Report Planning

Francesco Mecatti
299822@studenti.unimore.it

Preliminary Work: Parallelization

During this first stage, the code of the `frenet_optimal_trajectory.py` and `cubic_spline_planner.py` modules has been parallelized to minimize execution time.

`frenet_optimal_planning` function is found to be the major hotspot of the entire script, absorbing 0.25 seconds for each simulation time point, with base parameters (the ones provided in the initial script) at a speed of 10 m/s; the number of paths generated (at first, without considering colliding paths pruning) by this function are 400 with the just mentioned set of parameters. The execution time can be break down to: 0.089s for `calc_frenet_paths`, 0.097s for `calc_global_paths`, and 0.062s for `check_paths`.

The code has been refactored to exploit multi-core architectures. Parallelization takes place in the form of concurrent data-level parallelism for `calc_frenet_paths` and `calc_global_paths`, whereas such concurrency model has proved to underperform wrt sequential execution for `check_paths`, probably because of the smaller task size, the data movement overhead is higher than the benefits. This parallelization employs a `multiprocessing.Pool` to distribute the workload of paths generation and reference frame conversion on multiple processes (not threads due to Python's Global Interpreter Lock).

On top of this coarse "path-level" parallelism, a finer parallelism level is achieved through the use of vectorized (specifically SIMD) instructions to perform matrix-related operations, thanks to the Numpy library. Path generation, pruning, and check operations benefit from this code refactor. Some functions belonging to the `cubic_spline_planner.py` module have been rewritten to manage multiple input points instead of just one, thus not only saving function calls overhead, but also unlocking the possibility to exploit SIMD instructions. Specifically, `calc`, `calcd`, `__search_index`, and `calc_position`, and `calc_yaw` have been reworked to support Numpy vectors as input. These functions have been selected according to the result of `cProfile`, which showed much of the execution time spent in `calc_yaw` and `calc_position`; by tracing back the functions invoked by these two, it turned out that `calc`, `calcd`, and `__search_index` were the major responsible of such a high execution time.

The results, collected with the same conditions as described in the first paragraph (base params @ 10 m/s), show that the refactor of the `frenet_optimal_trajectory.py` module leads to 0.10s of execution time (x2.5 speedup); by further optimizing the `cubic_spline_planner.py` module, the execution time lowers to 0.05s (x5 speedup).

Exercise 1: Static Obstacle Avoidance at Low Speed

In exercise 1, an optimal Frenet-based planner has been successfully tracked with the Stanley geometric controller, first at a speed of 10 m/s, and then at a speed of 15 m/s. Furthermore, the size of the search space used for trajectory generation has been reduced, achieving a speedup of 2.4 in simulation wall-clock execution time.

The trajectory generated by the planner, with base parameters, requires 164.9s of simulated time and 193.1s of wall-clock time to be tracked at 10 m/s with the Stanley controller. Figure 2 and fig. 1 show two overtakes in different road conditions. The entire trajectory and the lateral error can be found in the attached *figures* folder.

