Optimal Racing Line Using Trajectory Optimization

Dalton Richardson

***Abstract*— Proposal and Project Draft Update for a Controller to model and determine optimal racing line using trajectory optimization methods**

I. PROJECT PROPOSAL

A. INTRODUCTION

In motorsports, such as Formula 1 and IndyCar, one of the most important aspects is the racing line you take around a track. This racing line determines when you should break, when you should turn, and how fast you should be going. An example of such a racing line is shown below in Fig. 1. Just being off by a few meters or milliseconds can make or break a race for a driver so it is incredibly important to know exactly what the optimal line is. For my project, I propose using trajectory optimization to determine this optimal line on any given track.

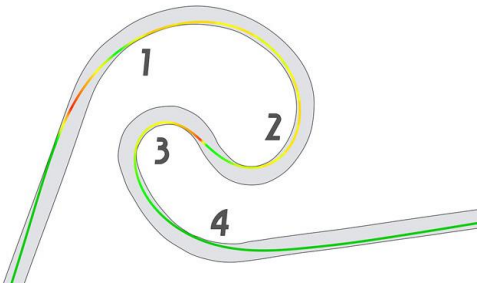


Fig. 1. Example of an Optimal Racing Line

B. SIMULATION

Simulating a system such as this will be the first challenge to tackle. My plan for the simulation, as well as most of the project, is to keep it simple at first and just get a working implementation. From there, as time permits, I can add onto the work and improve from there. The initial concept is to simulate a simple 2D track of constant width with a static environment with no air resistance or temperature constraints. Because of the constant width and 2D dimensions, I can model a track as a line from which I can calculate the distance from the car to determine if the car is on the track or not. Possible extensions could include making the track 3D by including inclines and such or having a changing width along different points of the track.

C. HYPERPARAMERS AND CONTROL INPUTS

There are a few hyperparameters for this system that must be chosen before modeling can take place. The most obvious of which is the track itself, including its width. These other parameters have to do with the car dynamics themselves. This includes the max acceleration, max deceleration, turn radius, and coefficient of friction.

As for the control inputs, it will consist of a vector of (X(t), Y(t)) coordinates indicating car position along the track.

D. THE CONTROL TASK

1) A. Objectives and Constraints

The objective of the problem will be to minimize the time it takes to drive a lap around the track. Constraints for this would include max and min speeds and accelerations, the car not leaving the track, turn radius, and the friction keeping the car on the track. This friction and turn radius constraints will be especially important to prevent the car from just speeding through corners.

2) B. Design Vector

The design vector will consist of a spline of points the car will cross along the track in a format similar to: $[X_1(t) \quad X_2(t) \ldots X_n(t) \quad Y_1(t)\, Y_2(t) \ldots Y_n(t)]^T$. From a vector such as this we can determine the optimal path, along with the optimal speed, to take.

E. CONTROLLER DESIGN AND SOLVER

1) Previous Works and Inspiration

An excellent source of information for me has been a paper published by MIT on this subject of optimization of racing lines [1]. It lays out in a fair amount of detail the formulas they used and methods. While I can't obviously take exactly from their work, it is an excellent starting point from which I can start the design of my own controller.

2) Proposed Design

My proposed controller would take use of the Multiple Shooting and Direct Collocation Trajectory Optimization techniques discussed in class to find an open loop solution. With a possible extension of adding Model-predictive Control if time permits as suggested by Dr. Hubicki. I am planning on using MATLAB to implement both the simulation and controller but want to investigate the benefits of using Python further before making this a concrete decision. I do not believe I will need any additional computational requirements for this project as of now.

II. PROJECT DRAFT

A. SIMULATION DETAILS

1) Representing The Track

The track is represented using in Frenet Coordinates using the referencePathFenet() function within Matlab. There were two distinct strengths to using this method. Firstly was that a track could be generated with this

function using a list of given (X,Y) points. This essentially meant I could give it a list of (X,Y) points and interpolate the entire track through these given points. The second reason was because of the function closestPoint(). This function could take in a pair of coordinates and output the point closest on the track to this point. What this gave me was an easy way to measure if a 'car' was on or off the track at any point. A few other important details about the track representation are that it currently is in 2D space meaning there are no elevation changes and that the track has constant width around the entire circuit.

