

About vehicle models and trimming them to equilibrium

Andrea Patrucco, Vehicle Dynamics Engineer, Applus+ IDIADA Spain, 17/9/2025

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

Recently Mike Law wrote a post on a step-by-step procedure to build a simple lap time simulator. That post sprawled a good number of responses and proposals, which not only brought up an interesting technical discussion, but also made me realize that the way I was thinking about a lap time simulator was kind of wrong.

I saw lap time simulators as glorified closed-loop controller problems: you have a track/trajectory, a vehicle model, you fit a controller onto the model and integrate it in time, and through some iteration you should get to a set of parameters, both for the controller and for the vehicle, that make it go through the track faster.

His post showed me that, while probably acceptable (and accurate) for, let's say, a double lane change simulation for a road vehicle, that approach was very inefficient compared to a GG-V based one; that is, compared to the concept where you know, for each speed, the curve describing the pareto-front of maximum acceleration in longitudinal and lateral direction, and you use that information and track curvature to find an equivalent speed history along the track.

An enormous question mark remained, however, compared to the more straightforward, controller-based approach: how do you determine, with some degree of accuracy, the GG-V diagram of a vehicle that you can use for the back and forth optimization?

The author provides, in his original post, the following answer in steps 2 and 3 of his proposed procedure:

2. Next step is the vehicle model, by which I mean a system of equations that will turn a state/driver inputs (steer angle, side slip angle, throttle and brake etc) into a set of accelerations (lateral, longitudinal, yaw) (...)

3. Next, you'll need to wrap an optimizer around the model to 'trim' it to a set of target conditions. You can use this to find out what combination of states/driver control inputs are required for the vehicle to achieve maximum acceleration along a given vector, subject to constraints like 'zero yaw acceleration'.

Which is a perfectly reasonable logical subdivision of the needed steps; however, it doesn't go too much into detail about which model (he suggests starting with a bicycle one) or which optimizer strategy one should follow. And really there's not an only solution to this, depending on how fast, how simple and how accurate you want your model to be.

I figured out that giving my personal interpretation of this might be interesting for those that are still looking for hints (the post is two months old, I came late to the party), as well as for me to receive some and improve on what I did.

So here it is: don't take it as a "guide to steps 2 and 3" but more like a realistic account of how, in some months of weekend work towards a lap-time optimizer, I *did not manage to build* a lap-time optimizer, but went through some interesting concepts, nevertheless.

Step 2: the vehicle model

A first consideration about any lap time optimizer tool is that whatever isn't in the original vehicle model won't be in the final information it'll be able to give us. We *know* that a vehicle who's capable of generating a higher lateral acceleration will be faster around a track, so we want to include some complexity and some physical parameters to be able to obtain meaningful information about them.

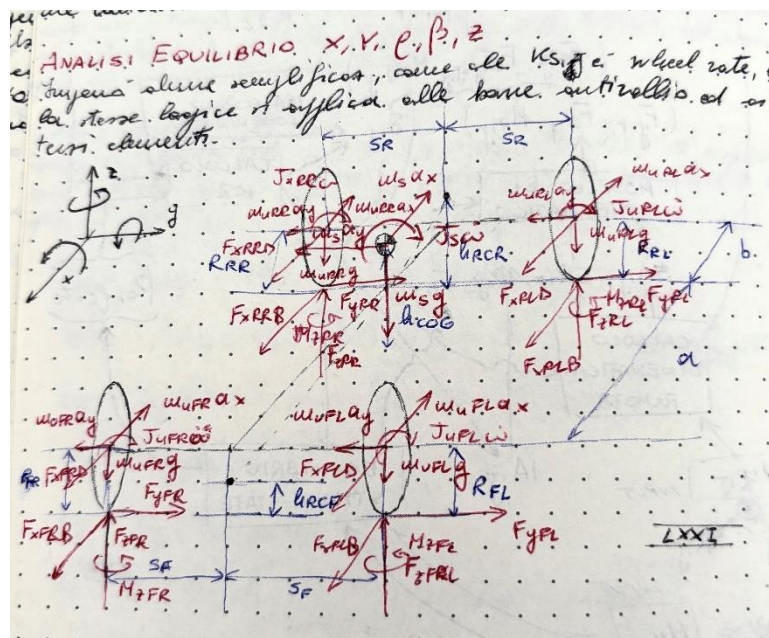
The downturn is, obviously, all those parameters have their cost in CPU time. For example, my "general purpose" model for full vehicle simulation has 14 degrees of freedom, plus engine speed. The "model" really consists of a big differential equation, in which, starting from a 30x1 state vector (all speeds and all positions), we obtain an array of 15 accelerations.