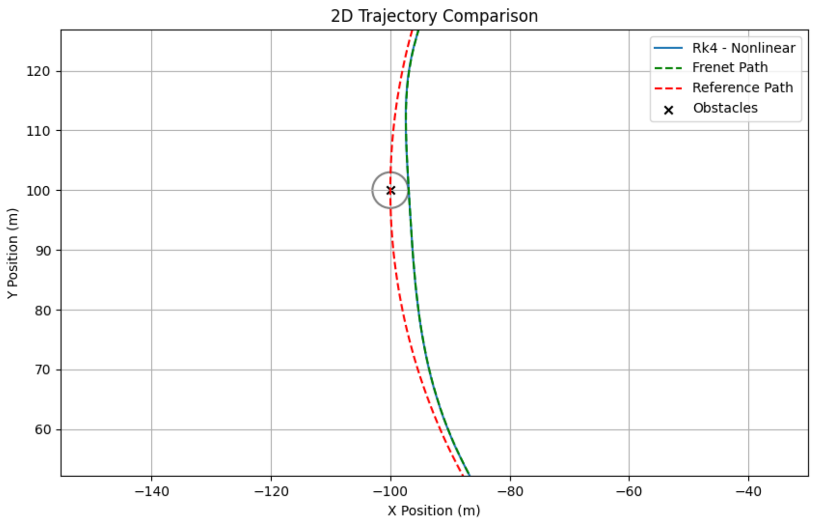

Figure 1: Mid turn overtake close-up @ 10 m/s

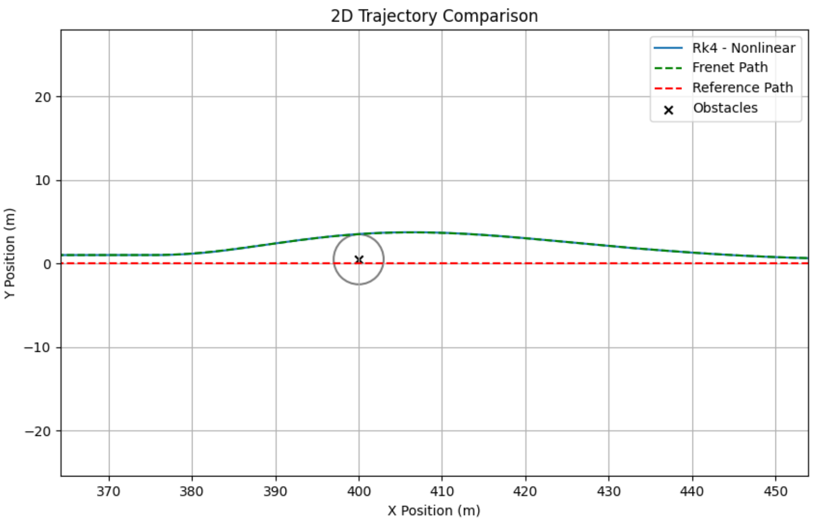

Figure 2: Straight line overtake close-up @ 10 m/s

At 15 m/s both the wall-clock and the simulated time decrease, because of the higher vehicle speed: 112.0s of simulated time and 134.7s of wall-clock time. Figure 3 and fig. 4 illustrate, respectively, the trajectory and the lateral error at a speed of 15 m/s. Figure 4 demonstrates how the lateral error consistently stays under the required 4 meters. Despite the *temporal consistency* ensured by the optimal frenet planner [2], the trajectory generated by the planner slightly suffers from a *stationary offset*, as highlighted by Figure 5.

