

# Lap Time Simulation: Going fast(er) at Karting Vendrell

*Andrea Patrucco, Vehicle Dynamics Engineer, Applus+ IDIADA Spain, 28/10/2025*

*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

Since I started working at IDIADA, the karting circuit ad El Vendrell, which is around 10km far from the office, has been a recurring venue for after-work fun with colleagues and friends. It is a 1.2km track built on a slightly hilly terrain, with a very nice variety of corners, and it's been formerly part of national events.



As most of such places, while also being open for private track sessions with own karts, Karting Vendrell offers some basic go-karts for rent, with the classic 10-minute session or “grand prix” option with a qualifying session followed by a 10-laps race.

While I always accepted a karting plan for the evening enthusiastically and I never had any ambitions of being a fast driver, I recently grew somewhat frustrated at my apparent inability to improve on my lap time. Karting Vendrell sends out an email with basic timing statistics each time you visit them: by analyzing the sequence of such documents over the course of the past four years, I could mathematically confirm that my performance remained almost perfectly stable, and that I could only improve my personal best from the 1'15"4 of the very first time I visited to 1'14"8 of the last, with some stochastic deviation due to traffic, weather, and kart variability along the years.

So, when I recently started trusting the results of my lap time simulation software, I could not help but wonder, can I use this to help me go faster around the track? Short answer, yes. Long answer, the twenty pages below.

## Trajectory Optimization – Why

Through some conversations with colleagues and acquaintances, I got the impression that the interest in trajectory optimization in motorsport and even in the lap time optimization field is fewer than in the other part of the problem, that is, given a trajectory, optimizing the set-up to make a vehicle go faster around it.

The general consensus seems to be that any good driver will be able to follow, with little variation, a *good* trajectory given a track and a vehicle class (LM, F1, GT4 for example), and that it is much easier to just work with that trajectory for example by logging its GPS data, than to theorize an optimal one. A track has its specificities and its quirks, for example different grip in different areas, curbs you can and you cannot jump onto, limits you can run onto and that you can't: modeling all of this to a degree where it becomes precise enough to suggest changes in the order of what a professional driver can do may be extremely difficult and time-consuming, and ultimately not worth it.

The basic assumption in all of this is, however, that the driver **will** find the optimal trajectory and consistently follow it, which may be the case for professional racing, but for sure isn't if the driver is me. While I consider myself decent at controlling the kart – feeling the limit in stability when braking or at high lateral acceleration, correcting early, mostly avoiding spins – I know I am rather terrible at understanding where I should make it go: where to aim for the apex, whether to sacrifice a corner or the following one, and similar issues.

I hence assumed that finding a good trajectory could help me bettering the “part of driving fast” I probably understand the least.

## Track and Trajectory

Here I'll briefly recall what I mean for *track* and for *trajectory* (in modeling terms) from my previous post about this topic [1]: a *track* represents the surface you're supposed to drive onto, while a *trajectory* is a line you are supposed to follow.

The former imposes constraints on the latter (for example, each point of the trajectory should lie within the limits of the track surface), and, given a “convenient” representation, it can generate it: as already described a couple of weeks ago, this applies to the *quadrilateral track* object I built in Python as part of my VD code library.

This representation of the track surface is based on an ordered sequence of quadrilateral elements. Each quadrilateral elements specifies a side you are supposed to enter it, and a side you are supposed to exit, plus two “lateral” limit sides. Considering the sequence as a whole, the track becomes defined by a sequence of N transversal segments delimiting sub-elements, and by their side lines.

By specifying N values bound between 0 and 1, and with a basic convention about, for example, 0 meaning at the complete right of a segment and 1 meaning at the complete left of it, we can approximate with an interpolating line all possible valid trajectories along a given track, which constitutes a very nice basis for constrained numerical optimization.

The first task we need to cover is, then, to build this track representation.

## Building the track

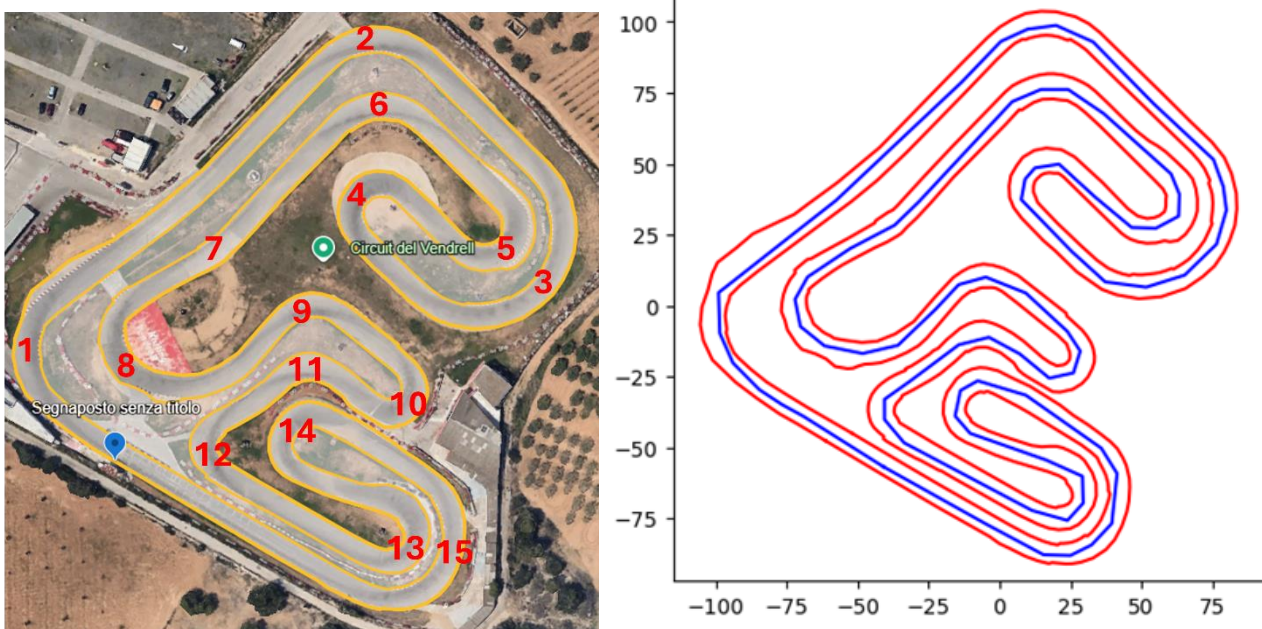
At the very beginning of this project, I toyed with the idea of building some sort of *Assetto Corsa track importer*, given that the Vendrell Circuit has been created as an Assetto track some years ago,

and this approach would immediately give my tool access to an enormous variety of tracks around the world. The issue was, even though there appear to exist tools to extract geometry data from AC.kn5 files, one is not really supposed to do it, since the format specifically contemplates encryption as a way to protect the creator's intellectual property.

Unfortunately, the only rendition of the track I found online was covered with said encryption, which means it was not meant to be shared, so I opted for a more classic approach of just hand-picking the geometry from Google Earth.

The most obvious way to define a track would be to manually draw the quadrilateral elements overlaid on the map, but it would be extremely tedious to do so in such a way to obtain an acceptable output resolution: instead, I decided to just draw three profiles on the map:

- Left track limit
- Right track limit
- Centerline



After an extremely boring hour of work, I obtained an acceptably accurate map of the track limits, where curbs have been included or excluded based on whether, according to my memory, you can reasonably use them as track space. It's clear that this is quite arbitrary: for example, in the first corner you technically can drive outside of the marked limit, however the surface will be uneven and with lower grip than in the main track, two aspects my simulator does not take into account.

The two left and right profiles can then be exported to the .kml format, which took ChatGPT around 5 seconds to build a good enough parser to read. I added a centerline for reasons I'll explain soon, used approximate Mercatore formulae to convert geographic coordinates into a distance in meters from a reference point, and I obtained the result shown to the right of the satellite image.

The strategy to transform this set of three profiles into a "quadrilateral element track" is, in synthesis, the following:

1. Define some relevant points along the centerline (i.e., sampled points themselves or some other set, for example equidistant ones).
2. For each point, find a line passing through the point and locally perpendicular to the centerline profile.

3. Find the intersections between the line we just defined and the left and right track margins
4. Define each quadrilateral element as being formed by the two intersections of the k-th perpendicular line, and the two intersections of the following one.

To do all of this, I had to take a small detour in the world of spline interpolation, that I'll summarize in the following sections. These are not essential to understand the logic path to trajectory optimization but may provide a quick reference to whomever may be interested in replicating the methodology.

### Cubic Splines - Definition

To interpolate between two successive points along a profile, I decided to use cubic Hermitian splines, mainly as a first attempt and due to the fact that they are reasonably simple and extensively documented (such as in Wikipedia: [2]).