It is a relatively complete model, with Pacejka tires and spline-based kinematics and compliances, which can be used to describe transients, damping effects and non-linearities. A single simulation for a sequence of two corners and 10 seconds takes around three minutes, so a single set-up for a GT car running around let's say, Laguna Seca might take in the order of half an hour to evaluate.

Even assuming we're imposing a constant body sideslip (we'll see why this is relevant soon), the model can take up to 1 second to reach its equilibrium, hence a single *working condition* of the vehicle can be found in around 20 seconds.

Most of this simulation effort is clearly wasted, since we are looking for equilibrium conditions to start with.

In fact, instead of painfully simulating a system of forces acting on a car body and slowly converging to an equilibrium position, we could just assume that such equilibrium is always being met. Forget roll, pitch and wheel jounce transients: let's just assume that, given a set of longitudinal and lateral forces (plus aligning moment) applied by the tires, the car will instantly find its equilibrium position.



Here you can see the first of the 25 pages of an A5 notebook that took me to find the corresponding equations describing what we just assumed. And yes, I know I could find them in some book, but where's the fun in that?

With the painful renouncement to damping effects (but hey, aren't race cars very well damped, anyways?), we got rid of 7 degrees of freedom:

- Body roll

- Body pitch
- Wheel jounces (4)

There were other, more subtle sacrifices I had to do to the gods of computational efficiency. For example, the rather sophisticated 2D splines to describe kinematics and compliance had to go in favor of linear proportionality coefficients; non-linear spring functions went with them replaced by a single “k” constant. On the other hand, speed increased to around 80% real-time, and to 400% by reducing the integration frequency from 1000 to 200Hz, where the model remained mostly stable.

The model we now have has a state array composed of the three planar motions (x, y, yaw) and of the four wheel rotations (plus the engine’s). These last five soon became the next victim of my simplifying rage.

For this application we’re targeting maximum accelerations. It makes little sense to keep simulating wheel speeds in time, just to then have to implement some controller to avoid going past peak friction, when we could just *assume* that each tire’s delivering the maximum amount of longitudinal force it can.

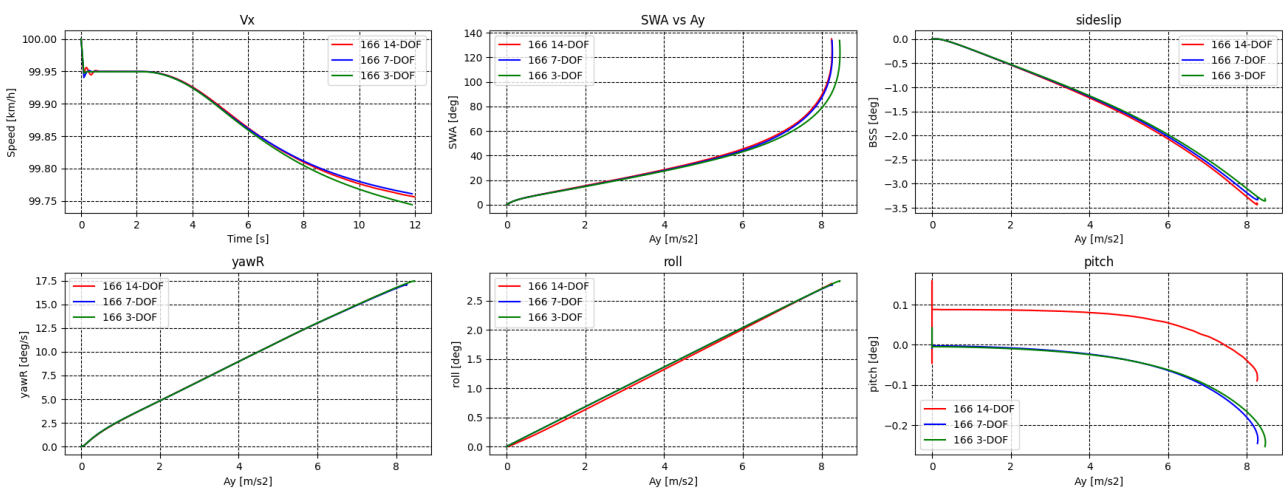
Here, the “simplified differential approach” I published here a couple of weeks ago came handy: given the wheel’s contact points rigid-body speeds, let’s just find a rotation giving us the torques withstanding torque and power balances determined by engine, brakes and the differential.

Again, more subtle approximations are involved: instead of the “complete” calculation of tire combined forces, we’ll now have to rely on a known F_x , and scale the corresponding F_y as it follows:

$$F_{y,combined} = F_{y,lateral\ only} * \sqrt{1 - \left(\frac{F_x}{F_{x,max}}\right)^2}$$

Where I took out of the formula the contribution of vertical and lateral X and Y shifts for clarity. Reasonable enough, even though it introduces a sort of “privilege” for longitudinal force, which in some way “takes the friction budget first” and leaves F_y with whatever is left.