2) Simulated Movement and Validation

Right now the decision vector for input into the simulation is made of the accelerations, current steering angle, and final time of the car in the form: $[Acc_1(t) \quad Acc_2(t) \ldots Acc_n(t) \quad \theta_1(t)\theta_2(t)\ldots\theta_n(t) \quad T_{final}]^T$. The equations of motion for the car from this point are simple and are mostly likely a source of improvement to come. As of now there are four equations used to determine the Theta (orientation), X, and Y positions of the car…

$$Theta_i = Theta_{i-1} + dT * \theta(t)$$
$$Speed_i = Speed_{i-1} + dT * Acc(t)$$
$$X_i = X_{i-1} + dT * Speed_{i-1} * cos(Theta_{i-1})$$
$$Y_i = Y_{i-1} + dT * Speed_{i-1} * sin(Theta_{i-1})$$

To validate the system I had a separate script that would allow me to control the car as it drove around the track. From this I could create a decision vector with which I could run the simulation on. To validate this system I checked the real driven path to the simulated path taken from the decision vector. The slight changes in the simulated driving between these two scripts meant that there were very slight differences in the path between these two since a small discrepancy could alter the path later on. But this difference is very marginal and can only be seen by the ending positions of the car and this is not something I expect to affect the final simulations since the drivable script will not be involved.

B. RUNNING A SIMULATION

There are four scripts that matter to this simulation with some variables and information exchanged between them using global variables.

The first is DrawTrack.m which allows the user to draw a track. Using the mouse to click on points, the program will automatically connect these points together. When finished with the track, you close the figure and it will automatically close the loop to complete the track.

The second is RenderTrack.m. This is pretty much a proof of concept just to draw the track. You can click on points on the figure and it will tell if that point is within the track limits or not.

Thirdly is DrivableTrack.m. As touched on before this will let you drive around the track with the arrow keys. It is recommended to not hold the arrow keys down but to just repeatedly tap them to control the car. After closing the figure it will automatically save the design vector as 'drivePath'.

Lastly is DriveSimulation.m. This final script is the actual simulation. Using the designed track and resulting decision vector, will simulate a lap around the track. This is the most important function and will be the basis as I start trying to add optimal control techniques to the system.

The scripts must be run in a certain order for them to work since some require a track or decision vector. To run a simulated lap, you must first draw the track and then drive around the track to create the appropriate decision vector. When completed with a script is it important to close the figure so that the global variables are updated and can be used in later scripts.

C. NEXT STEPS

Obviously the next steps are adding in optimal control techniques to the simulations to determine the optimal racing line. More constraints will be added to the car through this including speed limits, turning radius limits, and limits on cornering speeds. Before I move onto this I may take another look at the actual equations of motion to create a more realistic simulation using ODE45. One stretch goal of mine if time is on my side and I can get the solvers to run fast enough would be to integrate in the human driving to the simulation. This would allow you to alter the cars path as its being simulated and see its updated path plan in real time.

III. FINAL PROJECT REPORT

A. SIMULATION

1) Formulation

The equations of motion for this project were overhauled after the submission of the project draft to improve the simulation quality and realism. The formulation was created by drawing upon multiple different sources online and consists of two main parts, the acceleration from control of the car and the drag caused by motion.

First starting with the equations for the control of the car, I framed my equations from a webpage by Steven LaValle of the University of Oulu [2]. In his explanation he formulates his equations of motion to find the velocity of the car. Where u_s is the total speed of the car and u_θ represents the current angle of the wheels with respect to the car, the equations of motion are…

$$\dot{x} = u_s * cos(\theta)$$
$$\dot{y} = u_s * sin(\theta)$$
$$\dot{O} = (u_s / L) * tan(u_\theta)$$

… for the x velocity, $\dot{x}$, the y velocity, $\dot{y}$, and the cars change in orientation, $\dot{O}$. L represents the length between the wheel bases and θ is the current orientation of the car with respect to the world frame.

To model the drag on the car, I borrowed some ideas from a YouTube video by Christopher Lum [3]. He formulates drag as…

$$F_d = c_T * v^2$$

... where v is the velocity of the car and F_d is the total force from drag. Additionally he uses c_T as a coefficient for drag. In his video he describes a formula for finding this coefficient, but I just estimated it for my own formulation.

Combining these two formulations and differentiating them to find acceleration instead of velocity I found my final equations as…

$$\ddot{x} = -u_s * \sin(\theta) * \dot{O} + \mathring{u}_s * \cos(\theta) - c_T * \dot{x}^2$$
$$\ddot{y} = u_s * \cos(\theta) * \dot{O} + \mathring{u}_s * \sin(\theta) - c_T * \dot{y}^2$$
$$\dot{O} = (u_s / L) * \tan(u_\theta)$$

With these equations I can use ode45 to go from…

$$[x, \dot{x}, y, \dot{y}, \theta] \rightarrow [\dot{x}, \ddot{x}, \dot{y}, \ddot{y}, \dot{O}]$$

2) Correctness

The correctness of the simulation can be verified through the graphical display of the car's movement. Inputting the decision vector, which is described in more detail below, you can see the car's movement through the configuration space. Also, there is a script that allows the user to manually control the car using the same equations of motion that makes it easier to test edge cases. While the simulation physics works well, there are two noticeable error's that could be improved upon.

