Learning and Intelligent Systems

Technische Universität Berlin

Enhancing Sampling-Based Kinodynamic Motion Planning with Reinforcement Learning Policies

Master Thesis

Learning and Intelligent Systems

Fakultät IV Elektrotechnik und Informatik

Technische Universität Berlin

Alexander Weingart

Supervisors:

Joaquim Ortiz-Haro

Prof. Dr. Wolfgang Hönig

Examiners:

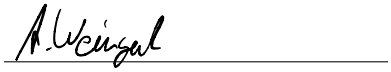
Prof. Dr. Wolfgang Hönig

Prof. Dr. Marc Toussaint

Declaration according to § 60 Abs. 8 AllgStuPO

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, August 18, 2023



Alexander Weingart

Abstract

Probabilistically complete sampling-based motion planners like Rapidly Exploring Random Trees (RRT) have been shown to be fast and effective solutions regarding feasible motion planning for a range of robotic systems. On a meta level, algorithms of this class iteratively build a graph or tree of the collision-free state-space, by randomly sampling a state and extending the nearest node towards it with collision-free trajectories. While the extension step is straightforward and fast in geometric planning, it can be challenging for complex non-holonomic kinodynamic systems, as there is often no efficient steering function available to calculate the optimal path between two states. In the following thesis, we approached this problem by training a control policy with Reinforcement Learning (RL) based on Proximal Policy Optimization and a custom Curriculum Learning approach. The trained model is subsequently used as a local planner for the commonly used sampling-based motion planner RRT, which is then integrated into the asymptotically optimal motion planning framework *AO-x*. We evaluated the resulting system using two car-like robotic systems (one velocity- and the other acceleration-controlled) in two different scenarios and compared their performance against two Monte-Carlo-propagation based extension strategies in a detailed analysis. Our experiments show, that the RL-integrated system finds its initial solution significantly faster than both alternatives, while the motion plans it produces have a noticeably lower cost.

Zusammenfassung / Abstract

Probabilistisch vollständige Sampling-Based Motion Planning (SBMP) Ansätze wie Rapidly Exploring Random Trees (RRT) haben sich als schnelle und effektive Lösung erwiesen für die Planung von realisierbaren und kollisionsfreien Bewegungen für eine Anzahl verschiedener Robotiksysteme. Algorithmen dieser Klasse bauen iterativ eine Graph- oder Baumrepresentation von kollisionsfreien Bewegungen im Zustandsraum, durch die zufällige Abtastung von Zuständen und die anschließende Erweiterung des nächstliegenden Knoten in die Richtung dieser Zustände. Für geometrische Planungsprobleme ist der Erweiterungsschritt einfach und effizient. Für komplexere nichtholonomische kinodynamische Systeme gibt es jedoch oft keine effiziente Lösung, um den optimalen Pfad zwischen zwei Zuständen zu berechnen. In der vorliegenden Arbeit evaluieren wir den Ansatz, diesem Problem durch das Training von Steueragenten mittels Reinforcement Learning zu begegnen. Das Training der Agenten erfolgte basierend auf Proximal Policy Optimization und einem speziell konzipierten Curriculum Learning Aufbau. Die trainierten Agenten wurden als lokale Planer in den Erweiterungsschritt des häufig verwendeten SBMP-Algorithmus RRT eingebaut, welcher wiederum in das asymptotisch optimale Bewegungsplanungsframework AO-x eingebaut wurde. Das resultierende Gesamtsystem wurde mit zwei verschiedenen Auto-ähnlichen Robotiksystemen evaluiert und mit zwei alternativen Monte-Carlo Erweiterungsschrittansätzen in zwei unterschiedlichen Szenarien verglichen. In unseren Ergebnissen fand das System mit RL-Integration seine initiale Lösung wesentlich schneller als beide Alternativen und seine produzierten Bewegungspläne hatten erkennbar niedrigere Kosten.

Contents

1 Introduction

In this chapter we introduce the problem motivating this thesis and give a first brief overview over our approach.

1.1 Motivation

In recent years, robotic systems are increasingly being embedded into many facets of both work and private life. Autonomous vacuum cleaners are populating a sizeable number of both homes and offices, prototype-state self-driving busses begin to support the public transport infrastructure[1] and both production floors and warehouses are becoming co-operated by autonomous robot platforms. The revenue of the global warehouse robotics market alone is predicted to hit 50 billion US$ in 2030 [1]. A significant challenge in enabling complex robots to successfully accomplish tasks safely and efficiently is to calculate collision-free plans achieving motion through cluttered real-world environments. Solving this problem has inspired a large amount of scientific publications and is the topic of a research field called *Motion Planning*.

Approaches in this field need to confront several complications when aiming for real-world deployment. Many robotic systems feature a high number of degrees of freedom. Their corresponding state spaces are often high-dimensional and mostly continuous, which makes them difficult targets for conventional optimal search strategies. The environments the systems are deployed in can be populated with diverse dynamic obstacles featuring complex non-convex shapes, whose explicit representation can quickly become infeasible. On top of that, the motion plans need to be found reasonably fast in many scenarios, while still satisfying various secondary targets like smoothness or energy-expenditure.

One popular approach that emerged to confront these challenges is called *Sampling-Based Motion Planning* (SBMP). Utilizing high-performant collision-checking modules, planners of this class build trees or graphs in the robot's state space by sampling collision-free states and connecting them with collision-free trajectories. In doing so, they iteratively built a representation of the connected obstacle-free space until a trajectory between initial and target state is found. While these techniques are reasonably straightforward in the domain of holonomic motion (where the optimal path between

[1] https://www.weforum.org/agenda/2023/01/autonomous-buses-geneva-project/

two points is just a straight line), they become more complex for non-holonomic systems, as a lot of these kinodynamical systems lack an efficient steering function.

Another research area that saw major advancements in the recent decade is *Reinforcement Learning* (RL). This branch of Machine Learning builds on the intuitive idea of learning from a reward indicator generated by interaction with the environment, in other words: learning by *Trial-and-Error*.

Research into various RL-methods has enabled interesting application for decades. In 1994, Tesauro et al. at IBM's Thomas J. Watson Research Center presented to the world a self-teaching program called TD-Gammon [2], which achieved master-level play in the game of Backgammon. In more recent years, the combination with successful techniques developed in the domain of supervised learning led to many new breakthroughs. In 2013, Mnih et al. [3] trained an autonomous agent to play Atari Games on a Superhuman Level solely based on the video output of the console(-emulation). Combining ideas from search, modern supervised learning and RL, Silver et al. [4] built a system called *AlphaGo*, which was able to reach master-level play in the game of Go, long considered the most challenging of classical games for artificial intelligence. In a now famous exhibition match, *AlphaGo* won against one of the world's best Go players, Lee Sedol, with a final score of 4-1 [5].

While many of the early success stories in RL dealt with discrete action- and state-spaces, subsequent research explored ways to extend the ideas into the continuous domain. This enabled many successful designs targeting diverse dynamical systems including among many others quadrupedal or humanoid robots.

In the following thesis we built on these advancements to approach the problem of steering for kinodynamic sampling-based motion planning.

1.2 Contribution

In this section we give a short overview over the contributions made by this thesis.

Using a state-of-the-art Reinforcement Learning algorithm, we successfully trained near-optimal control policies for two different car-like kinodynamic systems, exploring different reward function designs as well as a custom Curriculum Learning based approach. By reviewing the training results in detail, we provide a case-study regarding the design of a setup which could be useful for training control policies for a variety of dynamics.

The policies are subsequently integrated into the extension step of a widely used sampling-based planner, RRT, and integrated into the asymptotically optimal sampling-

based motion planing framework AO-x. This way, we were able to evaluate not only the effect the RL-integration has on the exploration behavior of RRT, but also how this impacts the performance of the overall asymptotically optimal planning setup. To investigate the change in performance, we compare the resulting system with two alternatives proposed by the research community and provide a detailed analysis of the results.

In Chapter 2 we introduce the relevant theoretical foundation our work is based on. After detailing our approach regarding the training of the policies, as well as the integration into the sampling-based planner in Chapters 3 and 4, we present the results of our evaluation in Chapter 5. Chapter 6 includes a contextualizing discussion as well as our thoughts regarding possible future work. In Chapter 7 we summarize our conclusion.

2 Background

The two main research areas our approach builds upon are *Motion Planning* and *Reinforcement Learning*. In the following sections we briefly introduce the relevant concepts and ideas from both.

2.1 Motion Planning

Planning describes the effort towards *"finding a sequence of valid states (i.e. a path) between specified positions (i.e. start and goal) in a search space"* [6]. The specific problem settings in which planning is performed vary enormously, depending on the domain and the solution criteria. The focus of the following sections and the thesis in general, is a subdomain of the planning problem, called *motion planning*, whose primary distinction to other variations is its focus on planning in continuous state spaces [7]. In this thesis, we use the following broad specification of motion planning (based on the formulation proposed by [8]):

Let $\mathbf{x} \in \mathcal{X}$ be the state of a robotic system, meaning its configuration and, depending on the specific control space of the setup, its derivatives. Additionally, let $\mathcal{X}_{free}$ describe the free space, i.e. the space of states, in which the system does not collide with any obstacles and does not violate any dynamical constraints.

Given an initial state $\mathbf{x}(0) = \mathbf{x}_{start}$ and a desired final state $\mathbf{x}(T) = \mathbf{x}_{goal}$, the objective in a motion planning problem is to find a time T and a set of controls $\mathbf{u} : [0, T] \rightarrow U$ such that the resulting motion satisfies $\mathbf{x}(T) = \mathbf{x}_{goal}$ and $\mathbf{x}(t) \in \mathcal{X}_{free}$ for all $t \in [0, T]$.

The motion of a system is defined by the differential equation

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) \tag{2.1}$$

where $\dot{\mathbf{x}}$ stands for the first-order derivative of the state with respect to time and f is encoding the system's dynamics.

A planner is said to be *complete* if it finds a solution to the motion planning problem if one exists and returns failure otherwise.

This version of the problem is already quite challenging for a lot of robotic systems. However, a significant portion of the motion planning literature aims to tackle an even

more difficult problem. In *optimal* motion planning, the objective is not only to find a feasible motion (if it exists), but to find the motion that *optimizes* specific criteria, like path-length, time-to-reach-target or actuator-effort.

In cases where the state space is discrete and can be modeled using graphs, a suitable approach is to rely on complete and optimal search algorithms, like **A*** [9]. When designing solutions for robotic motion planning problems, this becomes rather challenging however, because, as previously noted, state spaces are most often continuously valued. In theory, one could aim to approximate the continuous state space with a discrete representation, but depending on the chosen resolution, this comes with high performance costs and the resulting solutions will "only" be "resolution complete" and "resolution optimal". In general, selecting a suitable a priori discretization can be very challenging. If it is designed too sparse, a suitable solution could very well be impossible to find. If it is designed too dense, the computational effort required to discretize and search the space becomes too expensive for most real-world deployments.

Another common approach is to construct a parameterized representation of the path or controls and convert the motion planning problem into an *optimization problem*, where the objective is to solve for the parameters that minimize a specific cost function (while also satisfying the various system constraints) [8].

Optimization based techniques for kinodynamic motion planning have been studied extensively by the research community. In cases where the target system is differentially flat (like quadrotors [10] or even cars with a variable number of trailers [11]), optimization techniques otherwise used in the geometric planning domain, like spline optimization, can be almost directly applied to kinodynamic motion planning as well.

Another popular approach for solving the motion planning problem for non-convex dynamics is called Sequential Convex Programming (SCP). The main idea of SCP is to linearize the non-convex dynamics around an initial guess, constructing a convex problem based on the linearized dynamics and solving it with any convex optimizer. The results of the optimizer are subsequently used as the basis for the next iteration. This process is repeated until the motion plan fulfills all constraints of the original problem [12]. The two most relevant approaches in the class of SCP algorithms are currently successive convexification (SCvx) [13] and guaranteed sequential trajectory optimization (GuSTO) [14].

A popular alternative to SCP, called KOMO ([15]), defines the problem as a nonlinear program (NLP) instead, using the discretized configurations as the decision variables.

Derivatives are included via implicit Euler integration. Using the assumption that the cost and constraints only depend on the last k configurations (with $k = 2$ for acceleration constraints for example) nonlinear optimization can be done efficiently.

While trajectory optimization methods similar to the ones mentioned are able to produce near optimal solutions, they typically do not scale well with increasing time horizons (and a rising number of decision variables). On top of this, the quality and even feasibility of the solution can depend heavily on the initial trajectory ([16]).

As a response to the challenges of previous methods, a new class of approaches called *Sampling-Based Motion Planning* (SBMP) emerged. SBMP interleaves approximation and search, while trading weaker formal guarantees for significantly better performance.