You would expect the effect of these harsh approximations to be extreme on simulation results; but in reality, and for quasi-steady-state conditions, it’s relatively low, as shown in the next plot.



In steady-state conditions, the 7-DOF model is able to replicate almost 1:1 results obtained with the 14-DOF model; the 3-DOF model remains very close until the lateral acceleration grows significantly, probably as an effect of the simplified assumptions on combined friction. It sure helps 7 and 3 DOF that, even though the 14-DOF model would support non-linear kinematics and compliance, most of them is still defined with linear coefficients, almost eliminating this advantage.

In the following table, we see a comparison between the three models we defined.

	14-DOF	7-DOF	3-DOF
X pos	DOF	DOF	DOF
Y pos	DOF	DOF	DOF
yaw	DOF	DOF	DOF
Z pos	DOF	equilibrium	equilibrium
pitch	DOF	equilibrium	equilibrium
roll	DOF	equilibrium	equilibrium
jounce FL	DOF	equilibrium	equilibrium
jounce FR	DOF	equilibrium	equilibrium
jounce RL	DOF	equilibrium	equilibrium
jounce RR	DOF	equilibrium	equilibrium
spin FL	DOF	DOF	equilibrium
spin FR	DOF	DOF	equilibrium
spin RL	DOF	DOF	equilibrium
spin RR	DOF	DOF	equilibrium
engine	DOF	DOF	equilibrium
K&C	2D splines	linear	linear
Body transients	yes	no	no
Engine transients	yes	yes	no
Integration frequency	1000Hz	200Hz	100Hz
Sim time to equilibrium	1s	0.1s	0.1s
Speed (1 core)	5% RT	400% RT	800% RT

Step 3/A: Trimming the model

Milliken & Milliken dedicate an entire chapter (7) of their evergreen *Race Car Vehicle Dynamics* [1] to what they modestly call the *Milliken Moment Method*. I really won't compete with them nor with Claude Rouelle in trying to explain it and its interpretation; I'll just point out to whomever interested the video recording of one of Rouelle's seminars at the university of Moscow which is freely available on YouTube [2], and to a small OptimumG PDF [3] which may serve at a great introduction to the concept.

The basic concept is, let's impose a steering wheel angle (SWA) and a body sideslip (β) to a vehicle, and let's make some complementary assumption on the way it's moving: constant radius, or constant longitudinal speed. Depending on which basic set of assumptions we made, we can get rid of three additional degrees of freedom (the first ones listed in the table above).

Constant Speed

$$\begin{cases} V_x = V_{x,const} \\ V_y = V_x * \tan \beta \\ r = \frac{a_y}{V_x} \end{cases}$$

Constant Radius

$$\begin{cases} r = \sqrt{\frac{a_y}{R}} \\ V_x = \frac{a_y}{r} \\ V_y = V_x * \tan \beta \end{cases}$$

A particularly convenient way to implement these constraints is, in my opinion, to just write an abstract "container", or "testrig" class which encapsulates a vehicle model, and that, every time we try and access its status, "fixes" the status of the inner vehicle model by imposing the above conditions.