The first, and more minor of the two, is the effects of drag on top speed. While the drag does a good job slowing down the vehicle so that a constant acceleration force must always be applied, it also severely limits the top speed. This is since when applying a constant acceleration force to the car, it will eventually balance out with the drag in a sort of equilibrium that will stop the car's acceleration and keep a constant velocity. While this in itself is not a big deal, it limits the speed more than it realistically would. This is a much more minor problem though.

The second, more concerning, error is turning. When the car turns, it will slightly increase the velocity of the car. It is a very small amount but can be slightly noticeable in some cases. The problem with this comes from very fast movements with very harsh turns where it can be noticed much more. This can be seen the most when manually controlling the car and keeping a constant angle on the wheels of the car causing it to drive in a circle. In these cases, the velocity will start to exponentially increase until the entire simulation breaks. While this could be a problem in more complex projects, with the speed and angles the car is required to practically make in optimization this is a very small effect. It can be slightly noticeable in some cases, but only to a small degree.

3) Track Representation

Please see section A.1 of the Project Draft above for details of the track representation for the simulation.

B. CONTROLLER DESIGN

1) Decision Vector

The decision vector consists of three parts. The acceleration or force on the car, the wheel angle, and final simulation time. These first two will be inputted into the simulation equations above as $\mathring{u}_s$, for acceleration, and as u_θ, for wheel angle. The final form of the decision vector will be…

$$[\mathring{u}_{s\,1}, \ldots, \mathring{u}_{s\,N}, \; u_{\theta\,1}, \ldots, u_{\theta\,N}, T]$$

2) FMINCON

Due to the complexity of the simulation and the fact that this is a problem with non-linear constraints, I decided to use FMINCON to solve the problem. All constraints were treated as such in @nonlcon.

3) Cost Function

The cost function was formulated very uniquely for this project than many of the ways we discussed in class. One of the biggest problems I faced was with the original cost function which consisted of the last element of the decision vector *T*. The problem was that due to the complexity of the task, the constraints with this cost function were not enough to drive the decision vector towards a valid solution. For this reason, I worked towards essentially adding some constraints into the cost function that would decrease as the decision vector came closer to satisfying the constraints and would disappear from the cost completely when satisfied. Four constraints were placed into the cost function this way.

The first was endCost. This would add a penalty to the cost function when the car's final position in the simulation was not close enough to the finish line. The further away from the finish line the car was, the more of a penalty this would have on the cost value. If the ending state was within half of the track width of the finish line no penalty would be incurred.

The second was the pathCost. This was a penalty for how well the car stayed within the bounds of the track. At each point along the simulation, it would check how far outside the racetrack the car was and add a penalty for how far outside of it the car was. The further outside the track limits the car was, the larger this penalty would be. If the car stayed fully within the bounds of the track for the entire simulation this cost would be 0.

Thirdly was the lengthCost. This would compare the length of the path the car took over the entirety of the simulation to the length of the track. If the length of the path taken by the car was over twice as large as the track length a penalty would be incurred.

Lastly was the minLengthCost. This is similar to lengthCost but would cause a penalty if the path was less than 3/10ths the length of the racetrack. This was to ensure that the car actually attempted to go around the track when being optimized.

The strength of the penalty that each of these would have on the actual cost function would be proportional to the time *T*. For instance, the cost from endCost would be 20x as strong as the final time *T* until it was satisfied. Once these constraints have all been satisfied, it is left with a cost function that is just the final time *T*.

4) Constraints

Besides the 'constraints' kept inside the cost function, there were also the actual constraints that had to be met inside @nonlcon. There were three constraints in total all within the C vector meaning they had to stay under 0.

The first was distConstraint. This is very similar to pathCost in the fact it is about keeping the car within the track limits. Using the function closestPoint(), it would determine the distance the car was from the track's center line. Using this, it would utilize the track width to determine if the car was within the limits at each point of the simulation.

The second constraint was to make sure that the car did not go backwards and was called speed. It's a pretty simple constraint just to keep the car moving forwards.

Finally, was the constraint distEnd to determine if the car made it to the finish line. To determine this, it would check if the final point of the simulation was within the track width distance of the finish line.

