no. 7, pp. 1267-1294, 2020.

[2] A. Liniger, "MPCC," *github.com*. Available: https://github.com/alexliniger/MPCC [Accessed May 18, 2021].

[3] A. Liniger, "Path Planning and Control for Autonomous Racing," Ph. D. thesis, ETH Zurich, Wohlen bei Bern, Switzerland, 2018.

[4] "CppStyleGuide," *ros.org*, Jan. 5, 2021. [Online]. Available: http://wiki.ros.org/CppStyleGuide. [Accessed May 18, 2021].

[5] D. van Heesch, "doxygen," *github.com*. Available: https://github.com/doxygen/doxygen [Accessed May 18, 2021].

[6] *Formula Student Rules 2020*, SAE International, 2020.

[7] A. Bárdos, S. Vass, A. Nyerges, and Z. Szalay, "Path tracking controller for automated driving," In Advanced Manufacturing and Repairing Technologies in Vehicle Industry, 2017, pp. 10.

[8] A. Liniger, A. Domahidi, and M. Morari, "Optimization‐based autonomous racing of 1:43 scale RC cars," *Optimal Control Applications and Methods*, vol. 36, no. 5, pp. 4-5, 2017.

[9] J. Vazquez, M Bruhlmeier, A. Liniger, A. Rupenyan, and J. Lygeros, "Optimization-Based Hierarchical Motion Planningfor Autonomous Racing," In 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020, pp. 2.