

Lap Time Simulation: steps 5, 6 and 7, and some divagation

Andrea Patrucco, Vehicle Dynamics Engineer, Applus+ IDIADA Spain, 13/10/2025

The views, thoughts and opinions expressed in this article solely those of the author, and they do not necessarily reflect the official policy or position of Applus+ IDIADA.

Introduction

Referring to a now months-old post on this platform by Mike Law about steps to follow to build a lap time simulation software, last month I published here a little article which covered steps 2 to 3 in the proposed challenge, that is, roughly, converting a vehicle model into a collection of steady-state, zero-yaw-moment states.

While I am still thinking about how to tackle step 4 (converting such collection of states into the *performance envelope* of the vehicle), especially in a computationally-efficient way, I guess I produced enough material (and divagations) to talk about the following steps. Directly citing the author:

5. Now, it's time to introduce the circuit model. At its simplest, this is a vector of track curvature you want to follow, against the distance along the racing line that each curvature value represents.

6. In order to "drive" the G-G-V model around the lap, you will need to do three passes. The first is to retrieve the maximum speed you can go around each curvature on the circuit model, ignoring any longitudinal acceleration. The corner radius that each point on the G-G-V plot represents is given by v^2/a , where "a" is lateral acceleration.

7. You now have the maximum speed the car can go around the lap, however there's no continuity between the points in terms of longitudinal acceleration. You will now have to do sweeps through the lap in both directions to determine how quickly the car can accelerate on entry and exit to reach maximum speed!

Again, do not take this as a rigorous guide on how to do this, but rather, as an account of how I tried to, and some ideas which I might find way more interesting than they actually are, that I encountered along the process.

Circuit model – Track

In a relatively wide sense, we can define a track as a surface covered in asphalt or another material you are supposed to drive on, surrounded by grass or another material you are not supposed to drive on.

As any surface, this too can be approximated by a collection of quadrilateral elements. If we accept some coarse approximations when defining its lateral limits, we can try and define such quadrilateral elements as:

- An "entry" side (a cross section of the track you pass through *first*)
- An "exit" side (a cross section of the track you pass through *second*)
- Two lateral sides delimiting the track.

By placing many consecutive elements one after each other, we obtain a rather primitive, and completely planar description of the surface we're supposed to drive on with a clear indication on the direction we're supposed to follow. I find this implementation convenient for mainly two reasons:

- It's easy to write a reasonably robust function, based upon the sum of triangular areas, to check whether a point is located inside a track element, which allows to easily write functions to:

- Check whether the actual trajectory of a vehicle has always remained inside of the track throughout a simulation
- How long did it take for the vehicle to reach a specific point of the track (for example, the finish line)
- We can parametrize every possible way we can navigate the track as a sequence of values between 0 and 1 by using the entry and exit sides (the i -th exit becomes the $i+1$ th entry), for example 0 meaning completely right, 1 meaning completely left.

The attractiveness of the second point is that conceptually, for sufficiently close together entry and exit segments describing a track, we can describe whatever possible *trajectory* with a sequence of values between 0 and 1, which opens the door to a handful of probably useless but still funny optimization strategies.

Circuit model - Trajectory

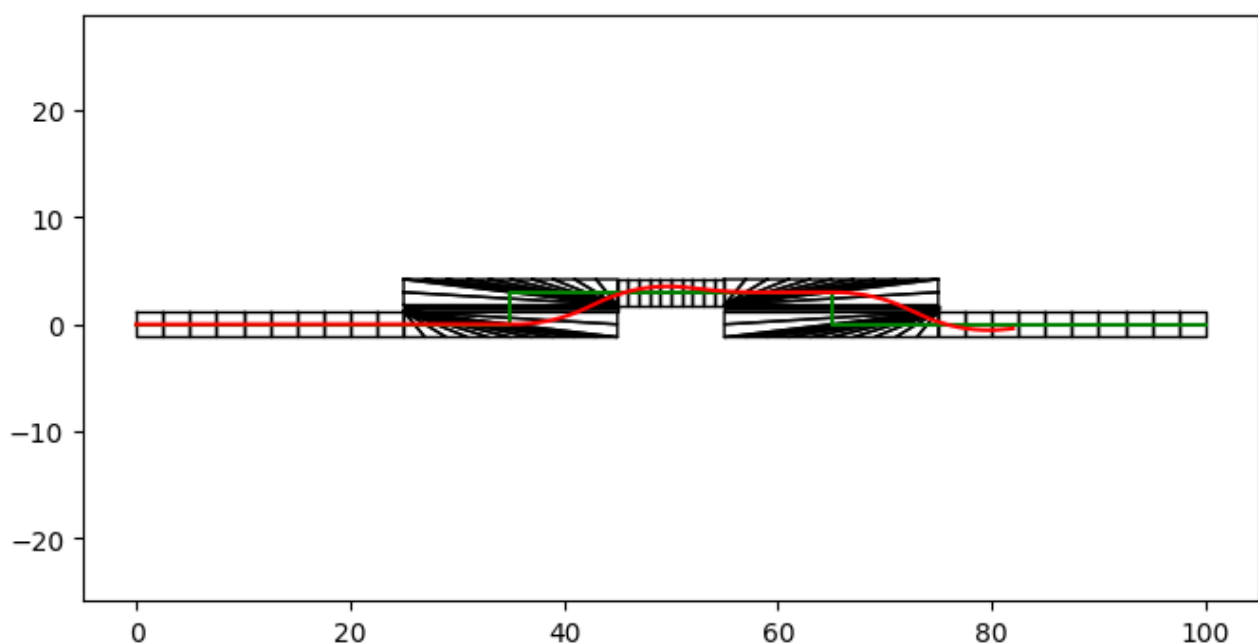
Without going too far, and again, neglecting the third dimension of the physical world, let's define a trajectory as a sequence of (x, y) points in a plane. Each straight line connecting two consecutive points will be a *segment*, which we'll be able to extract a *length* and an *orientation* from.

