Local Trajectory Planning in Unknown Environments using an Ensemble-Based Approach

1st Cristiano Souza de Oliveira
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Aldo von Wangenheim
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Planning local trajectories in real time that can converge, be feasible and smooth at the same time is one of the key challenges in autonomous ground vehicle navigation. Many techniques over the years try to overcome this challenge by partially solving a subset of those goals. We propose a new approach based on first allowing multiple path planning solutions to run in parallel and then choosing the right planned path based on selection criteria. We show that none of the selected local path planners running standalone is able to account for real time, smoothness, feasibility and convergence at the same time, but the parallel ensemble of local planners can achieve those goals. Moreover, the behavior of the vehicle adapts to different scenarios by local planner selection. We compare this approach with each of the selected local planner as a standalone solution both in terms of performance and of final planned path quality. Our approach outperforms each standalone local planner in all of the experiments.**

***Index Terms*—trajectory planning, ensemble, unknown environments, parallel planning, autonomous vehicles, self-driving cars, unstructured environments**

I. Introduction

A. Background

Research in self-driving cars has been intensively studied on public roads and city transportation in the past decades. Some partially-autonomous cars are being already pushed to the market [6]. In contrast, less attention has been paid to unknown unstructured scenarios, which include off-road driveways and some roads with poor maintenance. Unknown environments often lack knowlegde of parts of the configuration space [8], which implies that there is no information a priori about the terrain or what constitutes a driveway.

In this scope, planning feasible trajectories is one of the key challenges when navigating in such environments. Full trajectory planning task involves a synergy between a global planning, which first builds a low resolution high-level path, avoiding large known obstacles if previously known, and local planning, who plans high-resolution low-level path, dealing with the unknown [10], which may include the path itself.

Local planning in unknown environments is currently solved by intuition on what a good approach might be, combined with layers of heuristics for speedup. Since different groups come up with different solutions, there is a demand of a systematic approach to designing a planning solution best suited for a given application [2]. Thus, it seems unlikely that any local path planning algorithm alone can account for all of the unexpected variations, which opens a path to try solutions based on multiple local planning approaches.

We propose a local trajectory planning that can adapt to the changing conditions in unknown environments by executing an ensemble of local planners while keeping a low response time. To avoid increasing the overall execution time, we execute the ensemble of local planners in parallel and we select the final planned path based on quality metrics and execution hierarchy. The parallel planning is possible due to the current evolution of embedded systems, where multicore CPUs, GPUs and APUs are being increasingly available, allowing for the execution of multiple processing threads in real parallel pipelines. This allows for scaling the local response to new boundary conditions by stacking specific local planners, expanding the vehicle's ability to navigate in rough conditions, while keeping the time execution constraint. Several specialized approaches can be selected to handle boundary conditions, helping the vehicle to manage complex scenarios while keeping the overall complexity low. Similarly, a solution based on machine learning can also be stacked to learn from the best selected local planned path at any time, working as a self-adapting mechanism for unexpected future situations. Moreover, by sharing common data such as occupancy mapping or information about vector traversability, objects of pre-processing steps, we further improve the response time of each executor in the ensemble.

The main contributions of this research is to present an efficient local trajectory planning approach based on an ensemble of local planners, that allows for fast planning on multiple scenarios while maintaining the overall system complexity low. Moreover we present an efficient algorithm to fill gaps between the sparse milestones generated by a global planner operating in an environment with low data available.

B. Related Work

In [2] an ensemble of concurrent algorithms is used to plan trajectories for a unmanned aerial vehicle (UAV). To avoid NP-hard for optimization solving, a greedy approach with lazy evaluation was used as selection criteria. This approach works by using the learnt priors and applying conditional independence to select the planner with maximum marginal gain and then evaluating it with the real application problems.

In [13], a Determined Finite Automata (DFA) is used to select a local planning approach based on the vehicle perceived state. This state is determined by statistical analysis of vehicle states and environment sensor data. Typical conditions include the distribution of obstacles, the slope of the road and the current speed of the vehicle. To avoid oscillating planning mode switching, a probabilistic analysis on perception results within a continuous period is applied as the trigger condition. In [1] the local planning is defined before the mission starts. In order to follow the planned path, a parallel approach is proposed as a navigation method based on a DCS (Distributed Control System). Several navigation modules are executed in parallel, and they independently control the robot by using magnetic and geometric landmarks.

C. *Our work*

Our research focuses on planning local trajectories that can adapt to the changing conditions in low structured environments, by executing several local planners while keeping a low response time.