2.1.1 Sampling Based Motion Planning

SBMP, also referred to simply as "sampling methods", builds on the idea of searching and probing the configuration / state-space using a sampling scheme and a collision-detection module [7]. In a general sense, most approaches consist of the following components [8]:

- **State Sampling**: Function (random or deterministic) selecting samples from the state space.
- **Extension-Target Selection**: Function deciding which previous free-space sample (node) to extend. This is mainly done via nearest-neighbor selection based on a specified distance (pseudo-)metric.
- **(Simple) Local Planner**: Tries to connect and move toward the selected newly sampled state.

The two major sampling method classes, differ in the data-structure that is built using these components. Probabilistic Roadmaps (PRMs) comprise methods that represent the feasible motions in state-space with a roadmap graph. This means investing upfront time into constructing the graph (*learning phase*), but subsequent queries can build on the same graph (if the environment is static). Rapidly Exploring Random Trees (RRTs) on the other hand build a tree instead and focus on providing speedy results for single-query scenarios (like the highly dynamic environments of many robotic system deployments) without the need for exact connection between states. In the following section we will introduce the basic algorithm for building RRTs and discuss different design choices and extensions.

2.1.2 Rapidly Exploring Random Trees

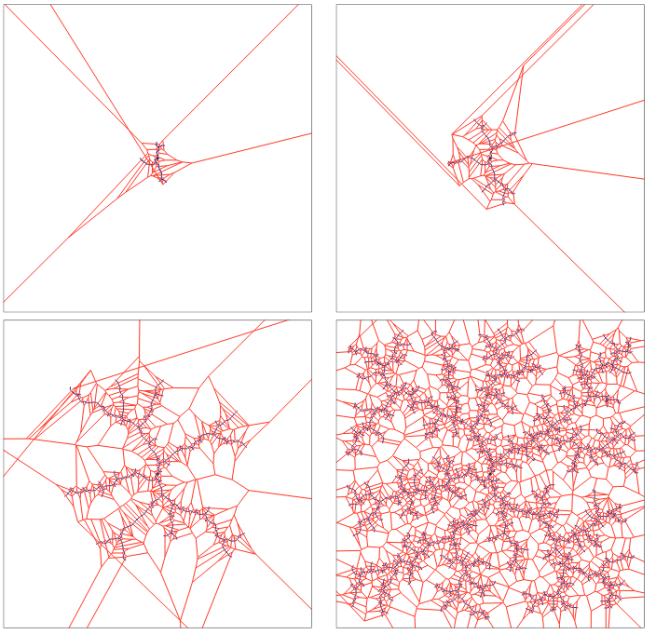


Figure 2.1: Visualization of the exploration behavior of RRT in the state space [17].

The concept of Rapidly Exploring Random Trees (RRT) was initially introduced by Steven M. LaValle et al. in 1998 [18], to address motion planning for non-holonomic kinodynamical systems. These mostly lack efficient local planners, able to exactly connect two states, which PRMs generally demand.

Their initial approach (see algorithm 1) constructed a tree $\mathcal{T}$ rooted in the initial state $\mathbf{x}_{init}$ by sampling a random state $\mathbf{x}_{rand}$ (line 3), selecting its nearest-neighbor in $\mathcal{T}$, $\mathbf{x}_{near}$ (line 4), and choosing the control input that when applied minimizes the distance between $\mathbf{x}_{rand}$ and $\mathbf{x}_{near}$ (line 5). As part of this step, the collision-detection module is queried to make sure that the state of the system stays in $\mathcal{X}_{free}$. Selecting this minimizing control is achieved by using an analytical function, if available. If there is no efficient analytical solution for the robotic system in question, the selection is based on forward simulation. If $\mathcal{U}$ is finite and relatively small, all controls can be simulated, and the best combination can be chosen. If $\mathcal{U}$ is continuous or disqualifyingly big, the best control input is often approximated by choosing the best from a set of sampled inputs [19]. The chosen control input is then applied for a given timestep Δt to obtain the new state $\mathbf{x}_{new}$ (line 6), which is added as a new vertex to $\mathcal{T}$ and connected to $\mathbf{x}_{near}$ with a new edge containing the applied control vector $\mathbf{u}$.

Algorithm 1 GENERATE_RRT($\mathbf{x}_{init}, K, \Delta t$)

$\mathcal{T}$.init($\mathbf{x}_{init}$)
for $k \leftarrow 1, K$ **do**
    $\mathbf{x}_{rand} \leftarrow$ RANDOM_STATE()
    $\mathbf{x}_{near} \leftarrow$ NEAREST_NEIGHBOR($\mathbf{x}_{rand}, \mathcal{T}$)
    $\mathbf{u} \leftarrow$ SELECT_INPUT($\mathbf{x}_{rand}, \mathbf{x}_{near}$)
    $\mathbf{x}_{new} \leftarrow$ NEW_STATE($\mathbf{x}_{near}, \mathbf{u}, \Delta t$)
    $\mathcal{T}$.add_vertex($\mathbf{x}_{new}$)
    $\mathcal{T}$.add_edge($\mathbf{x}_{near}, \mathbf{x}_{new}, u$)
end for
return $\mathcal{T}$

In their introduction to sampling methods, Lynch and Park [8] describe the basic RRT algorithm with a formulation that lends itself better to incorporating the various extensions proposed for RRT since its first introduction. In the following, we will use a version of the RRT algorithm, which is close to their description (as well as the version used in [19]), mostly exchanging their verbose descriptions for more concise function names:

Algorithm 2 FIND_MOTION_WITH_RRT

$T \leftarrow (E, V)$ with $E \leftarrow \emptyset$ and $V \leftarrow \{\mathbf{x}_{start}\}$
while $|V| < N_{E_{max}}$ **do**
    $\mathbf{x}_{samp} \leftarrow$ SAMPLE_STATE($\mathcal{C}_{free}$)
    $\mathbf{x}_{nearest} \leftarrow$ NEAREST_NODE($\mathcal{T}, \mathbf{x}_{samp}$)
    $\mathbf{u}(t), \mathbf{x}(t), \Delta t \leftarrow$ LOCAL_PLANNER($\mathbf{x}_{nearest}, \mathbf{x}_{samp}$)
    if COLLISION_FREE($\mathbf{x}(t)$) **then**
        $V \leftarrow V \cup \{\mathbf{x}(\Delta t)\}$
        $E \leftarrow E \cup \{(\mathbf{x}_{nearest}, \mathbf{x}(\Delta t))\}$, annotated with $\mathbf{u}(t)$
        if $\mathbf{x}_{new} \in \mathcal{C}_{goal}$ **then**
            return SUCCESS, CONSTRUCT_MOTION($\mathbf{x}(\Delta t)$)
        end if
    end if
end while

In this version, SELECT_INPUT and NEW_STATE are combined into LOCAL_PLANNER, to encompass RRT extensions, where a single edge can be composed of a set of different controls and forward propagation is interleaved with the control selection. Also, the COLLISION_FREE primitive is now explicitly stated.

In the following, we will briefly describe the challenges and subsequent design decisions which arise for each of the primitives.

2.1.2.1 SAMPLE_STATE

Sampling the state is an important task in any SMBP setup as the choices made in the implementation of this step can heavily influence the growth of the tree $\mathcal{T}$. One of the intuitive and most often used approaches is to sample from a uniform distribution over the state space $\mathcal{X}$. Often, the samples are drawn from an approximation of $\mathcal{X}_{free}$, via *rejection sampling* (Algorithm 3). In relatively uncluttered environments, this is a fast and easy-to-implement solution. In spaces where many obstacles are present however, other solutions might be necessary to achieve satisfactory results. Alternative proposed approaches to sampling can be based on motion primitives [20], potential functions [21] or more sophisticated approximations of the free state space [22]. State sampling does not even have to be random, some authors also explored deterministic solutions [23].

Algorithm 3 REJECTION_SAMPLING

1: **for** $i \leftarrow 0; i <$ MAX_ATTEMPTS$; i \leftarrow i + 1$ **do**
2: $\quad \mathbf{x}_{samp} \leftarrow$ SAMPLE_UNIFORM$(\mathcal{X})$
3: $\quad$ **if** COLLISION_FREE$(\mathbf{x}_{samp})$ **then**
4: $\quad\quad$ **return** $\mathbf{x}_{samp}$
5: $\quad$ **end if**
6: **end for**
7: **return** FAILURE

2.1.2.2 NEAREST_NODE

This primitive is the second part of the selection of an extension target in the tree. Selecting a nearest node, often called *nearest-neighbor*, hinges on the definition of distance in the particular state space. While this might be trivial in some cases (like geometric planning for a holonomic kinematic robotic system), it presents a difficult challenge for many others, as stating the real distance between two states of a complex kinodynamic system requires an (efficient) optimal steering solution which is mostly unavailable. In these cases, authors often deploy a weighted euclidean distance metric, where the weights are part of the configuration of the planer and need to be tuned based on heuristics.

Another practical challenge of the nearest-neighbor search is the runtime performance of the query, which heavily depends on the used data structure. While the naive brute force list comparison approach can work for small trees, its theoretical running time is $O(N)$, which heavily impacts the performance of the overall system, especially considering that the impact of the distance calculation itself is not yet factored in, whose time complexity is heavily dependent on the dimensionality of the state space and has to be done for every node in the tree. One of the most common ways to deal with this is

the usage of a data-structure called KD-tree ([24]), which extends the ideas of balanced binary search trees to multidimensional data and offers significantly improved runtimes for nearest-neighbor queries in large trees. For queries of K nearest-neighbors with d-dimensional entries, the time complexity with KD-Trees is $O(N^{1-\frac{1}{d}} + K)$ [25].

2.1.2.3 COLLISION_FREE

The goal of the COLLISION_FREE method is to return for any given trajectory (defined by the initial state $\mathbf{x}_{nearest}$ and the applied control $\mathbf{u}$) an answer stating if it is free of collision, i.e. fully part of $\mathcal{X}_{free}$. In some implementations, checking for collisions is done as part of the local planning step, for example by checking at each timestep of the forward propagation.

Even though many sophisticated techniques emerged over the years, and highly optimized collision checkers are packaged within many robotic frameworks, this operation can still be expensive computational wise for environments with a large amount of (complex) non-convex obstacles. Surveying the research in this domain could itself fill the entire thesis, but as our focus is on the LOCAL_PLANNER extension, covered in the next subsection, we will just direct any interested reader towards the introduction into collision checking theory in [26].

2.1.2.4 LOCAL_PLANNER

The local planner step, applied in Line 5 of Algorithm 2 is used to find a motion extending $\mathcal{T}$ from $\mathbf{x}_{nearest}$ towards $\mathbf{x}_{samp}$. For kinematic systems, this might be as trivial as returning a straight line. For kinodynamic systems with various dynamical constraints however, it is far less trivial and motivates a vast amount of research including this very thesis.

The initial approaches of the early RRT papers were based on *random propagation*, sometimes termed *Monte-Carlo-Propagation (MCP)* [27], of the controls. Algorithm 4 shows a one-shot approach. First, the random control vector $\mathbf{u} \in \mathcal{U}$ and a timestep $\Delta t \in (0, \Delta t_{max}]$ are sampled. By subsequently applying $\mathbf{u}$ for Δt, the system is then propagated forward (Line 4). The control vector, the sampled time and the final state of the trajectory are returned. Implementations differ regarding the selection of Δt, which is sometimes sampled and sometimes set to a constant.

Instead of propagating only ones, the system dynamics could be simulated multiple times with different random controls and timestep-lengths, before selecting the control input which brings the system's state closest to $\mathbf{x}_{goal}$. This version of the Local Planer is

shown in Algorithm 5. The number of sampled controls K becomes a part of the implementation parameters, which needs to be tuned according to the planning environment.

When using the guided version of MCP coupled with the fixed timestep, one has to be aware of the resulting limitations however. In a recent publication [19], researchers Kurz and Stilman proved that the claim of the initial RRT paper, that the described algorithm (selecting the best control input and a fixed timestep) was "probabilistically complete under very general conditions" [18], was not accurate. The authors describe a kinodynamic system for which this original approach would never be able to find a feasible motion plan, even though one exists. While the dynamics of the system they used for their proof were quite specific and unusual, selecting the timestep-length randomly avoids these issues and is therefore recommended. In later work, LaValle et al. abandoned their first version of the local planning step.