Given each couple of consecutive segments, we can define their *curvature* as the ratio between the difference between their orientation, and the distance between their center points. This corresponds to (an approximation of) the inverse of the radius of the ideal corner connecting the center points of each segment, tangent to the segments in such center points.

Now, the obvious and not so intriguing way to take advantage of this information is to write a function which minimizes either total curvature, total distance, or some weighted sum of the two, to de-couple the problem of finding a trajectory and driving onto it; I instead found it more appealing to do the wrong thing, and try to optimize them together.

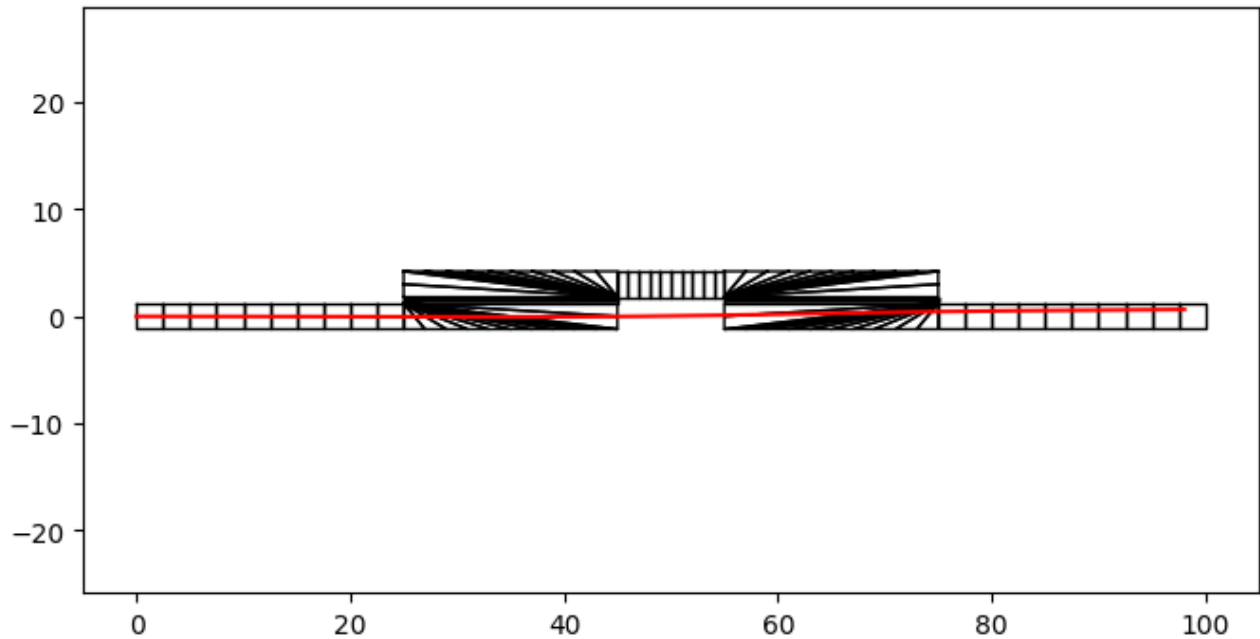
Sidetracked: concurrent trajectory + controller optimization

Once I had a basic implementation of a *Track* and a *Trajectory* classes, I immediately implemented the world's dumbest controller logic, coupled it to a 3-DOF vehicle model, defined a double-lane-change track and the most obvious choice of a trajectory, and the following simulation ensued:

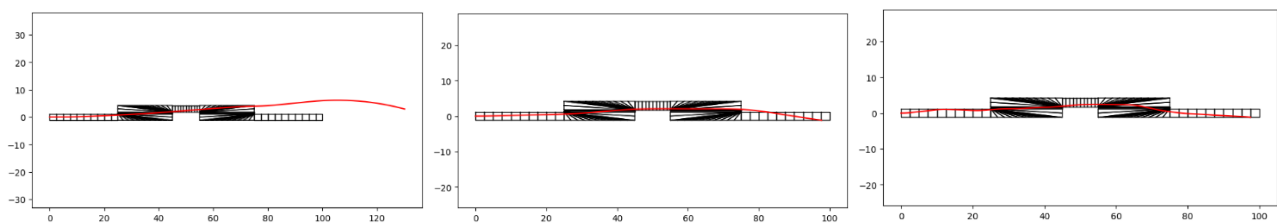


Where, in green, you can see the target trajectory and, in red, the actual one followed by the vehicle. It is clear that neither the trajectory (which just consists of the centerline for each entry point) nor the controller are particularly good at the task of going through the track fast: thinking about how to improve the performance of both, I thought about just attempting this in one shot: define a track, a vehicle model, and let a heuristic optimizer try and find *both* the best trajectory *and* the controller constants to navigate it.

Hence I defined a scalar objective function as a rather naïve sum of average speed, time to the finish line and added some penalties for going outside the track and, in a typical example of heuristic optimization humor, this happened:

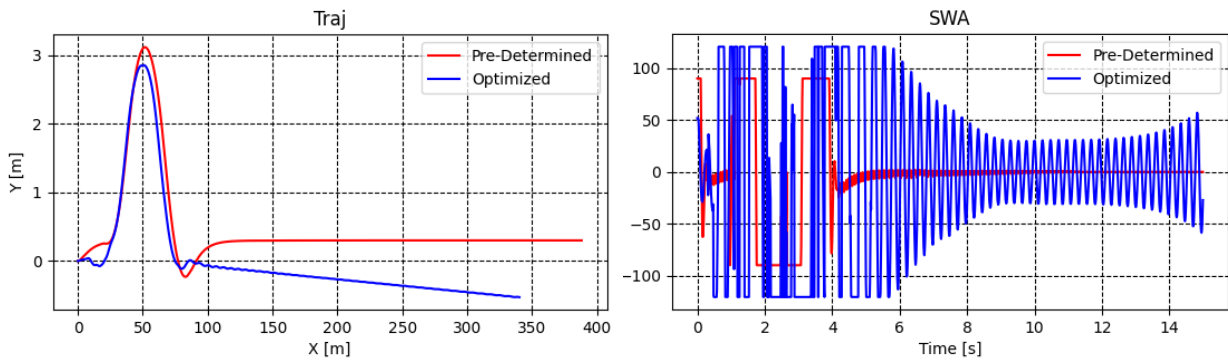


The vehicle didn't bother following the track, since the penalty for cutting it was too low, and the incentive to increase speed was too high. Revising the weighted function (for example, making penalty inversely proportional to the *last valid track element* the vehicle went through) and waiting far too long for such a simple case, I finally got this:



With the vehicle slowly *learning* to survive past the first lane change though missing the second one, then gently veering right after the first change, and finally grasping how to complete the maneuver without skips. The wording here is bit ambiguous; however it should be underlined that this is not machine-learning, just competing solutions combining themselves to more fit candidates at each iteration.

One may be tempted to compare results obtained this way with those of a more reasonable method, based upon successive optimization of trajectory (minimizing curvature) and controller, with the following result:



The concurrent trajectory+controller optimization (blue) indeed manages to go faster around the maneuver, but it does so through continuous changes in the steering wheel angle, which nervously jumps from the minimum to the maximum allowed values, while the pre-determined trajectory one (red) more elegantly turns left, then right, then left again.

This was, in the end, a far too brief attempt to claim the method is hopeless (and I'm sure I'll want to revisit it at some point, maybe with the point-mass approach I developed afterwards); I'm just reporting it here as a cautionary tale of how difficult it is to control the results of an optimization procedure when you're trying to optimize many things at the same time.

Furthermore, talking with friends and colleagues who know far better than me the topic, I finally realized that trajectory optimization is a rather *academic* approach to the issue, and that it can be forgotten for now in favor of the far more efficient one proposed by the LinkedIn post that started this all.

Circuit Model – *TracingInsights* trajectory data

I hence started with the *Trajectory* class I described some pages ago, and, instead of tediously mapping the profile of a racetrack passing through the *Track* class, I just looked for some collection of Formula-1 telemetry data, which is way easier to find than I thought.

By browsing the repository in [1], I could find practically every Formula 1 lap for the past 5 seasons. Data is presented in the open, human-readable .json format, and can be parsed with little effort in the collection of (x, y) points I needed to define a *Trajectory*.

