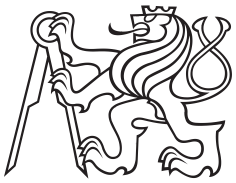
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Mission planning system for autonomous vehicle

Marek Boháč

Supervisor: Ing. David Vošahlík
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Boháč Marek** Personal ID number: **465967**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Mission planning system for autonomous vehicle

Master’s thesis title in Czech:

Návrh systému pro plánování mise autonomního vozidla

Guidelines:

The goal of this master’s thesis will be to design a mission planning system for autonomous vehicle. The system will consist of global planning module, that will be responsible for the over-all trajectory planning, and local planning module responsible for local situation the vehicle is facing at a given moment. The system receives camera-based measurements of surrounding environment and provides the trajectory plan as a sequence of reference states and vehicle inputs to be tracked.
The thesis will compose of following points:
1) Get familiar with mission planning systems, motion planning, sampling-based trajectory planning methods like RRT and its variations.
2) Design and implement global and local motion planning systems for autonomous vehicle.
3) The planning systems must follow safety rules and should minimize operation costs.
4) Design of mission planning system combining global and local motion plan.
5) Validate resulting planning algorithm using real world data from sub-scale demonstration platform.

Bibliography / sources:

[1] LaValle, Steven M.: Planning algorithms. Cambridge University Press, 2006.
[2] LaValle, Steven M., and James J. Kuffner Jr.: Rapidly-exploring random trees: Progress and prospects, (2000).
[3] Dieter Schramm, Manfred Hiller, Roberto Bardini – Vehicle Dynamics – Duisburg 2014
[4] Robert Bosch GmbH - Bosch automotive handbook - Plochingen, Germany : Robet Bosch GmbH ; Cambridge, Mass. : Bentley Publishers

Name and workplace of master’s thesis supervisor:

Ing. David Vošahlík Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **28.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

Ing. David Vošahlík	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

First, I would like to thank my supervisor Ing. David Vošahlík for his advice and guidance.

Next, I would like to thank team leaders doc. Ing. Tomáš Haniš, Ph.D. and Ing. Jan Čech, Ph.D. for the support throughout this difficult project.

Finally, I express my thanks to all the team members Bc. Adam Konopiský, Bc. Jan Švancar, and Bc. Tomáš Twardzik for all the work they have done. Without the endless hours we have spent in the office working on the project, it would not have been possible to finish this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 20 May 2022.

Abstract

Motion planning is a known challenge that allows further advancement of autonomous vehicles. Path-planning algorithms are well-established and efficient. However, designing an algorithm that would plan a state-space trajectory remains a challenge. This thesis focuses on designing and developing a motion planning algorithm for an autonomous vehicle. The designed random sampling minimum violation planning algorithm is a general algorithm that can be used on multiple nonholonomic models and respects traffic and safety rules in compliance with their priorities.

The goal is achieved by implementing an algorithm that combines random sampling algorithms (Rapidly-exploring random tree (RRT)) and Minimum-violation planning (MVP). The developed library allowed tests on real hardware of a subscale platform with recorded data from the drive, proving computational efficiency and practical results of the thesis.

The work is done as part of the team project which is aimed at designing and developing a full autonomous vehicle. Other parts of the pipeline, apart from motion planning, are not described in this thesis unless necessary for the goals of the thesis.

Keywords: motion planning, autonomous vehicle, Rapidly-exploring random tree, minimum-violation planning

Supervisor: Ing. David Vošahlík
Department of Control Engineering FEE

Abstrakt

Algoritmy pro plánování pohybu autonomního vozidla jsou známým problémem a jeho vyřešení by znamenalo významný posun v jejich vývoji. Oproti plánování cesty, což je problém, na který máme mnoho efektivních a rychlých algoritmů, plánování trajektorie ve stavovém prostoru je stále řešený problém. Tato závěrečná práce se zaměřuje na návrh a vývoj plánovacího algoritmu pro autonomní vozidlo, který by plánoval trajektorii v jeho stavovém prostoru. Navržený algoritmus je obecným algoritmem, který je možné nasadit na různé typy modelů vozidel a respektuje dopravní a bezpečností pravidla a jejich priority.

Navržený algoritmus je kombinací method postavených na náhodném vzorkování stavového prostoru vozidla a metody zvané MVP, tedy plánování které porušuje nejméně pravidel. Knihovna, které je vyvinuta v rámci této práce a která implementuje navržený algoritmus umožňuje testy na reálné platformě. K testování jsou použity výpočetní jednotky z platformy a kromě vytvořených testovacích scénářů jsou zpracovány také data získané z provozu vozidla.

Práce je součástí týmového úsilí k vytvoření plně autonomní testovací platformy. Autor se jako jediný v týmu zaměřuje na práci plánovacím algoritmu, nicméně pokud je to nezbytné, ostatní součásti autonomního systému a platformy jsou v této práci také popsány.

Klíčová slova: plánování pohybu, autonomní vozidla, RRT, MVP

Překlad názvu: Návrh systému pro plánování mise autonomního vozidla

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, massive investments have been made in the development of autonomous vehicles. The replacement of a driver with an autonomous system can result in a significant reduction in costs and is expected to prevent accidents, thus reducing causalities. Connected vehicles can also improve traffic flow on our congested roads by sharing its intentions and data about the actual state of the road.

These tasks used to be part of projects such as the DARPA Urban Challenge [1], but as these projects transition from experiments to roads used by human drivers, it is necessary to ensure safety by obeying law and other safety rules. We need to find algorithms that plan vehicle movements while obeying traffic and safety rules.

The autonomous vehicle pipeline can be divided into three sections. First, sensing, which is responsible for describing the environment, detecting obstacles, and determining the pose of the vehicle in the world. Second, there is decision making. On the basis of the data obtained from the sensing, the vehicle must decide which actions to perform to achieve its goals. The third and last part of the pipeline is the execution of the plan.

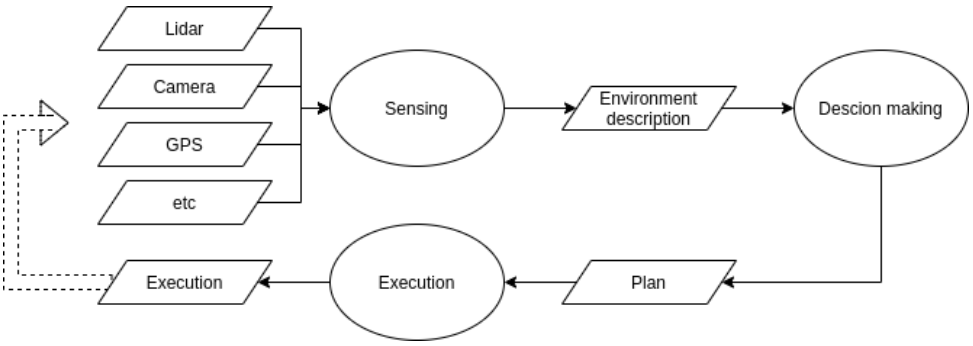


Figure 1.1: Autonomous vehicle pipeline

This thesis focuses on the decision making process; other parts (sensing and plan execution) are not within the scope of this thesis. Designing an efficient decision-making or motion planning algorithm is a well-known challenge. We can divide motion planning into two main categories based on the time and distance range. The first is the long horizon (possibly tens or hundreds of kilometers). The long-horizon plan is usually a high-level plan that describes (e.g., which street to take). Long-horizon planning usually uses long-range sensors such as Global positioning system (GPS) and map data. Second, we have short-horizon planning. The task of planning on small area covered by short-range sensors such as lidar and camera is more challenging, as the state space needs to be explored in a shorter time before the vehicle gets out of the planning area.

The thesis elaborates on both short- and long-horizon planning. The result of the thesis should be the algorithm that can be used on the real subscale platform.

1.2 Problem definition

The thesis describes the design and development of a mission and motion planning system for an autonomous subscale platform. Motion- and mission-planning algorithms will be part of the pipeline that can control the platform in a fully autonomous regime. The algorithm should be based on a family of RRT algorithms accompanied by Linear Temporal Logic (LTL) [2]. The systems responsible for the recognition of the surrounding environment, the location, and the execution of the plan (trajectory tracking controller) are not part of this thesis.

An autonomous vehicle must be able to navigate in an unseen environment. Some information can be provided to the system in the form of map data, but as it cannot be precise and up-to-date in an ever-changing world, the vehicle must react to what it senses at the moment. For these purposes, it is necessary to design a system that achieves a given goal (the vehicle reaches a given location) while avoiding obstacles and using what is considered to be true information about the road a few meters in front of the vehicle.

This problem is divided into two tasks which will be solved in the thesis. First, the motion planning system plans the path using map data. It does not process information about the environment provided by vehicle sensors other than localization. It is expected that the global goal will not change and the car will rarely deviate, thus this system can run on low rate with low computational requirements. Planning must be based at least on the kinematics of the vehicle. In this thesis, this system will be referred to as a global or mission-planning system.

The second motion planning system will process only information provided by vehicle sensors. The vehicle must always navigate in the environment; therefore, all constraints raised by planning must be considered soft. All constraints come with their priority. For example, as the vehicle navigates through space, it should stay on the road, but this condition has a lower priority than obstacle avoidance. Hitting an obstacle would mean crash; thus it should be avoided at all costs. In this thesis, this system will be referred to as a local planning system.

The local planning system processes data on the road a few meters in front of the vehicle. Because the vehicle is considered to be moving continuously and dynamically and because the information provided by the sensing systems is more accurate at shorter distances, the local planning system should run at high frequency to process the data before it is out of date and to take greater advantage of more accurate data. The local planner must also process the global path provided by the global planner to ensure the achievement of the global goal. The output of the local planner is directly fed to the trajectory tracking system; thus, the output trajectory has to comply with vehicle kinematics or dynamics to be directly executable.

Other requirements on both local and global planning system are raised as it should be part of onboard system thus communicating with other algorithms in pipeline, as well as running under limited computational power provided by embedded hardware. Both the autonomous pipeline and the subscale platform and its hardware are described in the following chapter.

1.3 Thesis structure

The work is divided into two parts, each consisting of two chapters. In the first part, the theory and state-of-the-art algorithms are summarized, the designed algorithm is presented, and the developed library is described. In the second part, a subscale platform is introduced and the designed and developed algorithm is tested with simulation and real-world scenarios.

At the end of the thesis, the results are summarized, and the conclusion is given together with suggestions for future work.

Chapter 2

Motion planning algorithms

Motion planning considering system dynamics is a computationally demanding task. Most of the algorithms developed focus on hard constraints such as a simple rule of path feasibility. When designing a motion planning algorithm for an autonomous vehicle, many safety rules with different priorities are required to be followed. In general, many motion planning algorithms were developed over the years of study of this field. Motion algorithms can be divided into three main classes: sampling-based, grid-based, and optimization-based.

Grid-based methods for planning under differential conditions sample state-space of the model and its inputs in order to generate a state transition system on which planning can be done. Planning on the grid can then be done with methods such as A* [3] or Dijkstra [4]. Although grid-based algorithms work in a deterministic way, they are not suitable as methods for high-dimensional problems. These methods suffer from the curse of dimensionality.

Optimization-based methods commonly employ optimization techniques to find a solution to the motion planning task given as a two-point Boundary value problem (BVP). In real-world problems, the given task is usually non-convex, which is a problem for most optimization solvers, as these can mostly solve only convex problems. Commonly used techniques are Model predictive control (MPC) [**qian_motion_2016**] or Dynamic programming (DP) [5].

Sampling-based methods are probabilistic methods for finding a trajectory in a configuration space. Instead of creating a grid in advance, the grid is gradually created while the algorithm is running. The representation of the sampled space can vary and depend on the chosen method. Algorithms such

as RRT [6] and its variants or Probabilistic road map (PRM) are commonly used. Algorithms based on PRM or RRT are commonly used in the literature to generate feasible trajectories under dynamical constraints [7]. Combining these algorithms with formal language such as LTL, which can be used to express traffic rules, may provide a computationally efficient algorithm to handle such complex tasks as navigating a vehicle in an environment shared with human drivers. For its properties, these methods were chosen as appropriate for the given task (a justification is given throughout the work) and will be further described in this thesis.

2.1 Random sampling algorithms

Given problem definition in the Section 1.2, both the planners of the thesis are subject to the system dynamics. Given the complexity of the problem and high number of dimensions, the family of the RRT algorithm, introduced in [6], was chosen as the appropriate method to solve the problem. RRT was proven computationally efficient [8] and is commonly used in robotics to solve motion planning for nonholonomic systems [9]. Given its simple data structure and graph background, it is pruned for fast growth and sampling of the state space. It is probabilistic complete, and most variations are also asymptotically optimal [10].

Other random sampling techniques such as PRM (e.g., [11]) or potential field algorithms (e.g., [12]) do not provide the unique feature of handling nonholonomic problems of RRT [6].

2.2 Rapidly-exploring random tree original and improved version

RRT is a randomized data structure for motion planning specifically designed for systems with nonholonomic constraints. Let us consider that we need to solve the path planning problem in the continuous configuration space $\mathcal{C}$, which is, for the sake of simplicity, obstacle free. However, the original algorithms did not consider an obstacle-free environment; for the purposes of this thesis, the simplification can be justified by recalling the definition of the problem in Section 1.2, where it is stated that all constraints of the system are considered soft; thus, there cannot be an obstacle in the environment. We

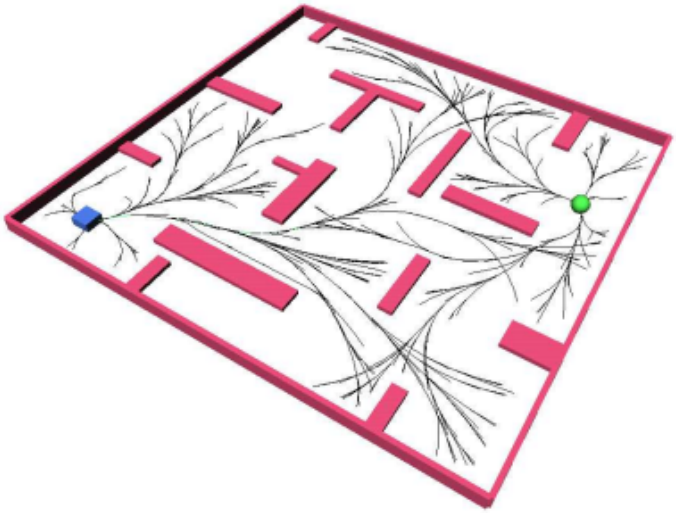


Figure 2.1: Example of RRT under kinodynamic model, adopted from [13]

have an initial configuration $q_0 \in \mathcal{C}$, the final number of vertices in the tree K, and the time interval Δt. See Algorithm 1.