Algorithm 4 MCP($\mathbf{x}_{init}, \mathbf{x}_{goal}$)

```
1: u ← SAMPLE_CONTROL(U)
2: Δt ← SAMPLE_TIMESTEP(Δt_max)
3: u(t), x(t) ← PROPAGATE_FORWARD(u, Δt)
4: return u(t), x(t), Δt
```

Algorithm 5 MCP-Guided($\mathbf{x}_{init}, \mathbf{x}_{goal}, K$)

```
1: u_res(t) ← ∅
2: x_res(t) ← ∅
3: Δt_res ← ∅
4: d_min ← inf
5: for i ← 0; i < K; i ← i + 1 do
6:     u(t), x(t), Δt ← MCP(x_init, x_goal)
7:     d ← DISTANCE(x(Δt), x_goal)
8:     if d < d_min then
9:         u_res(t) ← u(t)
10:        x_res(t) ← x(t)
11:        Δt_res ← Δt
12:        d_min ← d
13:    end if
14: end for
15: return u_res(t), x_res(t), Δt_res
```

2.1.3 Asymptotically Optimal Planning: AO-x

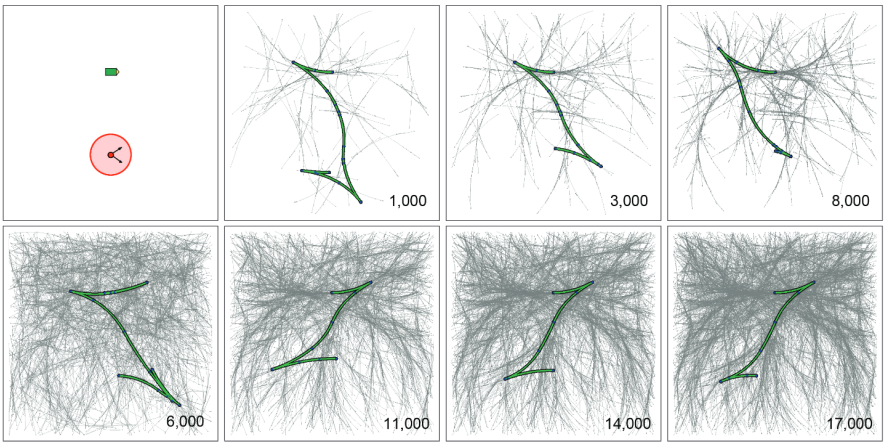


Figure 2.2: Snapshots from an AO-RRT planning attempt for a sideways maneuver of the Dubin's car [28]

In their paper "Asymptotically Optimal Planning by Feasible Kinodynamic Planning in State-Cost Space"[28], published in 2016, Kris Hauser et al proposed a meta-algorithm approach for asymptotically optimal motion planning. The approach is based on the theorem, that "any optimal planning problem can be transformed into a series of feasible planning problems in a state-cost space whose solutions approach the optimum" [28]. If provided with a *well-behaved* Planner $\mathcal{A}$, the probability that the resulting path $\mathbf{y}$ of the meta-planner *AO-x* is suboptimal is approaching zero, as the number of iterations n approaches infinity. The definition of *well-behaved* is fairly nonrestrictive:

1. If there exists a feasible solution and $\bar{c} > C^*$, then $\mathcal{A}$ terminates in finite time, and
2. Given a cost bound $\bar{c}$, the expected suboptimality of the computed path is shrunk toward C^* by a non-negligible amount each iteration

Here, $\bar{c}$ describes the current cost bound of the iteration and $C*$ the optimal cost. The proposed algorithm (Algorithm 11), operates in the state-cost space. The state $\mathbf{x}$, which the base planner initially uses, is supplemented with the cost-to-come c, resulting in the new state $z = (\mathbf{x}, c)$. This also applies for all states in the goal-region (transforming it into a cylinder with infinite extent in the cost direction). In the beginning, the base planner $\mathcal{A}$ is tasked to find a feasible solution to the problem with an infinite cost boundary $\mathcal{P}_{\text{inf}}$. If no solution is found, this is where the execution terminates with a failure. Otherwise, the cost of the resulting path is used as the cost boundary for the next iteration. This loop continues, until the maximum number of iterations is reached, at which

point the latest computed path y_n is returned. In most deployments of the algorithm, the iteration bound is exchanged for a bound on the execution time, for obvious reasons.

Algorithm 6 Asymptotically-optimal($\mathcal{P}, \mathcal{A}, n$)

1: Run $\mathcal{A}(\mathcal{P}_{\text{inf}})$ to obtain a first path y_0. If no solution exists, report "$\mathcal{P}$ has no solution".
2: Let $c_0 = C(y_0)$.
3: **for** $i = 1, 2, \ldots, n$ **do**
4: Run $\mathcal{A}(\mathcal{P}_{c_{i-1}})$ to obtain a new solution y_i.
5: Let $c_i = C(y_i)$.
6: **end for**
7: **return** y_n

2.2 Reinforcement Learning

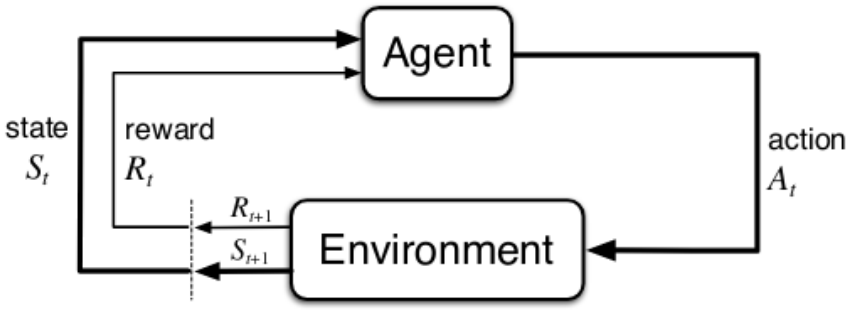


Figure 2.3: Typical depiction of the agent interaction model in a Markov Decision Process [29]

Next to *supervised* and *unsupervised* learning, *Reinforcement Learning (RL)* is currently one of the three major research fields in Machine Learning. On the foundational level, RL could be described as a computational approach following one of the fundamental ideas of most theories about learning and intelligence: "Learning from interaction" [29].

In the following we will briefly introduce the concepts from the Reinforcement Learning domain, which are necessary for the understanding of our approach. For a comprehensive introduction into the topic, we recommend "Reinforcement Learning: an Introduction" by Sutton and Barto [29], which is also used as the basis for the following sections.

2.2.1 Markov Decision Processes

To enable precise theoretical statements, the problem space in which RL approaches operate is often formulated as a Markov Decision Process (MDP), which is visualized in Figure 2.3. The basic setup of the MDP consists of a learning and decision-making entity called *agent*, and an *environment*, which is essentially everything outside the agent. At each discrete timestep $t = 0, 1, \ldots, T$ the agent

- receives a representation of the environment's *state* (*observation*) $S_t \in \mathcal{S}$
- chooses an *action* $A_t \in \mathcal{A}(s)$ based on this representation

As a result of the chosen action and according to the dynamics specified by a (possibly probabilistic) *transition function*, the environment's state changes from S_t to S_{t+1}, which is again fed to the agent, in addition to a *reward* signal $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$.

The agent's objective in the MDP, is to maximize the expected *return* G_t defined as:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.2}$$

To extend this objective to cases where the MDP is non-episodic, G_t can be defined as a *discounted return*

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

In this formulation $\gamma \in [0, 1]$ is called the *discount rate* and governs how valuable future rewards are for the agent.

While interacting with the environment, the agent's behavior is typically defined by a mapping of states to action-probabilities, called *policy* $\pi(a|s)$, which encodes the probability that the agent will choose action a if in state s. For any specific policy, the so called *state-value-function* $v_\pi(s)$ returns the expected reward the agent receives, when starting from s and acting according to π.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \text{ for all s} \in \mathcal{S} \tag{2.4}$$

The policy's *action-value function* q_π on the other hand, returns the expected return of the agent, if it starts in state s, chooses action a and afterwards follows the policy.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.5}$$

An *optimal policy*, often written as π_* is defined as a policy whose expected return is greater or equal to that of any other policy. In contrast to the optimal state- and action-value functions, v_* and q_* which are unique, many different equally optimal policies can exist [29].

Reinforcement Learning methods differ in what they are trying to learn. So called *value-based methods*, like Sarsa[30] or Q-Learning [31], are built on iteratively refining value-function-estimations. The estimations are updated based on the encountered states and the received rewards, by various different proposed algorithms. In the case of Sarsa for example, the update rule is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.6}$$

Where Q is an estimate for the optimal action-value function q_*. This update is performed, following each transition from a non-terminal state S_t to S_{t+1}. $Q(S_{t+1}, A_{t+1})$ is regarded as having the value zero, if S_{t+1} is terminal. The parameter α is usually called the *step-size* and governs, the size of the updates.

The value-function-estimations are in turn used as the basis of the policy, for example by using an ϵ-greedy selection method, i.e.choosing the action with the highest estimated action-value $Q(s, a)$, most of the time, but sometimes selecting randomly (to guarantee exploration). The pseudocode in Algorithm 7 (based on [29]) shows the general form of the Sarsa control algorithm as an example.

Algorithm 7 Sarsa

parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Init. $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
for i in $1 \ldots N_{episodes}$ **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    while S not terminal **do**
        Take action A, observe R, S'
        Choose A' from S' based on the policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$;$A \leftarrow A'$
    end while
end for

Another important class of RL-approaches are so-called *policy gradient methods*. Instead of using value-function-estimates for the guidance of the agent's actions, they learn a *parameterized policy* instead, which can select actions directly [29]. These policy-based methods, are interesting for robotic-applications, as they offer practical ways of handling large or even continuous action spaces.

2.2.2 Policy Gradient Methods

Policy Gradient Methods learn a *parameterized policy* π_θ defined as

$$\pi_\theta(a|s) = P(A_t = a|S_t = s, \theta_t = \theta) \tag{2.7}$$

where θ_t represents the policy's parameters at timestep t. The parametrization of the policy can be implemented in many ways, but $\pi_\theta(a|s)$ needs to be differentiable with respect to θ. Another constraint emerging in practice is, that the policy should never become fully deterministic, as to ensure at least a small amount of exploration.

The parameter vector θ is learned "based on the gradient of some scalar performance measure $J(\theta)$ "[29]. As the target is to *maximize* the performance, the parameters are updated in each round by approximating gradient *ascent* in J [29]:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_\mathbf{t})} \tag{2.8}$$

with $\widehat{\nabla J(\theta_\mathbf{t})} \in \mathbb{R}^{d'}$ specifying the stochastic estimate of the gradient of J (with respect to θ_t). The gradient estimator is also referenced as $\hat{g}$ and one of the most commonly used versions has the form [32]:

$$\hat{g} = \hat{\mathbb{E}}[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.9}$$

In this equation, the expectation $\hat{\mathbb{E}}[\dots]$ usually stands for the empirical average over a finite batch of samples, collected by an algorithm alternating between optimization and sampling [32]. $\hat{A}_t$ stands for an approximation of the advantage function at timestep t. When using automatic differentiation software, a common way to obtain $\hat{g}$ is to differentiate the objective

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.10}$$

A problem with this objective definition is however, that performing multiple steps of optimization on the same trajectory, empirically leads to contra-productively large policy updates [32]. This motivates the search for different objective functions.

2.2.3 Proximal Policy Optimization

One of the most impactful new approaches of the recent years in regards to Policy Gradient Methods was proposed as "Proximal Policy Optimization Algorithms" (PPO). [32] The central innovation with PPO is a new objective function, called "Clipped Surrogate Objective":

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\text{min}(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.11}$$

The min term is composed of two components. The first is based on the objective L^{CPI} proposed in [33]

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t] \tag{2.12}$$

where $r_t(\theta)$ denotes the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The second term is a modification of this surrogate objective. By clipping $r_t(\theta)$ between $(1-\epsilon)$ and $(1+\epsilon)$ it removes any incentive for r_t to move outside of the interval $[1-\epsilon, 1+\epsilon]$. By taking

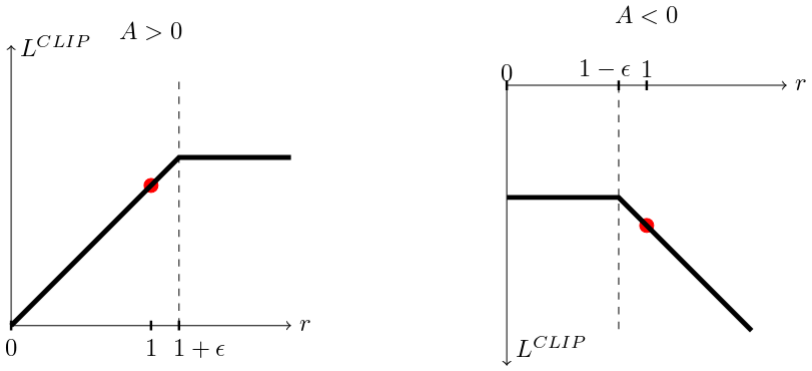


Figure 2.4: One timestep of PPO's surrogate function L^{CLIP} [32]

the minimum of the two terms (clipped and unclipped objective), L^{CLIP} basically represents a lower bound on L^{CPI}. Figure 2.4 shows the result of the objective for a single timestep as a function of the probability ratio. If the advantage function is bigger than 0, large probability ratios are quasi ignored.