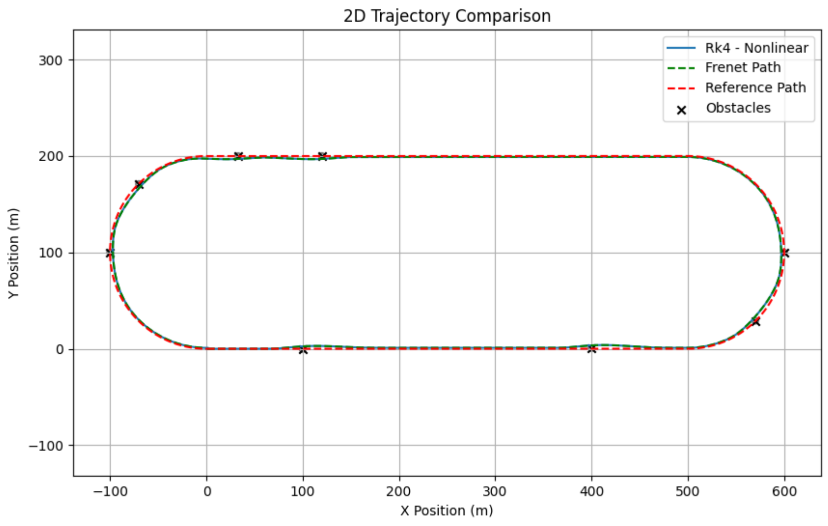

Figure 3: Trajectory @ 15 m/s

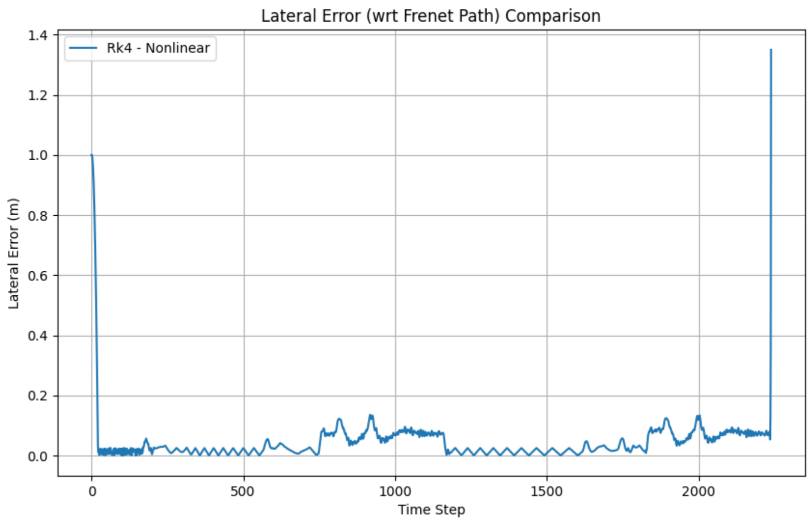

Figure 4: Lateral error @ 15 m/s

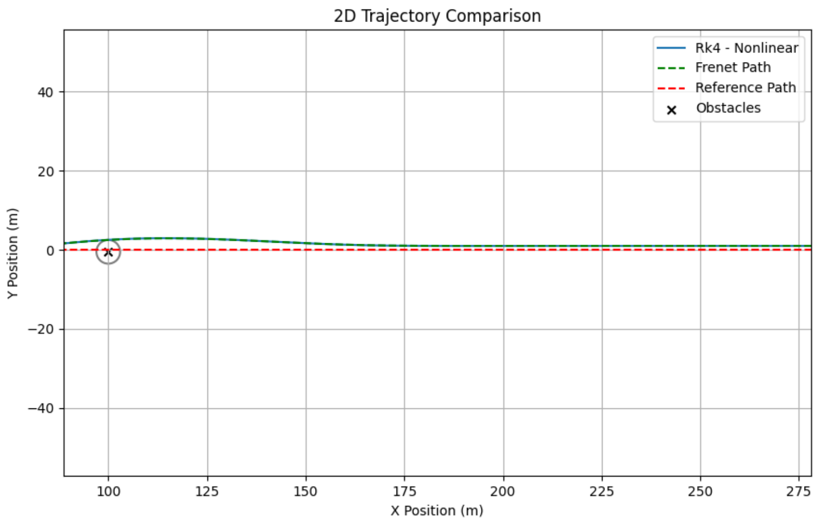

Figure 5: Steady state error @ 15 m/s

When it comes to cutting the execution time (at 15 m/s as before), the first approach tried is to force the planner to generate shorter/faster trajectories by weighting more the *needed time* [2] for the maneuver, since the natural consequence of a shorter simulated time is also a shorter wall-time. The best set of weights used for this experiment is:

```
K_J = 0.05
K_T = 1.0
K_D = 0.5
K_LAT = 1.0
K_LON = 0.5
```

However, it turned out that this set of weights leads to 111.9s of simulated time, meaning that there is not a huge difference with the simulated time with base weights, which is 112.0s as mentioned above. Therefore, this approach has been discarded.

Instead, a successful strategy turned out to be narrowing down the finite set of generated trajectories, which positively impacted not only the paths generation function, but also the downstream coordinate change and collision checking functions. The reduction of the "search" space is achieved by reducing road width and road sampling distance, as well as shortening the prediction time horizon delta (difference between max and min prediction time); these changes are reported in table 1. This way, the sampling space for the trajectory ending offset is narrower and coarser, effectively reducing the number of paths generated and evaluated.