When defining a local trajectory for an UGV, it is crucial that a maximum time constraint is followed, to allow the vehicle to perform a mission without risking it to navigate disoriented while waiting for planner to respond. Executing more than one local planner can be a problem if the first executor is unable to find a feasible path due to operation outside boundary conditions. If another local planner must now takeover, that means that the time frame for total execution is the sum of the two stacked planning solutions present in the system. If many solutions are present, them the total execution time is the sum of all of them in the worst case scenario.

In order to use several local planners while following the time constraint, our approach relies on executing them in parallel and then selecting the final resulting path based on some metrics and on execution hierarchy.

The main advantage of this approach is that a new local solution that responds to a specific set of boundary conditions can be stacked with the current known ones, expanding the vehicle's ability to navigate in rough conditions, while the time execution constraint can still be followed. Similarly, a solution based on machine learning can be stacked to learn from the best selected local solutions at any time, working as a self-adapting mechanism for unexpected future situations.

II. Overview

The goal of a local path planner is defined as to produce a feasible, ideally optimal and smooth, path from data gathered by environment perception, which complies with driving limitations. This data is often stored in some data structure which we define here as data representation. Some common quality metrics for evaluating a local trajectory planner are the ability to avoid obstacles, to respect the vehicle's dynamics, to execute in real-time and to find paths which cost as less as possible, whenever possible.

The cost to traverse a path may vary accordingly to the adopted criteria, such as path length, motion energy or smoothness. A trade-off is very common, such as increasing the total length to improve smoothness or to avoid a particular rough region of the terrain. Real-time trajectory planning and overall motion planning constraint are PSPACE-hard [18], so it often means that the total execution time should allow for a response time-frame compatible with the expected velocity. The performance of trajectory planning and motion execution is improved by using heuristics to avoid traversing the entire search space such as in [3] and by performing path tracking control with algorithms such as pure pursuit, stanley [7], PID or model predictive control achar aquela ref. do motion . To certify previsibility in execution time, a timeout is used for every local planner, which should output any partial results, if possible.

Car-like vehicles are non-holonomic systems, which means that differential constraints that cannot be fully integrated to remove time derivatives of the state variables, so future states depend on the path taken [11].Trajectory planning that account for the kinodynamic constraints of the vehicle is more accurate because they avoid choosing paths that are impossible to track, such as turning 90 degrees to the right of the current position. Due to the complex iterations between the car and the terrain, modelling the full physical movement is difficult. However, vehicles in sufficient low speeds that allow for movement without wheel slipping can be modeled by the Kinematic Bicycle Model [16]

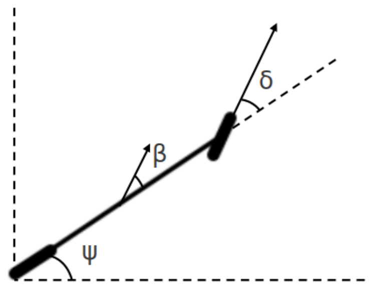


Fig. 1. The bicycle model for low speeds

Let ψ be the current vehicle's heading according to a reference coordinate system, δ be the instantaneous front wheel angle and β be the actual turning angle of the vehicle's center of gravity, V be the instantaneous velocity, L be the size of the vehicle and l_r the size from the rear wheel to the center of gravity, we can model the movement as the following series of differential equations.

$$\begin{cases} \dot{X} = Vcos(\theta+\beta) \\ \dot{Y} = Vsin(\theta+\beta) \\ \dot{\theta} = \dfrac{Vcos(\beta)tan(\delta)}{L} \end{cases} \tag{1}$$

$$\beta = tan^{-1}\left(\frac{l_r tan(\delta)}{L}\right) \tag{2}$$

Kinematic constraints allow for using motion primitives (figure 2) to accelerate the search space in local planning, such as in [17], by avoid visiting state candidates which are unreachable. Dubins [4] and Reeds-Shepp [5] curves can also be used to add non-holonomic constraints to planning algorithms such as in Hybrid A* [3].