Still, these are real GPS points with some slight discontinuities which are not well liked by the curvature estimation function. Aiming at a quick solution which did not require a more sophisticated approach (for example, splines), I found that smoothing (np.convolve, rectangular shape) X and Y values over 10 points, and again, the curvature output over 4-5 points resulted in a curvature array which is still close enough to the raw trajectory, but with much less discontinuity, resulting in a cleaner result.

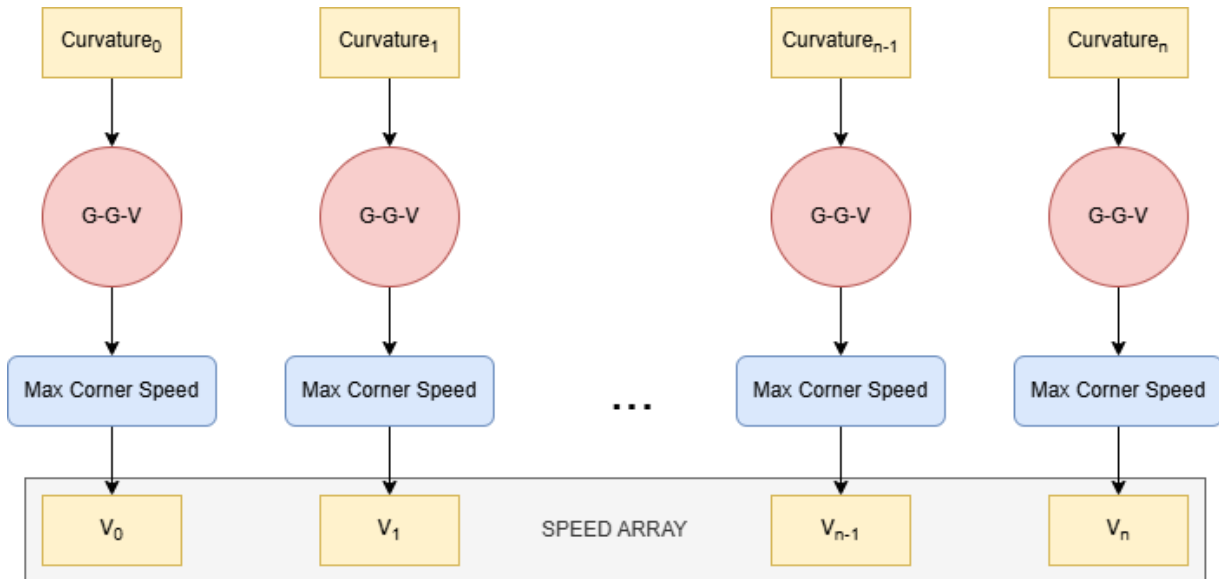
It should be noted that this strongly affects the output, and finding a compromise between cleaning data and avoiding the loss of detail is possibly one of the most evident areas of improvement of what I've currently developed.

Points 6 and 7: Driving the model around the track

Supposing that we are able to extract from our G-G-V model, at all speeds, the following information:

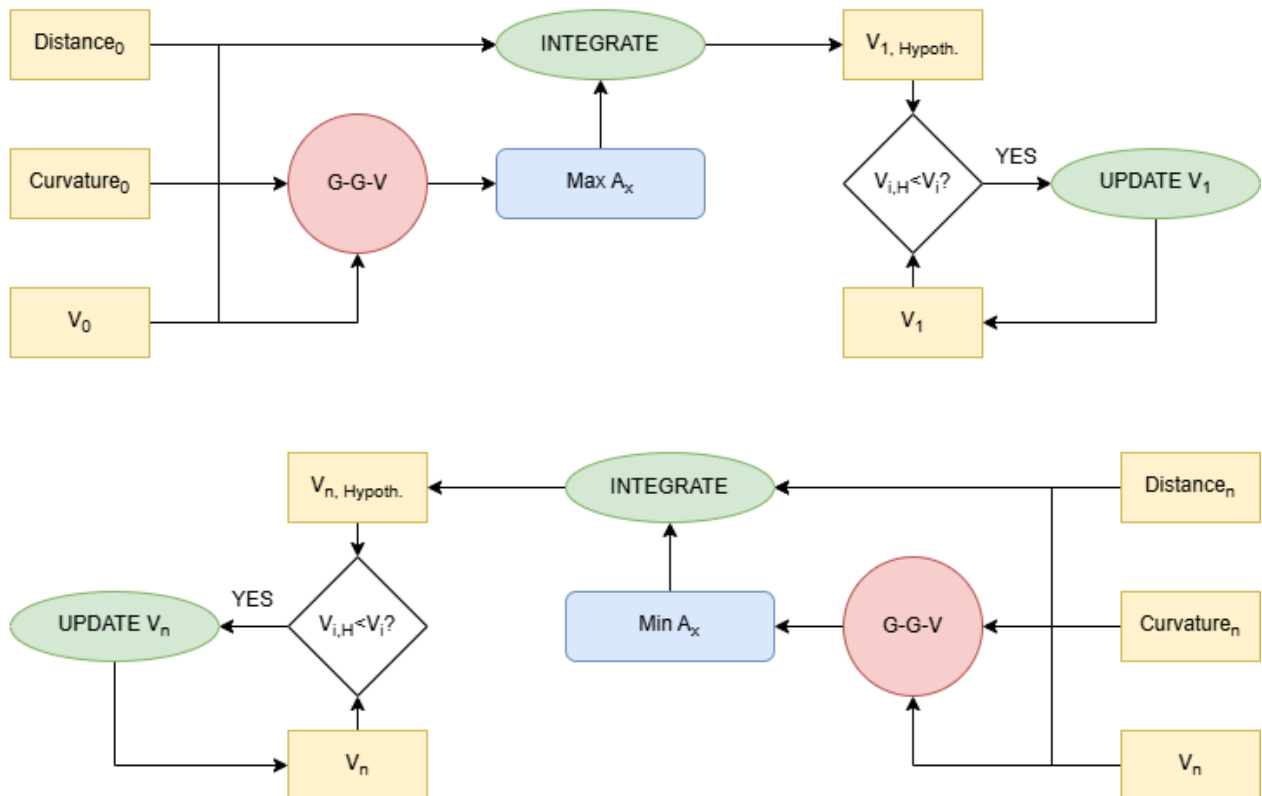
- Maximum speed, given a curvature radius
- Maximum/minimum longitudinal acceleration, given speed and a lateral acceleration (or speed and curvature)