Algorithm 1 Original RRT adopted from [6]

Require: q_0, K, Δt;
Ensure: $\mathcal{T}$ tree with K vertices;
  $\mathcal{T}$.init(q_0);
  while $\mathcal{T}$.count_vertices() $< K$ **do**
    $q_{rand} \leftarrow$ Rand();
    $q_{near} \leftarrow$ NearestNeighbour($q_{rand}, \mathcal{T}$);
    $u \leftarrow$ SelectInput(q_{rand}, q_{near});
    $q_{new} \leftarrow$ NewState($q_{near}, u, \Delta t$);
    $\mathcal{T}$.add_vertex(q_{new});
    $\mathcal{T}$.add_edge(q_{near}, q_{new});
  end while

First, the tree $\mathcal{T}$ is initialized with the root configuration q_0. In the next step, the Rand function returns uniformly distributed random samples from $\mathcal{C}$. NearestNeighbour function in the next step finds the closest configuration in tree $\mathcal{T}$ to q_{rand} followed by the SelectInput function, which finds the input whose trajectory minimizes distance to q_{near}. Lastly, the NewState function is called to evaluate the new state with respect to the original sample in the tree, input, time limit, and differential constraint $\dot{q}(t) = f(q, u, t)$. Tree $\mathcal{T}$ is then updated with this new configuration and new edge.

The original Algorithm 1 is probabilistic complete but it is not asymptotically optimal. Compared to PRM, it uses a much simpler structure with few vertices.

However, like in PRM, the solution is not guaranteed to be optimal. Later, the improved version of RRT called RRT* [14] solved the issue of optimality. See Algorithm 2. Recall that all previous assumptions are valid and the configuration space $\mathcal{C}$ is considered obstacle-free.

Algorithm 2 RRT* adopted from [14]

Require: q_0, K, Δt;
Ensure: $\mathcal{T}$ tree with K vertices;
  $\mathcal{T}$.init(q_0);
  while $\mathcal{T}$.count_vertices() $< K$ **do**
    $q_{rand} \leftarrow$ Rand();
    $q_{nearest} \leftarrow$ NearestNeighbour($q_{rand}, \mathcal{T}$);
    $u \leftarrow$ SelectInput(q_{rand}, q_{near});
    $q_{new} \leftarrow$ NewState($q_{near}, u, \Delta t$);
    $Q_{near} \leftarrow$ Near($\mathcal{T}, q_{new}$);
    $q_{min} \leftarrow q_{nearest}$;
    for $q_{near} \in Q_{near}$ **do**
      $c' \leftarrow$ Cost(q_{near}) + c(Line(q_{near}, q_{new}));
      if $c' <$ Cost(q_{near}) **then**
        $q_{min} \leftarrow q_{near}$;
      end if
    end for
    $\mathcal{T}$.add_vertex(q_{new});
    $\mathcal{T}$.add_edge(q_{min}, q_{new});
    for $q_{near} \in Q_{near} \setminus \{q_{min}\}$ **do**
      if Cost(q_{near}) > Cost(q_{new}) + c(Line(q_{new}, q_{near})) **then**
        $q_{parent} \leftarrow \mathcal{T}$.get_parent($q_{near}$)
        $\mathcal{T}$.remove_edge(q_{parent}, q_{near});
        $\mathcal{T}$.add_edge(q_{new}, q_{near});
      end if
    end for
  end while

The start of Algorithm 2 is the same as that of Algorithm 1 until q_{new} is created. Once q_{new} is found, a set of configurations Q_{near} in tree $\mathcal{T}$ is found within the specified state distance of q_{new}. The distance usually depends on the number of vertices in the tree $\mathcal{T}$. In the next step, the algorithm iterates on the set Q_{near} and, for each configuration, $q_{near} \in Q_{near}$ checks the cost of connecting q_{new} to the tree through q_{near}. After finding such a tree vertex, q_{new} is connected to the tree through it. The second iteration on Q_{near} is then performed. Now, we look for vertices that would have a lower cost if connected to the tree $\mathcal{T}$ through q_{new} instead of their current parent. If such a vertex is found, it is rewired.

Finding the best candidate for the parent and then rewiring the tree solves the problem of optimality, as proved in [14]. The tree is now guaranteed to

be asymptotically optimal. Thus, in a single query, we can find the shortest and optimal path.

2.3 Minimum-violation planning

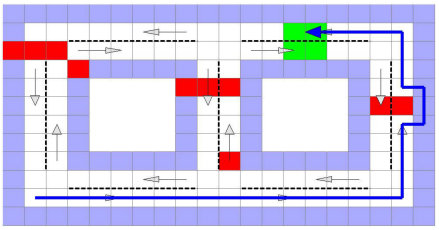
Until now, only system dynamics constraints have been considered, leaving safety and other rules as given in Section 1.2. In this chapter, the transition of traffic and safety rules to LTL and the adaptation of the RRT* algorithm will be discussed. Next, we will discuss how priorities can be introduced to these algorithms as required by the thesis specification. MVP is recent approach for solving motion planning for autonomous vehicles.

LTL is used in MVP to formulate safety rules. Because we are planning on a finite time horizon, we can restrict LTL to Finite Linear Temporal Logic (FLTL). FLTL formula is constructed from:

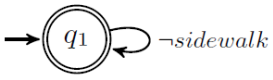
- finite set of atomic propositions Π
- logical operators $\neg$ and $\vee$
- temporal operators **U** (until) and **N** (next)

Using these fundamental operators, we can construct operators such as $/land$, $\Longrightarrow$, $\Longleftrightarrow$, true, and false. Also, we can construct the following temporal operators, such as eventually or always. Let us consider that we have a finite word $w = l_1..l_i..l_n \in (2^{\Pi})^n$ and formula FLTL ϕ over Π and then if w satisfies ϕ we write $w \models \phi$. According to [15], FLTL can be further reduced to si-FLTL$_{G_x}$ formula. This restriction is justified and completely acceptable according to [16] and [17] where it was proven that traffic rules can be expressed using si-FLTL$_{G_x}$ formulas.

MVP can be described using FLTL as follows. Similarly to feedback optimization methods approach, robot can be modeled using deterministic transitions model. The states of the transition system represent the vehicle configuration. Each transition system state can be assigned a set of atomic propositions. These propositions describe, for example, conflicts with safety rules, in the case of this thesis e.g. "Vehicle is on grass" or "Vehicle is conflicting with obstacle detected by lidar". The task is then defined using FLTL formulas and atomic propositions used to label the states of the transition system



(a) : Planned path avoiding as many sidewalks and obstacles as possible



(b) : Automaton for rule "Do not enter sidewalk"

Figure 2.2: Example of simple automaton and planned path, adopted from [19]

together with the priorities of these formulas. MVP is now solved by a trace on the transition system such that it satisfies as many high-priority tasks as possible [18].

In previous works such as [19] MVP was solved as follows. Each rule is translated into a weighted finite automaton. All weighted automata are combined into a single automaton with respect to their weights and priorities. Eventually, the weighted product of the transition system and the combined automaton are calculated. Now, the shortest path can be found. This approach is computationally demanding and grows exponentially. Furthermore, because it is based on the transition system, it is significantly restrictive for high-dimensional systems under differential constraint. In the next subsection, a state-of-the-art approach will be introduced that combines the power of RRT and MVP [15].

2.4 Minimum-violation planning with rapidly-exploring random tree

The state-of-the-art approach suggested in [15] combines the powers of RRT* described in Section 2.2 and MVP described in Section 2.3 was suggested. As opposed to the approach suggested in the previous section, instead of creating the product of the transition model and the weighted finite automaton, which is the product of the weighted finite automata constructed from safety rules and is exponential in size, the Kripke structure [20] is built incrementally using a weighted transition based on safety rules.

RRT algorithm is used to incrementally build the Kripke structure, while FLTL is used to express safety rules and calculate the weights of the transitions. See Algorithm 3 adopted from [15]. $\mathcal{K}$ is the Kripke structure. Other variables

remain the same as in Algorithm 2. The description of functions follows.

Algorithm 3 RRT*-MVP

Require: q_0, $\mathcal{K}$, K;
Ensure: Kripke structure $\mathcal{K}$ with K states;
  $\mathcal{K}$.init(q_0);
  while $\mathcal{K}$.count_states() $< K$ **do**
    $q_{new} \leftarrow$ Rand();
    $Q_{near} \leftarrow$ Near($\mathcal{K}, q_{new}$);
    $q_{min} \leftarrow q_{nearest}$;
    $min\_cost \leftarrow$ Cost($q_{nearest}$) + $c(q_{nearest}, q_{new})$;
    for $q_{near} \in Q_{near}$ **do**
      if steer(q_{near}, q_{new}) $\neq \emptyset$ **then**
        $c' \leftarrow$ Cost(q_{near}) + $c(q_{near}, q_{new})$;
        if $c' \prec$ Cost(q_{near}) **then**
          $q_{min} \leftarrow q_{near}$;
          $min\_cost \leftarrow c'$;
        end if
      end if
    end for
    $\mathcal{K}$.add_transition($q_{min}, q_{new}, min\_cost$);
    $\mathcal{K}$.add_state(q_{new});
    for $q_{near} \in Q_{near} \setminus \{q_{min}\}$ **do**
      if steer(q_{new}, q_{near}) $\neq \emptyset$ **then**
        $c' \leftarrow$ Cost(q_{new}) + $c(q_{new}, q_{near})$
        if $c' \prec$ Cost(q_{near}) **then**
          $q_{parent} \leftarrow \mathcal{K}$.get_parent($q_{near}$)
          $\mathcal{K}$.remove_transition(q_{parent}, q_{near}, c');
          $\mathcal{K}$.add_transition(q_{new}, q_{near});
        end if
      end if
    end for
  end while

Steer function returns trajectory from first to second argument if such exists and is finite. The trajectory is cropped if it is longer than the steer time limit T. The Cost function computes weights of all weighted transitions on the trace from the state to the root. The function c calculates the vector cost of the transition from the first state to the second state. The cost is based on the safety rules described using the formula FLTL. The comparison of costs is always considered lexicographic, denoted by the symbol $\prec$, which will be used in this thesis to express lexicographic less.

Algorithm 3 overcomes issues of most of the algorithms mentioned above.

Given the properties of the RRT* algorithm, it is probabilistic complete and asymptotically optimal. It is suitable for nonholonomic problems and is computationally efficient. Given the properties of MVP, it plans under the safety rules specified in FLTL, which is sufficient for traffic rules with different priorities, always preferring traces with lower cost in higher priority rules. As such, it was chosen as an appropriate solution to the problem of this thesis and will be implemented in the following chapter.

Chapter 3

Planning algorithm design and implementation

Based on the theory given in the previous chapter, the motion planning algorithm is designed. Design and development takes into account the practical limitations of the task and platform that are used for testing and final deployment.

For the purpose of this thesis, other autonomous systems than motion planning are considered to be working but are outside the scope of the thesis. However, a full pipeline is developed as part of the team project; thus, requirements for output of the motion and mission planning system will be raised, as well as requirements for input during the project, and are part of this thesis.

Finally, the motion planning algorithm will be implemented as a general library allowing for future development and multicase usage.

3.1 RRT*-MVP algorithm adaptation

Changes and adaptations of the RRT*-MVP algorithm introduced in Algorithm 3 in this section. The main goal of this thesis is to design and develop a motion planning system that operates on the vehicle specified in the Chapter 4.

Simplifications must be made so that the system works as expected, ensuring safe operation of the vehicle in fully autonomous mode.

The software and hardware architecture of the subscale platform is limited in computing time. From a hardware point of view, the main computing unit has low single-core computing capability. However, it offers a fast and reasonably large memory. From a software point of view, motion planning is time-constrained. The local planner has 500 ms not only to compute the trajectory, but also to handle all necessary communication with other systems. The more efficient the planner is in time, the more samples it can generate. Given the property of probabilistic completeness and asymptotical optimality, more samples provide a better trajectory.

Let us recall the Steer function used in Algorithm 3. The Steer function returns the trajectory between two states if such a trajectory exists and is finite. Both variants of the planner developed in the thesis use dynamics models, which means the Steer function must solve two-point BVP [21].

Finding an analytical solution of two-point BVP given by both models is computationally demanding because both models have nonlinear terms. In control theory, the two-point BVP is commonly solved using shooting methods, the gradient method, and quasilinearization [22]. All of these methods are computationally exhaustive. Solving the two-point BVP each time the steering function is evoked in Algorithm 3 would result in a significant demand for computing power [23]. Instead of creating a bottleneck in the algorithm and lowering its performance, a different approach is chosen.

Input sampling is introduced. In this way, BVP is reduced to Initial value problem (IVP), which can be solved with numerical integrators and the problem of finding the closest point. Furthermore, initial states can be sampled, which would result in loss of precision, but now all possible trajectories can be pre-computed and saved in memory before running the algorithm, sacrificing memory for speed. Due to the geometric meaning of some states (position x and y and heading ϕ), IVP can be considered independent of such states, allowing transformation of precomputed trajectories to actual initial values. IVP has to be solved only for states that would actually change its solution. Then the precomputed solution can be transformed. In the case of the models used in this thesis, the solution of IVP depends only on the velocity v in the local planner. Although this approach suffers from a slight loss of precision given by sampling both input and state, it is greatly overweighted by computational benefits.

Now, the first for loop of the algorithm is unnecessary. The algorithm is

now randomly selecting the configuration to cycle over near nodes to find the closest point in the pre-computed trajectories. Let us optimize random selection to save computing time as the main limiting factor. Heuristics are commonly used to improve the growth of the tree in a desired direction. The improvement in sample selection is made as follows. First, a random sample and a set of near nodes are generated in the same way as in the original Algorithm 3. Second, near-nodes are sorted in ascending lexicographical order on the basis of their cost. Lastly, a random endpoint from a random precomputed trajectory originating in the node that is the least violating near the node is selected as the new node.

The approach is based on the following assumptions. There is a presumption that the least violating near node will produce a better trajectory in terms of MVP as the basis is already the least violating near node. The second assumption is that with tree growth, the set of near nodes occupies a smaller subspace in configuration space. If the subspace given by the set of near nodes shrinks with the growing tree, the probability of expanding nodes that are relatively more violating will rise.

The final Algorithm 4 is presented together with the supporting methods explained in Algorithm 5, Algorithm 6, Algorithm 7, and Algorithm 8. All algorithms are commented on in the following paragraphs.