In their paper, the authors also introduce an alogrithm based on L^{CLIP} in an *actor-critic*[34] framework :

Algorithm 8 PPO, Actor-Critic Style

for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, N$ **do**
    Run policy π_{old} in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

In each iteration, the old policy (acting on the basis of the neural network based policy termed "actor") is deployed in the environment for a fixed number of timesteps. After this, the estimates of the advantages of each timestep are calculated (based on the, again, neural-network based value-function approximator termed "critic") and stored. At the end of N of these runs, the parameter vector is updated via optimizing the objective.

Even though the implementation of PPO is relatively straightforward, "requiring only few lines of code changes to a vanilla policy gradient implementation" [32], evaluation based on a variety of tasks (including robotics and Atari-games) show, that it has similar stability and reliability to previous approaches, but better overall performance.

2.3 Combining Reinforcement Learning and Sampling-Based Motion Planning

The combination of RL and SBMP has emerged as a promising research direction. Over the years, RL based enhancements have been proposed for many of the primitive operations underlying most SBMPs, including *sample generation*, *distance computation*, *collision-detection* and *steering*. In their recent survey [35] McMahon et al. provide an extensive overview over the various ways sampling-based planners have been extended with machine learning approaches, including many with RL. In the following we will focus on the ones targeting the *steering* operation, as they are the closest to our approach.

2.3.1 Integrating a RL trained Controller into PRM

In their 2018 publication [36], Faust et al. propose a hierarchical long-range navigation method based on the integration of a RL-trained close-range point-to-point steering policy into the Probabilistic Road Map planner framework (PRM). To evaluate their system, they trained policies for two robotic systems, a velocity-controlled differential drive platform (with observations incorporating noisy lidar scans), as well as an acceleration-vector-controlled aerial cargo delivery drone, both with continuous state- and action-spaces. Training was achieved via DDPG [37] in the case of the former and CAFV [38] in the case of the latter system.

To integrate the results of the local planner policies into PRM, they proposed a technique of nearly-connecting edges. If the distance of the final state of a trajectory is below a predefined threshold, a connection attempt query to the local planner is deemed successful. To be able to estimate to probability, that these connections are actually executable, the authors query the local planner multiple times for each edge (with states sampled close to the start and goal-states). The edge's distance is then set to the average of all queries, and it will only be added, if the connection-success rate is above a set threshold.

2.3.2 Integrating a RL trained Controller into RRT

In 2019, Chiang et al. [39] published their approach towards integrating a RL-based steering policy into SBMP, but instead of PRM, they combine RL with RRT. Their proposed system is based on two components. For the first one, they train an agent to steer a simulated version of the targeted robotic system through a cluttered map. Like Faust et al., they use simulated (noisy) lidar scans as the observation-input for their

system in addition to the robots configuration and velocities (in the case of second order systems). For the training, they utilized the parameterized proxy-reward

$$R_{\theta_{r_{DD}}} = \theta^T [r_{goal}\ r_{goalDist}\ r_{collision}\ r_{clearance}\ r_{speed}\ r_{step}\ r_{disp}] \tag{2.13}$$

where

- r_{goal} signals if the goal is reached (1 if reached, 0 else)
- $r_{goalDist}$ is the negative Euclidean distance to the target
- $r_{collision}$ penalizes collisions (-1 if the agent collides, 0 otherwise)
- $r_{clearance}$ rewards the distance to the closest obstacle
- r_{speed} is set based on the agent's speed
- r_{step} is a constant step penalty (value 1)
- and r_{disp} is the sum of displacements between the current position and the previous positions 3, 6 and 9 steps earlier.

The parameter vector θ is optimized via a special meta-training-framework called *AutoRL* [40].

After the agent is evaluated and found to achieve satisfying results on the steering task, it is employed in the target map, while its time-to-reach (TTR) is monitored. This data-generation period yields a set of (state, future-cost) tuples, which is fed to a supervised learning system to train the second component, a reachability estimator, informing the nearest-neighbor selection as well as the sample generation.

Based on the steering agent and reachability estimator, they subsequently build their RRT based system, using the steering agent for the tree expansion and a hierarchical combination of euclidean distance and the TTR estimation of the reachability estimator as the nearest-neighbor selector.

2.3.3 Integrating a RL trained Controller into DIRT

Similar to the previous two approaches, Sivaramakrishnan et al. [41] also used RL techniques to train a controller to reach a goal set. Instead of incorporating simulated sensor data however, they trained the agent in an obstacle free environment and used a

significantly less complex reward function, returning 0 if the current state is in the goal region and -1 otherwise.

$$r(x_t, x_G) = \begin{cases} 0, & if\ x_t \in \mathcal{X}_G \\ -1, & else \end{cases} \tag{2.14}$$

After evaluating the performance of a selection of different Deep Reinforcement Learning Algorithms, they settled for the best-performing, Soft-Actor-Critic (SAC), and combined it with Hindsight Experience Replay to train the controller for their three different robotic system targets (first- and second order differential drive vehicle and one Segway system in a more complex physical simulation). The trained controller is used as a local planner in the asymptotically optimal planner DIRT [42].

2.3.4 Positioning the approach of this thesis

After introducing the proposals made by other researchers in the space of integrating RL into SBMP, it seems fitting to briefly note how the approach of our work fits into the existing landscape.

While our design follows the approach of Chiang et al. [39] regarding the utilized base planner our problem domain and solution design diverge in a few ways. On top of implementation details, like the deployed RL-algorithm or the targeted robotic systems, the major differences are:

1. **Observation Space**: While Chiang et al. use obstacle-aware control policies, trained based on simulated sensor input, we chose to follow a different route in this regard and train our policies in *obstacle free environments*. While this increases the likelihood, that the agent returns infeasible trajectories, the significantly smaller observation space reduces the complexity of both the training and the integration.

2. **Distance Estimation**: Chiang et al. use rollouts of their trained policies to train a distance estimator with supervised learning, which is subsequently integrated into both the nearest-neighbor search and the sampling step, observing mixed results. We instead rely on a faster and less complex, but also less accurate *weighted euclidean distance approximation*.

3. **Reward Function Design**: Chiang et al. use a dense function design, which they call a "proxy-reward", for their training setup, to combat the problem of the

inherently sparse reward-signal of the training environment. We instead successfully train our agents with a Curriculum Learning approach and a reward function which is closer to the actual distance between the initial and the target state of a task.

Regarding the above points, our approach is much closer to the proposal made by Sivaramakrishnan et al. Another similarity with their design is, that we also investigate the integration into an asymptotically optimal motion planning framework. While we focus on the integration into *AO-x*[28] however, they extend *DIRT*[42] and explore variations of the goal-sampling step.

3 Training Control Policies with RL

As the goal of this thesis is to evaluate the integration of a Reinforcement Learning based local planner policy into SBMP, we need to design suitable dynamic systems and environments as well as a training setup for them. In the following we will introduce these important components of our evaluation approach.

3.1 Robot Systems and Environments

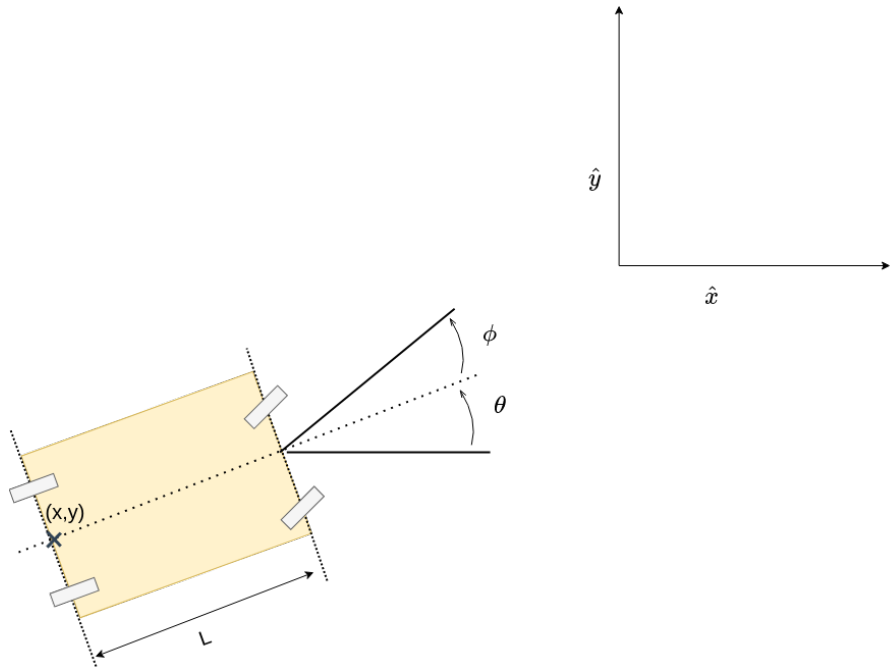


Figure 3.1: Visualization of the car-like system's geometry.

For our evaluation, we chose a wheeled mobile robotic system with car-like dynamics. To introduce different levels of complexity, the first system is controlled via linear velocity and steering angle, where the second is controlled via linear acceleration and the first-order derivative of the steering angle. Figure 3.1 shows the geometry of our setup. In the following we will introduce the dynamical equations for both the first- and second-order system.

3.1.1 First-Order Car

$$\mathbf{u} = (v, \phi) \in \mathcal{U} \subset \mathbb{R}^2 \tag{3.1}$$

$$\mathbf{x} = (x, y, \theta) \in \mathcal{X} \subset \mathbb{R}^2 \times SO(2) \tag{3.2}$$

The state of the first order car is defined by the position of the center of the rear axis in cartesian coordinates (x, y), as well as its orientation θ. To keep the dimensionality of the state space low, both position and orientation are given relative to the current target. The system is controlled via two inputs, the linear velocity v and the steering angle ϕ. Its dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ are defined by

$$\dot{x} = v \cdot \cos(\theta) \tag{3.3}$$

$$\dot{y} = v \cdot \sin(\theta) \tag{3.4}$$

$$\dot{\theta} = \frac{v}{L} \cdot \tan(\theta) \tag{3.5}$$

where L describes the car's length.

3.1.2 Second Order Car

The second order car is controlled via linear acceleration a and the change-rate of the steering angle ω. This means that the state space needs to include the linear velocity and the steering angle.

$$\mathbf{u} = (a, \omega) \in \mathcal{U} \subset \mathbb{R}^2 \tag{3.6}$$

$$\mathbf{x} = (x, y, \theta, v, \phi) \in \mathcal{X} \subset \mathbb{R}^2 \times SO(2) \times \mathbb{R}^2 \tag{3.7}$$

The expanded dynamic equations are consequently

$$\dot{x} = v \cdot \cos(\theta) \tag{3.8}$$

$$\dot{y} = v \cdot \sin(\theta) \tag{3.9}$$

$$\dot{\theta} = \frac{v}{L} \cdot \tan(\theta) \tag{3.10}$$

$$\dot{v} = a \tag{3.11}$$

$$\dot{\phi} = \omega \tag{3.12}$$

3.2 Environment

A major component of building any RL based approach is the careful design of the environment in which the agent is to be trained. Based on the general framework of the Markov Decision Process (see Chapter 2.2.1), the main components of the RL environment are:

- **observation space**: What is the nature of the observations the agent receives from the environment?
- **action space**: How can the agent interact with its environment?
- **transition/step function**: In which way does the state of the environment transition from one to the next, based on the agent's control signal?
- **reward function**: Which system state is rewarded, and by how much?

In the following, we will briefly describe the decisions we made regarding these primitives.

3.2.1 Observation and Action Space

In our system design, the agent is trained in a *directly observable environment*. This means, that the observation space and action space directly follow from the definition of the system's dynamics, outlined above. The dynamic boundaries are set via configuration parameters. In the case of the first order system, these are the speed limits v_{min} and v_{max} as well as the minimum and maximum steering angle, ϕ_{min} and ϕ_{max}. For the second order system the parameter set is supplemented by the limits on the acceleration, a_{min} and a_{max}, as well as the limits on the change of the steering angle, $\dot{\phi}_{min}$ and $\dot{\phi}_{max}$. As some algorithms can be sensitive in this regard, both the observations and the actions are normalized to the range $[-1, 1]$.