While these were good at letting the FMINCON instance know if the state it was in was valid, they did very little to help drive it towards a valid state thus why some constraints were baked into the cost function instead.

5) Initial Guess

The final key detail about the controllers' design is the initial guess given to the solver. Considering that there are many local minima that can be incurred by FMINCON when trying to solve the optimization, a good initial guess is very important. But there is no good general initial guess that will work for all cases. To solve this problem, there is an interactable simulation that allows you to drive the car manually and create an initial decision vector from the path you take. This is a good solution that works well but has a few things to keep in mind.

Firstly, it is pretty hard to drive and keep control of the car. For tips on how to better drive the car you can check the 'How to Run' section of this report below.

Secondly, when running an instance of manual driving, you have much more control over the car than a decision vector will have. For instance, while a decision vector may have a size of N, usually in the range 10 to 40, the manual control of the car is much more precise from moment to moment. This means the decision vector created from user inputs has to be scaled down from the actual inputs of the user. This can lead to sometimes small and sometime large discrepancies between the path driven by the user and the decision vector's simulated path. Obviously a larger N for the decision vector will mean the path of the initial guess's simulation will be closer to the path manually driven, but it also makes a more complicated problem for FMINCON. No matter what though, the initial guess given by this method is far better than using an initial guess vector filled with 1's.

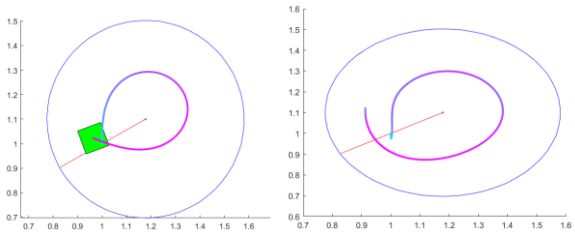
C. PERFORMANCE

1) Overall

The performance of the optimization is pretty mixed. Due to the number of local minima that can occur, it can quite often get stuck with solutions that aren't optimal or sometimes even on track. With the relatively simple tracks I have tested on, it can usually get a decent path but the initial guess can have a great effect on this.

Below I include three examples that help highlight where it succeeds and where it fails.

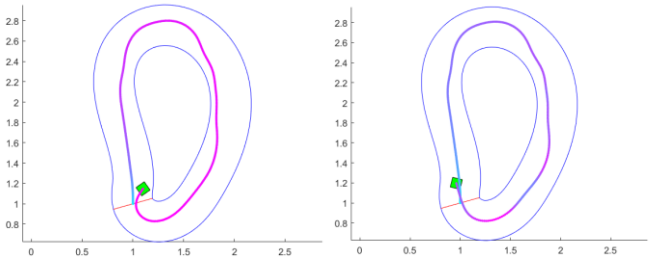
2) Example 1 (Easy Track) 32.75 -> 17.5



This first example is the best example of a route actually being optimized. While the intial guess and final optimization may look very similar, the difference is in the speed. While the original path finishes in 32.75 (The time scale is arbitrary and doesn't have a unit) while the final optimized route only takes 17.5. This is coming close to almost halving the time it takes to go around the track.

With an N of 10, it is able to experiment more quickly and try new decision vector values. But, what really makes this example work best is the simplicity of the track. A major problem which will become clear in the upcoming, more complex, is trying to change the beginning of the path/decision vector. This is since the earlier in the path a change is made, the large the effects will be later on in the path. But for this simple example, it can change earlier values in the decision vector while the path stays within bounds and the end near the finish line.

3) Example 2 (Hard Track with Good Initial Guess)



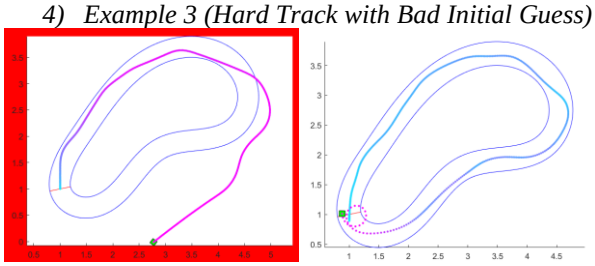
This is a good example of where it honestly failed to make much of a different. As you can see from the pictures the paths stays near identical from the intial guess to the optimized solution. And worst, the time between the two is exactly the same. But, saying this there is still a lot to take away from this example

First of all while there isn't much difference, most of the difference that does exist is towards the end of the cars path. This goes back to the previous example where I said that it is much harder to change the beginning of the path. Imagine having it speed up slightly more or take a little bit less of a turn at the beginning, it would massively change the end placement of the car. What this means for the optimizer is that by changing early values, it will massivly increase the cost function causing it to fall into a local minima of sorts. The reason the is more difference towards the end is that the optimizer is able to change the values much easier without massivly affecting the cost function.