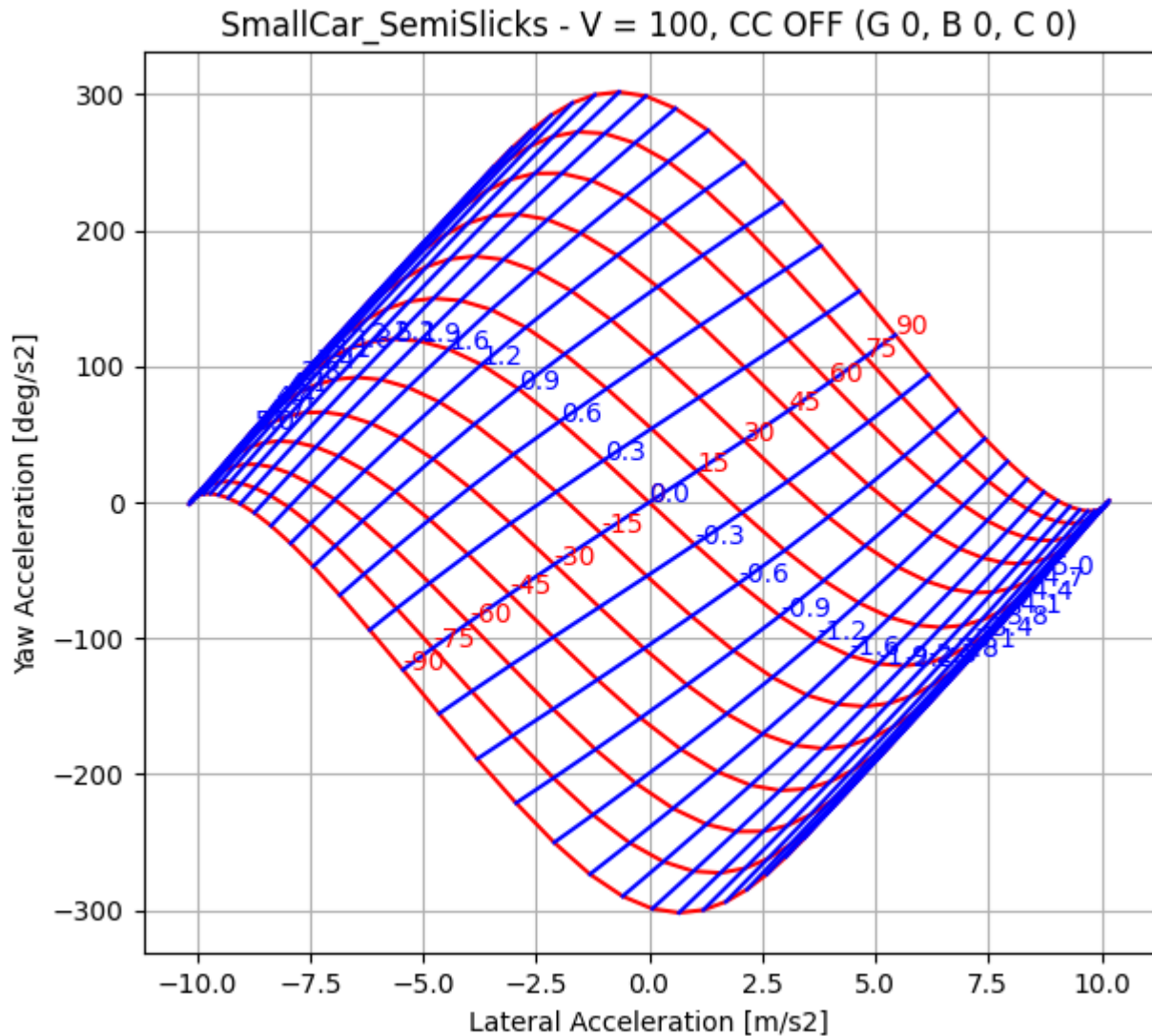
```
class YmdTestrig(OdeModel):
    def __init__(self, car, sideslip = 0., swa = 0., constantV = True, \
        V = 100., R = 100.):
        self.car = car                # OdeModel describing car
        self.sideslip = sideslip      # [deg], imposed sideslip
        self.swa = swa                # [deg], imposed steering wheel angle
        self.V = V                    # [km/h], speed; either initial or constant
        self.R = R                    # [m], radius for constant-radius mode
        self.constantV = constantV    # [-] True for constant-V mode, False for constant-R
```

We still need to calculate lateral acceleration a_y , which will be extracted from the underlying vehicle's equation of motion: for every simulation step, we'll:

- Take lateral acceleration from the estimation we made at the previous time step
- Calculate longitudinal and lateral speed according to either assumption

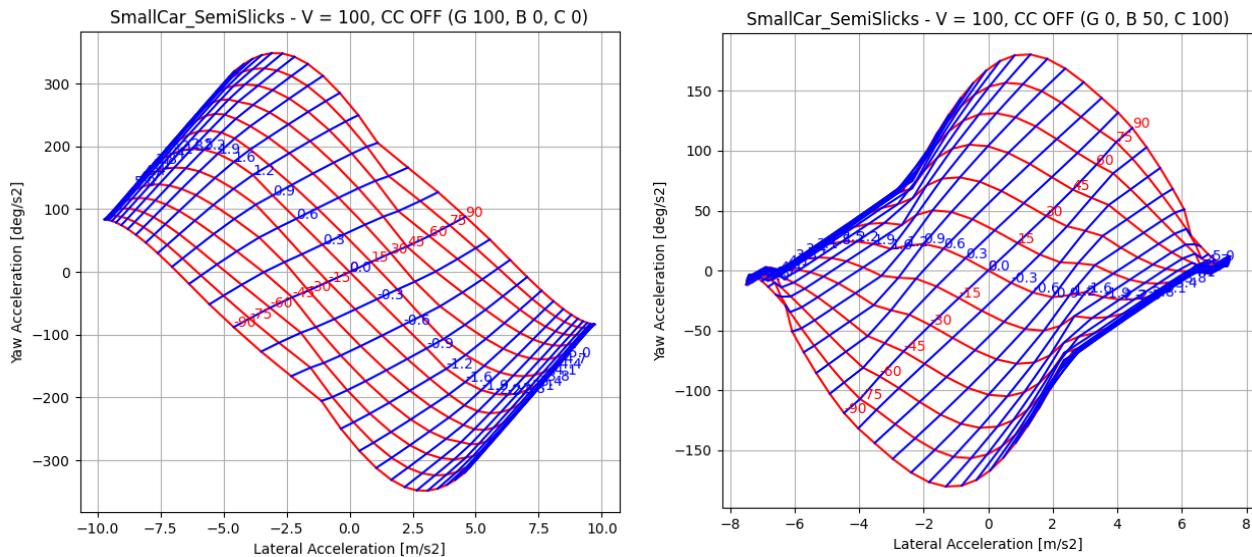
We'll then let the underlying model perform all calculations it usually needs to determine its status: slip angles, longitudinal wheel slips, et cetera, until an equilibrium condition is reached. The time we need to reach equilibrium not only depends on the model we chose (1s for 14-DOF, 0.1 for 7 and 3 DOF), but also on whether these models are fitted with some form of controller. If we want to be loyal to the *constant speed* assumption and we fit our model with a cruise controller, even simpler models require a higher amount of time, in the order of 0.5 to 1.0s.