Algorithm 4 Adjusted RRT*-MVP

Require: q_0, $\mathcal{T}$, K;
Ensure: The tree $\mathcal{T}$ with K nodes;
  $\mathcal{K}$.init(q_0);
  while $\mathcal{K}$.count_nodes() $< K$ **do**
    $q_{rand} \leftarrow$ Rand();
    $Q_{near} \leftarrow$ Near($\mathcal{T}, q_{rand}$);
    $q_{parent} \leftarrow$ SelectNode(Q_{near});
    $q_{new} \leftarrow$ SelectRandomTrajectory(q_{parent});
    $cost \leftarrow \mathcal{T}$.get_cost($q_{parent}$) + $c(q_{near}, q_{new})$
    $\mathcal{T}$.add_edge($q_{parent}, q_{new}, cost$);
    $\mathcal{T}$.add_node(q_{new});
    $Q_{near} \leftarrow$ Near($\mathcal{T}, q_{new}$);
    for $q_{near} \in Q_{near} \setminus \{q_{parent}\}$ **do**
      Rewire($\mathcal{T}, q_{new}, q_{near}$)
    end for
  end while

Algorithm 4 is described in the previous section. It creates a tree $\mathcal{T}$ with K nodes with root in configuration q_0. At each iteration, a random sample is created from the uniform distribution within the configuration space $\mathcal{C}$. The set of near nodes Q_{near} is then found using the Near() method, which

is described in Algorithm 7 and the following paragraphs. The SelectNode() method then selects the configuration q_{parent} from the tree $\mathcal{T}$ to be expanded. The method is described in Algorithm 5 and in the following paragraphs. The SelectRandomTrajectory() method described in Algorithm 6 and in the following paragraphs then selects a new configuration q_{new} using pre-computed trajectories originating in configuration q_{parent}. Configuration q_{new} is then added to the tree. The set Q_{near} is then reset with a new value using the Near method() and the configuration q_{new}. The tree $\mathcal{T}$ is then rewired in the set Q_{near} using the Rewire() method described in Algorithm 8 and in the following paragraphs. In this way, the tree is kept asymptotically optimal.

Algorithm 5 The SelectNode method used in Algorithm 4

Require: Q_{near};
Ensure: q_{best} is the most promising configuration in the set Q_{near};
  $q_{best} \leftarrow$ null;
  $min\_cost \leftarrow \infty$;
  for $q_{near} \in Q_{near}$ **do**
    $c \leftarrow \mathcal{T}$.get_cost(q_{parent});
    if $c \prec min\_cost$ **then**
      $q_{best} \leftarrow q_{near}$;
      $min\_cost \leftarrow c$;
    end if
  end for

The SelectNode method, described with Algorithm 5, selects the most promising configuration q_{best} in the set Q_{near}. The most promising configuration q_{best} is selected as the lexicographic minimum of the costs of the configurations in the set Q_{near}. The cost of each configuration is recovered from the tree $\mathcal{T}$, therefore, it is also dependent on the trajectory from the root of the tree $\mathcal{T}$ to the selected configuration. In other words, the cost also depends on the way in which the configuration is reached, not only on the configuration itself.

Algorithm 6 The SelectRandomTrajectory method used in Algorithm 4

Require: q_{parent};
Ensure: Configuration $q_{selected}$ is randomly selected node that succeeds q_{parent};
  $q_{selected} \leftarrow$ null;
  $input \leftarrow$ SelectRandomInput(q_{parent});
  $selected\_trajectory \leftarrow$ GetTrajectory($q_{parent}, input$);
  $max\_time \leftarrow$ min(length($selected\_trajectory$), $steer\_time\_limit$);
  $rand\_time \leftarrow$ RandomUniform($sampling\_period, max\_time$);
  $q_{selected} \leftarrow selected\_trajectory$.get_configuration_at_time($rand\_time$);

The SelectRandomTrajectory method, described with Algorithm 6, selects the configuration $q_{selected}$ that is preceded by the configuration q_{parent}. First, a random input is selected from the sample range. Second, the pre-computed trajectory, which is a solution to IVP q_{parent} and a constant input selected

in the previous step. The trajectory is cropped to a random length. The trajectory length is selected randomly from the uniform distribution from the sampling period of the model (the trajectory contains at least one sample) to the maximum time. The maximum time is set to the minimum of the two values: the current trajectory length and steer_time_limit which is a tunable parameter used to reduce the maximum expansion of a single node. The configuration $q_{selected}$ is the configuration from the selected trajectory at the time determined in the previous step.

Algorithm 7 The Near method used in Algorithm 4

Require: Tree $\mathcal{T}$, configuration q;
Ensure: Set of configurations Q near configuration q;
  $Q \leftarrow \emptyset$;
  for $q_i \in \mathcal{T}.get\_all\_nodes()$ **do**
    if weighted_state_distance(q, q_i) < dist_lim($\mathcal{T}$.count_nodes()) **then**
      $Q \leftarrow Q \cup \{q_i\}$;
    end if
  end for

The Near method, described with Algorithm 7, creates a set Q near the configuration q. The algorithm is straightforward; If the distance between the configuration q and the other configuration q_i of the tree is less than the distance limit, q_i is added to the set Q. A remarkable part of this method is that the state distance is weighted. These weights are use-case specific and, moreover, model specific. For example, given the definition of model used in the global planner, we place greater stress on the distance in the states x, y, and ϕ, but the velocity v has a lower weight. The main reason is that the tracking algorithm can tolerate jumps in the speed reference but jumps in heading and position main cause an instability. The second remark is on the distance limit. The distance limit is based on the number of nodes in the tree. It is a non-linear nonincreasing function; reasoning behind this was provided in the previous section. The function and its parameters are based on the use case and the configuration space and should be tuned together with steer_time_limit parameter used in the previous paragraph.

Algorithm 8 The Rewire method used in Algorithm 4

Require: $\mathcal{T}, q_{new\_parent}, q_{child}$;
Ensure: Optimal tree $\mathcal{T}$;
  $c' \leftarrow \mathcal{T}$.get_cost($q_{new\_parent}$) + $c(q_{new\_parent}, q_{child})$
  if $c' \prec \mathcal{T}$.get_cost(q_{child}) **then**
    $q_{parent} \leftarrow \mathcal{T}$.get_parent($q_{child}$)
    $\mathcal{T}$.remove_edge($q_{parent}, q_{child}, c'$);
    $\mathcal{T}$.add_node($q_{new\_parent}, q_{child}$);
  end if

The Rewire method, described with Algorithm 8, rewires the configuration q_{child} to the configuration q_{new_parent} if it had been lexicographically lower

cost than it currently is. The cost of q_{child} is calculated with respect to the trajectory of the root configuration; therefore, it also depends on the way the configuration was reached in the tree $\mathcal{T}$ and not only on the configuration q_{child} itself. Rewiring the tree $\mathcal{T}$ after adding each node ensures the asymptotical optimality of the algorithm.

This algorithm combines the powers of RRT* and MVP. It plans under differential constraints; other constraints are soft constraints with priorities. This algorithm can solve the problems defined in Section 1.2.

3.2 Practical considerations, data and requirements

In this section, the practical consideration given by the intended use case on the subscale platform is introduced. In addition, necessary details on the input of the algorithm given by the image processing and the output required by the trajectory tracking algorithm are introduced.

3.2.1 Global planning system

At the time of design and development, it is assumed that the Internet connection is not available and that the vehicle will drive in a limited environment; therefore, important landmarks and coordinates are given together with the task and are available to the executed program. However, given that the vehicle uses Differential global positioning system (DGPS), the origin of the coordinate system map will change at each start based on the position of the base station and the global path must be represented with respect to this frame.

The assumed area for the global planner is captured in Fig. 3.1. It is in the inner courtyard of the Czech Technical University in Prague in Karlovo náměstí 13, Praha 2.

The global planning system is subject to dynamic constraints expressed as

(a) : Map view [24] **(b) :** Orthophoto map view [25]

Figure 3.1: Area of planning of global planning system

Dubins vehicle [26]:

$$\dot{x} = V \cos\theta \tag{3.1}$$

$$\dot{y} = v \sin\theta \tag{3.2}$$

$$\dot{\theta} = V \frac{\sin\delta_f)}{L} \tag{3.3}$$

where x and y are coordinates in the plane, θ is the direction of the vehicle, V is the constant speed, and δ_f is input of the steering angle of the front wheels. The constant L is the distance between the axles.

Dubins vehicle model was selected as the best because of its simplicity. The global planner works as a reference generator for the local planner. It needs to guarantee that the trajectory is feasible and the vehicle can follow it, but at the same time, it should be simplified as much as possible to reduce the dimensionality of the problem.

The priority of constraints for the global planning system is as follows:

1. Surface type
2. Energy consumption

This order is straightforward and ensures that the vehicle avoids undesirable

(a) : Sett paving **(b) :** Grass **(c) :** Paving stone

Figure 3.2: A detailed view of the surface types

surfaces at the possible cost of a higher input cost.

3.2.2 The local planning system

The local planning system should be developed as a proof of concept and must adapt to future change in input. For the purposes of this thesis, the inputs for the local planning system are the Lidar scan and the occupancy grid of the 4x4 m area in front of the vehicle classifying surface of the road. As visible in Fig. 3.1, it has 3 surface types:

- Grass in the middle
- Sett paving around the central square that serves as a service road
- Flat and smooth paving stones in the central square that serves as a footpath

A detailed view of the surface types is shown in Fig. 3.2. Only pavement stones are considered desirable surfaces for driving; others will be negatively classified and should be avoided.

The local planning algorithm must be able to work with different types of input used either as soft constraints or as dynamic constraints. It is assumed that in future work this local planning system will be used in an unknown environment with input given by the predictor of surface friction [27] and should be easily interchangeable by reusing the planning algorithm itself. This

requirement ensures the continuous development of the autonomous system and raises high standards in the architecture and design of the algorithm and the planning system.

As specified in Section 1.2, the algorithm has to prioritize the inputs in the following order (the first has the highest priority):

1. Lidar scan and obstacle avoidance
2. Surface classification
3. Global goal reference
4. Energy consumption

The order ensures that any obstacle detected by lidar is avoided at all other costs. The vehicle will drive on the desirable surface unless it is obstructed by the aforementioned obstacle detected by lidar. The global goal is injected to direct the growth of the tree in a desirable direction. The different order of priorities and its results will be discussed in the following sections and chapters.

The local planning algorithm is subject to the following dynamic constraints of the kinematic model:

$$\dot{x} = v\cos\theta \tag{3.4}$$
$$\dot{y} = v\sin\theta \tag{3.5}$$
$$\dot{\theta} = u \tag{3.6}$$
$$\dot{v} = a \tag{3.7}$$

where states x and y are coordinates in the plane, θ is the angle of direction (or yaw), and v is the speed of the vehicle. The input a represents the acceleration of the vehicle and the input u is the yaw rate.

Compared to the model used in the global planner, the yaw rate is considered as direct input instead of being calculated from the steering angle. Although it may seem counterintuitive, let me first remind you that the rear wheels can also be steered. The yaw rate is calculated from both steering angles as follows.

$$\dot{\theta} = v\frac{\sin\left(\delta_f - \delta_r\right)}{L\cos\delta_r} \tag{3.8}$$

where δ_f and δ_r is the steering angle of the front and rear wheels, respectively, v is the vehicle velocity and L is the axle distance. Because the kinematic

model does not have state in relation to the side-slip angle of the vehicle, aligned-steering is never considered and vehicle can only be counter-steering.

The input range of the yaw rate is set so that the maximum yaw rate is equal to the full counter-steering of the vehicle at the lowest considered velocity. With increasing speed, the vehicle can achieve the maximum yaw rate with decreasing steering angles. Because the platform used is overactuated, it can easily lose traction. This way the maximum steering angle is limited by the velocity of the vehicle; thus it is ensured that maneuvers with shorter radius of the turn are preformed at lower speeds; therefore, operation of the autonomous is much safer.

Now, the vehicle will have an incentive to brake or accelerate. If same way to compute the yaw rate as in the global planner model

$$\dot{\theta} = V \frac{\sin \delta_f}{L} \tag{3.9}$$

would be used, then vehicle could always make the turn with the shortest possible radius at any speed. Now, the only way to make the turn with the shortest radius is to decelerate to the lowest speed.

3.3 Developed library

With this section, we get to the second part of the thesis. The practical approach to Algorithm 4 introduced in the previous section will be discussed. The algorithm needs to be written in code and run on an embedded computational unit. The algorithm was designed with the help of Python3, but, as expected, performance wise, it was insufficient.

The algorithm was incorporated into a single library written in C++ [28]. This approach allows modularity and facilitates future development. The library was used for both the benchmark scenarios and the ROS wrapper used on the platform discussed in the next chapter. The library contains 3 classes that are used to construct the tree.

In the practical approach, using formulas FLTL or building finite automatons is inefficient. It is a useful concept for the theory and design of the algorithm, but implementing it would mean that all nodes would have to be labeled with atomic propositions and then cycle over the atomic proposition to assign cost to the node. Instead, we simplify the implementation into a single run through

a set of cost functions. The result is the same, while saving computational time.

All classes are heavily based on the Eigen library [29], which is used for matrix operations.

3.3.1 TreeNode

TreeNode is a simple class that is used to store the data of the node in the tree. Saves all necessary node description such as parent node, trajectory from the parent node, cost of the trajectory from the parent, and the final configuration.

It provides a few useful methods. The user can ask for the cost and time to the parent node. These are recursive methods that recursively query the cost or time to root from the parent node and add the saved cost or time of the current node from the parent. The recursive approach is taken because of tree rewiring. Each time the node is rewired, the cost or time in all nodes that succeed it must be updated. Due to the simplification of the class and the better maintainability of the tree, the recursive approach was chosen as the best alternative. Another method is used to generate the Yaml [30] node to log node to postprocess the data and debug the algorithm.

TreeNode is also equipped with the necessary getter and setter functions. For full interface description, check the header files in the Appendix of this thesis. For a high-level overview of the library, see Figure 3.3.

3.3.2 Model

The Model class is an abstract class. It creates a dynamic model that is passed to the RRTGraph class described in the following section. Model class is used to create the model of a dynamic agent for which planning should be done.

The class provides a method that must be overridden by dynamic constraints. This function is used in the other methods to compute the trajectories. It also implements methods that directly copy the methods used in Algorithm 4,

namely, the state distance and select random trajectory functions. The state distance can also be overridden to completely change the method of computing the state distance; otherwise, the method computes the distance using a standard norm with weights on individual states.

$$d = \sqrt{\sum_i w_i (x_i - y_i)^2} \tag{3.10}$$

where d is the weighted distance between the configurations x and y, w_i is the weight of the ith state, x_i and y_i is the ith state of the first and second configurations, respectively.

The model class also precomputes the trajectories by numerical integration IVP. The trajectory is then stored in the object and used by other methods in the class.

The class also provides convenient methods to check whether the state is feasible and to generate random configurations, since it basically constructs the configuration space $\mathcal{C}$. Some of the attributes passed as constructor arguments were also mentioned in previous sections as tunable parameters, namely, the steer time limit and sampling period.