3.2.2 Transition Function

The transition function, also often called step function in the RL domain, is responsible for calculating the next state of the system, based on the last state and the agent's control input. The environments we designed for the training of our control policies are deterministic. To efficiently approximate the system's dynamics after each timestep of length Δt, we utilize Euler's integration. The resulting transitions are

$$x_{t+1} = v_t \cdot \cos(\theta_t) \cdot \Delta t \tag{3.13}$$
$$y_{t+1} = v_t \cdot \sin(\theta_t) \cdot \Delta t \tag{3.14}$$
$$\theta_{t+1} = \frac{v_t}{L} \cdot \tan(\theta_t) \cdot \Delta t \tag{3.15}$$

for the first order car and

$$x_{t+1} = v_t \cdot \cos(\theta_t) \cdot \Delta t \tag{3.16}$$
$$y_{t+1} = v_t \cdot \sin(\theta_t) \cdot \Delta t \tag{3.17}$$
$$\theta_{t+1} = \frac{v_t}{L} \cdot \tan(\theta_t) \cdot \Delta t \tag{3.18}$$
$$v_{t+1} = v_t + a_t \cdot \Delta t \tag{3.19}$$
$$\phi_{t+1} = \phi_t + \omega_t \cdot \Delta t \tag{3.20}$$

for the second order car.

In the setup we designed, the velocity and steering angle are bounded by the environment parameters v_{min}, v_{max} and ϕ_{min}, ϕ_{max} respectively and enforced by the environment through clipping, meaning when v and ϕ hit their limit, applying additional acceleration or velocity through the control input does not change the state-variables.

3.2.3 Reward Function

After encoding the dynamics of the environment, via observation, action and the transition function, another major component of the RL training setup is the design of a suitable reward function. When deciding on the specific formulation for the environment, the researcher needs to carefully weigh how much prior information is to be encoded, as doing so might bias the exploration of the agent towards suboptimal solutions. For the training of the agents we designed two different reward functions. One is sparse, meaning that the agent receives relatively few reward signals (in our case only when the goal is actually achieved). The other is dense, supplying the agent with constant feedback regarding its advancement towards the goal.

3.2.3.1 Sparse Reward Function

We initially decided on a minimalistic function design, solely based on the performance indicators we want to optimize: reaching the goal configuration in the least amount of steps. The resulting reward function $r_{t+1}(\mathbf{x}_t)$ is:

$$r_{t+1}(\mathbf{x}_t) = \begin{cases} 1, & if\ \mathbf{x}_t \in \mathcal{X}_G \\ -\frac{1}{N_{max}}, & else \end{cases} \tag{3.21}$$

In this formulation, N_{max} describes the maximum number of steps, before the environment is terminated. We normalize the step penalty, to keep the cumulative reward the agent can achieve in an episode in the range $[-1, 1]$, regardless of the specific maximum episode length setting.

The advantage of a comparatively minimal reward function like this is that it is easily applicable to all kinds of environments, as long as the dynamical constraints are enforced by the environment and the objective is to find the fastest trajectory. On top of that, it does not include any prior information on how to best achieve the objective, which could bias it towards suboptimal local minima.

The drawback is however, that the reward is sparse and for complex dynamics the agent might take a lot of time before reaching the goal for the first time. This is why we also designed a dense reward function, mostly with the training of the second-order-car environment in mind, which we introduce in the following subsection.

3.2.3.2 Dense Reward Function

The main goal in designing the alternative reward function was to increase the amount of reward signal, the agent gets in the early stages of training. Instead of only rewarding the accomplishment of the objective (i.e. reaching the target region), we want to also signal to the agent when it is improving its state towards the goal. For this we used a function we call *normalized-distance-reward* roughly similar to the one utilized in [39]:

$$d'_{\alpha_{t+1}} = \alpha_0 \cdot d_{n_t} + \alpha_1 \cdot d_{o_{n_t}} + \alpha_2 \cdot d_{v_{n_t}} + \alpha_3 \cdot d_{\phi_{n_t}} \tag{3.22}$$

$$r_{\alpha_{t+1}} = \begin{cases} \frac{N_{max}-n}{N_{max}}, & if\ \mathbf{x}_t \in \mathcal{X}_{goal} \\ (1 - \frac{d'_{\alpha_{t+1}}}{\alpha_0+\alpha_1+\alpha_2+\alpha_3}) \cdot \frac{1}{N_{max}}, & else \end{cases} \tag{3.23}$$

While the current state $\mathbf{x}_t$ is not in the goal region $\mathcal{X}_{goal}$, the parameterized reward $r_{\alpha_{t+1}}$, awarded in timestep $t+1$ is based on the weighted sum of the normalized distances between the current state and the goal state, with d_n denoting the normalized positional distance and d_{o_n} the normalized angular distance. The normalized absolute difference between current and target linear velocity and steering angle are noted as d_{v_n} and d_{ϕ_n} respectively. Because all components of the weighted sum are normalized between 0 and 1, its maximum is the sum of the parameters. The first part of the second term is therefore the normalized weighted distance substracted from one and multiplied with the maximum step reward. If the agent reaches the goal region, it receives the maximum possible amount for all remaining steps and the episode is terminated. This is calculated by $\frac{N_{max}-n}{N_{max}}$, where n describe the current number of steps. This way, the cumulative reward for each episode is in the range $[0, 1]$.

3.3 Curriculum Learning

While modifying the reward function proposed in Subsection 3.2.3.1 had the intention of providing a reward signal even though the chances of reaching the goal are initially low, another way to approach the same issue is to make the success of the early rollouts more likely. This can be achieved by using comparatively easier samples first, before iteratively increasing the difficulty.

This approach resembles ideas from the research field of Curriculum Learning, in which instead of directly learning the target task, the agent is presented with a sequence of incrementally more challenging tasks (the curriculum), before finally facing the target environment. While it is not consistently defined in the literature what constitutes a curriculum, the intuitive and in fact most common form is as an "ordering of tasks" [43].

In the case of learning our control policies, the general task is to generate a motion transferring the system from any initial state to a goal state. The difficulty of this task however can vary tremendously depending on initial and goal state. As we want to train the agent with a curriculum we now need to find combinations of initial and goal state with increasing difficulty. While it is certainly possible to craft some easy scenarios by hand, it gets increasingly tedious for more complex dynamics, like in the case of the second order car. We also might risk introducing a contra-productive bias into the training samples this way.

To combat these issue and to automate the process, we designed a way to generate samples with a bounded difficulty.

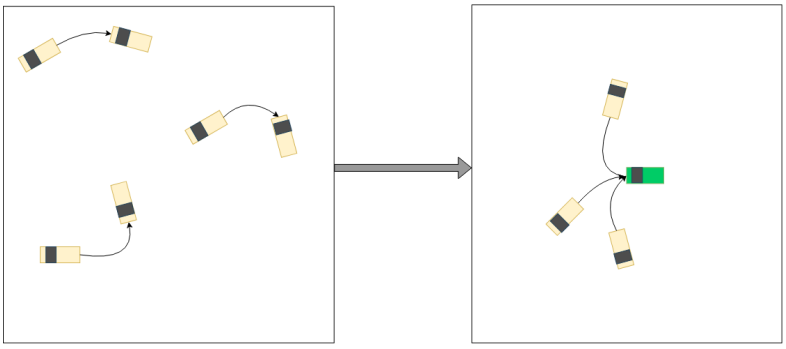


Figure 3.2: Setup to generate start positions with bounded maximum cost

Generating easy task configurations, for the robot systems we train, means selecting initial states in which even a RL-agent in the early stages of training can be expected to find a solution in at least a few rollouts (generating the first usable reward signal). There are a few different approaches, one could think of regarding this task. An intuitive one might be to generate random samples and feed them into an optimization based motion planning system to generate a near-optimal solution which could be used to specify the task configuration's complexity. While this is certainly doable for the dynamical systems we target, it requires additional computational time and effort.

Another less computationally intensive approach turns the problem on its head utilizing the translational and rotational invariances inherent in the dynamics of our target systems. Given an initial state $\mathbf{x}_t$ and a control vector $\mathbf{u}_t$ which causes the system to transition into the next state $\mathbf{x}_{t+1}$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + f(\mathbf{x}_t, \mathbf{u}_t) \cdot \Delta t \tag{3.24}$$

we can apply both translational and rotational transformations on the states, without invalidating the dynamical equation (as long as we apply the same transformations to both). If we would for example rotate $\mathbf{x}_t$ by 90 degrees, move it 1 unit along the y-axis and apply the initial control vector $\mathbf{u}_t$ to this transformed state $\mathbf{x'}_t$, the resulting next state $\mathbf{x'}_{t+1}$ would also be rotated 90 degrees and moved 1 unit along the y-axis compared to $\mathbf{x}_{t+1}$.

$$\mathbf{x'}_{t+1} = \mathbf{x'}_t + f(\mathbf{x'}_t, \mathbf{u}_t) \cdot \Delta t \tag{3.25}$$

This holds for both the first-order and the second-order car, where the additional state variables v and ϕ would not be touched by the transformations.

We are using this property for our approach. Instead of trying to measure the difficulty of specific configurations, one can also generate random trajectories with a fixed length,

by applying random controls for N timesteps. The start and end state of the resulting trajectory $\mathbf{x}_{start}$ and $\mathbf{x}_{end}$ are now known to have a solution with a maximum length of N steps.

We subsequently transform the initial state of the trajectory from its frame W to the frame F of the final pose of the trajectory. The homogeneous transformation matrix for the transformation $T_F^{\mathbf{p}}$ can be derived by multiplying the inverse of the transformation from W to F, ${T_W^F}^{-1}$, and the homogeneous matrix representation of the robot pose in world frame $T_W^{\mathbf{p}}$:

$$T_F^{\mathbf{p}} = T_F^W \cdot T_W^{\mathbf{p}} = {T_W^F}^{-1} \cdot T_W^{\mathbf{p}} \tag{3.26}$$

$$= \begin{bmatrix} \cos(\theta - \theta_f) & -\sin(\theta - \theta_f) & \cos(\theta_f)(x - x_f) + \sin(\theta_f)(y - y_f) \\ \sin(\theta - \theta_f) & \cos(\theta - \theta_f) & \sin(\theta_f)(x_f - x) + \cos(\theta_f)(y - y_f) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.27}$$

This shows, that the transformed trajectory pose $\mathbf{p}' = (x', y', \theta')$ can be calculated based on the final pose $\mathbf{p}_f = (x_f, y_f, \theta_f)$ and the untransformed pose $\mathbf{p} = (x, y, \theta)$ with

$$\theta' = \theta - \theta_f \tag{3.28}$$

$$x' = \cos(\theta_f)(x - x_f) + \sin(\theta_f)(y - y_f) \tag{3.29}$$

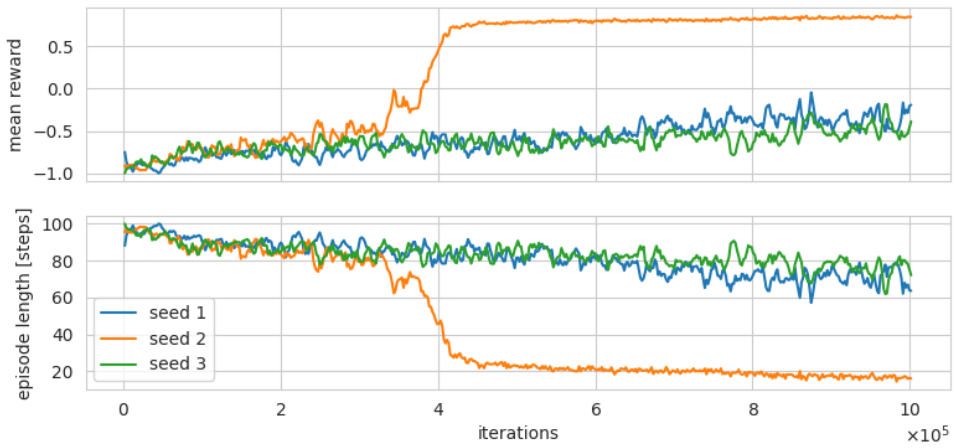
$$y' = \sin(\theta_f)(x_f - x) + \cos(\theta_f)(y - y_f) \tag{3.30}$$

3.4 Evaluation

Using the *stable-baselines 3*[44] implementation of PPO and our custom-built environments, we trained agents for both the first- and the second-order car-like robotic systems using similar hand-tuned parameters (see Appendix). In the following we will describe results we achieved during the training process, separated by the respective dynamics.