Naturally, the real joy of the moment method is repeating this operation over a grid of SWA and sideslip values, so that the result becomes a gorgeous $R2 \rightarrow R2$ transform from (SWA, β) to (a_y , r). The objective *beauty* of the diagram explains, I think, at least part of its success.



Now, complete as it seems, the above picture only describes the working conditions of a vehicle at a given speed, and at for a given configuration of driver input controls (for example, in the above plot the driver is not touching the gas, nor the brakes, nor the clutch). While every article I read about YMD reminds that the calculation should be repeated for different speeds and for different acceleration conditions, it's way easier to find examples of the former than of the latter.

While speed is important to determine aerodynamic effects on control and stability, I find very interesting to analyze what happens when we push the gas pedal, or the brakes, which you can find in the following two plots:



In this specific example (a powerful FWD car with pretty capable tires):

- When accelerating, the iso-SWA lines (red) become closer, indicating a reduction in *control*. Our ability to turn the car reduces due to the reduced friction budget available to the front tires, which, in the meantime, are working hard to accelerate the vehicle forward. The rear wheels are unaffected, hence *stability* improves, since the vehicle tends to rotate towards speed direction faster.
- When braking, the effect of load transfer to the front tires increases their cornering stiffness, likely more (linear range) than what they lose to combined friction. For 15deg of steering wheel angle we now reach around 4m/s² lateral acceleration, compared to around 2 without brakes. In the non-linear range, the vehicle reaches unstable conditions, however with the considerably reduced maximum lateral acceleration determined by combined friction (and load transfer) effects.

Step 3/B: Trimming the model to target conditions

A common criticism practical minds tend to do at the Yaw Moment Diagram is that it involves simulating the vehicle's position in a variety of points that it is unlikely to reach. And while certainly interesting, if we just stick to the original aim to impose a constraint of zero yaw acceleration, we'll see that most points above and below the $y = 0$ line are of little use in our case.

The issue is, there is no clear way to “reverse” the dynamic relations between accelerations and SWA and sideslip, at least without extreme assumptions about the vehicle's states. Hence, if we want to walk towards that $R2 \rightarrow R2$ transform *the other way*, we will have to resort to numerical solvers.