Considering a single sequence of points  $p(k)$ , we can evaluate the interpolated value  $y$  for every real value  $x$  in the interval  $(k, k+1)$  as follows:

$$t = \frac{x - x_k}{x_{k+1} - x_k}$$

$$y(t) = (2p_k + m_k - 2p_{k+1} + m_{k+1})t^3 + (-3p_k + 3p_{k+1} - 2m_k - m_{k+1})t^2 + m_k t + p_k$$

Where  $m_k$  depends on the fitting strategy, but in a very basic one (Finite Differences), we can write it (for a set of  $N$  points identified by indices going from 0 to  $N-1$ ) as:

$$m_k = \frac{1}{2} \left( \frac{p_{k+1} - p_k}{x_{k+1} - x_k} + \frac{p_k - p_{k-1}}{x_k - x_{k-1}} \right) \text{ for } k \in [1, N-2]$$

$$m_0 = \frac{p_1 - p_0}{x_1 - x_0}$$

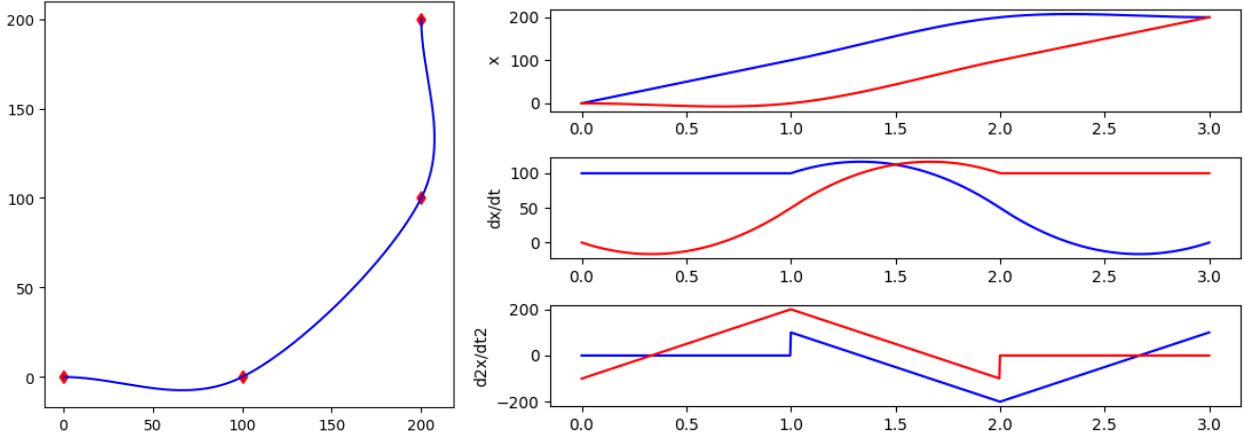
$$m_{N-1} = \frac{p_{N-1} - p_{N-2}}{x_{N-1} - x_{N-2}}$$

The polynomial formulation represented above is computationally efficient since we just need to calculate the coefficients once, allowing for reasonably fast interpolation.

To describe a sequence of  $(x, y)$  coordinates such as the ones representing a trajectory or a track limit, we can then apply the strategy above separately to  $X$  and  $Y$  data points, resulting, for each connecting segment between two consecutive points along the track, in two sets of coefficients describing the cubic trend of  $X$  and  $Y$  with “ $t$ ”:

$$\begin{cases} x(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \\ y(t) = b_3 t^3 + b_2 t^2 + b_1 t + b_0 \end{cases}$$

Which results, for an elementary example shown below, in a smooth connecting line between the points forming the initial dataset.



We should also point out that, while we are forcing the functions to have continuous first derivatives by imposing that the derivative value of each  $k+1$ th function for  $t = 0$  equals that of the  $k$ -th polynomial in  $t = 1$ , we are not imposing an equivalent condition for the second derivative, which results into some ambiguity if we want to pick  $d^2x/dt^2$  in correspondence of a given  $k$  value. This will have some repercussions on the estimation of curvature, as better detailed below.

### Cubic Spines – Properties

#### Tangent Line

For a given point along the spline profile, with the same definition of  $t$  as above, we can determine the equation of a line tangent to the profile by evaluating the first derivatives of the two polynomials, resulting in the following equation:

$$r_t(\varphi) = \begin{Bmatrix} x(t_0) \\ y(t_0) \end{Bmatrix} + \varphi \begin{Bmatrix} \frac{\delta x}{\delta t_{t_0}} \\ \frac{\delta y}{\delta t_{t_0}} \end{Bmatrix} = \begin{Bmatrix} a_3 t_0^3 + a_2 t_0^2 + a_1 t_0 + a_0 \\ b_3 t_0^3 + b_2 t_0^2 + b_1 t_0 + b_0 \end{Bmatrix} + \varphi \begin{Bmatrix} 3a_3 t_0^2 + 2a_2 t_0 + a_1 \\ 3b_3 t_0^2 + 2b_2 t_0 + b_1 \end{Bmatrix}$$

$$= \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} + \varphi \begin{Bmatrix} \alpha_0 \\ \beta_0 \end{Bmatrix}$$

#### Perpendicular Line

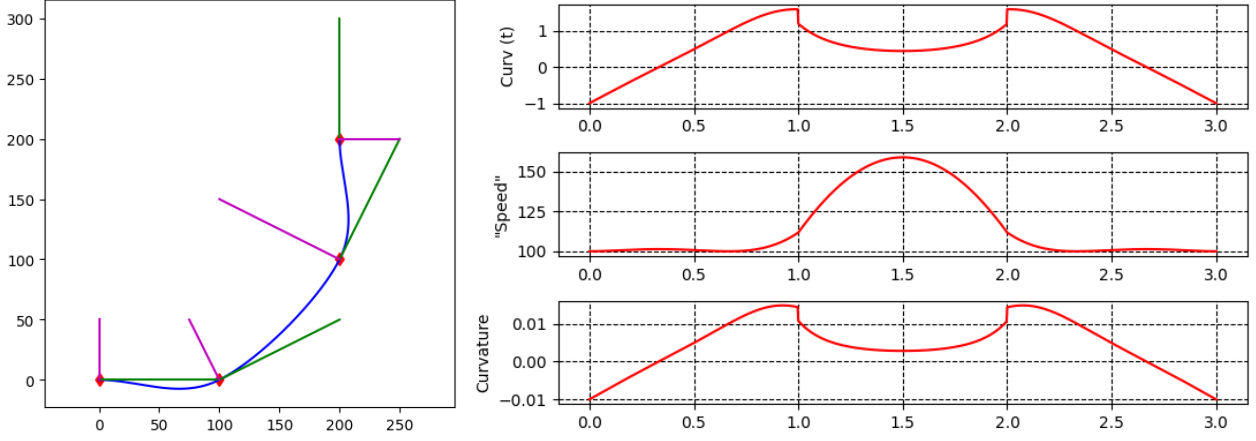
From this first definition, we can also determine the equation of the line passing through a point along the spline profile, and perpendicular to it:

$$r_p(\varphi) = \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} + \varphi \begin{Bmatrix} \gamma_0 \\ \delta_0 \end{Bmatrix}$$

Where we can define  $\gamma_0$  and  $\delta_0$  according to the value of  $\alpha_0$ :

$$\text{If } \alpha_0 \neq 0, \delta_0 = 1; \gamma_0 = -\frac{\beta_0}{\alpha_0}; \text{ else, } \gamma_0 = 1; \delta_0 = -\frac{\alpha_0}{\beta_0}$$

For the same case as before, we can see an example of tangent and perpendicular lines in the spline nodes below:



The plot to the right also presents results of the analytical calculation of curvature. Such calculation was introduced in hopes of allowing for a smoother estimation of curvature compared to a discrete one based on angle differences and distances between two contiguous segments of a trajectory. However, by effect of the aforementioned lack of continuity in the second derivative, “jumps” in curvature trends are still visible, and the analytical calculation of curvature did not lead to significant improvements in smoothness.

#### Intersection between a spline sector and a line

Given a line described by the form

$$r(\varphi) = \begin{Bmatrix} x_0 \\ y_0 \end{Bmatrix} + \varphi \begin{Bmatrix} \alpha_0 \\ \beta_0 \end{Bmatrix}$$

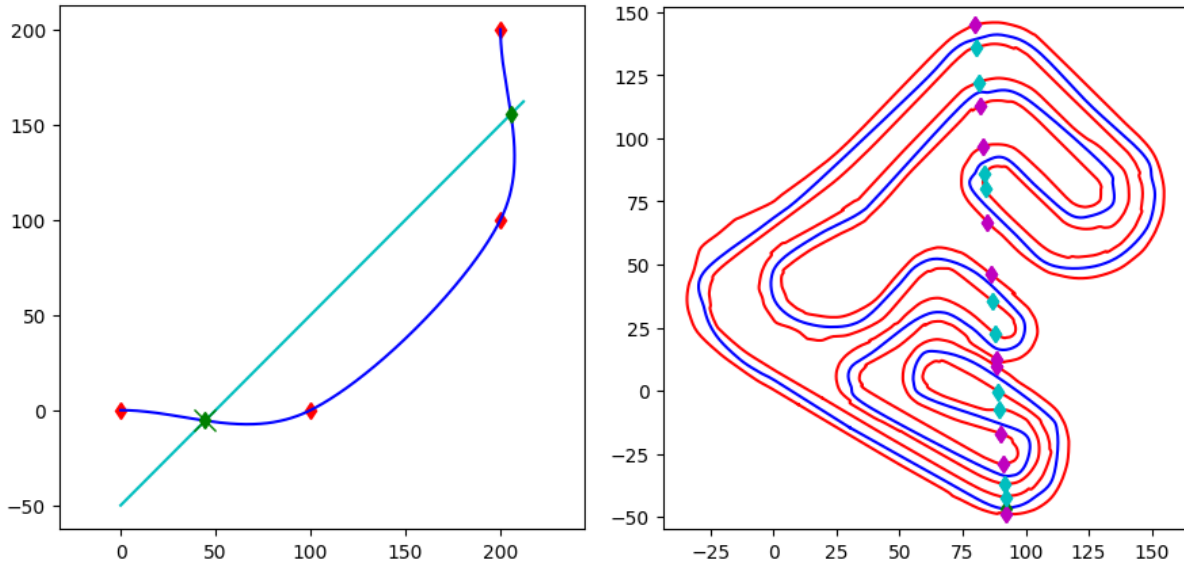
And a spline sector in polynomial form, as described in the previous section, we can find intersections by placing all equations in a system:

$$\begin{cases} x = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \\ y = b_3 t^3 + b_2 t^2 + b_1 t + b_0 \\ x = x_0 + \varphi \alpha_0 \\ y = y_0 + \varphi \beta_0 \end{cases}$$