And knowing (from the Trajectory class) curvature for every point along the track, we can code an iterative algorithm whose steps are represented below. The initial step, which we'll call just once, is the following:



For all points along the track, we compute, given the curvature, the maximum cornering speed. This will provide an initial estimate for the speed array, which, at every point, will only be limited by the maximum lateral acceleration the vehicle can produce in a corner.

Then, we will alternately call the following two procedures, iteratively:



Here, we'll go *back and forth*: for all positions along the speed array, we will first calculate, considering current curvature, lateral acceleration, which we will feed into the G-G-V model to retrieve the maximum achievable longitudinal acceleration.

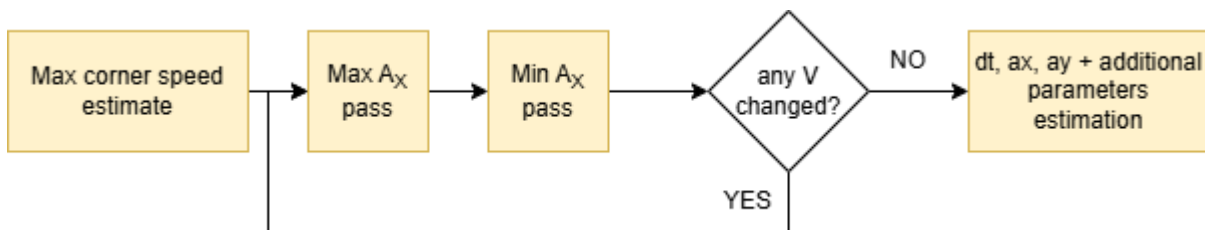
Knowing the distance between the current and the following point along the track, we will integrate to find final speed.

If such found speed is greater than the one we had found at the previous step, it means the constraint introduced by longitudinal acceleration continuity is less severe than the previous one (either maximum cornering speed, or a previous iteration of forward/backward passes), and we need not to modify its value. If it is lower, we will replace the previous speed estimation with the current one.

We then need to repeat the same procedure, but going backwards: again, we calculate lateral acceleration using speed and curvature, and we ask the G-G-V model to retrieve the minimum longitudinal acceleration the vehicle can produce. We again integrate knowing distance, and we compare, with the same criterion, speed with its previous estimation.

We will keep track of whether any speed has been replaced during these passes: whenever our speed array survives a sequence of forward and backward pass with no correction, we can end the procedure, with a known speed array.

With the following and final step, we can now compute, knowing distance, initial and final speed, and curvature, longitudinal and lateral acceleration for each step, plus, obviously, the elapsed time. If our G-G-V model is complete enough (mine, at this time, isn't), we can even extract information about driver controls, sideslip, and other relevant previously mapped quantities.