The model class is also equipped with a function that can generate the yaml node as a description of the model to postprocess the data and debug the algorithm. It is also equipped with the necessary setter and getter functions. For full interface description, check the header files in the Appendix of this thesis.

3.3.3 RRTGraph

RRTGraph is the last class in the library. It is a wrapper for the aforementioned Algorithm 4. From a user point of view, it is a simple library with only a few methods. Once object is created with Model class object and TreeNode object used as root. The user can call the method to run Algorithm 4 to achieve the specified number of nodes, run a single attempt adding a single node, or retrieve all nodes in the tree, or query directly for a solution to the path planning problem.

The class is also equipped with the method of generating a yaml node of the tree, which is constructed as a sequence of yaml nodes from tree nodes.

For full interface description, check the header files in the Appendix of this thesis.

In the following chapter, results obtained while using this library will be presented and properties of Algorithm 4 will be discussed on examples.

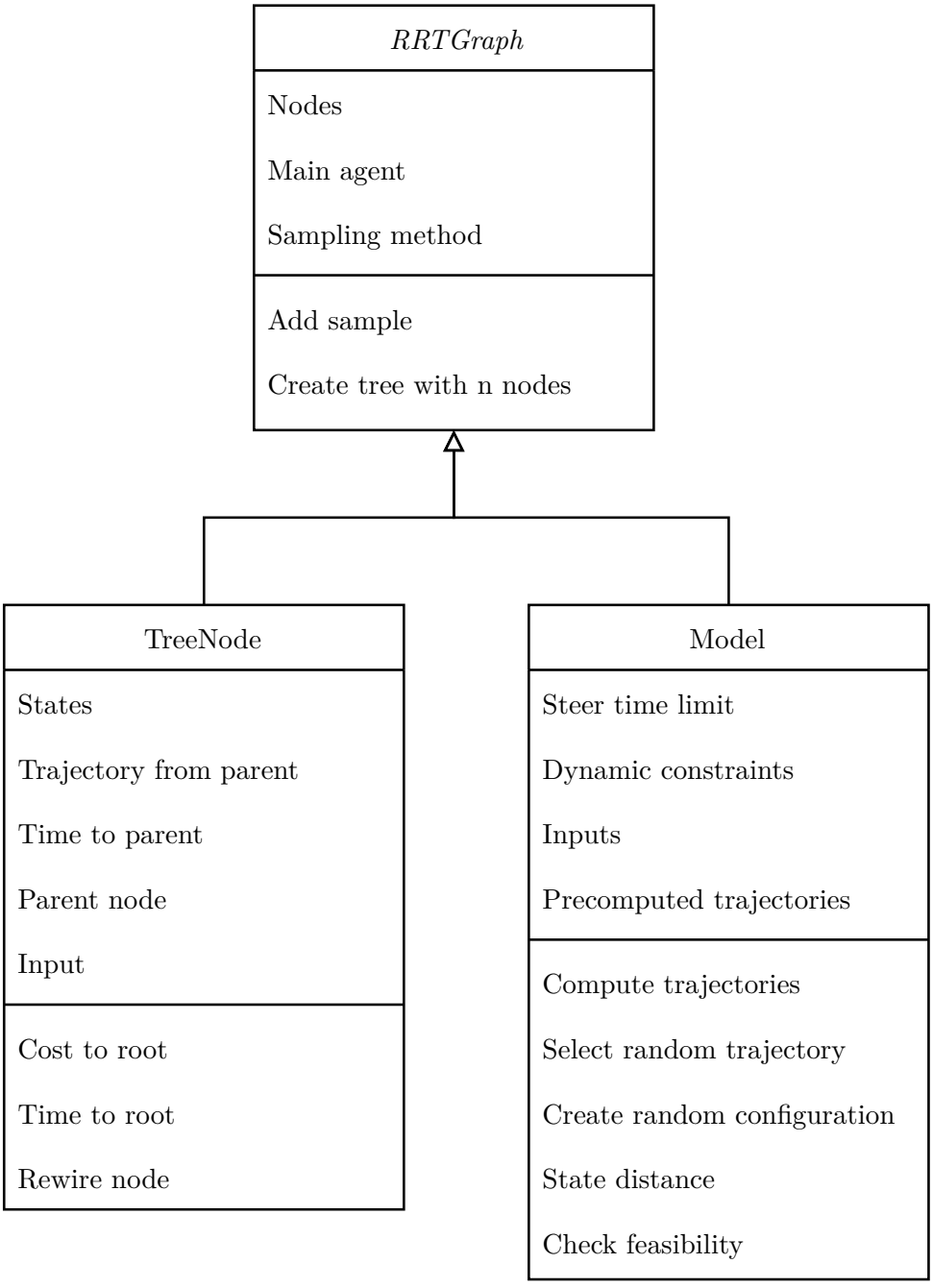


Figure 3.3: High-level library overview

Chapter 4

Subscale platform

The subscale platform used for testing is described in this section. The algorithm was designed to run on the platform. Respectively, the platform is used to validate the algorithm in real-world scenarios.

4.1 Platform base

The autonomous vehicle is built on the Losi 1:5 DBXL-E platform [31] which was redesigned and rebuilt for the purposes of this project [32]. The redesigned platform has each wheel individually steered. It is driven by the rear wheels. The fully loaded vehicle weighs 22.5 kg and can reach speeds of more than 10 ms^{-1}. The power to the motors is delivered from two 4S LiPo batteries. Another 4S LiPo battery is used to power the main computational unit, and the vehicle also has a 5V source for ultra-low voltage units.

The platform is equipped with the following sensors (only relevant to the topic of the thesis are presented):

- Stereolabs camera's ZED2 stereocamera
- Emlids's Navio2
- Hokuyo's URG-04LX-UG01 lidar

Figure 4.1: Photo of the Losi platform

- 4 hall effect sensors for accurate measurement of wheel's rotations
- DGPS with two antennas on vehicle for accurate localization and heading

The platform is also equipped with an RC controller and is capable of running under manual command. This ability was kept, and the autonomous mode is an optional mode that can be switched on and off anytime.

4.2 Hardware architecture

Platform was equipped with the following computing units:

- NVIDIA's Jetson AGX Xavier

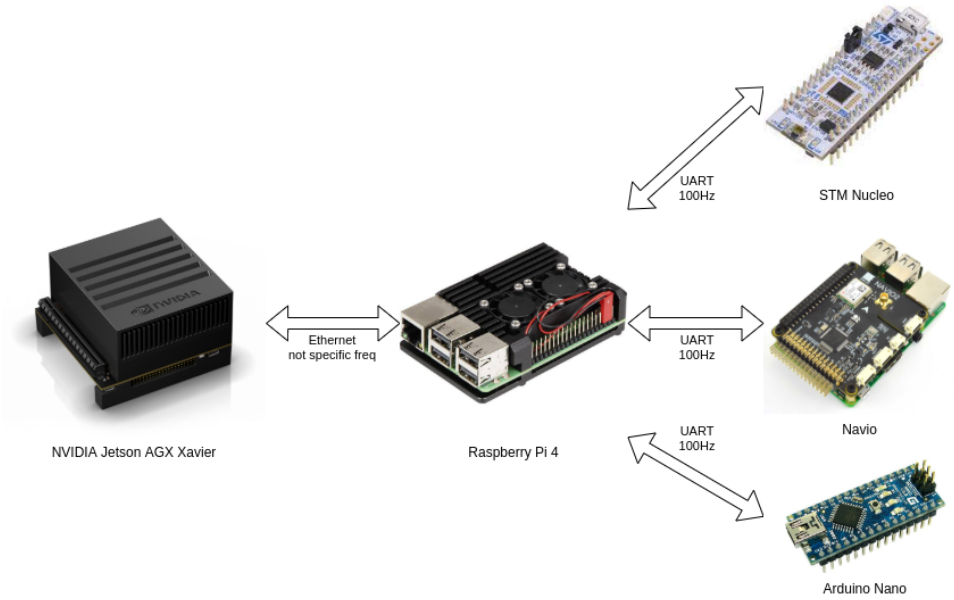


Figure 4.2: Hardware overview

- Raspberry Pi 4
- Raspberry Pi 4 with Navio2
- Arduion Nano
- STM Nucleo

NVIDIA Jetson Xavier is the main processing unit for the autonomous system and is the unit that handles computing of both global and local planners. Raspberry Pi 4 and Raspberry Pi 4 with Navio2 shield are used for lower control of motor and steering; these are also for manual control mode. STM Nucleo and Arduino Nano are lower processing units which do not provide computation power for autonomous system, but process data from sensors.

As illustrated in Fig. 4.2, sensor data are available to Raspberry Pi 4 over UART at 100Hz frequency. Raspberry Pi 4 then processes these data and provides the necessary information to the processes running on Jetson AGX Xavier.

4.3 Software architecture

Autonomous system uses ROS Foxy middleware [33, 34] to handle communication and data recording. The distribution of the nodes on the computation units and all nodes can be seen in Fig. 4.3. As can be seen, NVIDIA Jetson Xavier handles most of the computations of the autonomous system. For this reason and because NVIDIA Jetson Xavier is commonly used in these projects, it will be used for benchmarking of planning algorithms.

The autonomous system follows a standard pipeline in which the camera image is processed, where the surface is segmented and labeled [35]. The global planning algorithm receives the position. The local planning algorithm uses processed data from the camera and the lidar, as well as velocity and position. Local planning algorithm also processes the planned global route in the global planning node. The local planner's output is then fed into the trajectory tracking algorithm [36]. Now, for safety purposes, the pipeline that processes commands and controls motors and steering servos is running on the Raspberry Pi 4. This way if higher functions of the pipeline or hardware itself stop working, the vehicle is still controlled via Raspberry Pi 4. It also processes sensors and GPS signals and provides odometery [37]. As NVIDIA Jetson Xavier is quite occupied by processes, it is more pruned to latching and failing, separating direct motor control to Raspberry Pi 4 ensures that it has enough computation capacity at all times and can shut down the vehicle if autonomous system fails.

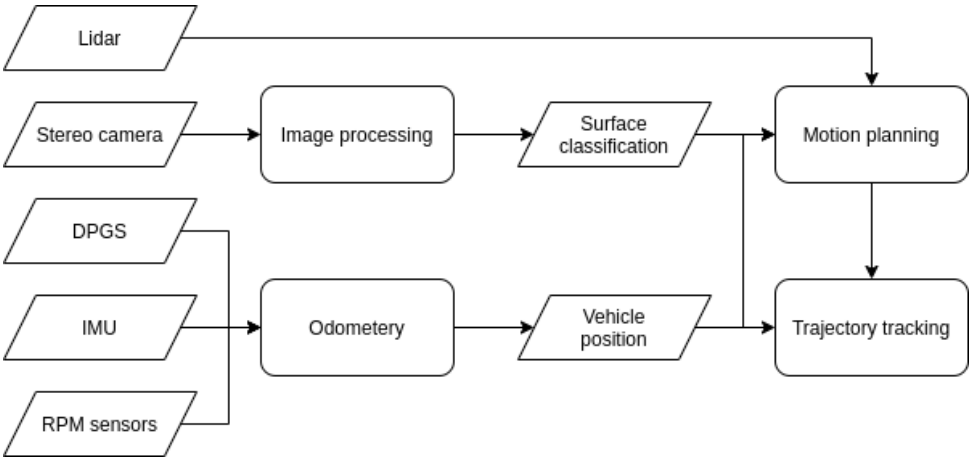


Figure 4.3: Subscale platform pipeline

The output frequency of the image processing is set to 2 Hz. This frequency is given by the method used to process the image and compromises the

computational demands of all systems running on NVIDIA Jetson Xavier.

Most of the codebase is written in C++17, but some of it is written in Python3. The algorithm used in this thesis was designed and tested in Python3, but all the results presented in this thesis are from C++17, which is deployed in the platform.

Chapter 5

Algorithm results on testing scenarios and real-world data

Practical usage of the library designed and described in previous chapter is presented in this chapter. Test scenarios were designed to test the desired behavior of the algorithm. These scenarios and its results are shown in the Section 5.1. Test were done on the platform specified in Chapter 4. That is, benchmark tests were performed on the Nvidia Jetson Xavier AGX unit.

In Section 5.2, the results of the processing of real-world data are shown. The data were post-processed although the algorithm is running properly on the unit itself. The reasons for this are purely practical. While running on the subscale platform, splitting the time between computation and logging data necessary to generate figures for the thesis would severely degrade the algorithm performance. Therefore, the input data is recorded, as these are recorded all the time for debugging purposes, and the output presented was not computed while the full pipeline was running. The performance of the algorithm during post-processing is capped by the performance experienced while running the pipeline.

5.1 Benchmark tests

In this section, benchmark tests are presented. Benchmarking was performed on the Nvidia Jetson AGX Xavier computing unit. Tests are standalone

scenarios that are run multiple times as proofs of the correct design of the algorithm and performance tests.

Note that the graphs in this section show the planned path in the plane. Planning is carried out in the configuration space of 4 or 3 dimensions for the local or global planner, respectively, which means that even if some nodes shown in the figures below overlap in the x-y plane, they are not the same configuration.

5.1.1 Local planner

Local planner benchmark test is done on the designed aritificial input from image processing system. These were chosen so that they stretch the algorithm and show that the algorithm is well designed. In the following text, each scenario is described and its purpose is explained.

The local planner uses two inputs as given by the model described in Section 3.2. Each input has 21 samples. The only state that must be sampled for the precomputation of the trajectories is the velocity that is sampled with step of 0.1 ms-1 within range $[0.5, 3.0]$. The area is limited to 4x4m square and the ranges x and y are set to $[0, 4]$ and $[-2, 2]$, respectively.

The local planner always starts to plan from the point $(0, 0)$ with the heading of 0 radians. The image is captured in the vehicle coordinate frame, so no further transformation is needed. Another remark on the grid is that for the purposes of the planner, the grid is dilated, so that vehicle dimensions are taken into account.

Before diving into scenarios, some global properties of the algorithm can be analyzed. In Figure 5.1, the dependence of the nodes in the tree on the time of growth is shown. The more the configuration space is sampled, the more samples might be rejected because they conflict with the already sampled configuration. This gives the following non-linear dependency of number of nodes in tree on time of the growth. For each data point, 100 runs were performed. The vertical line represents the standard deviation of the execution times.