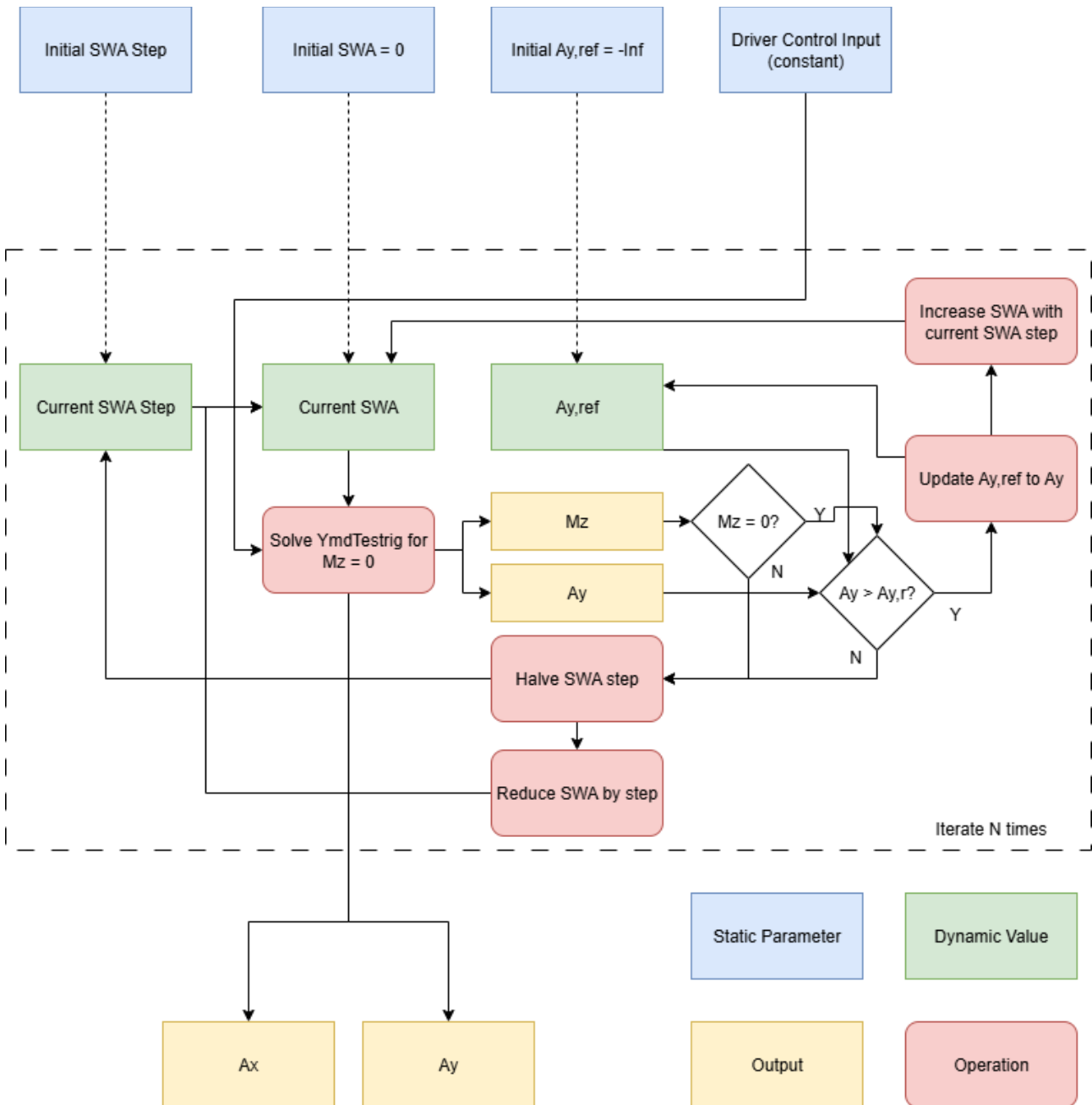
While it is probably technically possible to optimize a problem in which $YawAcc = 0$ trying to find specific pedal and steering wheel angle positions needed to reach a given couple of (A_x, A_y) , the underlying “objective function” is still a time-integrated function, which will need, in the best case scenario, around 0.1ms to run. This will need multiple evaluations in order for the optimizer to build a jacobian matrix and proceed, and, even if we succeed, we'll still have to pre-determine a grid of (A_x, A_y) values, some of which will be impossible to reach.

Instead of this, I went for a simpler, iterative solution.

For a given pedals position and a given SWA, we will still need a solver to find β such that yaw moment equals zero. In my case, I'm just using `scipy.optimize.minimize` with standard settings. Note that even the existence of this solution isn't to be taken for granted: we might (and will, for oversteering

configuration) reach conditions in which, no matter the increase in sideslip, the rear axle will reach its limit and the minimum (absolute) possible yaw acceleration will remain greater than zero.

So, found this way to solve $\text{YawAcc} = 0$ for a given input controls configuration and a base criterion to determine if the solution is valid, we can inline it into the following loop on SWA values:



First, we define some initial steering wheel angle (SWA) step, and a set of driver control inputs (acceleration, brake, clutch pedal position). We also set an initial reference value of lateral acceleration, which, for the time being, we'll set to negative infinite value, that we'll use as a reference for each iteration.