Replacing x and y from the third and fourth equations into the first two, and then solving first for  $\varphi$  from either the first of the second depending on whether  $\alpha_0$  is null, we obtain either of the following polynomials:

$$\begin{aligned} \left(b_3 - \frac{\beta_0}{\alpha_0} a_3\right) t^3 + \left(b_2 - \frac{\beta_0}{\alpha_0} a_2\right) t^2 + \left(b_1 - \frac{\beta_0}{\alpha_0} a_1\right) t + b_0 - \frac{\beta_0}{\alpha_0} a_0 - y_0 + \frac{\beta_0}{\alpha_0} x_0 &= 0 \\ \left(a_3 - \frac{\alpha_0}{\beta_0} b_3\right) t^3 + \left(a_2 - \frac{\alpha_0}{\beta_0} b_2\right) t^2 + \left(a_1 - \frac{\alpha_0}{\beta_0} b_1\right) t + a_0 - \frac{\alpha_0}{\beta_0} b_0 - x_0 + \frac{\alpha_0}{\beta_0} y_0 &= 0 \end{aligned}$$

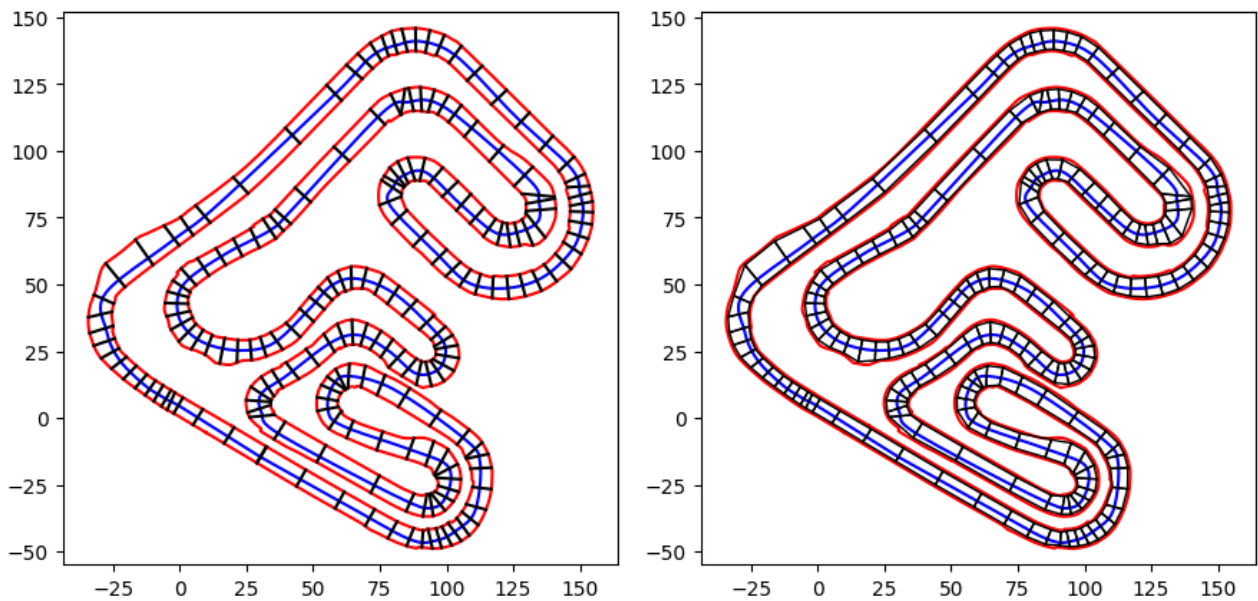
By then finding the roots of the polynomial (*numpy.roots*) and filtering real solutions, we obtain the values of  $t$  identifying a variable number of intersections.



In the above plots, we can see the results of the intersection strategy for a simple example, and over the spline-interpolated track. Intersections used to build quadrilateral elements are simply the closest ones to each point along the centerline.

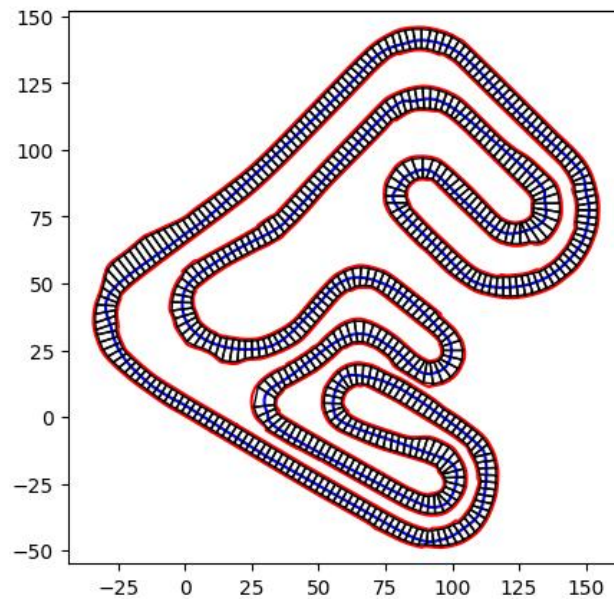
### Building the track with splines

By repeating the procedure we presented some pages ago along the centerline, we finally obtain the sequence of transversal track sectors shown in the plots below:



Which we can further, slightly refine by imposing a symmetrical reduction in width to each section as in the example shown in the right plot. This is made in order to account for the fact that we should not allow the center of gravity of the vehicle to get too close to the track limits, as this condition would correspond to that of putting two wheels on the grass, or into a barrier.

For this example, I assumed a track width of 1.2m for the vehicle I was simulating, and I hence reduced the width of each segment by 0.6m on both ends.



Another improvement which proved useful in the following optimization phases is to attempt to equally space transverse sectors in distance rather than using the original spline subdivision.

### Trajectory Optimization

Now that we have defined a track, we can focus on the next obvious part of the problem, that is, getting the trajectory which gives the best possible lap time. We already have a way to generate trajectories (by transforming an array of bounded values) and, as an objective function, we can clearly use lap time itself (for some reason and possibly just out of habit, often dealing with possibly negative ones, I'm minimizing *squared* lap time). Hence, the problem reduces to choosing a minimization strategy that somehow brings us to this optimal lap.

The choice and development of this strategy was, compared to other projects I did, way less obvious than I expected, and while the final solution I found technically works, I am not too happy about the amount of small trickery and adjustments I had to implement to achieve this. Here I'll give a brief recount of what I did.

#### Scipy.optimize.minimize

Even if unconvinced about the continuity of the solution, one should always attempt classic optimization methods, which `scipy.optimize.minimize` offers a large collection of (see its documentation at [3]). In the simplest possible attempt, you can feed it a function defined as in the following snippet of code, where the first line uses the track object to generate a collection of points, and the second one builds a trajectory object (Course) which then gets fed into the `iterate_coursegg_basic` function, which is responsible for finding the speed profile following the logic I described in my previous post [1].

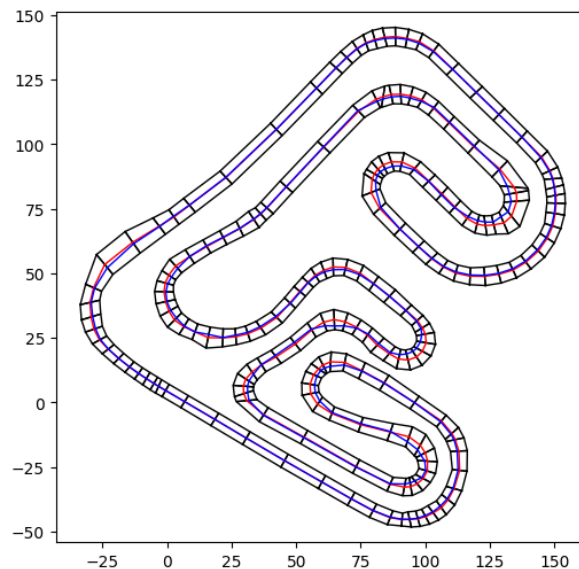
```
# Define an objective function - note that base Track is already read in one of the imports
def minfcn_laptime(x):
    pts = track.transform_trajpoints(x)
    c = Course.from_trajpoints_planar(pts)
    #c = SplineCourse.from_trajpoints_planar(pts, N_ups = 500)
    arr_conv = None #[0.2 for i in range(5)] # Use this if Spline based
    ctc, stc = iterate_coursegg_basic(c, g, verbose = False, init_speed = np.inf/3.6, \
                                     arr_convolve = arr_conv, n_max_iter=10, \
                                     dist_correction = 1., closed_loop=True)#hst['tel']['dista

    ctc['distance'] = ctc['dx'].cumsum()
    ctc['time'] = ctc['dt'].cumsum()
    out = ctc['time'].max()*2.
    return out
```

The issue with this implementation is (or at least, I think it to be) that, almost certainly due to discontinuities in the curvature estimation, the BFGS algorithm has a hard time finding a trustworthy gradient to determine optimization direction, and will stop after a huge number of function evaluations without having improved the initial trajectory in a significant way.