All scenarios selects tree with 2500 nodes as final node. This threshold was chosen because 2500 nodes is reasonable number of samples given the dimensions of the area. Also these scenarios should test viability of the

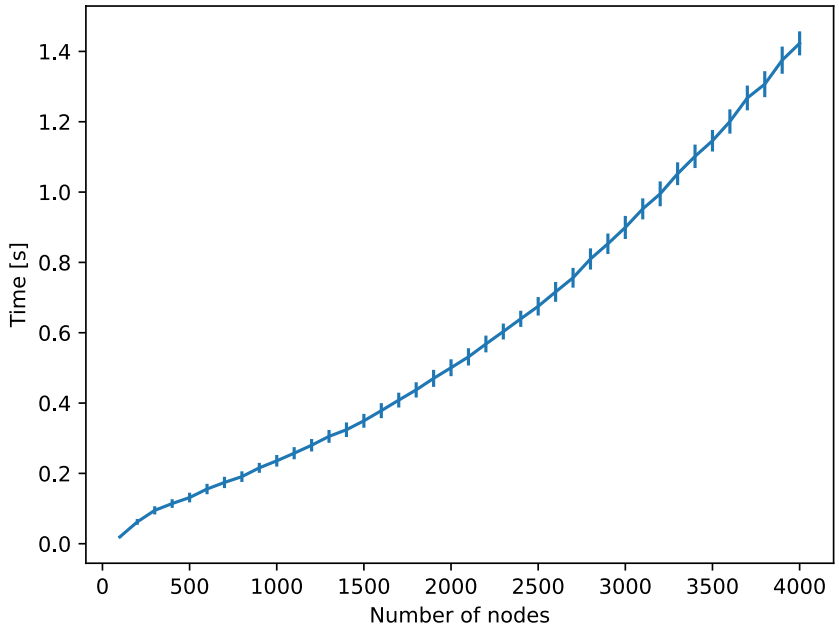


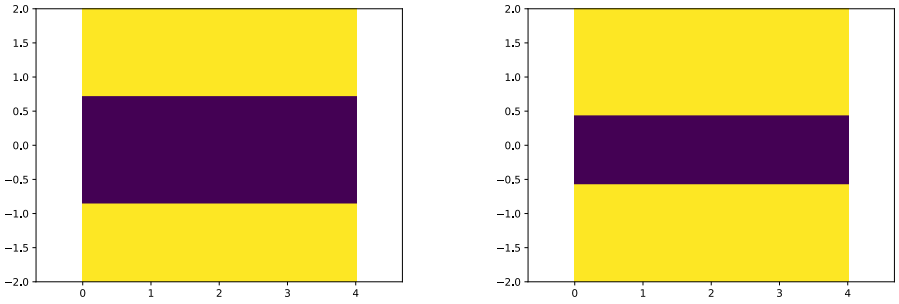
Figure 5.1: Dependency of number of nodes on time of growing the tree

implementation for the subscale platform and as specified in Section 3.2 tree is reset (new data is received) approximately every 0.5 seconds thus 2500 nodes is reasonable goal given data shown in Figure 5.1.

Scenario 1 - straight

In this scenario, the vehicle faces the straight segment of the road. The purpose of this scenario is to prove that the tree is built correctly and that the planner can navigate in a relatively tight corridor. The underlying grid for this scenario can be seen in Figure 5.2. The vehicle has an initial speed of 0.5 ms^{-1}, which is the minimum speed of the model. The global goal is set to the point $(4.0, 0.0)$.

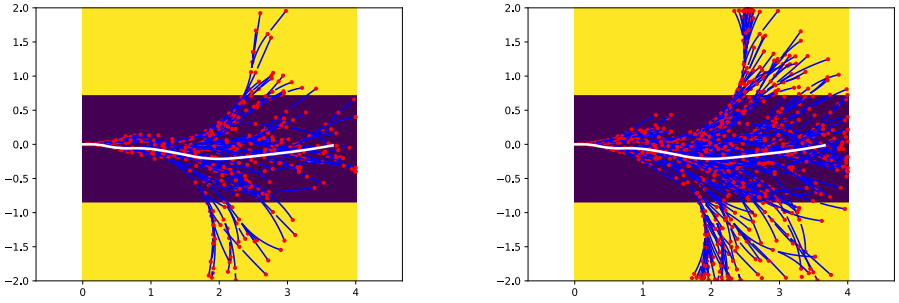
The progress of tree growth can be seen in Figure 5.3. The tree keeps a relatively straight line and navigates successfully through the grid. The correct branching of the tree can also be seen, as the tree is expected to be branching from the center line. The final tree with 2500 nodes and the best trajectory is shown in Figure 5.5.



(a) : Underlying mask classifying surface - original

(b) : Underlying mask classifying surface - dilated

Figure 5.2: Local planner benchmark scenario 1, underlying mask: undesired surface (yellow) and preferred surface (violet)



(a) : Tree with 500 nodes

(b) : Tree with 1000 nodes

Figure 5.3: Local planner benchmark scenario 1 - tree growth. Red points - nodes, blue lines - trajectories, white - chosen path

Scenario 2 - turn

In the second scenario, the ability of the model and the tree to generate a curve is shown. The local goal is set at $(2.0, -2.0)$ The underlying grid is shown in Figure 5.6. The resulting tree is shown in Figure 5.7. For a further test of the planning ability, the vehicle has an initial speed of 3.0 ms^{-1}, which is the maximum speed allowed by the model.

In Figure 5.7, growth of the tree and all random samples togheter with connecting trajectories are shown. As seen in Figure 5.8, where the final tree is shown, the planned trajectory takes the turn correctly.

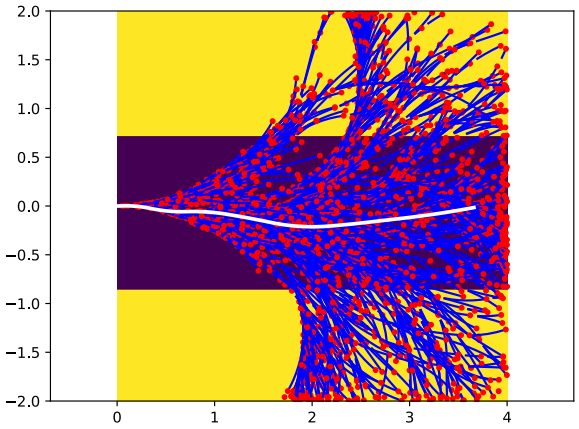


Figure 5.5: Local planner benchmark scenario 1 - final tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white - chosen path

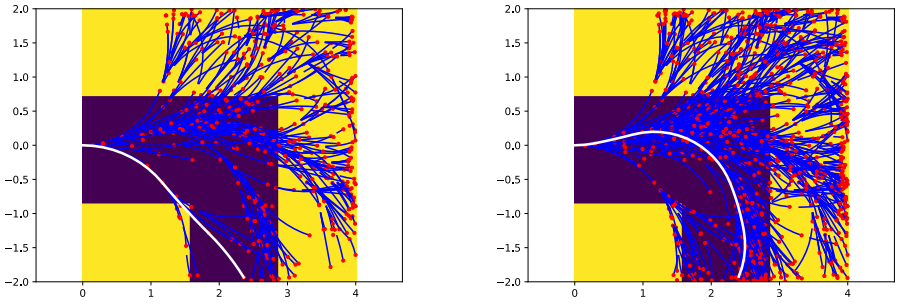
(a) : Underlying mask classifying surface - original

(b) : Underlying mask classifying surface - dilated

Figure 5.6: Local planner benchmark scenario 2, underlying mask: undesired surface (yellow) and preferred surface (violet)

Scenario 3 - zigzag

In the third scenario, the vehicle is headed for a zigzag road. The vehicle has the lowest possible speed of 0.5ms^{-1} in the beginning. The grid and its dilated variant are captured in Figure 5.9. Tree growth and final results are in Figure 5.10 and Figure 5.11, respectively.



(a) : Tree with 500 nodes

(b) : Tree with 1000 nodes

Figure 5.7: Local planner benchmark scenario 2 - Tree growth. Red points - nodes, blue lines - trajectories, white - chosen path

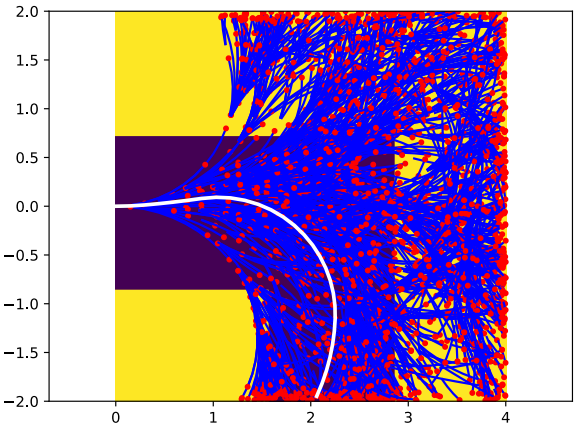
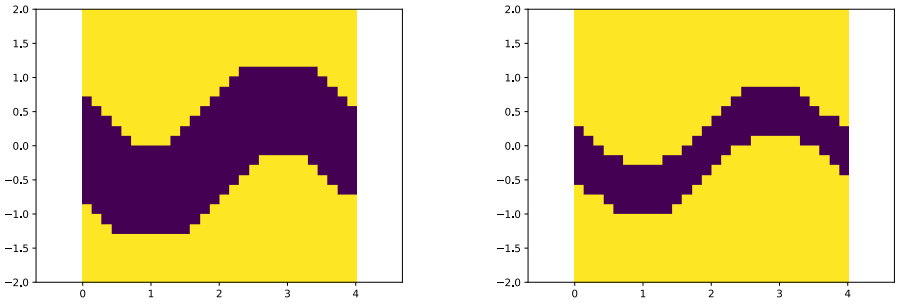


Figure 5.8: Local planner benchmark scenario 2 - Tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white - chosen path



(a) : Underlying mask classifying surface - original

(b) : Underlying mask classifying surface - dilated

Figure 5.9: Local planner benchmark scenario 3, underlying mask: undesired surface (yellow) and preferred surface (violet)

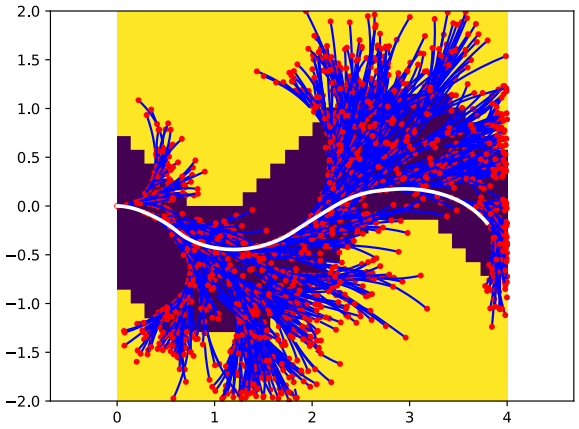
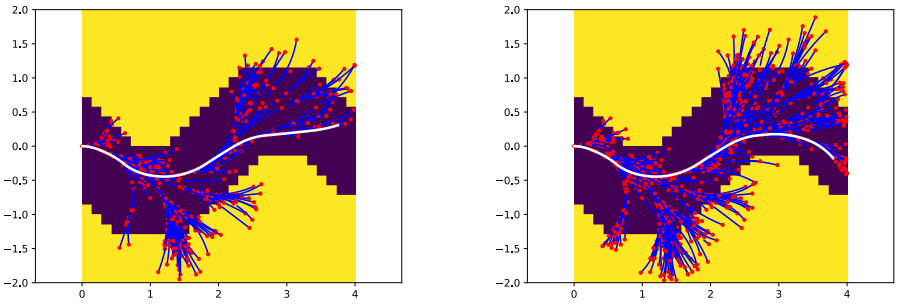


Figure 5.11: Local planner benchmark scenario 3 - final tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white - chosen path

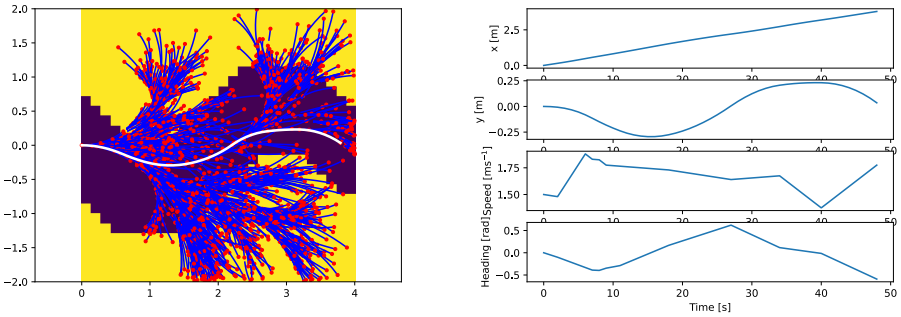


(a) : Tree with 500 nodes

(b) : Tree with 1000 nodes

Figure 5.10: Local planner benchmark scenario 3 - tree growth. Red points - nodes, blue lines - trajectories, white - chosen path

The vehicle successfully takes both turns but, therefore, slightly breaks the rule of not going on the undesired surface, but as soon as possible, the vehicle is navigated on the road and starts copying its shape. Note that the underlying grid shown is not dilated. As can be seen in Figure 5.12, the vehicle gradually increases speed, but is capped at 1.5 ms^{-1}.



(a) : Tree and best path **(b) :** State trajectory

Figure 5.13: Planned path with initial conditions of 1.5 ms^{-1}

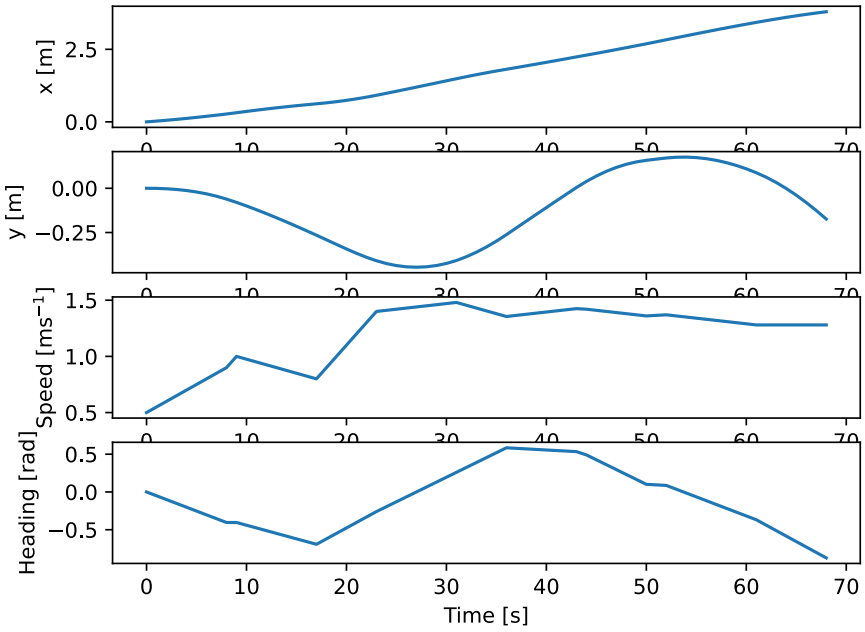
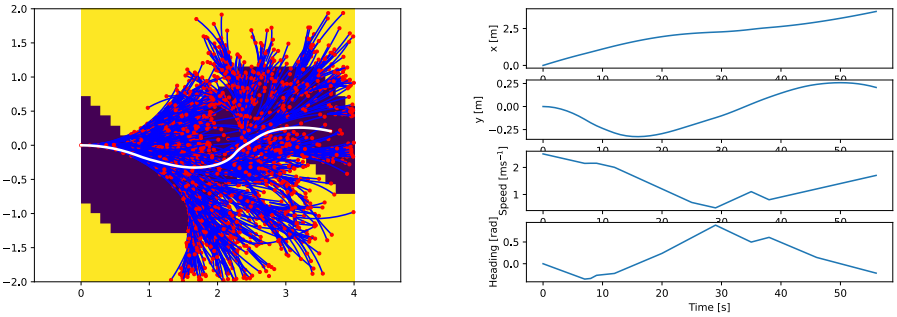


Figure 5.12: State development in zigzag scenario

For comparison, two scenarios with the same underlying grid but different initial conditions. First, in Figure 5.13, the vehicle starts at a speed of 1.5 ms^{-1}, and second, in Figure 5.14, the vehicle starts at a speed of 2.5 ms^{-1}. In both, it can be seen that the vehicle successfully decelerates and accelerates keeping optimal speed and plans the optimal route.