3.4.1 First-Order Car

In the training for the first-order car, we purely relied on the reward function described in Subsection 3.2.3.1. Figure 3.4.1 shows the mean reward as well as the mean episode length during one million training iterations using 3 different seeds. While all three seeds show signs of improvement regarding both the reward (increasing) and the episode length (decreasing), one of the seeds drastically outperforms the others of the same batch, converging towards an episode length of 18, which is close to the optimum. These results highlight the vulnerability of the training procedure to the randomness inherent in the sampling of the states it observes during training. While the results suggest, that all of the seeds shown in the graph should eventually converge, testing multiple seeds in the early stages of the training, can drastically speed up the process.



3.4.2 Second-Order Car

The second order car environment proved to be significantly more challenging than the one for the first order car, which is not surprising, considering that the dynamics of the

system are more complex and the observation space needs to include 4 more dimensions:

$$\mathbf{x}_t = (x_t, y_t, \theta_t, v_t, \phi_t, v_{target}, \phi_{target}) \tag{3.31}$$

When reusing the same parameters and reward function as described in the above section for the first-order car, none of the seeds show any noticeable improvement, after one million steps (see Figure 3.6). As looking for a set of parameters that significantly improved this performance was not successful, we turned to other ideas. Our first attempt was to investigate different reward functions, which motivated the design of the *normalized-distance-reward* already described in Subsection 3.2.3.2.

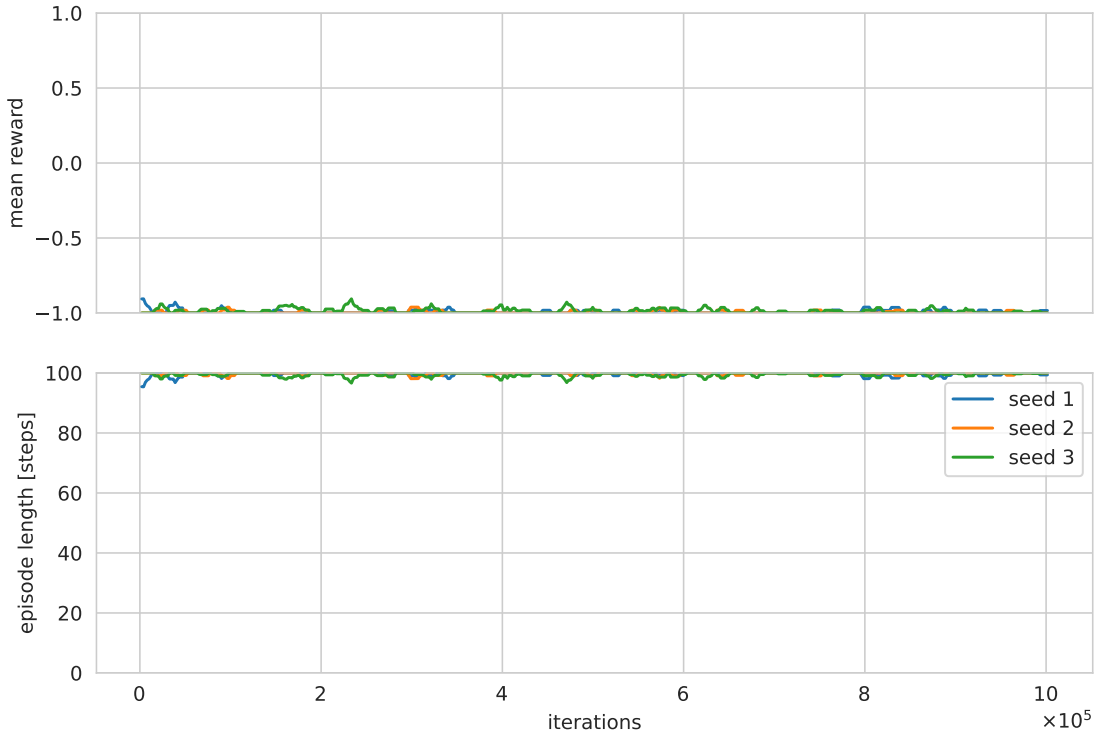


Figure 3.3: Results for 3 different seeds of PPO with similar parameters as in Subsection 3.4.1 and the reward function *step-penalty-target-reward*

3.4.2.1 Using a Dense Reward Function

Figure 3.4 shows the results for 3 different seeds, in the second-order-car environment, using the described dense reward function, *normalized-distance-reward*.

While the mean reward rises in the initial phase of the training, it quickly converges and stays around 0.7. The episode length meanwhile barely changes and hovers near

the maximum. As the episode is terminated early, in case the agent reaches its goal, the graph is clearly showing, that no useful control policy was learned.

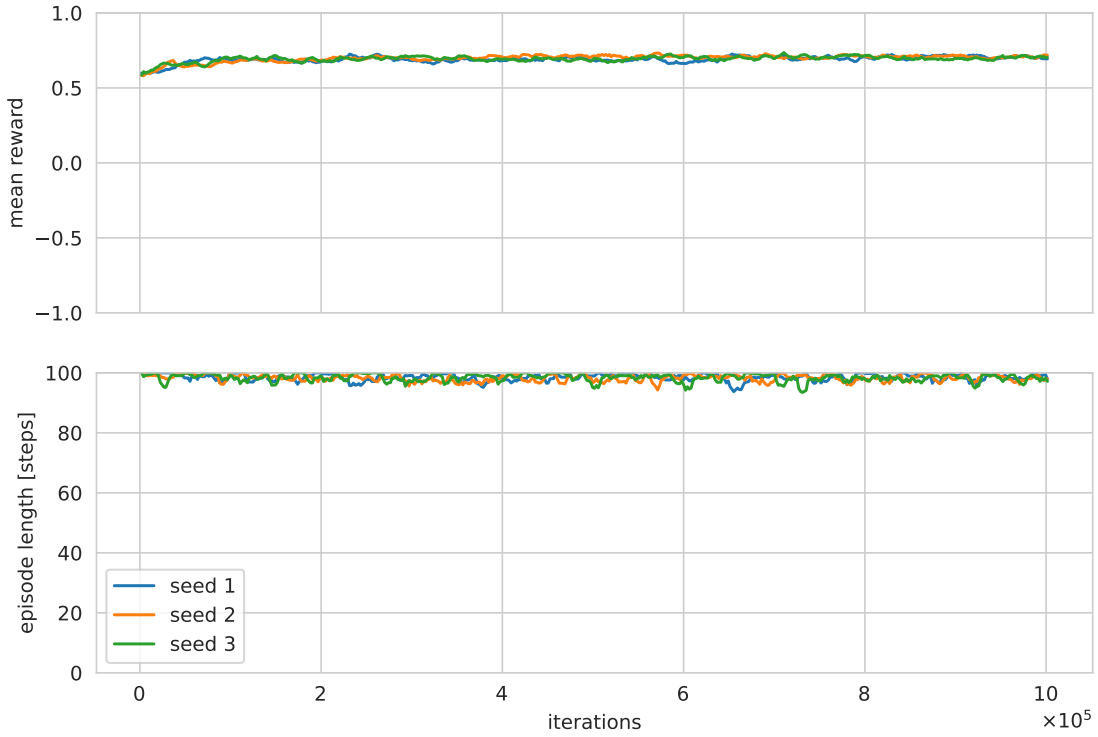


Figure 3.4: Results for 3 different seeds and PPO with similar parameters as in 3.4.1 and the reward function *normalized-distance-reward*

While it might be the case that the performance could be increased with the perfect set of parameters, found in a time-intensive tuning process, we instead successfully deployed the Curriculum Learning strategy described in 3.3 to further the training efforts, which we detail in the next section.

3.4.2.2 Curriculum Learning

Following the approach introduced in Subsection 3.3, we generated 1000 starting configurations with maximum cost of 20 steps and introduced them gradually into the pool of starting configurations used for training. Figure 3.5 shows the curriculum setup as well as the result of the training. $N_{samples}$ denotes the number of samples (from the generated list) introduced. For the first 2M iterations, we used a pool of 100 different samples. After that, we gradually increased the size of the sample-cohort every 500k iterations to 250, 500 and then 1000. At 3.5M steps, we opened the sampling up to random sampling. As can be seen in the graph, the performance converges to roughly 17 steps for

the pool of size 100. After increasing to pool size, the performance barely changes. When opening up the start-configuration-selection to random-sampling however, both the mean reward and the episode length take a significant hit, before converging to a new set-point of about 0.75 mean reward and 30 steps.

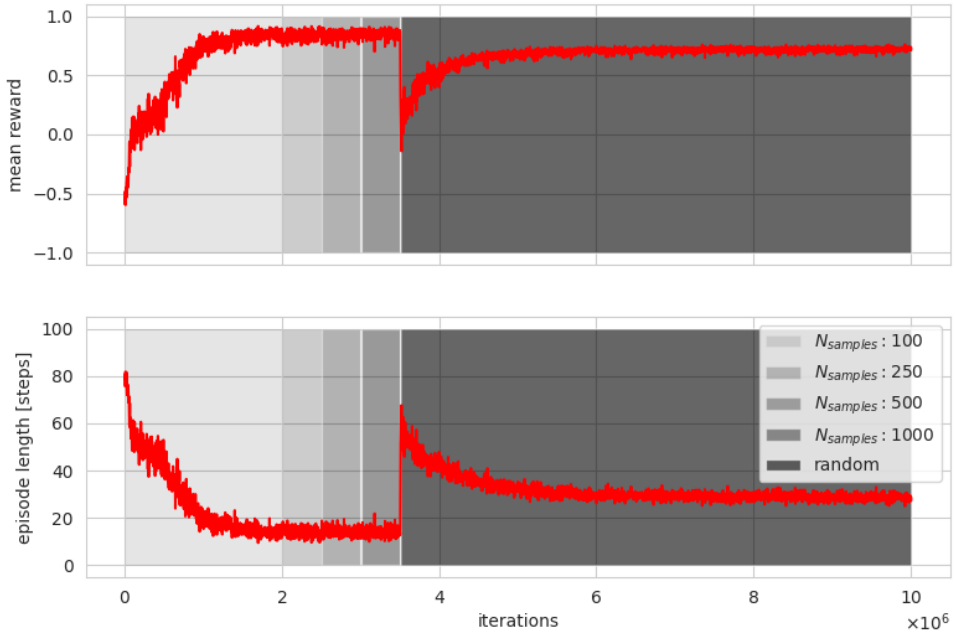


Figure 3.5: Mean reward of the RL-Policy in the second-order car environment over the course of 10M training iteration steps. The color of the background indicates the curriculum-phase.

This should be close to the optimal performance and manual inspection of the generated trajectories (see Figure 3.6 for examples), showed promising results. Because of this, we decided to use the trained model for the RRT-integration.

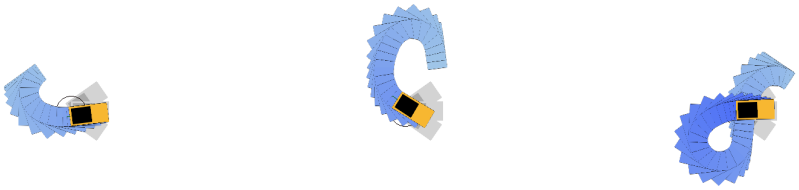


Figure 3.6: Policy rollouts for 3 different starting configurations. Earlier states have a brighter color. The size of the goal region is indicated by the circle for the position and the gray car silhouettes for the orientation.

4 Integrating RL-Based Policies into Sampling-Based Motion Planning

After successfully training the control policies, the second necessary component is to integrate them into the overall RRT-Planner Framework, as part of the `LOCAL_PLANNER` primitive. As introduced in Section 2.1.2.4, the local planner is used to extend the planner's tree from a selected node towards a sampled goal state (line 5 in Algorithm 2). It receives the system state $\mathbf{x}_{nearest}$ of the selected node, as well as the goal state $\mathbf{x}_{samp}$. Based on these (and some system parameters, like the maximum number of extension steps), it generates a list of control vectors $\mathbf{u}$, which, if applied, should advance $\mathbf{x}_{nearest}$ towards $\mathbf{x}_{samp}$.

When integrating our RL policies into this step we face the problem, that the workspace of the overall motion planner is most often significantly larger than the one the model is initially trained in. We therefore need to select an intermediate target in the general direction of the randomly selected expansion goal, but close enough to the current position to be reachable by the trained agent.