Does it work? A trivial attempt

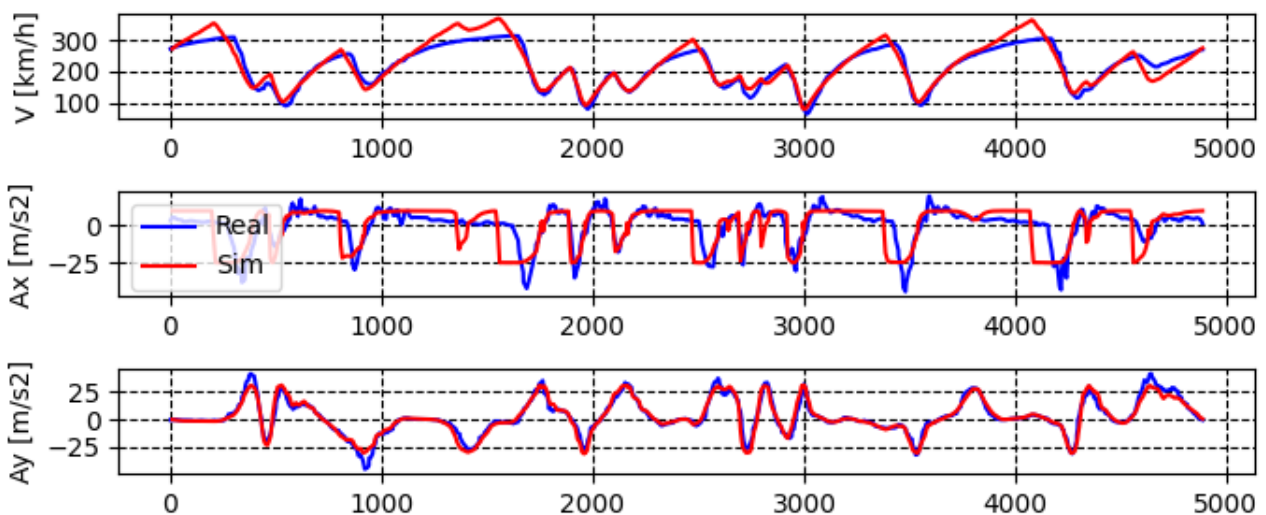
To verify the performance of the algorithm without, for the time being, adding the uncertainty of a full vehicle model previously mapped to a GGV, I defined a simplified G-G-V class as a vehicle which is able, at any speed, to deliver an *elliptical* A_x - A_y shape, defined with few, simple constant properties.

```

55 # Extremely simple, no offset, no nothing, different levels of
56 # longitudinal acc/brk and lateral
57 class EllipticGGV(BaseGGV):
58     def __init__(self):
59         self.max_ax_acc = 8.0           # m/s2, maximum longitudinal acceleration, acc phase
60         self.max_ax_brk = 12.0          # m/s2, maximum longitudinal acceleration, brk phase
61         self.max_ay = 10.0              # m/s2, maximum lateral acceleration
62         self.swa_max_ay = 100.          # deg, SWA at maximum lateral acceleration
63         pass
64
65     def get_max_ay(self, vel, **kwargs):
66         return self.max_ay
67
68     def get_max_ay_knowax(self, vel, ax, **kwargs):
69         if ax > 0:
70             axm = self.max_ax_acc
71         else:
72             axm = self.max_ax_brk
73         # Sanitize input
74         axr = min([abs(ax), axm])
75         aym = self.max_ay * (1. - (axr / axm)**2)**0.5
76         return aym
77
78     # Generalization of the following two functions
79     def _get_max_ax_knoway(self, vel, ay, acc = True):
80         if acc:
81             axm = self.max_ax_acc
82         else:
83             axm = self.max_ax_brk
84         # Sanitize input
85         ayr = min([abs(ay), self.max_ay])
86         ax = axm * (1. - (ayr / self.max_ay)**2)**0.5
87         return ax

```

I coupled it with the track data from Lando Norris' 2024 pole position at Singapore GP, and (after some parameter tweaking) I obtained this:



Which I found honestly surprising. Sure, it did not closely match the speed profile, and all the shortcomings of a basically speed-unlimited powertrain are clearly visible in the Ax plot; nevertheless, most peaks and valleys are clearly matched and, where the friction-limited-acceleration hypothesis *more or less* holds (the area around x = 2000m), all three time histories are quite decently followed by the model.

A slightly less trivial attempt – basic F1 model

The generalization of a generic GGV simulation into a true G-G-V object is a problem I am still looking for a satisfactory solution to, especially desiring to somewhat limit the computational time

required. While working at a hypothetical class based on a single GG simulation, plus multiple max-Ay simulations and multiple max/min-Ax ones, I thought about a stopgap model to easily draw a GG and the required ancillary functions with just a handful of known data and very strong constitutive assumptions to help me debug the *main* class.