This strategy resulted in a 47.5s of wall-clock time, with 113.5s of execution time: the wall-clock time is reduced by a factor of almost 3, while the simulated time worsens due to the smaller sampling space, which presumably leads to a less optimal solution, compared to the solution of a denser sampling space.

Param. Name	Base Value	New Value
MAX_ACCEL	10.0	10.0
MAX_CURVATURE	2.0	2.0
MAX_ROAD_WIDTH	5.0	4.0
D_ROAD_W	0.5	0.8
MAX_T	5.0	3.0
MIN_T	4.5	2.7
D_T_S	0.5	0.5
N_S_SAMPLE	1	1

Table 1: Parameters configuration used to successfully reduce the execution time

Exercise 2: Static Obstacle Avoidance at High Speed

In this exercise, the same path is tracked at higher speeds (20 m/s and 25 m/s) while different controllers are tested, specifically Pure Pursuit (PP), Stanley and MPC with both kinematic and linear ST model. In addition, a maximum speed of 31 m/s is achieved by employing an MPC controller w/ a linear ST model, exceeding the maximum speed of 29 m/s reached in the previous report.

No appreciable differences are evident in the results obtained with the three different controllers. The same behaviors emerged in the previous report are visible: PP has the most oscillatory lateral velocity, followed by the Stanely controller, with MPC performing the best in terms of stability and amplitude of lateral velocity, thanks to its predictive capabilities. This is visibile comparing the plots of fig. 6, fig. 7, and fig. 8. The tendency explained above – the shorter the simulated time, the shorter the wall-time – is followed even in this case.