4.1 Determining an Intermediate Target

An intuitive first approach to this would be to take a step of length r from the initial position $(x_{nearest}, y_{nearest})$ towards the target (x_{rand}, y_{rand}) to reach (x', y'):

$$\Delta\mathbf{x} = \begin{bmatrix} x_{nearest} \\ y_{nearest} \end{bmatrix} - \begin{bmatrix} x_{rand} \\ y_{rand} \end{bmatrix}$$
$$\widehat{\Delta\mathbf{x}} = \frac{\Delta\mathbf{x}}{|\Delta\mathbf{x}|}$$
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + r \cdot \widehat{\Delta\mathbf{x}}$$

The relation between original distance $d = |\Delta\mathbf{x}|$ and the new distance r could then be used to scale the difference between the remaining initial- and goal-state variables:

$$
\begin{aligned}
\gamma &= \frac{r}{d} \\
\theta' &= \theta_{init} + \texttt{ANG\_DIST}(\theta_{init}, \theta_{goal}) \cdot \gamma \\
v' &= v_{init} + (v_{goal} - v_{init}) \cdot \gamma \\
\phi' &= \phi_{init} + (\phi_{goal} - \phi_{init}) \cdot \gamma
\end{aligned}
$$

In the above set of equations, ANG_DIST stands for the angular distance.

The resulting new intermediate goal $\mathbf{x}' = (x', y', \theta', v', \phi)$ could subsequently be used as the target state for a complete rollout of the RL-policy, with the goal of reaching the intermediate goal (within the defined tolerances for position and orientation). Summarizing the scaling procedure described above under the function-name $\texttt{SCALE}(\mathbf{x}_{nearest}, \mathbf{x}_{rand}, r)$, this approach is outlined in Algorithm 9.

Algorithm 9 SINGLE_INTERMEDIATE_GOAL($\mathbf{x}_{nearest}, \mathbf{x}_{rand}, r_{max}$)

1: **if** $d > r_{max}$ **then**
2: $\quad \mathbf{x}' \leftarrow \texttt{SCALE}(\mathbf{x}_{nearest}, \mathbf{x}_{rand}, r_{max})$
3: **else**
4: $\quad \mathbf{x}' \leftarrow \mathbf{x}_{rand}$
5: **end if**
6: $U,\ X \leftarrow \texttt{FULL\_POLICY\_ROLLOUT}(\mathbf{x}_{nearest}, \mathbf{x}')$
7: **return** $U,\ X$

The problem with the *single-intermediate-goal* approach is, that it empirically leads to curvy paths. While the intermediate state $\mathbf{x}'$ lies in the general direction of the goal, it is generally not on the optimal path to the target for the kinodynamic systems. Trying to fully reach both the orientation and position goal in one rollout can therefore be counterproductive. Figure 4.1 illustrates this problem.

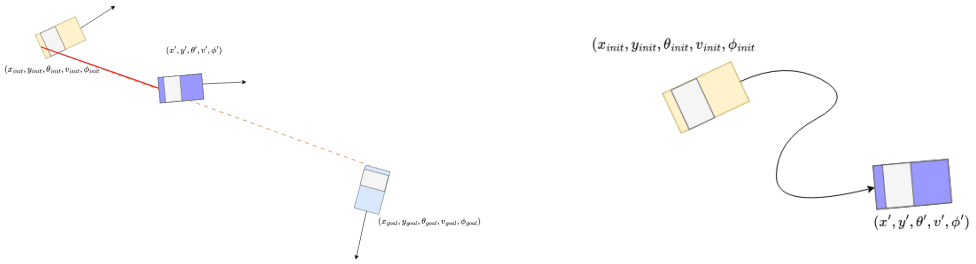


Figure 4.1: Single-intermediate-goal generation (left) and resulting trajectory (right).

The poor results for this initial extension mode, lead us to the design of an *intermediate-goal-per-step* approach. Instead of generating one intermediate goal and then trying to reach it, we use the scaling approach introduced above once for each new step, in a *sliding-window* fashion.

Algorithm 10 INTERMEDIATE_GOAL_PER_STEP($\mathbf{x}_{near}, \mathbf{x}_{rand}, r_{max}, N_{steps}$)

$i \leftarrow 0$
$\mathbf{U} \leftarrow \emptyset$
$\mathbf{X} \leftarrow \emptyset$
$\mathbf{x}_{last} \leftarrow \mathbf{x}_{near}$
while $i < N_{steps}$ **do**
    if $d > r_{max}$ **then**
        $\mathbf{x}' \leftarrow \texttt{SCALE}(\mathbf{x}_{last}, \mathbf{x}_{rand}, r_{max})$
    else
        $\mathbf{x}' \leftarrow \mathbf{x}_{rand}$
    end if
    $\mathbf{x} \leftarrow \texttt{TRANSFORM}(\mathbf{x}_{last}, \mathbf{x}'_{goal})$
    $\mathbf{U}_i, \mathbf{X}_i \leftarrow \texttt{RL\_POLICY\_STEP}(\mathbf{x}_{last})$
    $\mathbf{x} \leftarrow \texttt{TRANSFORM\_BACK}(\mathbf{x}, \mathbf{x}'_{goal})$
    $\mathbf{U} \leftarrow \mathbf{U} \cup \{\mathbf{U}_i\}$
    $\mathbf{X} \leftarrow \mathbf{X} \cup \{\mathbf{X}_i\}$
    $i \leftarrow i + 1$
    if $\mathbf{x} \in \mathcal{X}_{goal}$ **then**
        break
    end if
end while
return $U, X, i \cdot \Delta t$

The pseudocode in Algorithm 10 describes the general idea. The RL-policy is given an initial state $\mathbf{x}_{near}$, a goal state $\mathbf{x}_{rand}$, as well as the parameters extension radius r_{max} and maximum steps N_{steps}. The propagation loop runs until it reaches the goal region or the maximum of N_{max} steps, whatever happens first. Before each query of the policy, we determine an intermediate goal using (`GET_INTERMEDIATE_GOAL`) and transform the robot's pose into the intermediate goal's frame (`TRANSFORM`) (analog to the transformation described in Section 3.3). The RL-policy is subsequently called for one step (`RL_POLICY_STEP`), based on this transformed initial pose. The resulting next state is then transformed back into the initial (world-)frame and appended to the return trajectory. The first 3 steps of a possible extension attempt based on this approach are visualized in Figure 4.2. The advantage of the *intermediate-goal-per-step* approach is, that it leads to superior state-exploration, as the tree expansion is pulled towards the originally sampled x_{rand} instead of the intermediate goal.

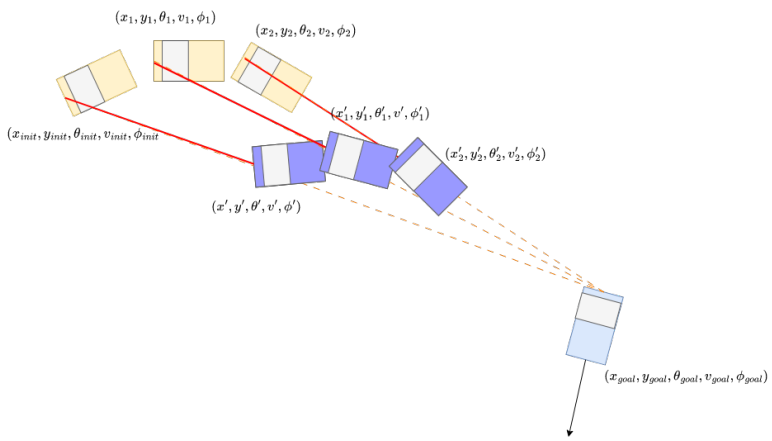


Figure 4.2: First 3 steps of the intermediate-goal-per-step approach.

4.2 Exporting the Trained Model

As our first experiments revealed that querying the control policies using the default stable-baselines3 API was taking longer than we expected, we decided to export the model to the standardized ONNX format and use the optimized ONNX-runtime [45] for python instead. Figure 4.3 shows the analysis of the querying-times of 1000 extension-steps (without the sliding-window mode) using either the ONNX-runtime or stable-baselines-3, visualized as a boxplot. The boxes span 50% and the whiskers 75% of the sample-results. As can be seen, the mean-runtime of the ONNX-version is significantly shorter and has less variance than the stable-baselines version.

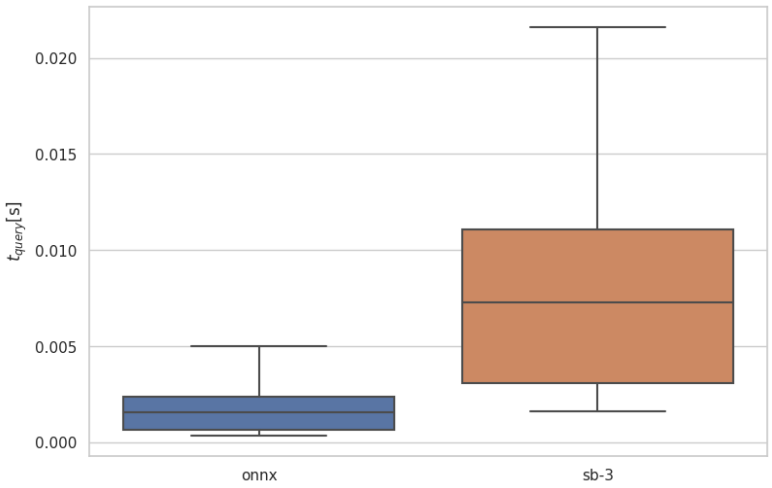


Figure 4.3: Comparing the runtime of querying the trained network, using the stable-baselines 3 interface (right), and the ONNX-runtime (left).

4.3 Integration into AO-x

As the target of the thesis is to explore the impact of the RL-policy integration on asymptotically optimal sampling-based motion planning, we integrate the RL enhanced RRT planner into the *AO-x* framework proposed by Hauser et al. [28] which we already introduced in Section 2.1.3.

As described, the integration is fairly straightforward. We mainly need to supplement the system's state space with an additional dimension - the cost-to-come c, meaning the number of steps taken from the tree root before reaching a particular state. The cost-dimension is integrated into both the state sampling and the nearest-neighbor search (with a comparatively low weight). For the sampling, we follow the approach described by Kleinbort et al. in their refined AO-x analysis [46], and sample the cost from $[0, c_{max}]$, where c_{max} is the current cost boundary of the iteration.

Initially we run the base RRT algorithm once with no cost boundary, while the max sampled cost c_{max} is set to the maximum cost currently observed in the tree. After this initial round found a feasible solution, all subsequent iterations are provided with the last attempt's cost as the boundary. The motion planning query runs until a set maximum runtime is reached, at which point the last found solution is returned. Both the initial *AO-x* paper [28] and the subsequent analysis in [46] propose pruning the tree between iterations as an enhancement. This did however not significantly improve the runtime of our planners in our preliminary test-runs, and we therefore chose to use the basic version of the algorithm instead, in which the base-planner builds a new tree for each query.

Algorithm 11 AO-RRT($\mathbf{x}_{init}, \mathbf{x}_{goal}, t_{max}$)

```
c ← ∞
U ← ∅
X ← ∅
while current-runtime < t_max do
    U, X, c ← RRT(x_init, x_goal, c)
end while
return U, X
```

4.4 Impact on the Probabilistic Completeness

When integrating the RL-planner in the way we described in the previous sections, one has to consider the drawbacks the RL-extension has regarding the probabilistic completeness of the overall system. While this theoretical attribute was proven for RRT with extension based on Monte-Carlo Propagation and a randomly sampled timestep,

this does not necessarily transfer to RRT with a RL-policy based extension step. The causes for this are twofold. First, the behavior of the RL-agent is based on the output of a neural network. Proving probabilistic completeness, would necessitate making assumptions on the general behavior of the network which we currently can not achieve in a theoretical sound way. Second, as the RL-policy is trained and deployed in an obstacle-free environment, there might be situations where, depending on the dynamical bounds of the system, the agent might nearly always return plans, which are infeasible in the global workspace. Consider for example the edge case visualized in Figure 4.4. In this map, the car needs to drive backwards to reach its goal. Given the velocity controlled car and a configuration with a relatively narrow turning radius and a significantly higher absolute velocity forward than backwards, the optimal strategy in the obstacle free environment might be to drive a forward curve instead of backwards, for most if not all possible collision free samples $\mathbf{x}_{samp} \in \mathcal{X}_{free}$.

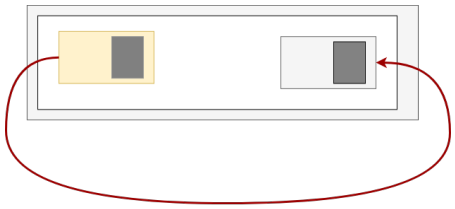


Figure 4.4: Scenario in which it would be impossible for the RL-RRT approach to find the existing solution, for specific dynamical configurations. The initial state is drawn yellow, the goal state grey.

One way for the system to still achieve probabilistic completeness however, is to introduce a (small) chance, that the extension step uses Random Propagation instead of the RL-policy. While we did not do this in the system we evaluated, it would be straight-forward to add this implementation detail in any possible future deployment. For further ideas regarding this topic, we refer to Section 6.