At first, I launched the optimizer using, as a starting point, the trajectory defined from an array of “0.5” values for each point along the track, that is, the centerline. Thinking that this could be too far from the “realistic” solution that the algorithm may find comfortable to start with, I defined two additional “simpler” solutions to provide a more fitting starting point: minimum curvature, and minimum distance.

L-BFGS-B (bounded) had no issue finding the trajectory with the minimum possible total length, however this did not prove of any help as a better starting point for GG iteration. Curvature minimization, instead, appeared to be plagued by the same issues affecting the general GG solution, namely small discontinuities in point-to-point curvature evaluation.



The best solution I found with BFGS is that of the above plot, and still starting from the centerline for the initial guess. The solver stopped after 27000 function evaluations with status 2, i.e. “desired error not necessarily achieved due to precision loss”. You can see that the overall *direction* of the improvement is correct – it is reducing cornering radius in some cases, and it is generally smoothing out some apparent discontinuities in the trajectory, however all of these changes only resulted in lowering the lap time (of a F1 vehicle whose model I already had defined in [1]) from 56 to 54 seconds.

Now, the *more elegant solution that I didn't implement*, and that would require possibly more than a weekend of pen-and-paper mathematical design would probably be:

- Define splines in such a way that they are C-2 continuous (i.e. that their second derivative is always continuous along the path)
- Find partial derivatives of curvature with respect to point coordinates of the trajectory.
- Find partial derivatives of point coordinates with respect to the input  $[0, 1]$  array.

This way, a solver could be provided with at least an evaluation of the function gradient, greatly improving its efficiency; this does not guarantee, however, that it'll be able to find a global minimum, nor that the minimum curvature solution will be of any help as a starting point to minimum lap time, which is the final goal.

One could also think of also calculating the partial derivative of vehicle speed with respect to curvature to go even further in the “initial solution” evaluation, in such a way that the first guess is not the one identifying the lowest-curvature option, but rather a “cleverer” curvature minimizer, which already takes into account the fact that further curvature reduction past a given speed (for example, the vehicle's maximum speed) is of no advantage for the subsequent step.

As I said, these developments would require a large amount of research and of code development time, hence I just decided to take advantage of the huge computational power we humans of this century enjoy, and switch to heuristic optimization methods.

### **Genetic Algorithm Trajectory Optimization**

I hence threw my evergreen genetic algorithm implementation at the problem, by roughly using the already introduced objective function, but starting from a population of random trajectories, i.e. mapping every “lateral position” along the track to an 8-bit variable describing the 0-to-1 position with 256 possible steps.

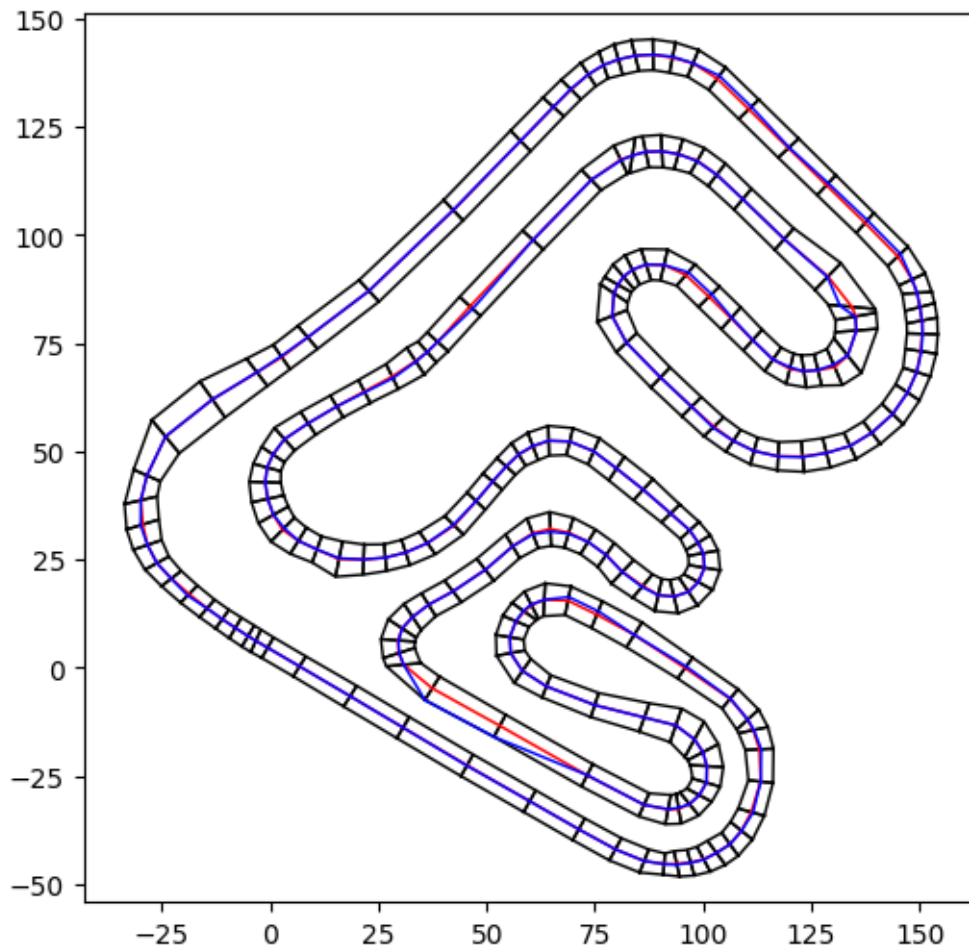
It didn't converge, not even by far. I made the first attempts at this by using the F1 point-mass model: the lap time by always staying on the track centerline would be around 56 seconds, while the first genetic attempts were struggling to break the 1-minute barrier. This gave me the opportunity to test a genetic algorithm technique I had heard about but never tested.

#### Genetic Algorithm with Inbreeding

In those cases where you are using a genetic algorithm starting from a generic set of individuals, but you already know a *relatively good* solution, you might want to “cheat” in the initial population description, artificially adding that relatively good solution to the mix.

In my case, I tried forcing the first individual of the otherwise random set with a single “cheater” whose genetic code just read 0.5 for every position in the array, resulting into the definition of the centerline.

Since, as I commented, the results of the genetic algorithm were not managing to beat this basic solution, my hope was that they could at least improve on it, i.e. that, by mixing random individuals with this single determined one, some better trajectory than the initial one could be born. And to my amazement, it worked: through the first ~30 generations of genetic algorithm iterations, the best individual remained the centerline one; however, after a while, new, improved individuals started to emerge that were still *mostly* the centerline, but with some tweaks improving speed on specific corners.



Sadly, when plotting the trajectory, I understood that the full second I had gained with respect to the centerline solution was just this: the algorithm had learnt how to get out of a corner slightly faster. And while it kept shaving some milliseconds for each iteration, the overall convergence speed was not compatible with a practical solution, hence I abandoned it.

#### An endless list of things that didn't work

In the attempt of feeding the algorithm some fitness function that he would be able to deal with, I attempted various strategies, involving, in part, conceptually redefining the solution itself, and I employed various techniques which did not bring me to significant improvements on the basic implementation, which I will list here as a warning for whomever tries to replicate this (or as ideas that might deserve further investigation):

1. Switching to an array describing “lateral movement” instead of “lateral position”: the aim here is to reduce the irregularity in initial trajectories by implicitly forcing a maximum lateral displacement between two consecutive transverse elements of the track. That is, with the previous implementation it might well happen that the vehicle is forced to “bounce”, in a tight corner, between the inner to the outer margin, like  $(0, 1, 0)$ : here, I’m forcing it to, at most, do  $(0., 0.25, 0.)$ , or  $(0., 0.25, 0.5)$ . This did help the algorithm start from more reasonable values, but constrained it to converging to something which was far from optimal.
2. Reducing the amount of bits describing lateral position along the track from 8 to 4: I hoped this would at least reduce convergence times, it didn't. It was unclear whether this reduction was detrimental to the quality of the final solution, but, since it did not offer any trade-off advantages in terms of speed compared to that, I just reverted to 8 bit.

3. Replacing the basic Course object with a spline-based one: this indeed resulted in a “cleaner” curvature profile and it allowed to apply some smoothing (because I could up-scale the trajectory by adding virtual points); however it was ultimately less efficient and effective compared to a better static re-definition of the basic track. I still think this would make a dramatic difference if splines were chosen to ensure curvature continuity.
4. Replacing the “bits mapped to lateral position of each transversal section” paradigm with a “hybrid” one where:
  - a. Some bits describe a lateral position
  - b. Some bits describe the index along track elements to which such lateral position refersThe rationale here was that in general a driver tends to navigate the track by alternatively reaching the left and right margin of it: for example, when he has to navigate a corner, it will likely position the vehicle to the outside, then he will put it to the apex, and then out again. My hope was that, by letting the algorithm free to define apex points and the “outside points” by giving it some 40 points along a 15 corner track, the solution would still be generic enough to represent a close approximation of the best possible trajectory: it wasn't. It did converge relatively fast, but it could not improve on the centerline time by more than 3-4 seconds.