The model needs the following data:

Symbol	Description	Unit
P	Engine power	kW
m	Vehicle mass	Kg
μ_x	Friction coefficient – Longitudinal direction	-
μ_y	Friction coefficient – lateral direction	-
C_{DS}	Drag coefficient (multiplied by front area)	m ²
C_{LS}	Lift coefficient (multiplied by front area)	m ²
φ_s	Fraction of static vertical force acting on tractive wheels	-
φ_A	Fraction of aerodynamic lift force acting on tractive wheels	-

The last two coefficients are needed in order to limit our vehicle's ability to transmit a tractive force in a different way than braking, due to it having to deal with two rather than four wheels. In such a model, we can express maximum lateral, and max/min longitudinal accelerations as a function of speed as in the following formulae:

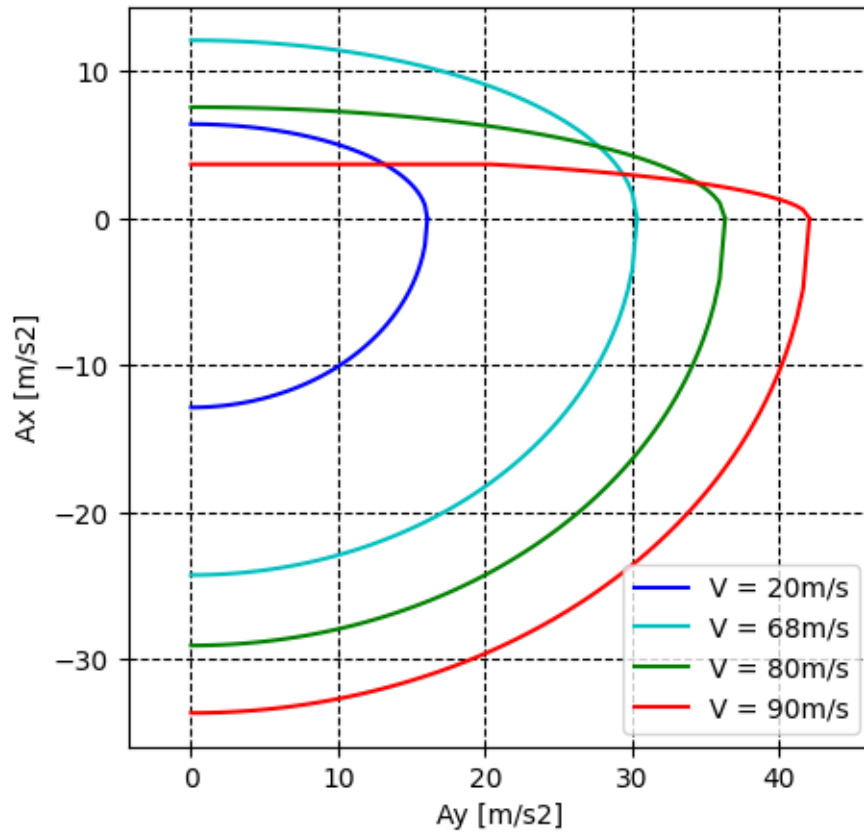
$$\begin{aligned}
 a_{y,max} &= \mu_y \left(g - \frac{1}{2m} \rho C_L S V^2 \right) \\
 a_{x,min} &= -\mu_x \left(g - \frac{1}{2m} \rho C_L S V^2 \right) \\
 a_{x,max} &= \min \left(\mu_x \left(\varphi_s g - \varphi_a \frac{1}{2m} \rho C_L S V^2 \right); \frac{1}{m} \left(\frac{P}{V} - \frac{1}{2} \rho C_D S V^2 \right) \right)
 \end{aligned}$$

In other words:

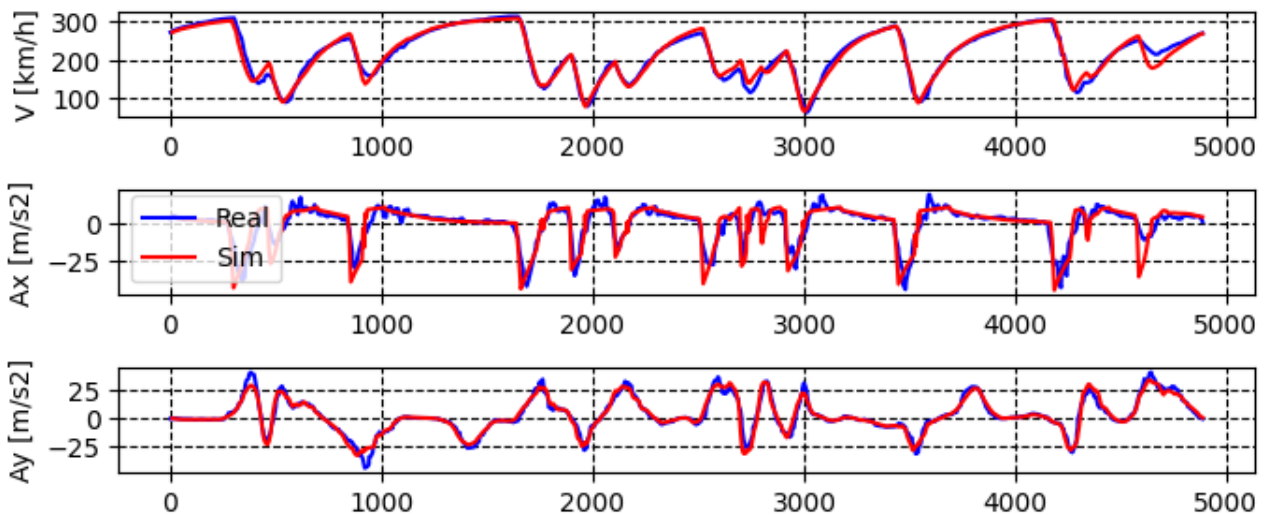
- Maximum lateral acceleration is determined by lateral friction and by the lift force
- Minimum longitudinal acceleration is determined by longitudinal friction and by the lift force
- Maximum longitudinal acceleration is given by the minimum between a fraction of total available friction force (with separate static/aero scaling factors), and power-limited force.