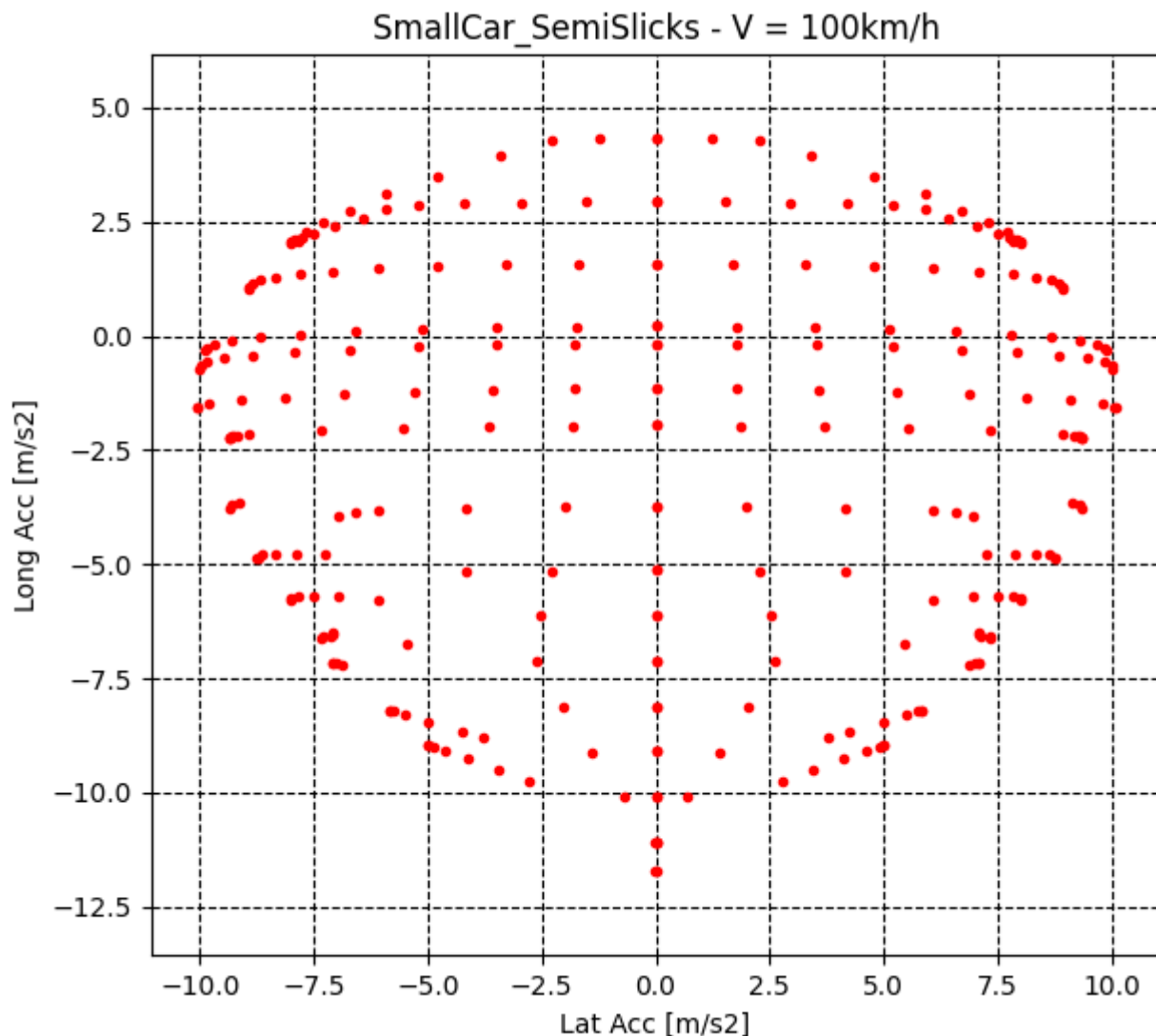
We will set $\text{SWA} = 0$ initially, and we'll use `scipy.optimize.minimize` to iterate on body sideslip (in a `YmdTestrig` object) to find such a sideslip value that minimizes yaw moment M_z . This first solution will generally be stable: then, we'll update the reference lateral acceleration value from what we found with it (which, unless the model is asymmetric, will be zero).

We'll then increase SWA by a step which we copied from an initial, static estimate, and we'll repeat the call to *minimize* to find another value of body sideslip, and the corresponding lateral and yaw accelerations. If the solution is not stable (i.e. the solver stopped before being able to bring yaw moment close to zero), the vehicle likely reached the rear axle limit: we cannot further increase lateral force to compensate, in moment, force produced at the front axle.

Hence we need to reduce steering wheel angle, and at the same time we'll halve SWA step so that we'll try to find a new solution between the last valid SWA and the one we found to be unstable.

If instead the solution is stable, we still need to verify that it resulted in a greater lateral acceleration than the previous best. If this holds, it means we should keep increasing SWA to find a potential higher- A_y steady-state solution. We'll update A_y to this new best and iterate. If instead, lateral acceleration decreased with respect to the previous solution, we have gone past the front axle lateral force limit, and, as in the previous "unstable" case, we need to reduce SWA (and step) in search of a potential optimum point between the better A_y point and the current one.

After N iterations, we *can* (let's just do) assume that A_y has reached its maximum value, or at least a local maximum, and we can extract its value, together with A_x , as a point in the A_y , A_x plot.