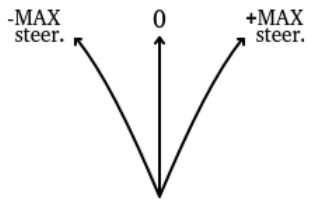


Fig. 2. Movement primitives

Errors in the model or from the sensor data can lead to collisions even when the car is correctly following the planned path. To guarantee a collision-free path, we need to establish a minimal distance constraint for each candidate waypoint. This check is performed on visual perception data. First, we compute an occupancy grid representing the bird's eye view of the surroundings. Since the environment is unknown, this occupancy map serves as source for pathway discover and obstacle avoidance. Then, we check, if every planned waypoint respects the minimal width and height, adjusted by the expected heading. Thus, the vehicle's current and future heading plays a key role in defining which pixels of the Occupancy Grid should be checked, so a rotation must be performed on the expected coordinate points. To define the heading between two waypoints, we compute the mean value of the heading relative to the closest k points to avoid abrupt changes caused by discretization at low resolution, k = 3. Let n be the number of distinct path waypoints, the heading is computed using eq 3. To check for minimal distances, we compute a simple 2D rotation around the Z axis for each angle θ (eq 4).

$$\begin{cases} \frac{1}{k}\sum tan^{-1}(p_i, p_{i+j}), & j \in (1,k) \quad i+j < n \\ \frac{1}{k}\sum tan^{-1}(p_i, p_{i-j}), & j \in (1,k) \quad i-j >= 0 \end{cases} \tag{3}$$

$$\begin{bmatrix} x' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ z \\ 1 \end{bmatrix} \tag{4}$$

The set of all waypoints representing the states and its minimal distances form a tunnel as seen in figure 3. To increase efficiency, we perform those operations in parallel using CUDA, instead of using the CPU, if the length of the path is big enough that compensates for the I/O delay when copying information to the videocard. Experiments show that it is faster to compute a path of with at least 40 waypoints using the GPU on RTX3060 and with at least 140 waypoints using Jetson Orion Nano (see table I). We use a selector to decide which processing unit should compute it, depending on the waypoint count and the running hardware.

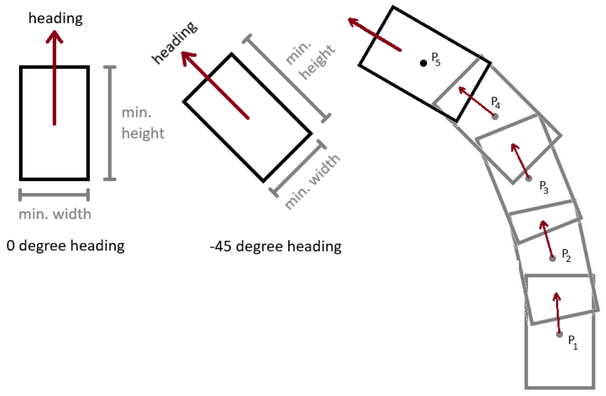


Fig. 3. Distance constraints in movement

TABLE I
COMPARING EXECUTION TIME FOR WAYPOINT MINIMAL DISTANCE CHECK IN CPU AND GPU

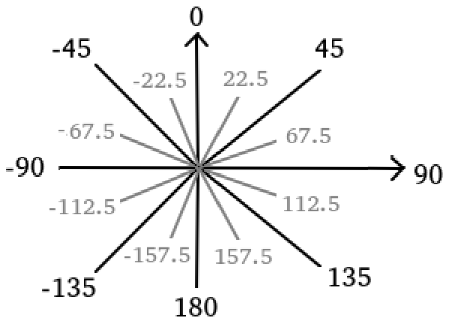
Size	CPU	GPU	Equipment
20	0.36 ms	0.5 ms	Xeon E5-1650, RTX 3060
20	0.56 ms	2.3 ms	Jetson Orin Nano
40	0.75 ms	0.62 ms	Xeon E5-1650, RTX 3060
40	1.18 ms	2.63 ms	Jetson Orin Nano
80	1.45 ms	0.69 ms	Xeon E5-1650, RTX 3060
80	2.21 ms	2.83 ms	Jetson Orin Nano
140	2.67 ms	0.86 ms	Xeon E5-1650, RTX 3060
140	3.71 ms	3.06 ms	Jetson Orin Nano
160	3.07 ms	0.98 ms	Xeon E5-1650, RTX 3060
160	4.23 ms	3.2 ms	Jetson Orin Nano

A. Search space

The search space consists of a 3 channel occupancy grid (OG), built from the semantic segmentation of RGB images in a Bird's Eye View format [12]. The semantic segmentation outputs the segmentation class for each pixel in the original image on the first OG channel. The other two channels are used to pre-compute the traversability and euclidean distance to the goal point using CUDA [15].