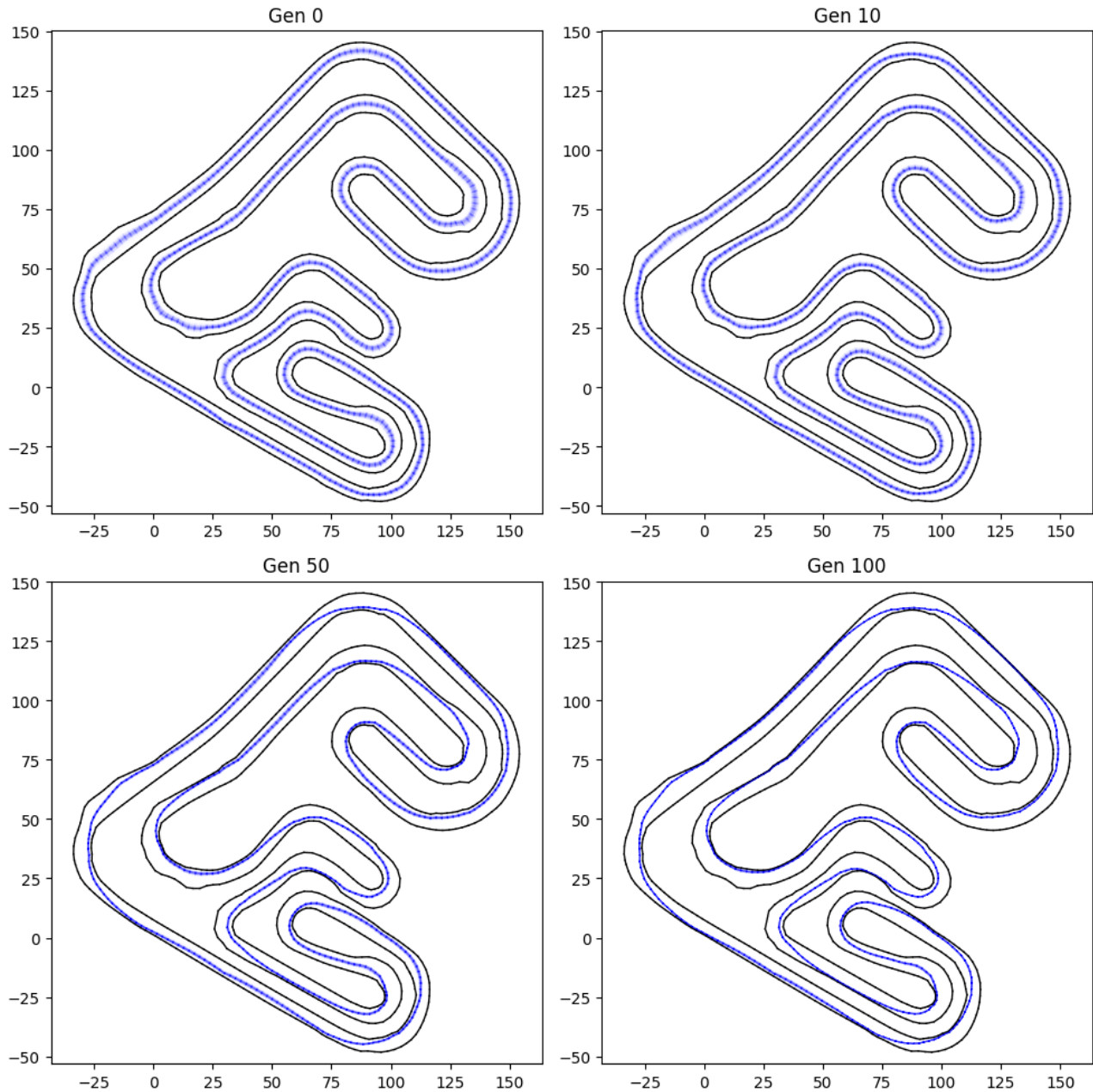
#### Some tweaks that did work

In the end, what really worked wasn't a re-definition of the problem logic nor too much parameter tweaking, but just some key improvements to the target function, that, together, finally made the algorithm converge in an acceptable amount of time, and with good repeatability.

- **Equidistant transversal track sections:** in some of the previous track plots you might have noticed that the transverse sections in corners are much closer together than the ones in straight section. This was not a specific design decision, but rather a byproduct of how I instinctively drew the centerline profile in Google Earth, which I accepted since I expected more resolution to be needed in corner sections. It turned out, lacking detail in the entry and exit sections of the corner brought the algorithm to choose unusually *outer* lines in most corners, possibly due to the mere fact of a low-curvature line being easier to draw in that part of the track. I just redefined the sections to be almost equidistant, and this tendency disappeared.
- **Lateral position smoothing:** by this, I mean that, after I transform the genetic code of an individual into a sequence of values between 0 and 1, and before I feed it into the trajectory generation function, I smooth the array by applying *np.convolve* with a triangular weighing array. This addresses the same discontinuity issues that I had tried to solve with ideas 1 and 4 of the previous list, but I guess leaves more freedom to the algorithm in lateral positioning. I still do not like the fact that, conceptually, lateral position at a given section now depends on some 10 or 20 values instead of one – we implicitly lost the clean, obvious interpretation of the genetic code into a solution – but I'll accept this for the sake of convergence speed.
- **Curvature smoothing:** this means that the curvature I extract from the trajectory is smoothed on some 5 or 10 points before being fed into the GGV iterator. Again, this is a bit controversial as there is no *physical justification* to do this. Furthermore, smoothing curvature does tend to remove curvature peaks, thus affecting greatly the maximum cornering speed, and hence, resulting in a different solution than that achieved without it. What I can say by experience is that, by smoothing over 5 to 15 meters of track, the output curvature signal is “cleaner” than the original one while respecting the underlying trends, while anything above that distance starts flattening the curve. This rather weak conclusion is, however, only valid for this specific track, and mileage may vary with a different profile or with tighter corners.

With all of these modifications, the algorithm, launched with 1000 individuals and a 200-individuals reproductive pool, is able to find an optimal trajectory in some 100 generations, which translates to 80200 function evaluations. Each evaluation, for a simplified vehicle model, takes around 50ms,

resulting in a time of around one hour and seven minutes to find the optimal trajectory. The program runs on a single core, hence it is possible to parallelize multiple attempts; instead, due to the relatively brief duration of a single function evaluation, the multi-threaded implementation of the genetic algorithm did not lead to significant time improvements, likely due to the influence of some overhead to create, launch, and destroy multiple processes.



In the plot, we have an overlay of the 100 best individuals from each generation in the track profile. As you can see, not only the trajectory becomes more and more definite with advancing generations (a typical effect of this algorithm type), but the also appear to *move*: this is most evident in corner 2 (the uppermost one here), where the line gets closer and closer to the apex, the entry moving further outside between generations 50 and 100. This is caused by the smoothing filter applied to the solutions before transforming them into a trajectory, removing at least visually some of the underlying point variability.

A nice property of this optimization procedure is that, as I verified for several runs, the algorithm will converge to extremely similar solutions when launched multiple times with the same parameters,

fastest lap time only diverging in the order of milliseconds. This suggests that the solution is unique, apart from the physiological tendency of genetic optimization to find it in different ways.

Still, I think this procedure is way under-optimized with respect to some other possible formulations. A particularly interesting example I found proving this is [4], the webpage describing *SET-UP Driving*, an application by NT-Project, a company owned by engineer Omar Tabacchi. Apparently, the tool was able to optimize trajectory depending on kart characteristics: since the copyright on the page states 2005-2019, I assume the author managed to do all of this with the computational power from 20 years ago, in a much more efficient way than I did: this might motivate me to invest more time into a more *deterministic* solution in the next months, however, for the sake of sleeping sometimes, I'll live with the shortcomings of the current one.