(a) : Tree and best path

(b) : State trajectory

Figure 5.14: Planned path with initial conditions of 2.5 ms^{-1}

5.1.2 Global planner

Given the problem that the global planner is solving, the only way to benchmark the global planner is by changing its initial pose. In this section, multiple scenarios with different poses are tested to demonstrate the viability of the algorithm. The description of each scenario is given in its subsection, respectively.

Similarly to the local planner, in Figure 5.15, the dependence of the number of nodes in the tree on the growth time is depicted. The dependence is exponential. Vertical lines represent the standard deviation of the measurements. For each data point, 100 runs were performed.

The underlying grid is the same for all scenarios and is captured in Figure 5.16.

Scenario 1

In this scenario, the vehicle is placed at $(-35.0, 15.0)$ with a heading of $-\frac{\pi}{2}$ radians. The goal is set to $(0, 0)$. This scenario is a general test of the planner's ability to plan across the entire grid. Tree growth can be seen in Figure 5.18. The results are shown in Figure 5.17.

The heurstic bias can be seen in tree growth. Tree prefers to create samples on the paving stone, occasionally exploring the grass. Although grass sections are not explored extensively, it is assumed that the vehicle will not

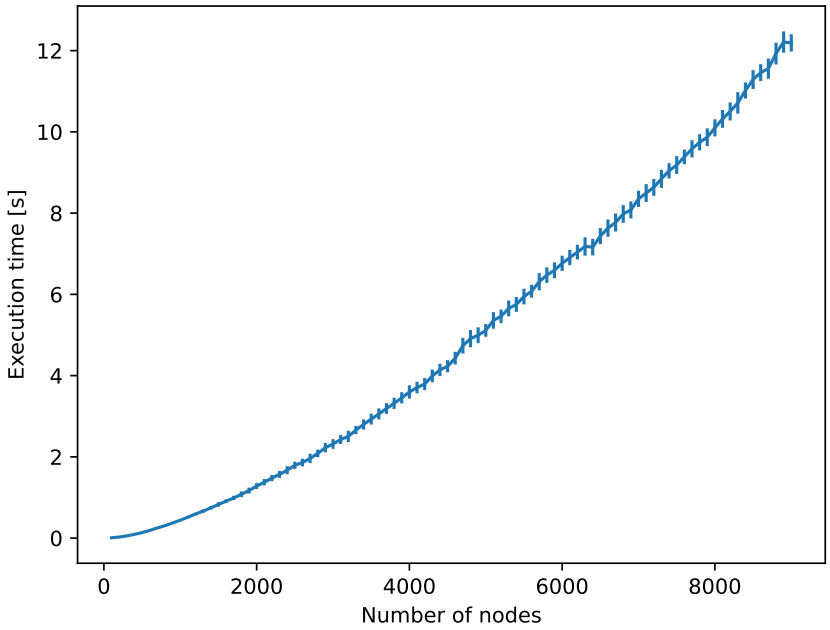


Figure 5.15: Dependency of number of nodes on time of growing the tree

be commanded to navigate to the grass segment; therefore, a global goal will be set on the paving stone footpaths.

The development of the cost based on the number of nodes in the tree is shown in Figure 5.19. Recall that the order of the cost functions follows: surface, distance from the global goal, and input. Vertical lines are the standard deviation of the cost over 100 runs. As seen, at the first level, the highest priority cost function increases. This is caused by the distance to the global goal; At fewer nodes, a node farther from the goal is selected. The farther away from the goal, the less chances you have of violating the highest priority rule. After enough nodes are selected to select the path leading to the global goal, the highest priority rule cost function starts to decrease.

5.2 Processing data from platform

In this section, selected cases captured in the data sets are analyzed and shown. Data were captured live and the trajectory was computed live; however, the results presented here are generated later. Reasoning is given at the beginning of the chapter. The general performance of the ROS wrapper for planners is

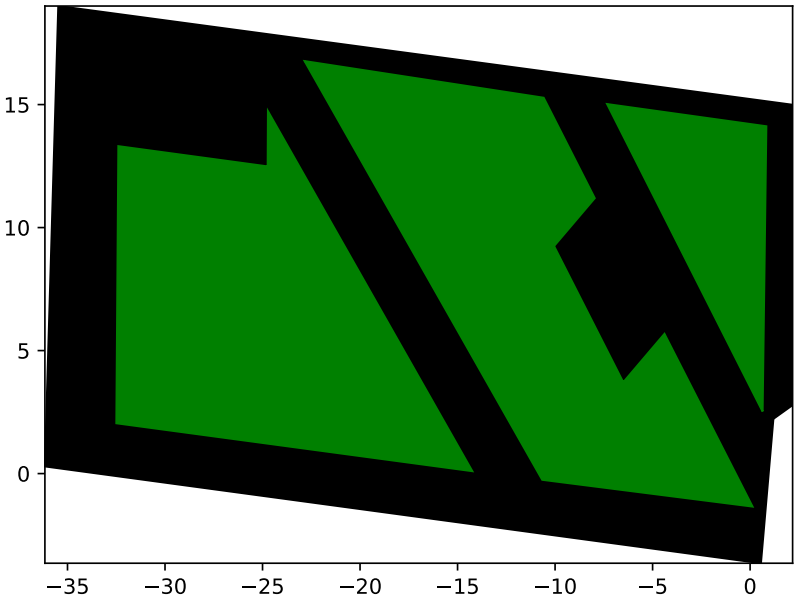


Figure 5.16: Underlying grid for the global planner

given in each subsection, respectively.

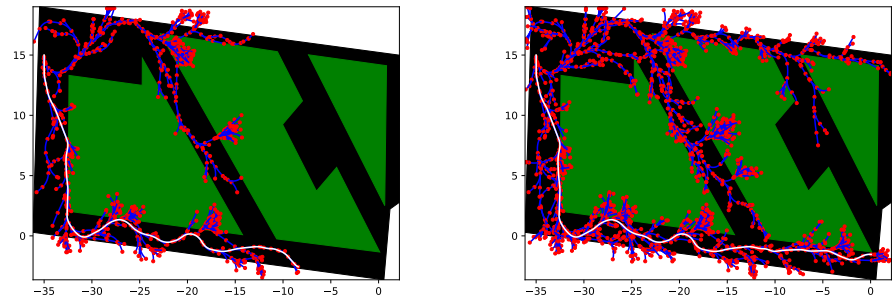
Scenario 1: local planner - split

In this scenario, the car has detected a branching road and is supposed to navigate through it. See Figure 5.20 for details on the input. The input also contains false detections of the undesirable surface on the path. For the second input, Lidar does not return any detections in this scenario.

As specified in the previous section and its respective subsection, the local planner always starts to plan from the point $(0, 0)$ with the heading 0 radians.

During post-processing, the local planner is bounded by the number of nodes it reaches on average while running live, as this parameter is being logged. The local planner on average reaches 2012 nodes (measured on 1000 runs).

This scenario was chosen for the following purposes:



(a) : Tree with 500 nodes **(b) :** Tree with 1000 nodes

Figure 5.17: Local planner benchmark scenario 3 - tree growth. Red points - nodes, blue lines - trajectories, white - chosen path

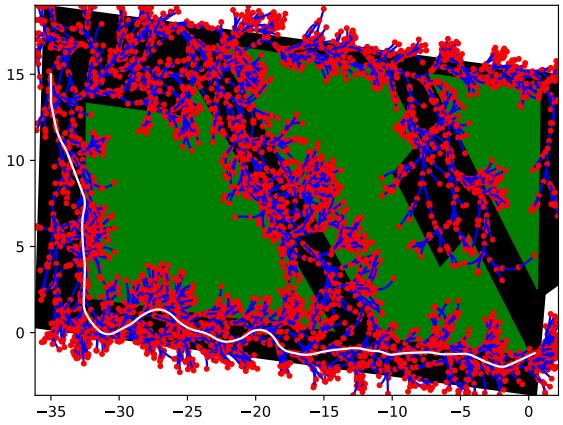


Figure 5.18: Local planner benchmark scenario 3 - final tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white - chosen path

- Proving the ability of the vehicle to steer and accelerate.
- The vehicle should have a tendency to avoid false detections, but is allowed to cross them for a short period of time.
- Selection of the trajectory based on the global goal.

The growth of the tree is depicted in Figure 5.21. The tree was guided towards the goal, given by the global planner at approximately $(2.4, -2.0)$. Results are in Figure 5.22

The result of the same task but with the goal moved to $(4.0, 1.0)$ can be seen in Figure 5.23. In this case, the location of the goal is artificially chosen to prove that the car is guided towards the right goal.

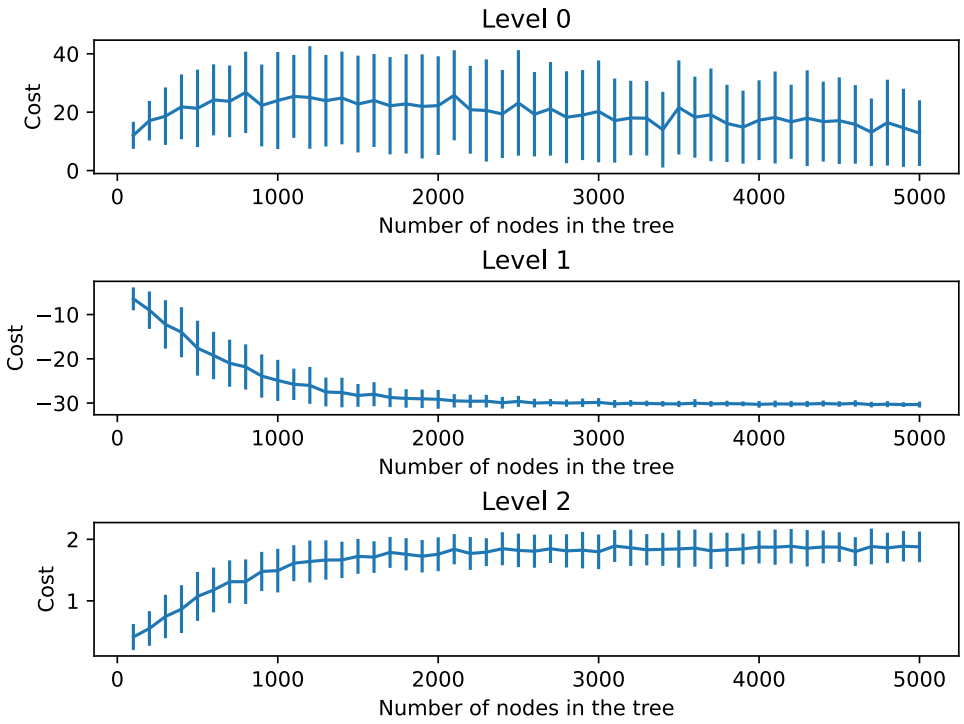


Figure 5.19: Cost functions development

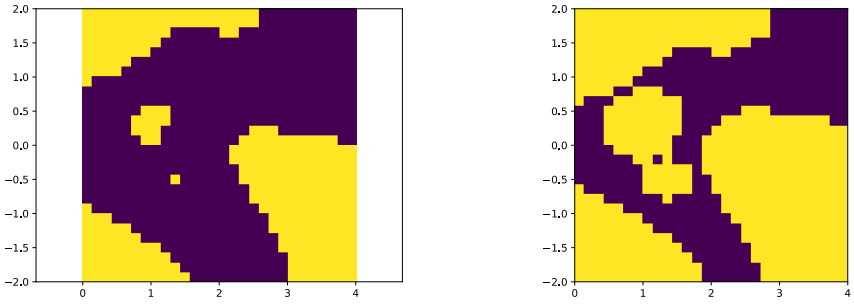
As can be seen in both Figure 5.21 and Figure 5.23, the constraints given by the underlying grid are soft. The tree is growing, so it avoids these areas but occasionally explores them. It is also visible that the planner works properly with the dynamics model and that both control inputs work as intended. The full state-space description of the chosen path is depicted in Figure 5.25.

Scenario 2: full system showcase

In this section, a complete overview of the system is presented. Data for this scenario were captured while driving. In Figure 5.27 image from left and right lens of the stereocamera is shown.

In **??**, the transformed into bird-eye view of the 4x4m area in front of the vehicle, and next to it the results of the surface classification. Note that local frame respects NED coordinate. Origin of the frame is in front of the vehicle, x-axis extends forward in the direction of the vehicle, y-axis increasing to the right of the vehicle.

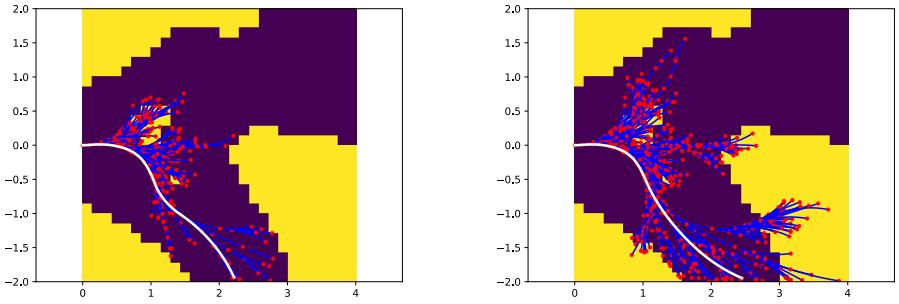
Next, Figure 5.28 shows the global view of the vehicle in the map coordinates.