Traversability is calculated relative to the vehicle's dimensions in the search space. It holds information about which directions are allowed to be chosen in a particular state, given the minimal distance. Assuming that the minimal distances are symmetrical in the center of the search state, we use 8 bits to represent 16 possible headings as seen in figure 4. This is possible because if a state is traversable with a heading of 0 degrees, then it must also be traversable with a heading of 180 degrees. We use the GPU to compute this function in parallel, taking 2 ms on average in RTX 3060 and 5.6 ms on Jetson Orin.

Fig. 4. Pre-computed traversability angles for each state candidate



B. Navigating the unknown

The global planner outputs a low-resolution path, which is a list of latitude and longitude positions that we expect the vehicle to follow, in order to reach the final destination. This list can be more sparse if the global planner has low information about the environment. We project each of those milestones on the vehicle coordinate system using Mercator (eq. 6), which is enough for short distances. This approximation is initially calibrated by the starting position of the vehicle, representing the coordinate (x,y) = (0,0). This allows for using simple euclidean equations, requiring less computing time than performing geodetic calculations.

$$scale = cos(radians(lat\_origin)) \tag{5}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} scale * 6378137.0 * radians(lon) \\ -scale * 6378137.0 * log(tan(\pi * \frac{90+lat}{360})) \end{bmatrix} \tag{6}$$

Since the environment is unknown, we don't have information a priori of which constitutes a path. Moreover, the next (x,y) position given by the global goal planner (GG) may be out of vision from the current ego posision. Therefore, it is necessary to chose a local goal (LG) that approximates the global goal. A simple projection in local coordinates may lead to an unfeasible goal due to minimal distance constraints or the presence of unexpected obstacles. To address this, we propose an algorithm to efficiently find a feasible local goal point that is the best local approximation within the boundaries of the OG frame for the given input data.

Let g_1 be the local representation of the first global goal for which the car must currently head and let g_2 be the next global goal, for which the car will be heading if g_1 is successfully reached. We want to find a $g_1^{'}$ that represents the best approximation in terms of distance and heading to g_1. Then, there are three possible scenarios:

1) g_1 is within the boundaries of the OG.
2) g_1 is outside the boundaries of the OG but not far.
3) g_1 is far from the boundaries of the OG.

In the first scenario, $g_1^{'}$ can potentially be g_1 itself, heading towards g_2, which is the best solution. In case g_1 is not feasible, then we search for the entire search space for a waypoint that can lead to the midpoint between g_1 and g_2 using the algorithm II-B. We use the GPU to paralellize this search, which takes, on average, 1.51 ms on RTX 3060 and 6.38 ms on Jetson Orin. If we can't find a waypoint, then we search again for a waypoint that has the same heading as $g_1 \rightarrow g_2$, which is less ideal, since the waypoint position on the grid may increase unnecessary turnings. Finally, if no local goal is found, we search for the goal closest to g_1 whose heading has the lowest heading error, as in eq 8.

$$best\_heading = heading(start, goal) \tag{7}$$

$$heading\_err = |best\_heading - a|, a \in direction \tag{8}$$

We limit this search for heading 'a' at each waypoint candidate by a precision of 5 degrees in the direction of the waypoint location in the OG for performance. If the waypoint is at the top-left position, we search from -90 to 0 degrees. If it is at the top-right position, we search for 0 to 90 degrees. Since we do not support reverse driving, we ignore the bottom directions to improve performance. As we demonstrate in fig. 5, p is a candidate for local goal, which has the goal being approximated placed at its top-right position. Thus, we only need to investigate the heading angles between 0 and +90 degrees, since any other range will lead to a wrong direction. In order to efficiently find a solution, we use the algorithm II-B to iterate on the entire search space in parallel.

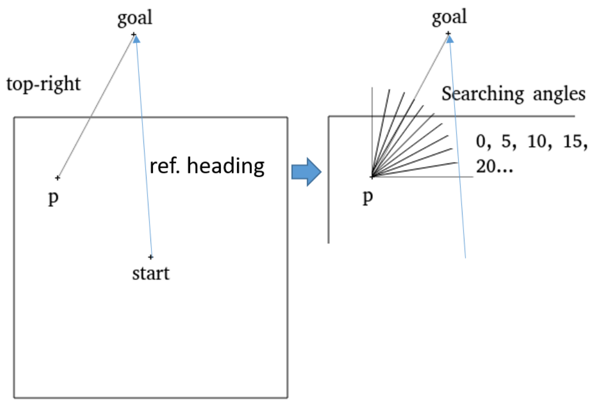


Fig. 5. Searching for a local goal based on direction

```
Algorithm 1 Parallel best local goal for direction
Input: frame, pos, distant_goal
Output: valid_goal
    Initialisation :
 1: cost = BestCostInDirection(frame, pos, distant_goal)
 2: return FirstWaypointWithBestCost(frame, pos, distant_goal, cost)
```