The last assumption vaguely coincides to assuming that the gearbox is always able to fully transmit the maximum engine power notwithstanding engine and wheel speeds. Naturally the choice of what to consider and what not is completely debatable: most notably the choice to part with longitudinal load transfer to remove wheelbase and height of the CoG from the parameter set is likely to be controversial.

About combined accelerations (i.e. the *shape*) of the G-G-V diagram, I am using two separate strategies to handle the case in which longitudinal acceleration is limited by friction, or power. In the first case, the shape of the curve is elliptical, with an amplitude given by the maximum acceleration. In the second case, the curve gets “flattened” in its positive-acceleration part, leading to the following shapes at different speeds:



Attempting the same exercise of the previous case, and with some optimization of coefficients, we obtain the following plot:



Which, while still rough, drastically improves correlation in speed and longitudinal acceleration.

Some fun with optimization

After some attempts at manually trimming the coefficients of the model to more closely match the real time histories, I decided to feed the problem into a genetic algorithm to see whether it could do it better: obviously, it could. More interestingly, after imposing mass and power as constant values (respectively, to 800kg and 768kW), I noticed that multiple runs of the same optimization problem led to rather consistent results.

Could this hyper-simplified model extract some insights about vehicle set-up in different racetracks? The following is a synthesis of results of averages of three runs (per each track) of genetic optimization (500 individuals, 40 generations), imposing, as a target, the best lap during qualifying (car and driver may and do vary), and minimizing speed differences along the array:

Variable	Min	Max	N_bit	Singapore	Monza	Monaco	Austin
μ_x	1,0	5,0	8	1,71	1,55	1,24	1,59
μ_y	1,0	5,0	8	2,01	2,06	1,74	2,45
CDS	0,3	5,0	16	1,89	1,54	2,65	1,79
CLS	-10,0	-0,3	16	-3,72	-2,77	-5,16	-2,51
ϕ_S	0,1	0,9	8	0,46	0,47	0,61	0,57
ϕ_A	0,1	0,9	8	0,27	0,64	0,90	0,87
P	768			768	768	768	768
m	800			800	800	800	800

Where we can see that the friction estimation is *to some degree consistent*, and that, more interestingly, we can clearly spot the dependency we expected between track type and aerodynamic coefficients. The constants placed to scale the effects of static and aerodynamic vertical load are far less noteworthy, as it “feels” that the algorithm just uses them as additional degrees of freedom to fine-tune friction limiting effects on positive acceleration, rather than spotting physical relations.

Aerodynamic coefficients in themselves are higher than I expected, which may be caused by an over-estimation of available torque at about every speed but the maximum one, which forces the optimal set of values to be quite limiting in that regard.

Nevertheless, as to how simple the model is, I find it to be a neat estimator for these two set-up variables.

Conclusion

Through these past months attempt to build lap time optimization software, I ended up finding what might be the most convoluted way ever to discover that aerodynamic load is higher in Monaco than in Monza. I know it is common knowledge, and the algorithm in itself is well-known, too.

Which perhaps makes for a good occasion to answer the inevitable question of why should one invest time into this, which today I think I found a fitting similitude for.

The automotive industry is an incredibly interesting place to work in, but it’s also understandably utilitarian. You come across spots you’d like to know more, to study in more detail, to visit in a calmer way, and oftentimes you just have to run somewhere else, because that’s where the project is going, and what makes sense to do.

This kind of research is just coming back as a tourist.

References

- [1] Tracing Insights, "Tracing Insights 2024 season," 2024. [Online]. Available: <https://github.com/TracingInsights/2024/tree/main>. [Accessed 13 10 2025].