(a) : Underlying mask classifying surface - original

(b) : Underlying mask classifying surface - dilated

Figure 5.20: Underlying mask classifying surface into two classes: undesired surface (yellow) and preferred surface (violet)



(a) : Tree with 500 nodes

(b) : Tree with 1000 nodes

Figure 5.21: Growth of the tree. Red points - nodes, blue lines - trajectories, white line - chosen path

The global goal and the initial position of the vehicle are connected to the planned global trajectory. The approximate area captured by the image processing is also shown. The tree was capped at 5000 nodes. Based on tests done, this provides enough sampling to safely navigate in the environment and provides approximate period of 5 seconds between publishing global path.

If Figure 5.28 and Figure 5.27 are compared, a slight difference can be observed between the areas captured by the camera and the approximate areas based on odometery. This is the reason to use the local planner. Because odometery and the global map are never perfect, the decision making on short-range ensures safe navigating on short horizon. Finally, Figure 5.29 shows the trajectory planned with the local planner together with the local goal given by the global planner.

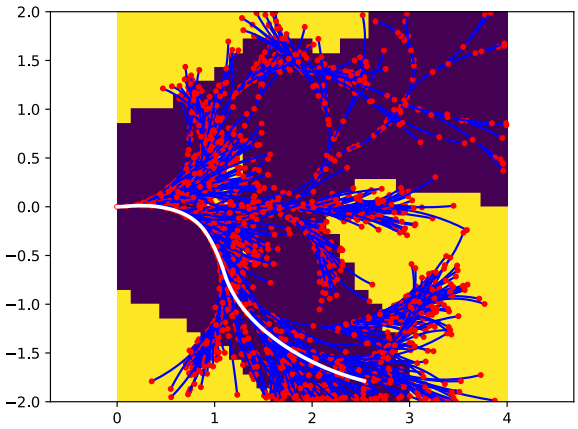
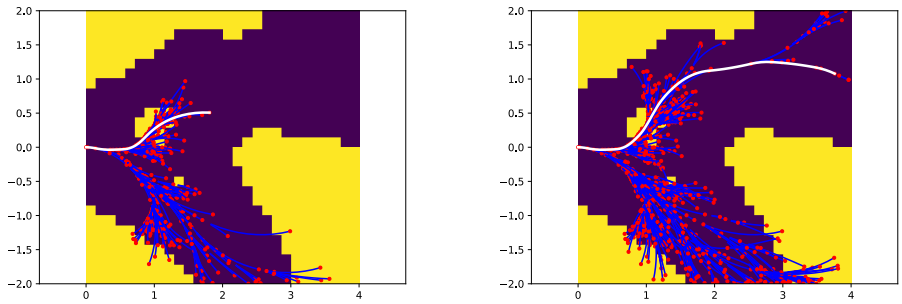


Figure 5.22: Tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white line - chosen path



(a) : Tree with 500 nodes

(b) : Tree with 1000 nodes

Figure 5.23: Growth of the tree. Red points - nodes, blue lines - trajectories, white line - chosen path

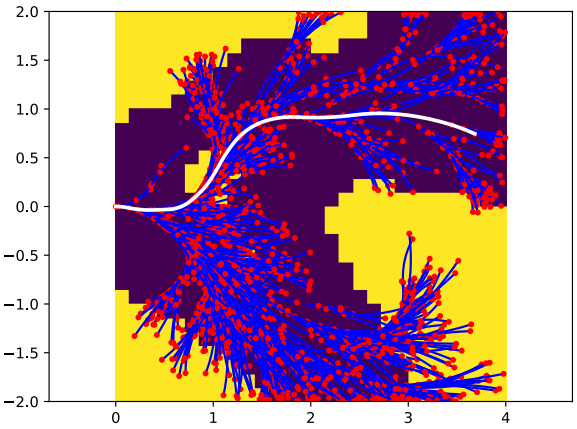
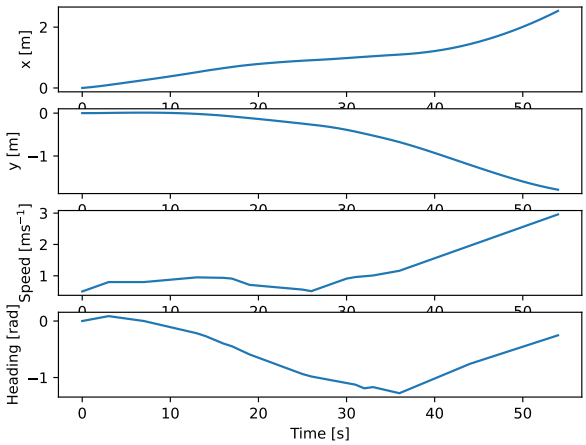
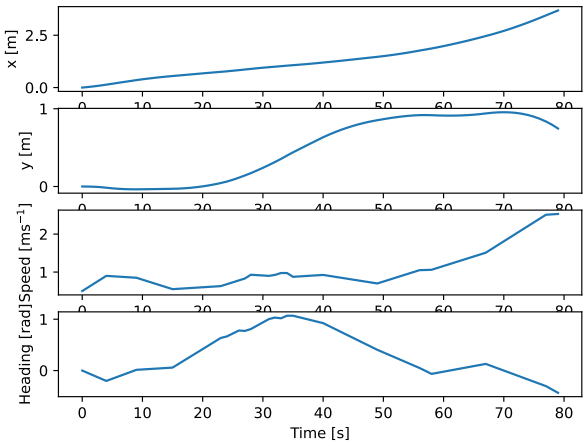


Figure 5.24: Tree with 2500 nodes. Red points - nodes, blue lines - trajectories, white line - chosen path



(a) : State-space trajectory for chosen path in test-case 1 in scenario1 of the local planner



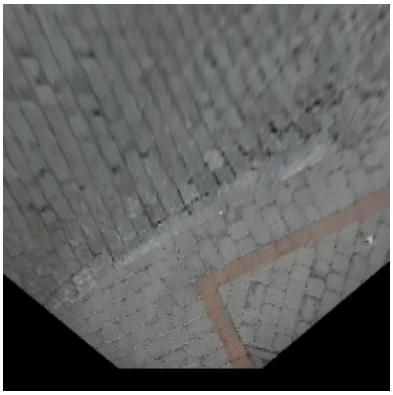
(b) : State-space trajectory for chosen path in test-case 2 in scenario1 of the local planner

Figure 5.25: State-space trajectory for chosen path in test-case 1 and 2 in scenario1 of the local planner

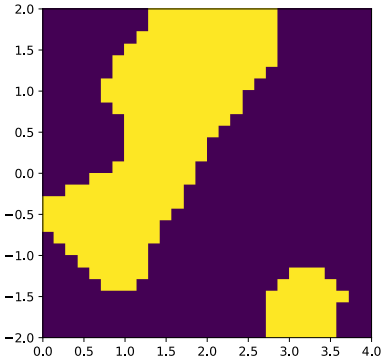
(a) : Original image captured by the left lens of the stereocamera

(b) : Original image captured by the right lens of the stereocamera

Figure 5.26: View on image input captured by the stereocamera



(a) : Bird-eye view transformation of the input image



(b) : Classification grid

Figure 5.27: Image processing

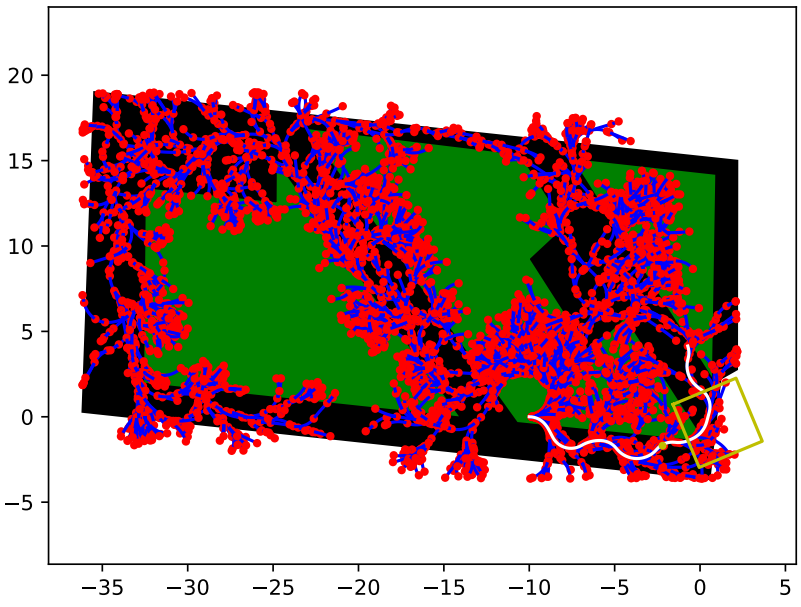


Figure 5.28: Global map with tree (nodes - red, connections - blue) planned trajectory (white) and approximate area captured by image processing (yellow square)

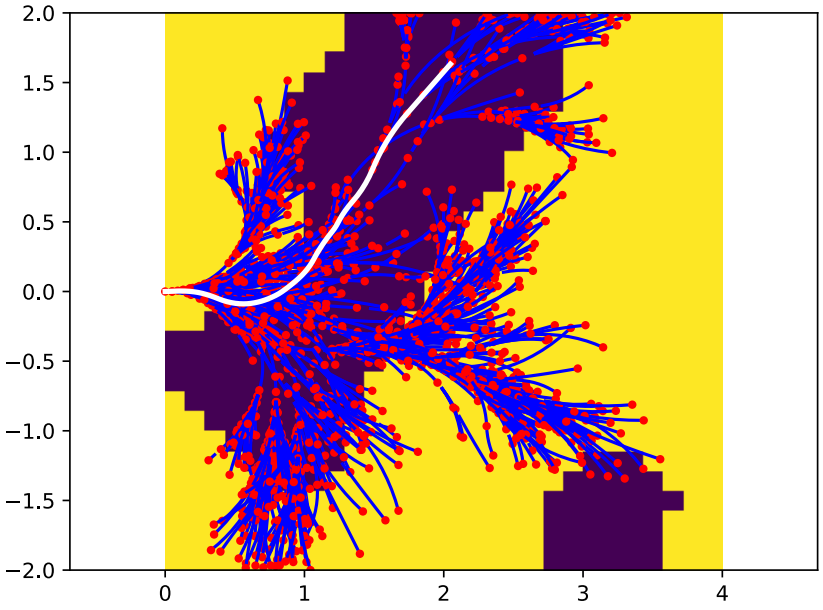


Figure 5.29: Local image frame with planned trajectory (white)

Chapter 6

Conclusion

The motion planning algorithm for an autonomous vehicle was designed in the thesis. The theory of the work was summarized in Chapter 2. In Chapter 3, an algorithm is designed. The designed algorithm is capable of planning under dynamic system constraints given by the vehicle model. It respects traffic and safety rules and their priorities. This makes the algorithm suitable for implementation as part of a fully autonomous pipeline. The algorithm is based on random sampling methods, namely RRT, combined with MVP.

The algorithm designed was written in a C++ library, whose API is described in Chapter 3. In Chapter 5, the library was tested in artificial scenarios and real hardware to demonstrate computational vialability and efficiency. The tests were carried out on real hardware used on the subscale platform described in Chapter 4. Later, the code was also deployed on the aforementioned platform, and the results of the real-world test are also presented as part of Chapter 5.

The algorithm was proven efficient and successfully solved all the tasks given in the thesis specification. In addition to the thesis specification, the library was developed. Furthermore, using the library, the ROS wrapper of the library was deployed on the subscale platform. The library enables the future and continuous development of the autonomous system used on the subscale platform.

As future work, I would recommend further analyzing possibilities of heuristics and testing more vehicle models with the library. The first would allow faster computation of the trajectory, which would result in saving

computational time of the main computational unit, easing deployment of other algorithms that are currently not used on the platform. The second is needed for the intended transition of the underlying mask given by the image processing system. As the mask should give the surface friction coefficient, the model needs to be adapted to allow for usage of such data.

I believe that the work done is viable for both these tasks and gives the user a good starting point for future work to be done.

Appendix A

Bibliography

1. LEONARD, John; HOW, Jonathan; TELLER, Seth; BERGER, Mitch; CAMPBELL, Stefan; FIORE, Gaston; FLETCHER, Luke; FRAZZOLI, Emilio; HUANG, Albert; KARAMAN, Sertac; OTHERS. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*. 2008, vol. 25, no. 10, pp. 727–774. Publisher: Wiley Online Library.
2. HUTH, Michael; RYAN, Mark. *Logic in computer science: modelling and reasoning about systems*. 2nd ed. Cambridge [U.K.] ; New York: Cambridge University Press, 2004. ISBN 9780521543101.
3. ZENG, W.; CHURCH, R. L. Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science* [online]. 2009, vol. 23, no. 4, pp. 531–543 [visited on 2022-05-20]. ISSN 1365-8816, ISSN 1362-3087. Available from DOI: 10.1080/13658810801949850.
4. WANG, Huijuan; YU, Yuan; YUAN, Quanbo. Application of Dijkstra algorithm in robot path-planning. In: *2011 Second International Conference on Mechanic Automation and Control Engineering*. 2011, pp. 1067–1069. Available from DOI: 10.1109/MACE.2011.5987118.
5. BELLMAN, Richard. *Dynamic programming*. Dover ed. Mineola, N.Y: Dover Publications, 2003. ISBN 9780486428093.
6. LAVALLE, Steven M; OTHERS. *Rapidly-exploring random trees: A new tool for path planning*. Ames, IA, USA, 1998. Available also from: http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf.
7. LAVALLE, Steven M. *Planning algorithms*. Cambridge university press, 2006.

8. LAVALLE, Steven M.; KUFFNER, James J.; JR. Rapidly-Exploring Random Trees: Progress and Prospects. In: *Algorithmic and Computational Robotics: New Directions*. 2000, pp. 293–308.

9. VONASEK, Vojtech; PENICKA, Robert. Computation of Approximate Solutions for Guided Sampling-Based Motion Planning of 3D Objects. In: *2019 12th International Workshop on Robot Motion and Control (RoMoCo)* [online]. Poznań, Poland: IEEE, 2019, pp. 231–238 [visited on 2022-05-20]. ISBN 9781728129754. Available from DOI: 10.1109/RoMoCo.2019.8787344.