Algorithm 2 BestCostInDirection

```
Input: frame, pos, invalid_goal, heading, *best_cost
Output:
 1: if $((x, z) \in (ego\_bounds))$ then
 2:     return
 3: end if
 4: if $((x, z) \in (unfeasible\_segmentation\_class))$ then
 5:     return
 6: end if
 7: ref_heading = $\frac{\pi}{2} - atan2(-(distant\_goal.z - z), (distant\_goal.x - x))$
 8: if $ref\_heading > \pi$ then
 9:     $ref\_heading = ref\_heading - 2 * \pi$
10: end if
11: if distant_goal.z $>$ start.z then
12:     return
13: end if
14: if distant_goal.x $<$ start.x then
15:     $angle\_start = -90$
16:     $angle\_end = 0$
17: else if distant_goal.x $>$ start.x then
18:     $angle\_start = 0$
19:     $angle\_end = 90$
20: end if
21: for $heading = angle\_start$ to $angle\_end$ do
22:     for $i = -\frac{min\_height}{2}$ to $+\frac{min\_height}{2}$ do
23:         for $j = -\frac{min\_width}{2}$ to $+\frac{min\_width}{2}$ do
24:             xl = round($j * cos(heading) - i * sin(heading) + x$)
25:             zl = round($j * sin(heading) + i * cos(heading) + z$)
26:             if $(xl, zl \neq borders)$ then
27:                 continue
28:                 if $(xl, zl \in (unfeasible\_segmentation\_class))$ then
29:                     return
30:                 end if
31:             end if
32:         end for
33:     end for
34:     heading_err = $|ref\_heading - heading| + 1$
35:     distance = $(distant\_goal.x - x)^2 + (distant\_goal.z - z)^2$
36:     cost = $heading\_err^2 * distance$ % MAX_INT
37:     atomic_update_min(*best_cost, cost)
38: end for
39: return
```

In the second scenario, g_1 is currently out-of-reach, so we can't use it directly, but since it is within a short distance, the heading $g_1 \rightarrow g_2$ can still be valid. We use the method described in the algorithm II-B to find a feasible $g_1^{'}$ that can go toward g_2. If no local goal can be found, we then search for any $g_1^{'}$ with heading to the midpoint between g_1 and g_1. This is less ideal because g_2 can be very far from our location, given that not even g_1 is within reach. When still no goal is found, we search for the goal closest to g_1 whose heading has the lowest heading error, as in eq 8 (see II-B).

In the third scenario, g_1 is far from our position, so we can't use it directly and the heading $g_1 \rightarrow g_2$ is probably not valid. Therefore, the best action is to head towards g_1. To avoid heading to a far goal, we first try to head to the mid point between g_1 and our current position. We use method **??** to search for a feasible $g_1^{'}$ with such heading. In case we can't find a solution, then a direct heading to g_1 is tried. When still no goal is found, we search for the goal closest to g_1 whose heading has the lowest heading error, as in eq 8 (see II-B).

Algorithm 3 Parallel best local goal for heading

```
Input: frame, pos, invalid_goal
Output: valid_goal
    Initialisation :
 1: cost = BestCostForHeading(frame, pos, invalid_goal)
 2: return FirstWaypointWithCostAndHeading(frame, pos, invalid_goal, cost)
```

Algorithm 4 BestCostForHeading

```
Input: frame, pos, invalid_goal, heading, *best_cost
Output:
 1: cost = $(invalid\_goal.x - x)^2 + (invalid\_goal.z - z)^2$ % MAX_INT:
 2: if $((x, z) \in (ego\_bounds))$ then
 3:     return
 4: end if
 5: if $((x, z) \in (unfeasible\_segmentation\_class))$ then
 6:     return
 7: end if
 8: for $i = -\frac{min\_height}{2}$ to $+\frac{min\_height}{2}$ do
 9:     for $j = -\frac{min\_width}{2}$ to $+\frac{min\_width}{2}$ do
10:         xl = round($j * cos(heading) - i * sin(heading) + x$)
11:         zl = round($j * sin(heading) + i * cos(heading) + z$)
12:         if $(xl, zl \neq borders)$ then
13:             continue
14:             if $(xl, zl \in (unfeasible\_segmentation\_class))$ then
15:                 return
16:             end if
17:         end if
18:     end for
19: end for
    all (xl, zl) are feasible, then (x, z) is feasible for the heading
20: atomic_update_min(*best_cost, cost)
21: return
```

III. Parallel Local Planning

Once we finish defining a local goal that approximates the next global goal, we perform the local trajectory planning to create a list of (x, y, heading) states to be compared to the current vehicle position in the path tracking step.