The second noticable thing is that the timing between the two is identical. My prediction for why this is is that

changing the timing will change when each input from the decision vector is inputted into the system. For instance if the time in the decision vector was 70 instead of 80, it would go through the the inputs of the decision vector over a smaller amount of time which would dramatically effect the path. Thus, it was less costly for the optimizer to just keep the time constant.

4) Example 3 (Hard Track with Bad Initial Guess)



This final example does a good job showing the optimizer working on just finding a path on the track itself. I purposfully gave it a very poor intial guess to highlight how in many cases it can at the very least meet the constraints.

D. HOW TO RUN

There are multiple scripts each with their own purpose which will be discussed here.

1) DrawTrack

DrawTrack allows the user to draw a track by placing waypoints that the track must drive through. It will use referencePathFrenet to interpolate a track in between the waypoints. This is also where you can change the parameters of the track. In the script you can change two variables before running. The first is trackWidth. This is the width of the track on both sides. For instance, if set to .2, the entire track will be .4 across. The other is trackResolution which allows you to change how many points are used to render the track. The lower the number, the blockier and more disjointed the track will be. Once the track is finished, you can just close the graph and it will automatically save the details to the global variable `TrackParams`.

2) RenderTrack

This must be run with a valid `TrackParams` loaded in the current workspace. This will display the track as it will be used for the simulations to come. Additionally, the user can click at any point on the graph and it will indicate whether that point is in the track or not.

3) DriveTrack

This must be run with a valid `TrackParams` loaded in the current workspace. There are a few parameters at the top that can be changed related to how the car will run but I would leave them as is. The only variable that I would recommend looking at is `N`. This is the size of the initial guess decision vector that will be created. This `N` will be used for the rest of the scripts that come after it. This current script when run will allow the user to manually drive around the track to create this initial guess decision vector. You use the up and down arrow keys to control acceleration and the left and right arrow keys to steer. For steering, I highly recommend rapidly tapping to increase the steering angle as it is much easier to control. Once you complete your lap you need to close the current graph and the initial decision vector will be saved. After running this script the global variables `N`, `XInit`, and `carParams` will be added to the current workspace.

4) SingleOptimization

This is the script that will attempt to optimize the decision vector. To run you need to have valid instances of the global variables `trackParams`, `XInit`, and `carParams`. `N` will be determined from `XInit`. Additionally at the top there is a choice of the FMINCON options that can be selected from the commented lines. For simple tracks, the options line with just 'MaxFunctionEvaluations' is best, but for longer more complex tracks using the options with more settings can help. Once a run is started, it will graphically show you the different simulations and strategies FMINCON is searching through. Once finished, the final decision vector will be stored in `decisionVector`.

5) SimulateVec

This final script will graphically simulate either `XInit` or `decisionVector` depending on what is commented out at the top of the script. It will visually show you the car driving the path decided by the control inputs of the vector.

6) Running The Examples

Running one of the provided examples is a simple process. The workspaces for the examples are provided in the same folder as the scripts. After loading one in, just run 'SingleOptimization' to start the process. Once complete, you can use 'SimulateVec' to see both the final decision vector and the initial guess.

7) Running Custom Tracks and Optimization

The process of running this all from scratch is pretty simple as well. First I would suggest starting with a new and empty workspace though this isn't entirely necessary. First you start with 'DrawTrack' following most of what was said above about this script. I would suggest leaving the parameters alone besides trackWidth, but you can change whichever variables you want. Once this is drawn you can use 'RenderTrack' to view the final version of the track. If you are not happy you can simply just rerun 'DrawTrack'. After this you need to run 'DriveTrack' to create your initial guess. While it is hard to control the car, with simpler tracks this is less important. The most important part of this is choosing the size of the decision vector through N in the script. Once you complete this, you have two options. The first is to view your final initial guess using 'SimulateVec' with the 'simTrajectory' call for `XInit` not commented out at the top. Or you can just go straight to 'SingleOptimization' to run the optimizer and then call 'SimulateVec' afterwards to see the solution.

REFERENCES

[1] Y. Xiong, *Racing Line Optimization* , Massachusetts Institute of Technology, 2010

[2] LaValle, Steven M. A Simple Car.
http://msl.cs.uiuc.edu/planning/node658.html.
[3] https://www.youtube.com/watch?v=kbGal6xKLB4