10. SOLOVEY, Kiril; JANSON, Lucas; SCHMERLING, Edward; FRAZZOLI, Emilio; PAVONE, Marco. Revisiting the Asymptotic Optimality of RRT. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)* [online]. Paris, France: IEEE, 2020, pp. 2189–2195 [visited on 2022-05-18]. ISBN 9781728173955. Available from DOI: 10.1109/ICRA40945.2020.9196553.

11. KAVRAKI, Lydia E; SVESTKA, Petr; LATOMBE, J-C; OVERMARS, Mark H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*. 1996, vol. 12, no. 4, pp. 566–580. Publisher: IEEE.

12. AMATO, N.M.; WU, Y. A randomized roadmap method for path and manipulation planning. In: *Proceedings of IEEE International Conference on Robotics and Automation* [online]. Minneapolis, MN, USA: IEEE, 1996, vol. 1, pp. 113–120 [visited on 2022-05-07]. ISBN 978-0-7803-2988-1. Available from DOI: 10.1109/ROBOT.1996.503582.

13. LAVALLE, Steven M.; KUFFNER, James J. Randomized Kinodynamic Planning. *The International Journal of Robotics Research* [online]. 2001, vol. 20, no. 5, pp. 378–400 [visited on 2022-05-08]. ISSN 0278-3649, ISSN 1741-3176. Available from DOI: 10.1177/02783640122067453.

14. KARAMAN, S.; FRAZZOLI, E. Incremental Sampling-based Algorithms for Optimal Motion Planning. In: *Robotics: Science and Systems VI*. Robotics: Science and Systems Foundation, 2010, vol. abs/1005.0416. Available from DOI: 10.15607/rss.2010.vi.034. Journal Abbreviation: CoRR.

15. WONGPIROMSARN, Tichakorn; SLUTSKY, Konstantin; FRAZZOLI, Emilio; TOPCU, Ufuk. Minimum-violation planning for autonomous systems: Theoretical and practical considerations. In: *2021 American Control Conference (ACC)*. IEEE, 2021, pp. 4866–4872.

16. CASTRO, Luis I. Reyes; CHAUDHARI, Pratik; TUMOVA, Jana; KARAMAN, Sertac; FRAZZOLI, Emilio; RUS, Daniela. Incremental sampling-based algorithm for minimum-violation motion planning. In: *52nd IEEE Conference on Decision and Control*. IEEE, 2013. Available from DOI: 10.1109/cdc.2013.6760374.

17. WONGPIROMSARN, Tichakorn; KARAMAN, Sertac; FRAZZOLI, Emilio. Synthesis of provably correct controllers for autonomous vehicles in urban environments. In: *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2011. Available from DOI: 10.1109/itsc.2011.6083056.

18. TUMOVA, Jana; CASTRO, Luis I. Reyes; KARAMAN, Sertac; FRAZZOLI, Emilio; RUS, Daniela. Minimum-violation LTL planning with conflicting specifications. In: *2013 American Control Conference*. IEEE, 2013, pp. 200–205. Available from DOI: 10.1109/acc.2013.6579837. Backup Publisher: IEEE.

19. TUMOVA, Jana; HALL, Gavin C.; KARAMAN, Sertac; FRAZZOLI, Emilio; RUS, Daniela. Least-violating control strategy synthesis with safety rules. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control - HSCC '13*. ACM Press, 2013. Available from DOI: 10.1145/2461328.2461330.

20. KRIPKE, Saul. Semantical considerations of the modal logic. *Studia Philosophica*. 2007, vol. 1.

21. ZWILLINGER, Daniel. *Handbook of Differential Equations.* [online]. Saint Louis: Elsevier Science, 2015 [visited on 2022-05-08]. ISBN 978-1-4832-2096-3. Available from: http://qut.eblib.com.au/patron/FullRecord.aspx?p=1901454. OCLC: 1041861798.

22. DIEHL, Moritz; GROS, Sébastien. Numerical optimal control. *Optimization in Engineering Center (OPTEC)*. 2011.

23. TURNOVEC, Petr. *Optimal planning and control of vehicle dynamics*. 2021. Available also from: http://hdl.handle.net/10467/95322. MA thesis. Czech Technical University in Prague.

24. OPENSTREETMAP FOUNDATION. *OpenStreetMap* [online]. 2022-05-20. [visited on 2022-05-20]. Available from: https://www.openstreetmap.org/#map=19/50.07645/14.41759.

25. SEZNAM.CZ, A.S. *Mapy.cz* [online]. 2022-05-20. [visited on 2022-05-20]. Available from: https://mapy.cz/zakladni?x=14.4178885&y=50.0763904&z=20&base=ophoto.

26. DUBINS, L. E. On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents. *American Journal of Mathematics*. 1957, vol. 79, no. 3, p. 497. Available from DOI: 10.2307/2372560. Publisher: JSTOR.

27. VOSAHLIK, David; CECH, Jan; HANIS, Tomas; KONOPISKY, Adam; RURTLE, Tomas; SVANCAR, Jan; TWARDZIK, Tomas. Self-Supervised Learning of Camera-based Drivable Surface Friction. In: *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 2773–2780. Available from DOI: 10.1109/itsc48978.2021.9564894.

28. *RRT-MVP Gitlab repository* [online]. 2022-05-20. [visited on 2022-05-20]. Available from: `https://gitlab.fel.cvut.cz/hanistom/toyota-cross-modal-training/-/tree/master/Code/Cpp/rrtmvp-planner`.

29. *Eigen library* [Eigen] [online]. 2022-05-20. [visited on 2022-05-20]. Available from: `https://eigen.tuxfamily.org/`.

30. *The Official YAML Web Site*. 2022-05-20. Available also from: `yaml.org`.

31. HORIZON HOBBY, LLC. *Losi | Horizon Hobby RC Cars, RC Trucks, and RC Vehicle Parts* [online]. 2022-05-20. [visited on 2022-05-20]. Available from: `https://www.horizonhobby.com/losi/`.

32. RUTRLE, Tomas. *Development of verification platform for overactuated vehicles*. 2020. Master's Thesis. Czech Technical University in Prague.

33. QUIGLEY, Morgan; CONLEY, Ken; GERKEY, Brian; FAUST, Josh; FOOTE, Tully; LEIBS, Jeremy; WHEELER, Rob; NG, Andrew Y; OTHERS. ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software*. Kobe, Japan, 2009, vol. 3, p. 5. Issue: 3.2.

34. THOMAS, Dirk; WOODALL, William; FERNANDEZ, Esteve. Next-generation ROS: Building on DDS. In: *ROSCon Chicago 2014* [online]. Chicago, IL: Open Robotics, 2014 [visited on 2022-05-08]. Available from DOI: `10.36288/ROSCon2014-900183`.

35. KONOPISKÝ, Adam. *Surface Properties Prediction from Images Using Self-supervised Learning*. [N.d.]. Master's Thesis. Czech Technical Univeristy in Prague.

36. ŠVANCAR, Jan. *Autonomous vehicle trajectory tracking algorithms*. [N.d.]. Master's Thesis. Czech Technical Univeristy in Prague.

37. TWARDZIK, Tomáš. *Autonomous vehicle position data fusion*. [N.d.]. Master's Thesis. Czech Technical Univeristy in Prague.

Appendix B

Motion planning library header files

B.1 RRTGraph

```
#include <cstdlib>
#include <vector>
#include <string>
#include <cmath>
#include <Eigen/Dense>
#include <Eigen/Core>
#include <random>
#include "Model.h"
#include "TreeNode.h"
#include <yaml-cpp/yaml.h>


#ifndef RRTMVP_PLANNER_RRTGRAPH_H
#define RRTMVP_PLANNER_RRTGRAPH_H


class RRTGraph {
public:
    RRTGraph(TreeNode* root,
             std::vector<std::vector<std::function<
                 double(Eigen::RowVectorXd,
                                                    Eigen
                                                        ::
                                                    RowVectorXd
```

```
                ,

            std
                ::
                vector
                <
                double
                >,

            Eigen
                ::
                MatrixXd
                ,

            Eigen
                ::
                VectorXd
                )
                >>>

                cost_fun
                ,

                Model * main_agent,
                int sample_selection_method=4,
                double dist_lim_coefficient=1.);
    ~RRTGraph();
    void create_tree(unsigned int n_nodes);
    std::vector<TreeNode *> find_shortest_path(Eigen::
        RowVectorXd goal, double xy_dist_lim=4.);
    int add_sample();
    const std::vector<TreeNode *> &get_nodes() const;
    YAML::Node generate_yaml() const;

private:
    Eigen::VectorXd compute_cost(const Eigen::
        RowVectorXd& start,
                                 const Eigen::
                                     RowVectorXd& goal,
                                 const std::vector<
                                     double>& inp,
                                 const Eigen::MatrixXd&
                                      trajectory,
                                 const Eigen::VectorXd&
                                      time_vec);
```

```
    std::vector<std::vector<std::function<double (Eigen
        ::RowVectorXd,
                                                  Eigen
                                                      ::
                                                      RowVectorXd
                                                      ,
                                                  std::
                                                      vector
                                                      <
                                                      double
                                                      >,
                                                  Eigen
                                                      ::
                                                      MatrixXd
                                                      ,
                                                  Eigen
                                                      ::
                                                      VectorXd
                                                      )
                                                     >>>

                                                      cost_functions
                                                      ;
    std::vector<TreeNode*> nodes;
    double dist_lim_coefficient;
    double dist_lim;
    Eigen::MatrixXd main_agent_states;
    int sample_selection_method;
    Model * main_agent;
    std::default_random_engine generator;
};


#endif //RRTMVP_PLANNER_RRTGRAPH_H
```

B.2 Model

```
#include <cstdlib>
#include <vector>
#include <string>
#include <cmath>
#include <Eigen/Dense>
#include <Eigen/Core>
#include <random>
```

```
#include <iostream>
#include <yaml-cpp/yaml.h>

#ifndef RRTMVP_PLANNER_MODEL_H
#define RRTMVP_PLANNER_MODEL_H


class Model {
public:
    Model(std::vector<std::vector<double>> inputs = std
        ::vector<std::vector<double>>(),
            std::vector<double> upper_state_limits = std
                ::vector<double>(),
            std::vector<double> lower_state_limits = std
                ::vector<double>(),
            std::vector<bool>
                solution_dependent_on_states = std::vector
                <bool>(),
            bool load = false,
            std::string data_dir = "",
            std::string precomputation_method = "runge",
            std::vector<double> distance_weights=std::
                vector<double>(),
            double steer_dist_threshold = 0.2,
            std::vector<double> state_sampling = std::
                vector<double>(),
            double steer_time_limit = 1.0,
            double sampling_period = 0.05);
    int get_n_states() const { return n_states; }

    virtual Eigen::VectorXd state_distance(const Eigen
        ::MatrixXd& state1, const Eigen::RowVectorXd&
        state2) const;
    virtual Eigen::VectorXd state_distance(const Eigen
        ::RowVectorXd& state1, const Eigen::MatrixXd&
        state2) const;
    virtual Eigen::VectorXd state_distance(const Eigen
        ::RowVectorXd& state1, const Eigen::RowVectorXd&
         state2) const;

    virtual std::vector<bool> check_feasible(Eigen::
        RowVectorXd state);
    virtual std::vector<bool> check_feasible(Eigen::
        MatrixXd states);

    virtual bool check_collision(const Eigen::
```

```
        RowVectorXd& state1, const Eigen::RowVectorXd&
        state2) const;

    Eigen::RowVectorXd generate_random_sample();

    int select_random_trajectory(Eigen::RowVectorXd
        start, Eigen::MatrixXd & trajectory, std::vector
        <double> & inp);

    double get_sampling_period() const;

    int steer(const Eigen::RowVectorXd& start, const
        Eigen::RowVectorXd& goal, Eigen::MatrixXd &
        trajectory, std::vector<double> & inp);
    void compute_trajectories();
    YAML::Node generate_yaml() const;

protected:
    virtual Eigen::RowVectorXd apply_input(const Eigen
        ::RowVectorXd & state, const Eigen::RowVectorXd
        & input) const = 0;

private:
    std::vector<std::vector<double>> inputs;
    std::vector<std::vector<double>> inputs_product;
    std::vector<double> upper_state_limits;
    std::vector<double> lower_state_limits;
    std::vector<double> state_sampling;
    int n_states;
    std::string precomputation_method;
    std::string data_dir;
    double steer_t_limit;
    std::vector<double> distance_weights;
    std::vector<bool> solution_dependent_on_states;
    double steer_dist_threshold;
    bool load;
    double sampling_period;
    std::vector<std::vector<Eigen::MatrixXd>>
        trajectories;
    std::vector<Eigen::RowVectorXd> initial_values;
    std::vector<Eigen::MatrixXd> mapping;

    void load_trajectories();


    static std::vector<std::vector<double>>
```

```
        get_cartesian_vector_product(const std::vector<
        std::vector<double>>& inp);


    void solve_ivp(const Eigen::RowVectorXd & iv, const
         Eigen::RowVectorXd & inp, std::vector<double> &
         time_vec, Eigen::MatrixXd & solution);

};


#endif
```

B.3 TreeNode

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <string>
#include <chrono>
#include <ctime>
#include <Eigen/Dense>
#include <yaml-cpp/yaml.h>

#ifndef RRTMVP_PLANNER_TREENODE_H
#define RRTMVP_PLANNER_TREENODE_H

class TreeNode {
public:
    TreeNode() = default;
    TreeNode(TreeNode * new_parent,
             Eigen::VectorXd cost_to_parent,
             Eigen::RowVectorXd main_agent_state,
             double time_to_parent,
             Eigen::MatrixXd trajectory_from_parent,
             const std::vector<double>& input);
    TreeNode(const TreeNode&) = default;
    TreeNode(TreeNode&&) = default;
    TreeNode& operator=(const TreeNode&) & = default;
    TreeNode& operator=(TreeNode&&) & = default;
    Eigen::VectorXd get_cost_to_root() const;
    double get_time_to_root() const;
    TreeNode * get_parent() const { return parent; };
```

```
    double get_time_to_parent() const { return
        time_to_parent; }
    Eigen::VectorXd get_cost_to_parent() const { return
        cost_to_parent; }
    Eigen::RowVectorXd get_main_agent_state() const {
        return main_agent_state; }
    void rewire_node(TreeNode *new_parent,
                     const Eigen::VectorXd&
                         cost_to_new_parent,
                     double time_to_new_parent,
                     const Eigen::MatrixXd
                         trajectory_from_new_parent,
                     const std::vector<double>& input);
    const Eigen::MatrixXd &get_trajectory_from_parent()
        const;
    YAML::Node generate_yaml() const;

private:
    TreeNode * parent;
    Eigen::RowVectorXd main_agent_state;

    Eigen::VectorXd cost_to_parent;
    double time_to_parent;
    Eigen::MatrixXd trajectory_from_parent;

    std::vector<double> input;
};

#endif
```