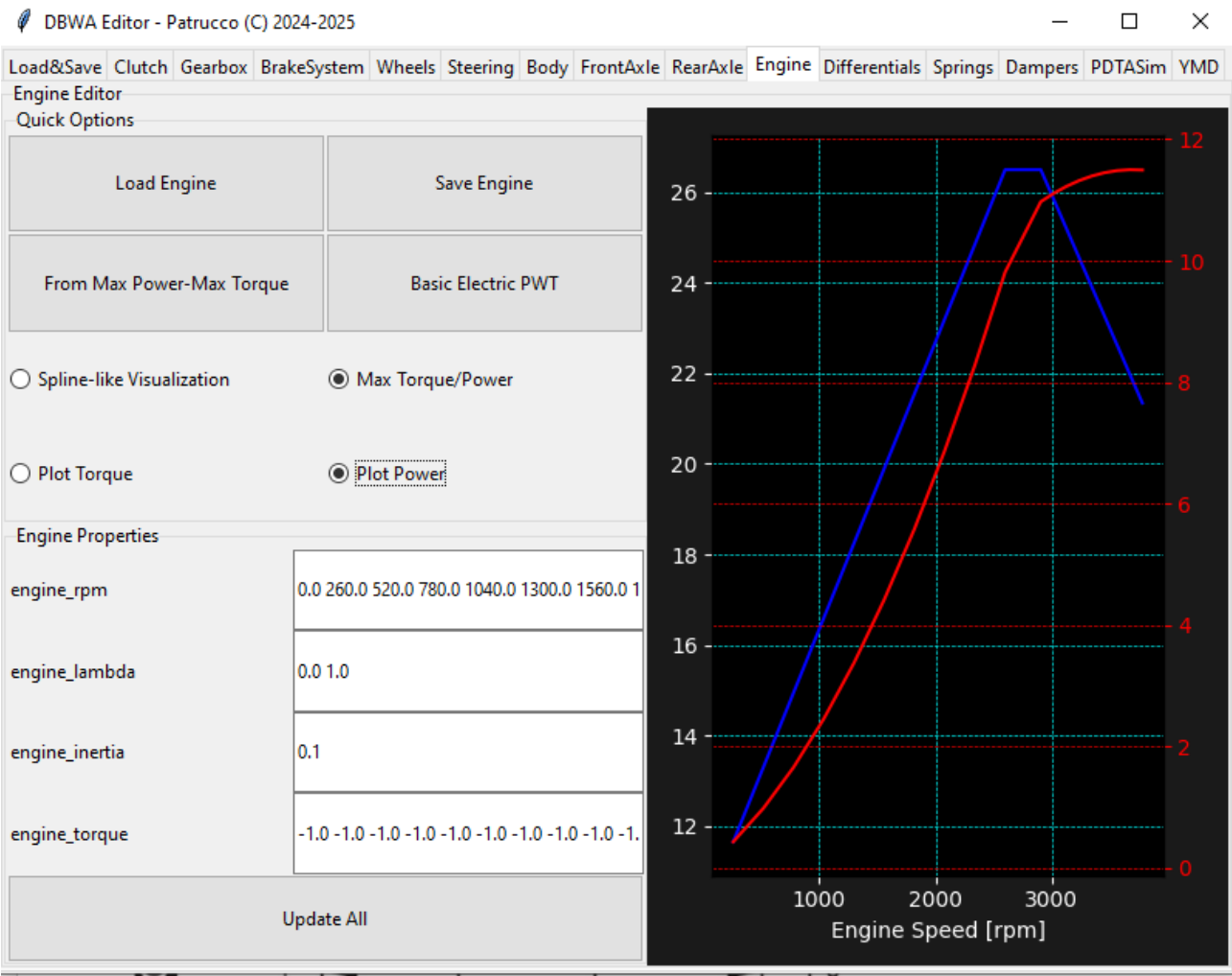
### **Building a Kart Model**

A big element we still haven't described is the vehicle model: under the assumption that the vehicle parameters have an influence on the choice of trajectory, this part of the problem is of enormous relevance when searching for a solution. Since I still haven't got a way to transform a vehicle model into a proper GG-V "object", I will just use the model as a reference to then manually draw the shape of a GG diagram, as explained in more detail further below.

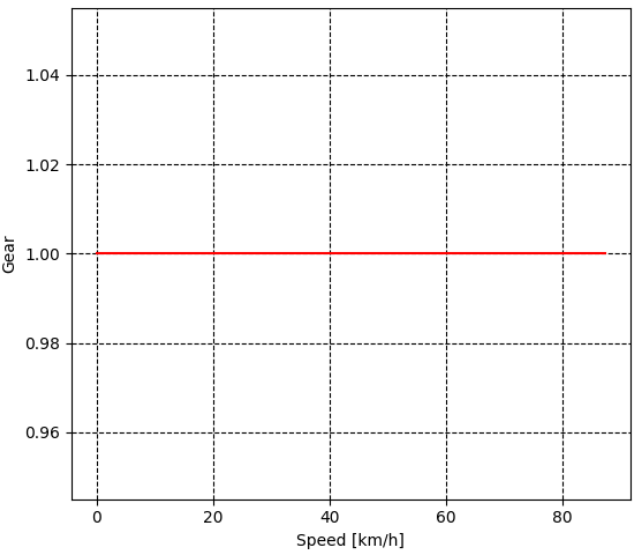
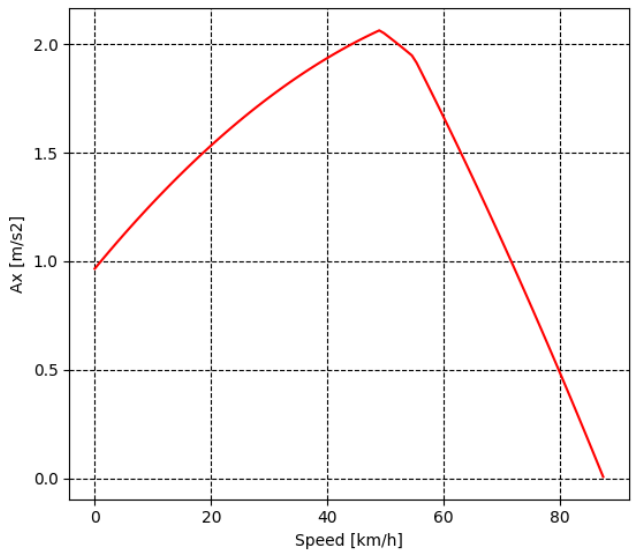
I could not identify the make and type of the rental karts available at Karting Vendrell, so I just made some assumptions on their characteristics to help me build a *first guess* model. My first assumptions were:

- 215kg mass (145kg kart, 70kg driver)
- 1.1m wheelbase, 40% front weight distribution
- 1.2m track, 55% rear lateral transfer balance
- 1.5g maximum lateral acceleration
- 150mm wheel radius (rim + tire)
- Rear axle-only brake, capable of producing enough force to lock the wheels at maximum pressure
- Parallel steering geometry

For the engine and transmission, I assumed the rental karts to be fitted with a Honda GX390 engine, whose torque curve I found in [5] and replicated into my model.

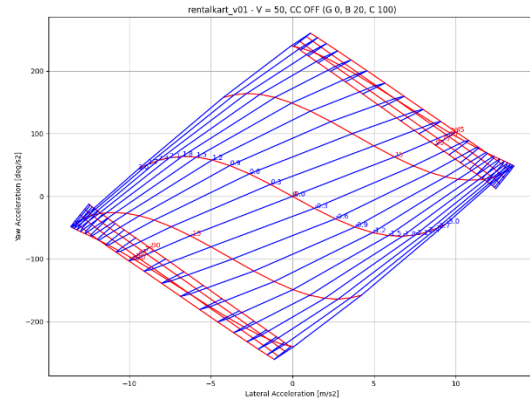
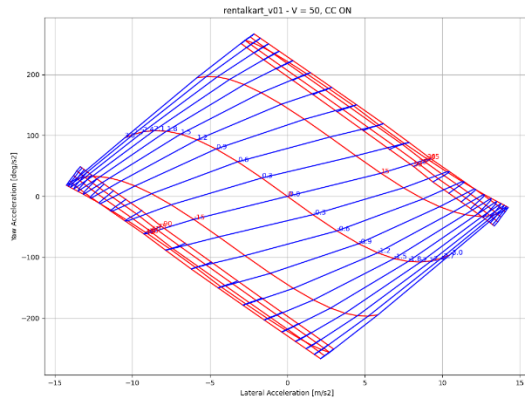


I assumed a total transmission ratio of 3 between engine and wheel speed, which was **a mistake** since it appears that the maximum rotational speed of the Honda GX390 engine is 3850rpm, while my solution leads to a 4500rpm maximum engine speed. Since I realized about this after obtaining most conclusions in this article, I will still assume the longitudinal acceleration profile I extracted from it to be realistic.



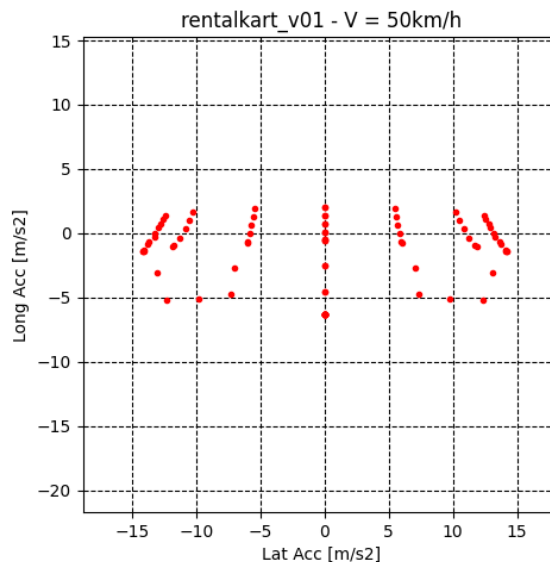
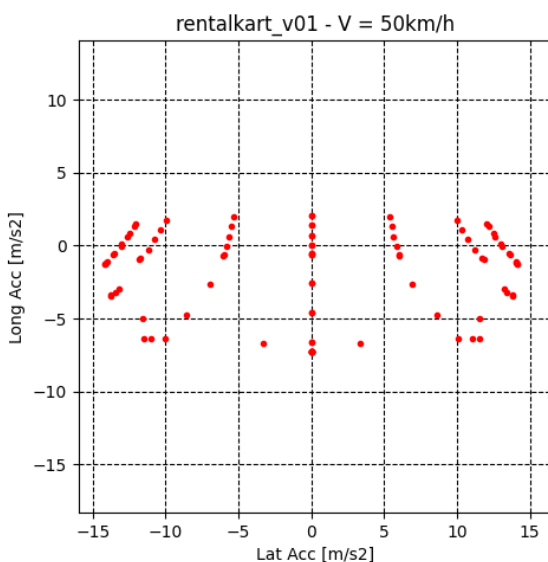
In these settings, maximum speed is around 85m/s, with a maximum longitudinal acceleration of  $\sim 2\text{m/s}^2$  obtained at circa 50km/h. Maximum braking acceleration is of about  $8\text{m/s}^2$ , as a result of high grip tires and of only one braking axle.