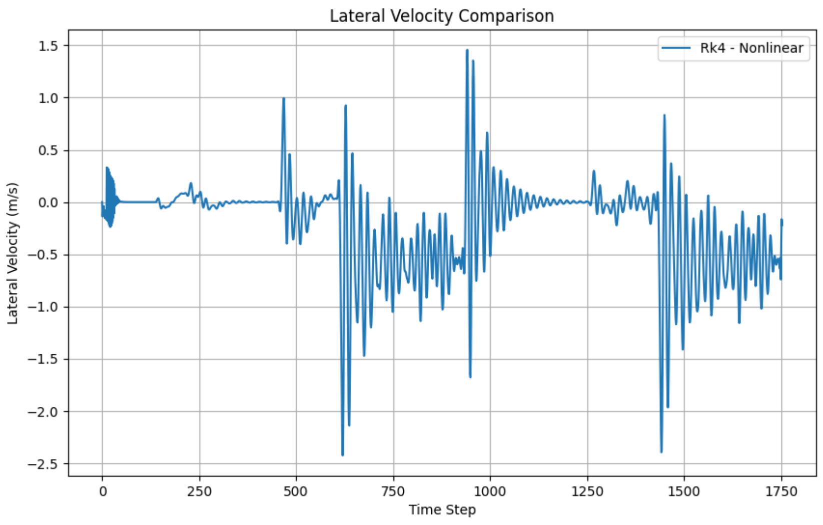

Figure 6: PP lateral velocity @ 20 m/s

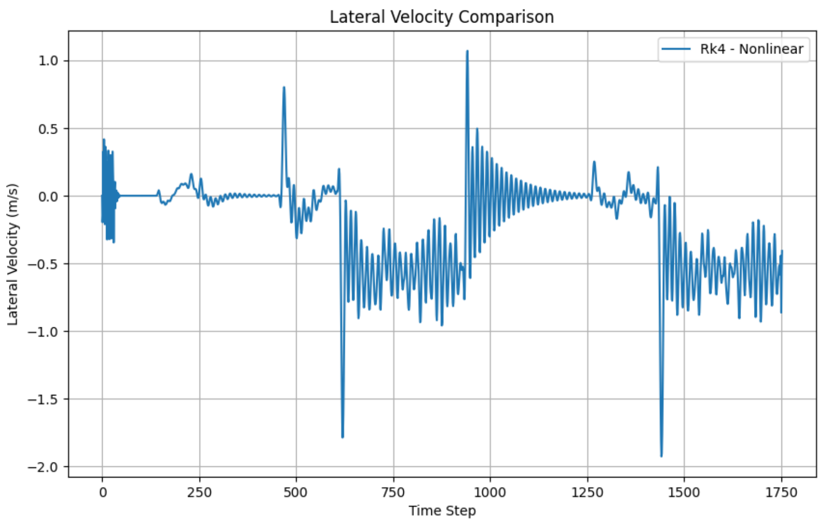

Figure 7: Stanley lateral velocity @ 20 m/s

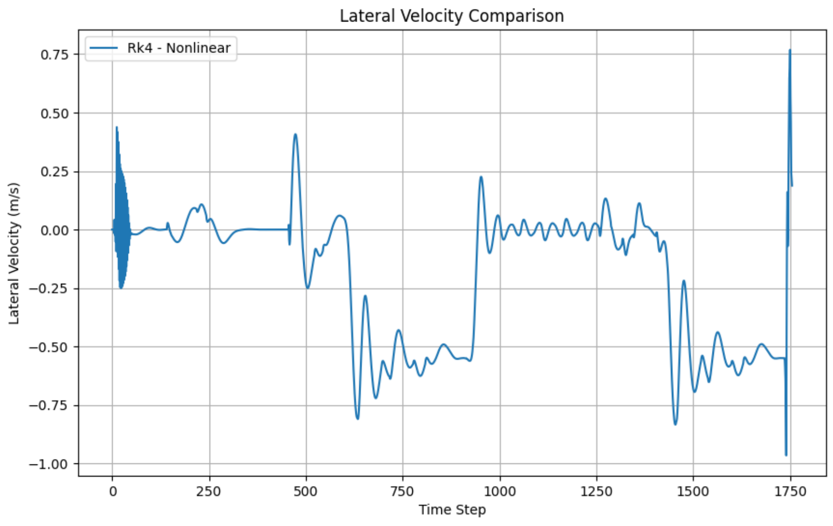

Figure 8: MPC w/ kinematic model lateral velocity @ 20 m/s

A longitudinal velocity of 25 m/s is achieved, with linear single track (ST) MPC, without relaxing the requirements. The result, in terms of lateral error, is not much different from the MPC w/ kinematic model @ 20 m/s, as the findings *Kinematic and dynamic vehicle models for autonomous driving control design* [1] anticipated.

A maximum speed of 31 m/s is also reached, with a linear ST MPC, without relaxing the constraints. The linear ST model is the same as that used in the last experiment of the previous report to achieve a speed of 29 m/s, meaning the best-performing model found in the previous assignment. However, the maximum speed found this time is higher than the max speed of the previous report; this means that the planner synergistically helps the controller by providing a smoother and less aggressive trajectory to reach the reference path. The resulting trajectory, lateral error and lateral velocity are shown, respectively, by fig. 9, fig. 10, and fig. 11.