By iterating on multiple gas, brake and clutch pedal position, and through the magic of *mirroring the plot so we can pretend we verified the procedure for right hand turns*, we can finally obtain the GG

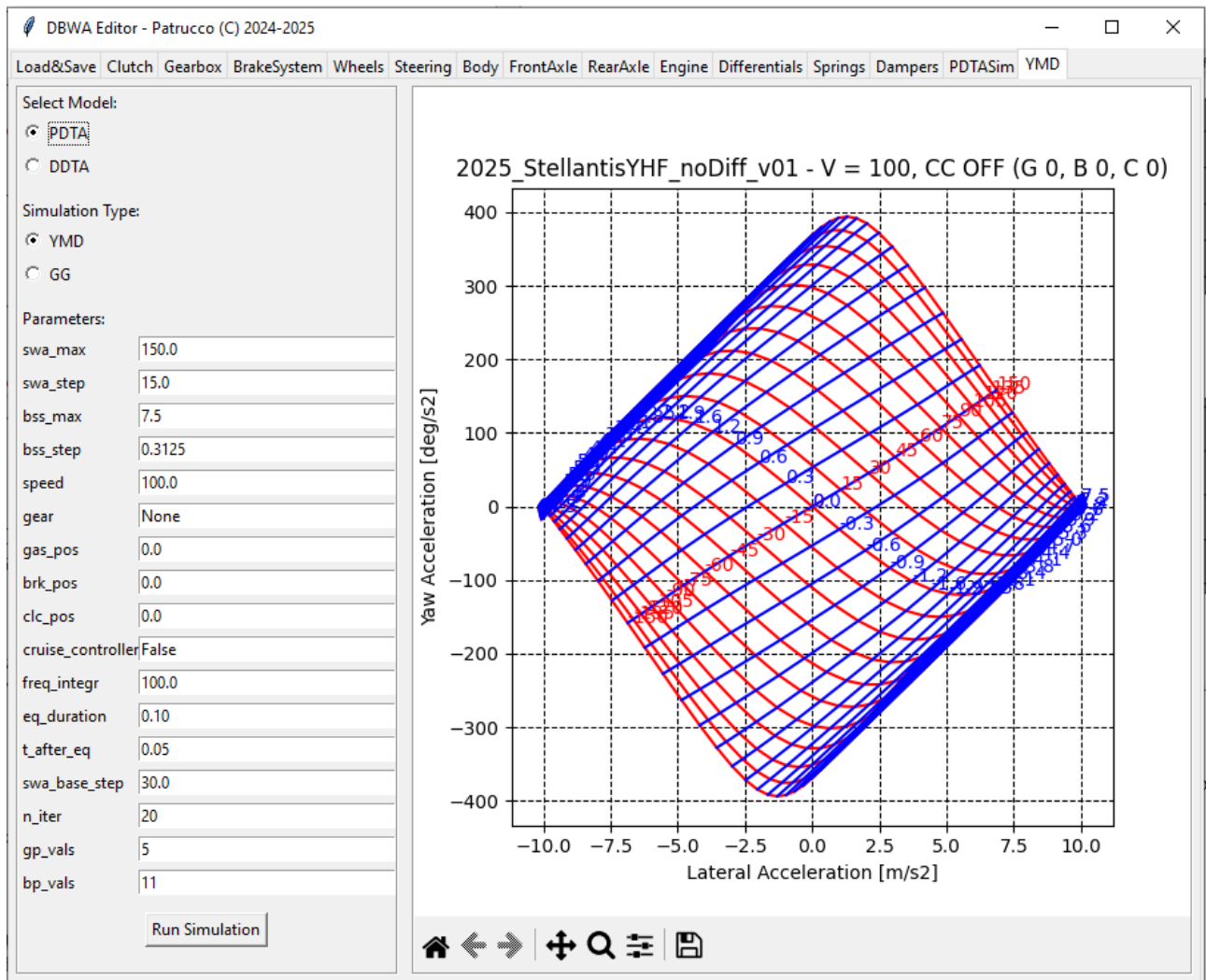
Diagram, where each point represents a separate steady-state equilibrium position. We now know, for each point in the plot:

- Gas, brake, clutch pedal positions
- Steering wheel angle
- Sideslip

Which should be a good starting point, in terms of basic vehicle representation, to proceed to the next phase, i.e. a two-way laptime optimizer.

Conclusions

All of this may sound obvious to the expert, however it's kind of amazing to have this kind of information in a couple of seconds. The reward of being able to *see* what happens if you change static geometry, or suspension kinematics, or anti-roll bar stiffness at a glance is already worth the effort. I integrated both YMD and GG plots into the GUI I use to build vehicle models as a further incentive to throw spare time at this.



Bibliography

- [1] D. L. M. William F. Milliken, Race Car Vehicle Dynamics, SAE International, 1995.
- [2] C. Rouelle, "Getting the most of grip, balance, control and stability - Claude Rouelle (FS Autumn School)," 2021. [Online]. Available:
<https://www.youtube.com/watch?v=6fyxlvDrx3M&t=4694s>. [Accessed 16 9 2025].
- [3] Optimum G, "Getting more from your yaw diagrams," *racecar-engineering.com*, 2017.