I thought it would be fun to try and build some yaw moment diagrams for this vehicle:



In the left plot, we see the YMD of the kart at 50km/h at constant speed: the set-up is understeering and stable at low lateral acceleration; while still characterized by limit understeer, the amount of *stability* noticeably reduces due to the rear tires reaching the limit.

In the right plot, we see the YMD of the kart at 50km/h applying 20% of brake pedal position: at lower lateral accelerations, the kart remains stable (though the margin is now lower than before); moving towards higher lateral acceleration values, stability noticeably reduces, and the vehicle is now prone to *terminal oversteer*. This is at least “conceptually” realistic, as, with such low-powered kart, it is very difficult to spin out on acceleration sections, while reaching an unrecoverable slide during a braking phase is much more common.

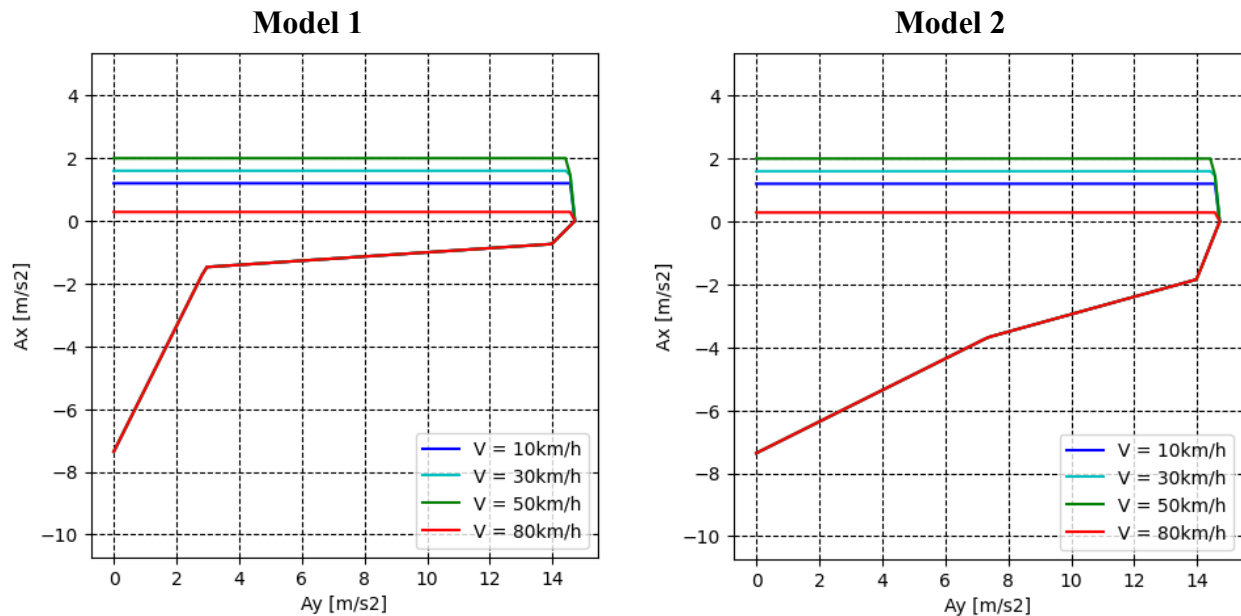


Similar considerations can be made when simulating zero-yaw-moment conditions (as I described in [6]), obtained the *theoretical GG* diagram: the difference between the left and the right plot is weight distribution: 40:60 F:R to the left, 50:50 to the right: we can notice that a more rear-biased mass distribution results in a more stable vehicle in braking phase: the driver is able to reach longitudinal acceleration values very close to the absolute maximum even while producing significant lateral

acceleration: this becomes trickier with a more front-biased distribution, in which not only the minimum  $A_x$  is limited by a different static distribution, but also, such minimum value can only be reached with a completely straight steering wheel, since every disturbance results in a yaw moment contradicting the initial “steady state” assumption.

### Drawing GG Diagrams

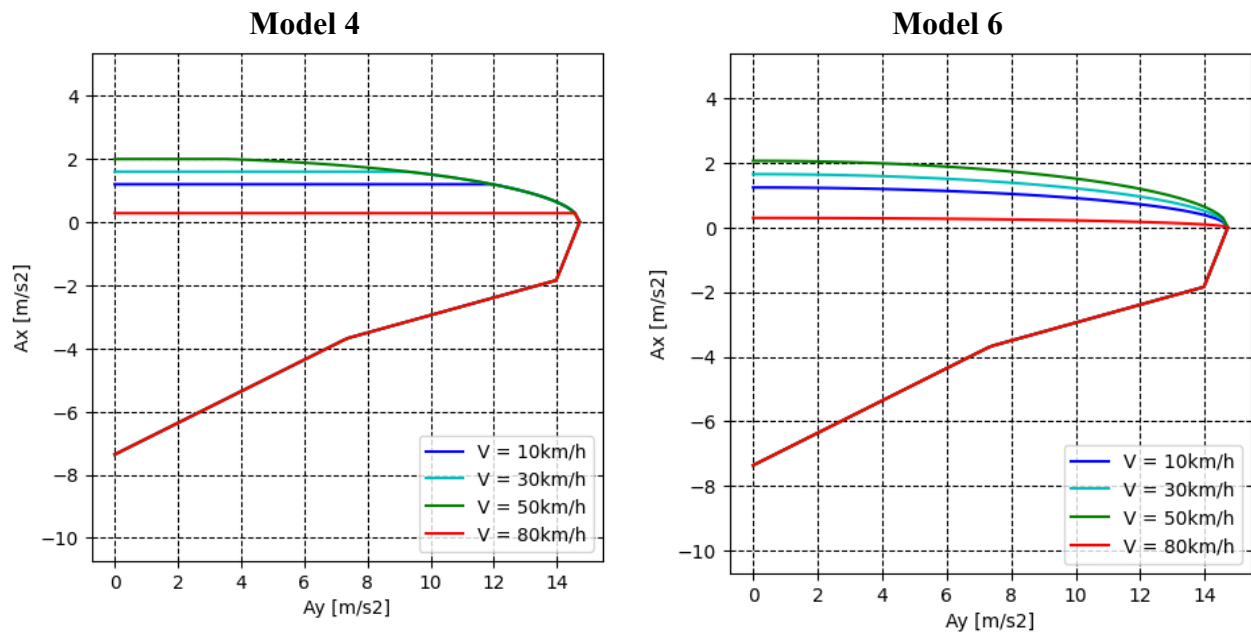
Such dependency of results on mass distribution led me to build two separate GG shapes, which you find in the plots below:



You may notice that I was still very pessimistic, about braking-and-cornering capabilities, in the definition of these lines, compared to the GG diagrams I had found. This is quite arbitrary, and mainly justified by wanting to achieve a GG profile I *could be expected to reach*, as a non-professional driver with very limited left foot sensitivity. Since I was particularly unsure about where exactly to place the line of achievable acceleration, I created two options, out of which a more “cautious one” (Model 1), in which the driver is basically only allowed to brake in a straight line, and a more “flexible” one, where the driver takes, to some degree, advantage of the relative stability of the kart when braking.

You may also see that the line describing limit power-on acceleration is perfectly straight: this is not reflected by the GG diagram, where, even when longitudinal acceleration is power-limited, the maximum reachable amount still sees the influence of lateral acceleration production, probably due to the longitudinal component of lateral force produced by the front tires.