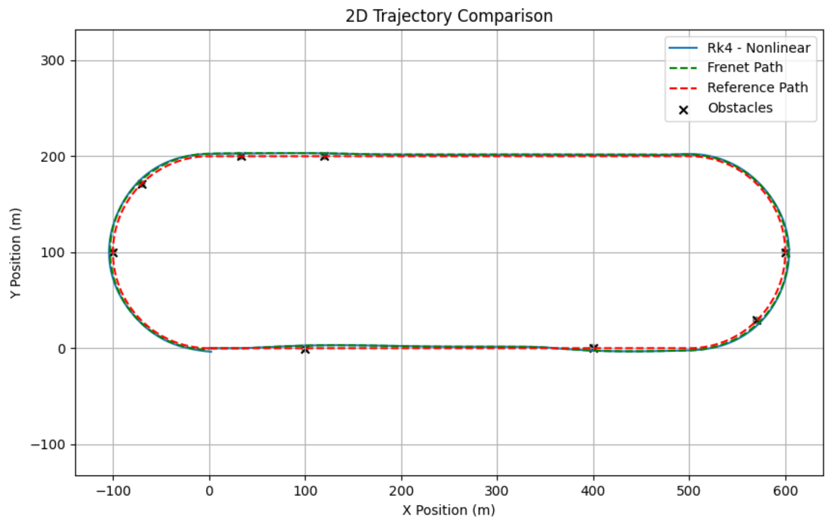

Figure 9: MPC w/ linear ST model trajectory @ 31 m/s

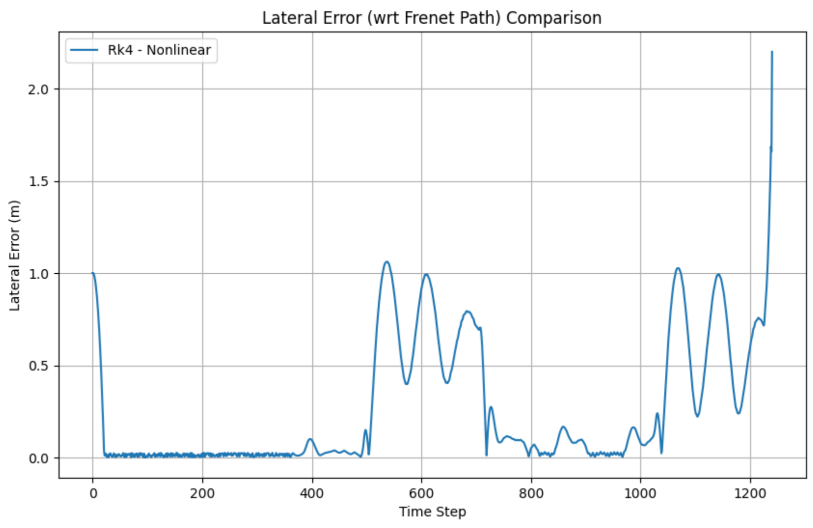

Figure 10: MPC w/ linear ST model lateral error @ 31 m/s

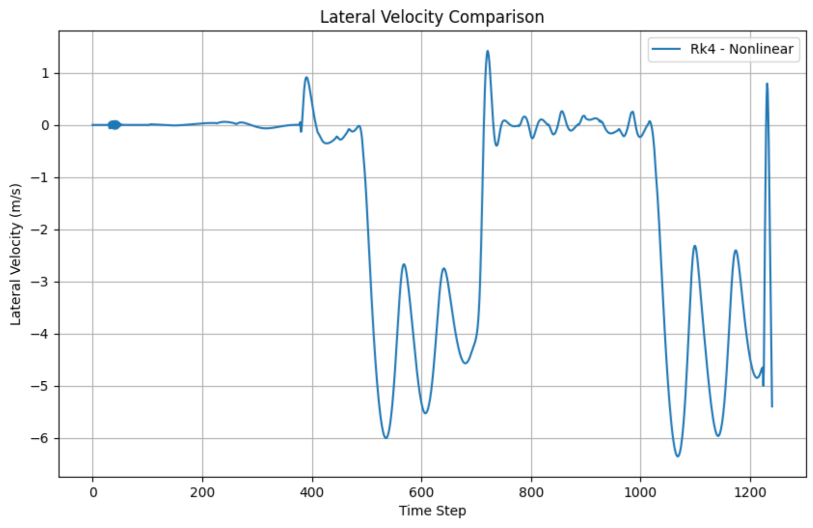

Figure 11: MPC w/ linear ST model lateral velocity @ 31 m/s

References

[1] Jason Kong, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. pages 1094–1099, 06 2015.

[2] Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun. Optimal trajectory generation for dynamic street scenarios in a frenet frame. pages 987 – 993, 06 2010.