Assuming that any local planner L_i in the local planner set S support the occupancy grid and will only perform writing operations on its local data, we can assure that there is no need to protect the shared memory with mutual exclusion or semaphore locking. Thus, we avoid serial execution. Assuming that every local planner in the set is preemptive, implying that a non-optimal or a null response can be given at any time, we are able to run the parallel execution with two possible approaches: to give priority response for the faster local planners or to run all of them at the same priority level and then select the best planner response in terms of the rate of change of acceleration (jerk).

In the first approach, the maximum execution time is equals to the execution time of the slower local planner in the set, but on average, it runs faster, since the selection by hierarchy ensures that the slower general solution planners such as RRT* and Hybrid A* are ignore in case of a feasible path is produced by the faster local planners, see fig. 6. The main disadvantage of this proposal is that it may not select the optimal in terms of jerk. The second approach always select the best jerk, but the average execution time is equals to the execution time of the slower local planner in the set. In all cases, a timeout threshold limits total execution time to ensure responsiveness.

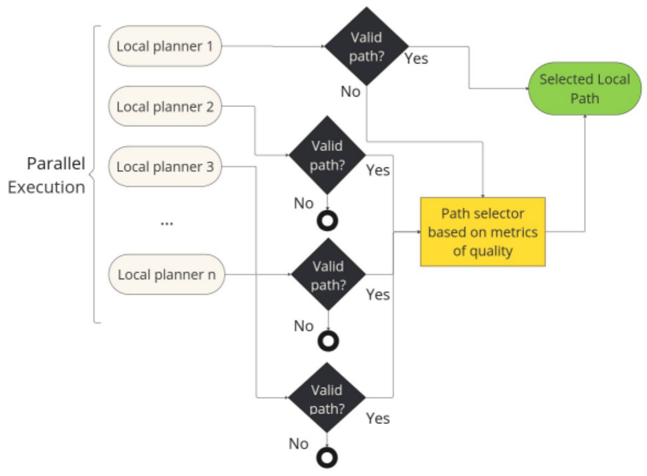


Fig. 6. The architecture of the hierarchical selection

If no local planner reaches a valid path, then the Ensemble found an invalid path and must recover from it, which means finding alternative routes to reach the goal. This task is not reachable from the perspective of the local planning, since the goal should not be on the vehicle's vicinity. A full stop is then expected here, while the global planner access available offline data to find a new route. If this task also fails, then driving is not possible anymore and a human operator must intervene.

IV. Testing Architecture

To test our solution, we adopted a three-layer architecture: a simple global planner provides a fixed list of goal points, being unable to plan or to re-plan global paths; a local planner implementing parallel hierarchical approach and a simple motion controller, using feedback correct variations between the planned trajectory and the real resulting motion. The motion controller is divided into longitudinal controller, which keepts the velocity around an specified value usind PID and a lateral controller based on stanley [7]. A reduced SLAM module performs ego location,by fusing data of GNSS, compass and IMU sensors using an Extended Kalmann Filter [14]. This step also applies Mercator's projection, resulting in a final pose given by the tuple (x, y, heading), relative to the initial calibration position. That means that the initial pose after calibration is (0, 0, 0). In case of being unable to plan, the vehicle will simply perform a full stop. The overall architecture is shown in figure 7

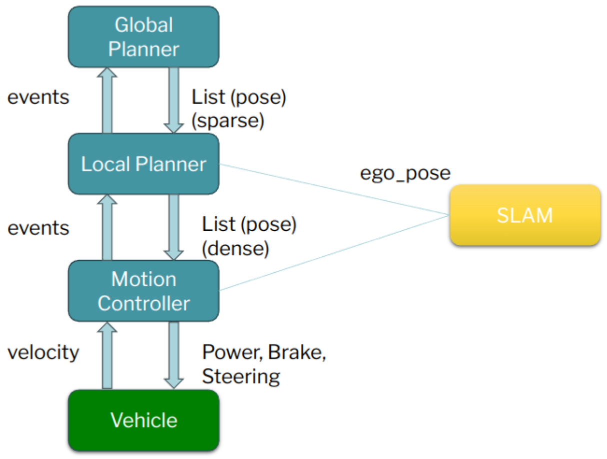


Fig. 7. Architecture of the parallel hierarchical execution

A. Local Planners

In order to test our approach, we have implemented four local path planners: Interpolator, Overtaker, Hybrid A* and RRT* (see table II).