When analyzing results, I had some doubts about the influence of this specific shape of longitudinal acceleration, which brought me to define some additional models:

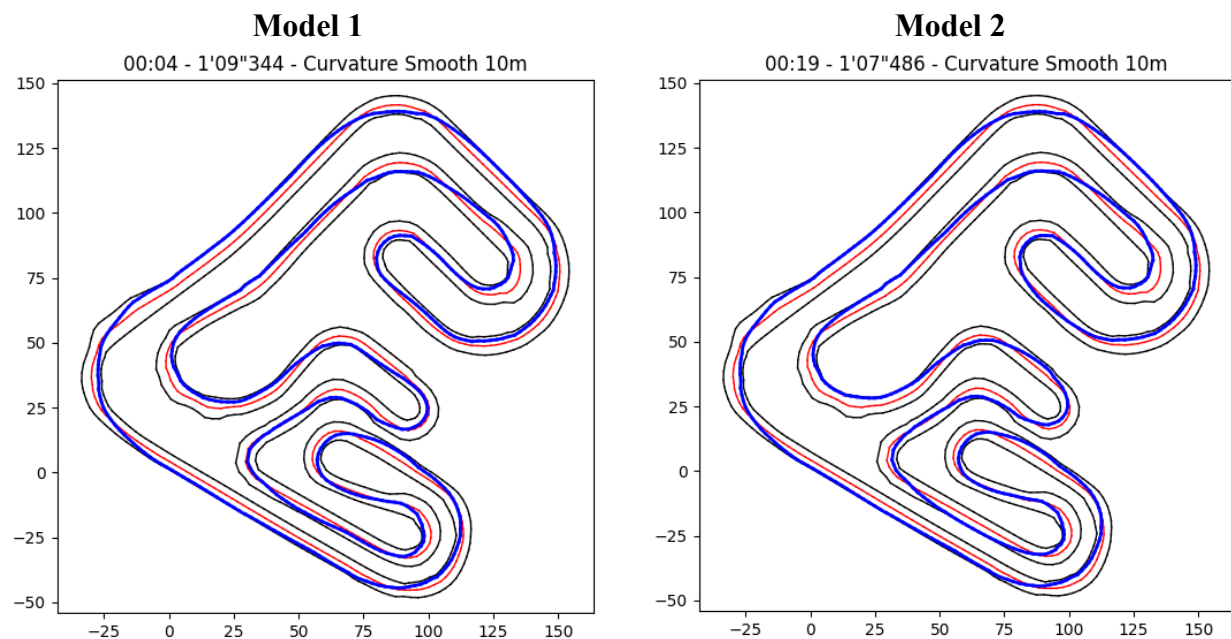


Which basically consist of *Model 2* with a smoother transition between the straight and the elliptical shape of lateral acceleration (*Model 4*) or a completely elliptical one (*Model 6*). This, however, did not produce the changes in the trajectory I was expecting, as better explained further below.

It might be worth to note that the “*Model 2*” GG diagram achieved a lap time of 1'23"4 on the *centerline* trajectory, which initially led me to believe it to be quite close in performance to the real vehicle (at almost nine seconds slower than my time, obtained with a more reasonable trajectory). I later realized that the performance I am obtaining are quite over-estimated, probably due to the mistake in the transmission ratio I explained before, and maybe due to some optimism in maximum lateral acceleration, as well.

## Results

So finally, with a reasonably-tuned trajectory optimizer and some arbitrary assumptions about the kart model, I was ready to launch the solver to the following results:



Which I find of particular interest because, while the two models “agree” on the line to follow for most of the track, they diverge in two specific areas:

- The section between Curve 3 and Curve 4: here Model 1 moves towards the inside only to get to the outside again.
- Section between Curve 9 and Curve 10: here, again, Model 1 remains at the inside only to move towards the middle just before negotiating the corner; Model 2 follows a much more *natural* line inside-outside-inside.

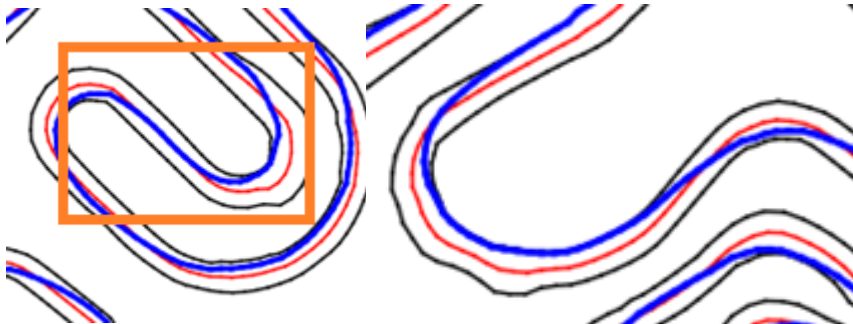
I have no certain explanation about why this happens, my best guess being that Model 1 is so limited in braking stability that it tries to achieve a perfectly straight section of track before even the entry phase of each corner just to enter it without spinning.

A very interesting byproduct of this double-model analysis is that we can answer the question: “what happens if we switch trajectories?” i.e., what happens if we run the GG-V iterator for Model 1 on the optimal trajectory found for Model 2, and vice versa?

| Trajectory\Model | Model 1 | Model 2 |
|------------------|---------|---------|
| Trajectory 1     | 1'09"3  | 1'07"9  |
| Trajectory 2     | 1'10"2  | 1'07"5  |

That is, both models go slower around the track when following the trajectory optimized for the other one. This not only reinforces the feeling that the procedure is, indeed, finding the *optimal trajectory*, but must importantly, it confirms that the optimal trajectory does indeed depend on the vehicle set-up.

This is not enough to disprove the basic assumption that many lap time optimization tools follow, of just using a fixed trajectory to optimize the set-up (to do that, we would need to prove that *the optimal set-up is different for the two trajectories*), but I think it at least casts some doubts.



A specific section of track that made me quite doubtful about the trustworthiness of my method was the one highlighted here above (left). Both models seemed to agree that the most convenient way to get out of curve 4 and into curve 5 is by exiting the former in the middle of the track without exploiting full track width, then directly entering the latter without attempting to increase the entrance radius.

This looked odd to me and to everybody I showed this, as we thought a more curved profile would be faster, or at the very least it could be advantageous to pursue a late apex in the first corner to then enter the following one faster, enjoying a higher speed for a very large section of track.

I thought the explanation was that, since the GG of both Model 1 and Model 2 is completely flat in its upper part, the algorithm was not finding any further advantage in reducing curvature exiting Corner 4: to try and prove this, I defined Model 4 and Model 6 with various degrees of continuous decrease in  $A_x$  for  $A_y$  going from zero to its maximum value. In both cases, the algorithm kept converging to the same, counter-intuitive profile.

The other section of the track in which the algorithm suggested something completely different from what I did was Corner 8 (right plot): a tight corner with an increasing radius which I tended to “split” into two corners entering rather fast and re-approaching the inside towards its end in order to prepare for Corner 9. This was less surprising, as I always felt I was losing lots of time there, hence I was more prone to accepting the algorithm’s suggestion.

So, realizing I had run out of theoretical explanations for my algorithm results, I resorted to the most obvious and supreme verification tool I could think of: just going to the actual track and trying.

### The reality test

So last Friday I just went to Karting Vendrell and booked two 10-minute sessions. My intention was to use the first one to evaluate some of the suggestions by the algorithm, and the second one to just find an average between what I usually did and the “new strategy”.

The timetable of the first session read the following:

| Pos. | 1         |
|------|-----------|
| Lap  | Patrucco  |
|      | Kart 12   |
| 1    | 01:20.174 |
| 2    | 01:15.181 |
| 3    | 01:14.271 |
| 4    | 01:13.857 |
| 5    | 01:13.947 |
| 6    | 01:13.677 |
| 7    | 01:14.469 |
| Avg. | 01:15.082 |

It took three laps to improve my previous best by six tenths of a second; over the following three I could further lower it by another 5, leading to a  $\sim 1.1$  improvement over my all-times best.

I was far from the *theoretical* time of 1’07”4, however at least part of the difference can be attributed to model inaccuracies – the track record with rental karts is currently standing at about 1’11”.

### Conclusions and further work

At least for significant differences in the GG diagram shape, I found that the optimal trajectory does depend on the set-up, or at least on how close a driver is willing to get to instability. It would be interesting to try and prove the same conclusion for a more “professional racing” case to understand the importance of the simplifying assumption most software takes. Or even to try and define in a more rigorous fashion the “driver’s willingness to approach instability” in its effect over a GG, and, conversely, lap times.

Before that, there are other interesting but simpler-to-analyze variables deserving some sensitivity analysis: friction, mass distribution (affecting braking force), driving torque, and most of all total mass.

The direct conversion of a vehicle model into a GG-V object remains, for me, an open point I still need to solve to complete the original “challenge” of building a simulator, and I plan on orienting my hobby work there, after this slight detour that took very close to 100% of my spare time for two weeks.

Which I don’t really regret. Engineering is about identifying a problem, making sensible assumptions, find a reasonable solution, come back and see if you solved it. When you did, it’s just beautiful. I could have, for sure, improved my karting skills by taking a course or asking a better driver to show me how he did.

But when, after two weekends and countless evenings of model/algorithm attempts, refinements and retries, I finally crossed the finish line and saw *it had worked*, I got just a moment of sheer joy I think I couldn’t have bought in any other way.

## References

- [1] A. Patrucco, "Lap Time Simulation: steps 5, 6 and 7, and some divagation," *LinkedIn*, 15 10 2025.
- [2] Wikipedia, "Cubic Hermite spline," [Online]. Available: [https://en.wikipedia.org/wiki/Cubic\\_Hermite\\_spline](https://en.wikipedia.org/wiki/Cubic_Hermite_spline). [Accessed 26 10 2025].
- [3] Scipy, "Minimize - Scipy v1.16.2 manual," [Online]. Available: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html>. [Accessed 26 10 2025].
- [4] NT Project, "SET-UP Driving," 2005. [Online]. Available: [https://www.ntproject.com/software\\_pks\\_eng.htm](https://www.ntproject.com/software_pks_eng.htm). [Accessed 26 10 2025].
- [5] EngineSpecs.net, "Honda GX390 Engine Specs," [Online]. Available: <https://www.engine-specs.net/honda/gx390-k2.html>. [Accessed 26 10 2025].
- [6] A. Patrucco, "About vehicle models and trimming them to equilibrium," *LinkedIn*, 17 September 2025.