V. Results and Conclusion

We tested our solution within the following scenarios using the Carla simulator. We've modified a town to reflect a unstructured terrain by applying texture to the driving pathway. To check the performance of our proposal, We've compared the average jerk (m/s^3), total path size and average execution time of our approach with with each individual local planner in the following scenarios:

1) Long obstacle-free straight-line with close goal points (within range)
2) Long obstacle-free straight-line with sparse goal points (2x the vehicle's vision range)
3) Long path with close goal points and sparse static obstacles
4) Long path with sparse goal points and sparse static obstacles
5) Short path with dense static obstacles
6) Very long path with close goal points and curvy high terrain

Table III presents the results for the execution of those scenarios for each local planner. We can see that our approach

TABLE II
IMPLEMENTED LOCAL PATH PLANNER EXECUTORS

Local planner	Description	Constraints
Interpolator	Interpolates a cubic spline from the origin to the goal point, with the next goal point as one of the control points, in order to build continuous paths in the long run. The final path is checked for traversability.	-
Overtaker	Generates a dublin path from the start to the goal, then checks for traversability. If not feasible, it delocates the goal point to the left and to the right, generating new paths, trying to avoid simple obstacles.	Dubins
Hybrid A*	Hybrid A* [3] is a variant of the well-known A* search algorithm that applies 3D kinematic to the state space of the vehicle and uses a modified state-update rule that captures the continuous state of the vehicle in the discrete nodes of A*. We use the bicycle model to generate primitives in the continuous state and we approximate them to the search space, where they are checked for feasibility using the GPU.	Kinematics
RRT*	RRT* [9] is a variant of the well-known RRT sampling algorithm that finds an initial feasible solution quickly, but unlike the RRT, almost surely converges to an optimal solution, as it updates the optimal path when new samples are collected from the search space.	-
Parallel Ensemble	Our approach, based on executing the four local planners parallel (interpolator, overtaker, hybrid A* and RRT*) and selecting the best output in terms of jerk to control the vehicle.	
Hierarchy Ensemble	Our approach, based on executing the four local planners in parallel and selecting their output based on the priority of selection: first the interpolator, then the overtaker, then hybrid A* and finally RRT*.	

is capable of producing the most optimal path in terms of jerk when local planners are selected in parallel, but it increases the average execution time to values close to the timeout threshold. The execution using hierarchy produces a slightly less optimal path in terms of jerk, but the mean execution time is much better than the first option. Therefore, we conclude that selecting path output based on hierarchy, quickly handling the expected average driving scenario is the best option.

The execution of the local planners in parallel reduces the complexity of local trajectory planning by allowing the use of specialized solutions to expected local scenarios. In our testing, simple interpolation works as a specialized local planner that can handle the most frequent scenario: driving on a free path ahead, while more generic solutions were ready to takeover when needed. The mean execution time was improved, since the generic slower planners such as hybrid A* and RRT* could be ignored in most situations. The main disadvantage of this approach is that the selection based on hierarchy is manually performed, which can lead to a biased choice towards the scenario selected for testing.

REFERENCES

[1] Naoki Akai et al. “Development of magnetic navigation method based on distributed control system using magnetic and geometric landmarks”. In: *ROBOMECH Journal* 1.1 (Nov. 2014), p. 21. ISSN: 2197-4225. DOI: 10.1186/s40648-014-0021-8. URL: https://doi.org/10.1186/s40648-014-0021-8.

[2] Sanjiban Choudhury, Sankalp Arora, and Sebastian Scherer. “The planner ensemble: Motion planning by executing diverse algorithms”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 2389–2395. DOI: 10.1109/ICRA.2015.7139517.

[3] Dmitri Dolgov et al. “Practical Search Techniques in Path Planning for Autonomous Driving”. In: *AAAI Workshop - Technical Report* (Jan. 2008).

[4] Lester E. Dubins. “On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents”. In: *American Journal of Mathematics* 79 (1957), p. 497. URL: https://api.semanticscholar.org/CorpusID:124320622.

[5] T. Fraichard and A. Scheuer. “From Reeds and Shepp’s to continuous-curvature paths”. In: *IEEE Transactions on Robotics* 20.6 (2004), pp. 1025–1035. DOI: 10.1109/TRO.2004.833789.

[6] David González et al. “A Review of Motion Planning Techniques for Automated Vehicles”. In: *IEEE Transac-*

TABLE III
LOCAL PLANNER EXECUTION RESULTS

Scenario	Planner	total. jerk (m/s³)	Path size (meters)	Avg exec. time
1	H-Ensemble	0.00	101.91	6.27 ms
1	P-Ensemble	0.00	99.79	10.71 ms
1	Hybrid A*	34.23	122.12	286.54 ms
1	Interpolator	0.00	100.37	3.95 ms
1	Overtaker	23.16	97.63	2.90 ms
1	RRT*	0.00	108.81	301.62 ms
2	H-Ensemble	0.00	97.44	8.64 ms
2	P-Ensemble	0.00	100.11	8.74 ms
2	Hybrid A*	31.48	116.64	15.44 ms
2	Interpolator	1.14	97.44	4.02 ms
2	Overtaker	0.56	101.73	2.36 ms
2	RRT*	0.00	110.86	321.21 ms
3	H-Ensemble	3.35	129.20	123.30 ms
3	P-Ensemble	3.88	131.32	576.30 ms
3	Hybrid A*	46.58	144.78	299.03 ms
3	Interpolator	-	fail	-
3	Overtaker	-	fail	-
3	RRT*	0.00	110.86	321.21 ms
4	H-Ensemble	7.95	131.92	370.51 ms
4	P-Ensemble	3.87	131.20	293.66 ms
4	Hybrid A*	48.48	138.62	1132.02 ms
4	Interpolator	-	fail	-
4	Overtaker	-	fail	-
5	H-Ensemble	3.04	69.83	321.92 ms
5	P-Ensemble	4.19	72.74	1212.70 ms
5	Hybrid A*	29.71	83.08	480.11 ms
5	Interpolator	-	fail	-
5	Overtaker	-	fail	-
6	H-Ensemble	8.31	363.13	103.72 ms
6	P-Ensemble	6.15	351.81	362.45 ms
6	Hybrid A*	129.63	407.49	277.17 ms
6	Interpolator	-	fail	-
6	Overtaker	-	fail	-

tions on Intelligent Transportation Systems 17.4 (2016), pp. 1135–1145. DOI: 10.1109/TITS.2015.2498841.

[7] Gabriel M. Hoffmann et al. “Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing”. In: *2007 American Control Conference*. 2007, pp. 2296–2301. DOI: 10.1109/ACC.2007.4282788.

[8] Lucas Janson, Tommy Hu, and Marco Pavone. “Safe Motion Planning in Unknown Environments: Optimality Benchmarks and Tractable Policies”. In: (Apr. 2018).

[9] Sertac Karaman et al. “Anytime Motion Planning using the RRT*”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1478–1483. DOI: 10.1109/ICRA.2011.5980479.

[10] Weria Khaksar et al. “A review on mobile robots motion path planning in unknown environments”. In: *2015 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*. 2015, pp. 295–300. DOI: 10.1109/IRIS.2015.7451628.

[11] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006.

[12] Hongyang Li et al. *Delving into the Devils of Bird’s-eye-view Perception: A Review, Evaluation and Recipe*. 2023. arXiv: 2209.05324.

[13] Ning Li et al. “DFA based autonomous decision-making for UGV in unstructured terrain”. In: *2017 IEEE International Conference on Unmanned Systems (ICUS)*. 2017, pp. 34–39. DOI: 10.1109/ICUS.2017.8278313.

[14] Qiang Li et al. “Kalman Filter and Its Application”. In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. 2015, pp. 74–77. DOI: 10.1109/ICINIS.2015.35.

[15] *NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 12.3, NVIDIA Corporation*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2024-06-01.

[16] Philip Polack et al. “The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?” In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[17] Yao Qi et al. “Hierarchical Motion Planning for Autonomous Vehicles in Unstructured Dynamic Environments”. In: *IEEE Robotics and Automation Letters* 8.2 (2023), pp. 496–503. DOI: 10.1109/LRA.2022.3228159.

[18] Kai Zhang et al. “An efficient decision and planning method for high speed autonomous driving in dynamic environment”. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 806–811. DOI: 10.1109/IVS.2017.7995815.