In the following Section we will present the approach we took for the evaluation of the integrated system, before presenting the results we observed.

5 Evaluation

In this chapter, we present the setup we used to evaluate the performance of our integrated system as well as the results we observed.

5.1 Setup

To evaluate the performance of the RL-enhanced Motion Planning System, we compare it against two alternative candidates for the extension step of RRT:

- **Monte-Carlo Propagation (MCP)**: Random (valid) controls propagated for a random amount of time ($\Delta t \in [0.1, 5]s$)
- **MCP-Guided**: Similar to MCP, but instead of only simulating once, we sample and simulate 10 different control-vector-time-tuples and select the one which comes closest to the extension target.

The resulting 3 versions of RRT (RRT-RL, RRT-MCP and RRT-MCP-Guided) were integrated into the AO-x framework according to the method described in 4.3 and queried for both robotic systems in two different maps we designed. The evaluated maps are visualized in Figure 5.1. The first, *Bugtrap*, features a slightly easier wide-opening variant of the often used bugtrap problem [28]. The second map consists of a simple maze in which the robot needs to navigate from the start-state to goal-region by following a zigzag-course. Again, a commonly used map idea, similar to the one used for the evaluation of PRM-RL [36] for example.

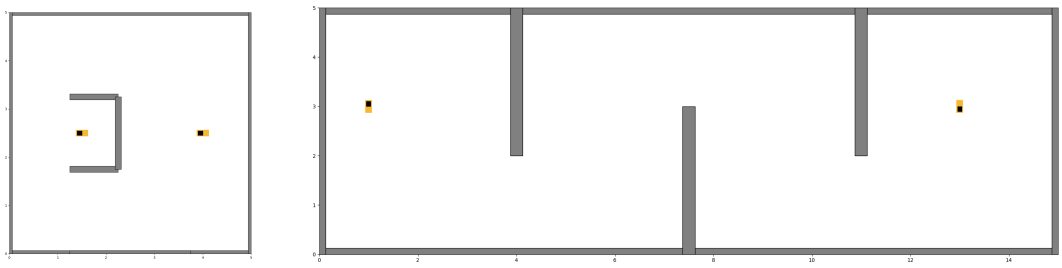


Figure 5.1: Evaluation Maps: *Bugtrap* (left) and *ZigZag* (right)

In our comparison of the different extension methods, we focus on two different metrics, the *success ratio* (the percentage of runs that found at least one solution to the motion planning problem) and the total *path-execution-time* Δt_{path}. While the former is

a measure of how fast and reliable the feasible motion planning problem is solved, the latter shows the quality of the found solution, as well as its convergence rate over time.

For the evaluation experiments, we use our own python-implementation of the AO-RRT planner. Together with the policy-training-tools, and the implementation of the overall experiment setup, it can be found on GitHub[1].

In the following we will present the result of the described evaluation setup, structured by the targeted robot system.

5.2 First-Order Car

Controlling the first order car-like system, the RL-based approach achieved 100% success-rate significantly faster than both MCP-based versions in both of the two maps (see the bottom section of Figures 5.2 and 5.3). The guided MCP version had the worst performance regarding this metric, reaching 100% roughly 50 seconds later than the one-shot MCP version in the *Bugtrap* map. In *ZigZag* the difference is even more stark, with a gap of about two minutes. The RL-enhanced system is again exhibiting superior performance in our second metric, the path-execution-time Δt_{path}, which is shown in the top section of the Figure 5.2, where the solid lines represent the median of the best (shortest) path-execution-time, of all planner queries at this point in time (set to infinity, if no solution was found yet), starting at the timestep at which at least 50% of planners have found a solution. The standard deviation is also visualized, starting at 100% success rate. As shown in the plot, the initial trajectories found by our system are noticeably shorter, then both MCP planners. Interestingly though, the performance of MCP and MCP-Guided is flipped compared to the success rate. Regarding the path-execution-time, MCP-Guided starts lower than MCP and keeps this position until the end of the evaluation runtime.

While the solution-quality improves quite a bit in the early stages, for all planners, it is rather stagnant in the later stage of the runtime. As expected (and noted by other authors [28]), the algorithm spends a large amount of its runtime for small scale improvements in the end, while achieving larger scale improvements comparatively fast in the early stages.

[1] https://github.com/alexanderweingart/rl-sbmp

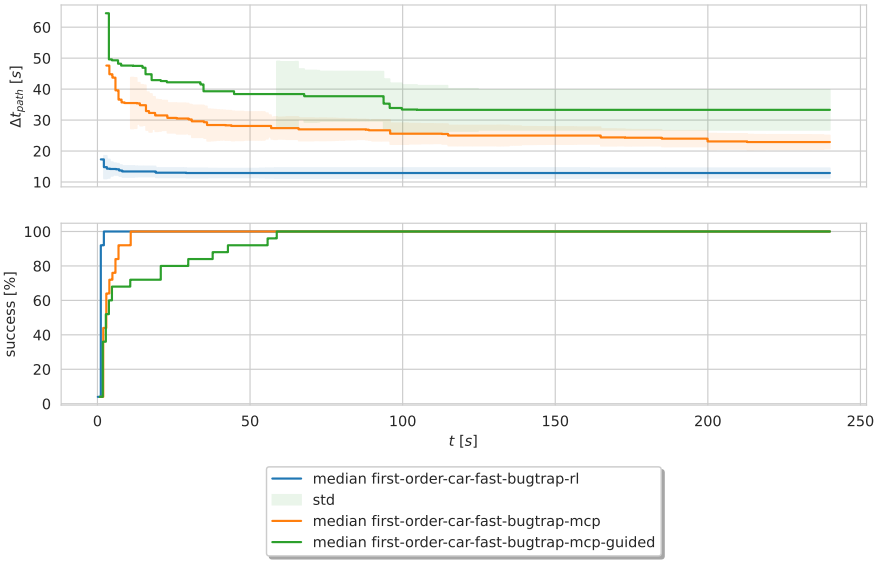


Figure 5.2: 25 runs of AO-RRT for the first-order system in the *Bugtrap* map.

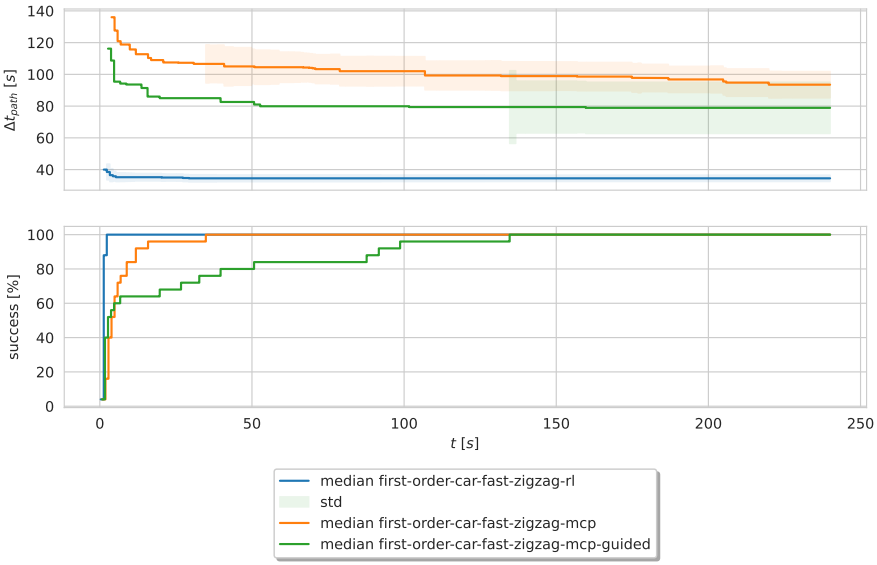


Figure 5.3: 25 runs of AO-RRT for the first-order system in the *ZigZag* map.

5.3 Second-Order Car

The second-order car-like system proved to be more difficult for all planner variants. Like for the first-order car, the RL-enhanced variant was able to get to a 100% success-rate faster, while achieving superior solution quality than both alternatives. While MCP and MCP-Guided are fairly close to each other regarding the path-execution-time in the *Bugtrap* map, MCP-Guided outperformed the unguided version in *ZigZag*. This is only true regarding the solution quality though, as again mirroring the results from the first-order system, MCP-Guided is significantly slower to reach 100% success rate in the first map. In the second, there were even some runs, where no solution was found at all, which is why the success rate never reaches 100%.

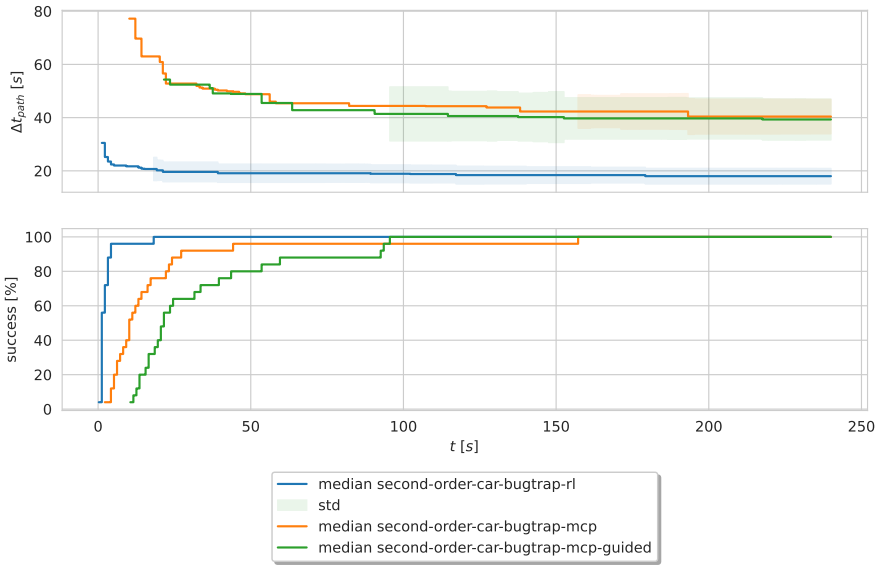


Figure 5.4: 25 runs of AO-RRT for the second-order system in the *Bugtrap* map.

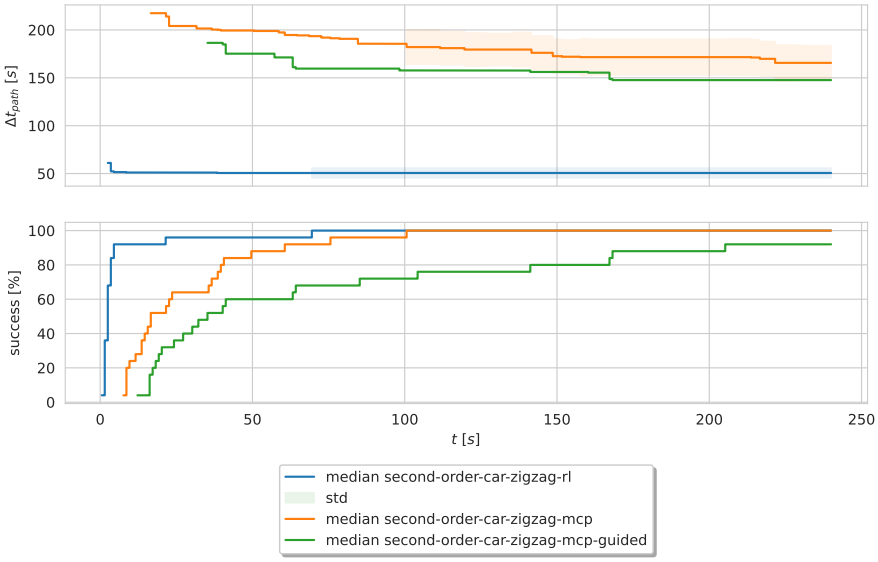


Figure 5.5: 25 runs of AO-RRT for the second-order system in the *ZigZag* map.

5.4 Examining the Trajectory Progression

To investigate the trajectories found by the compared planners, we select one set of (intermediate-)trajectories for each variant in the *ZigZag* map and the first-order car-like system. The specific run was selected based on the number of intermediate trajectories. We choose runs for each system where 6 trajectories where found as part of the solving attempt. The results can be seen in Figure 5.4. In the selected trajectory progressions, we can see that compared to MCP and MCP-Guided, the trajectories of the RL-variant feature more straight sections. Figure 5.4 visualizes the car states on the final trajectory of the runs shown in Figure 5.4. As the time-interval between states on the trajectory is fixed, the visualization enables an understanding of the velocities the car exhibits while executing the motion plan. Sections where the car is moving faster, show more of the car's black roof. In this way, it can be seen, that the motion plan produced by the RL-variant is significantly faster in most sections of plan. This also mirrors the results discussed in 5.2.

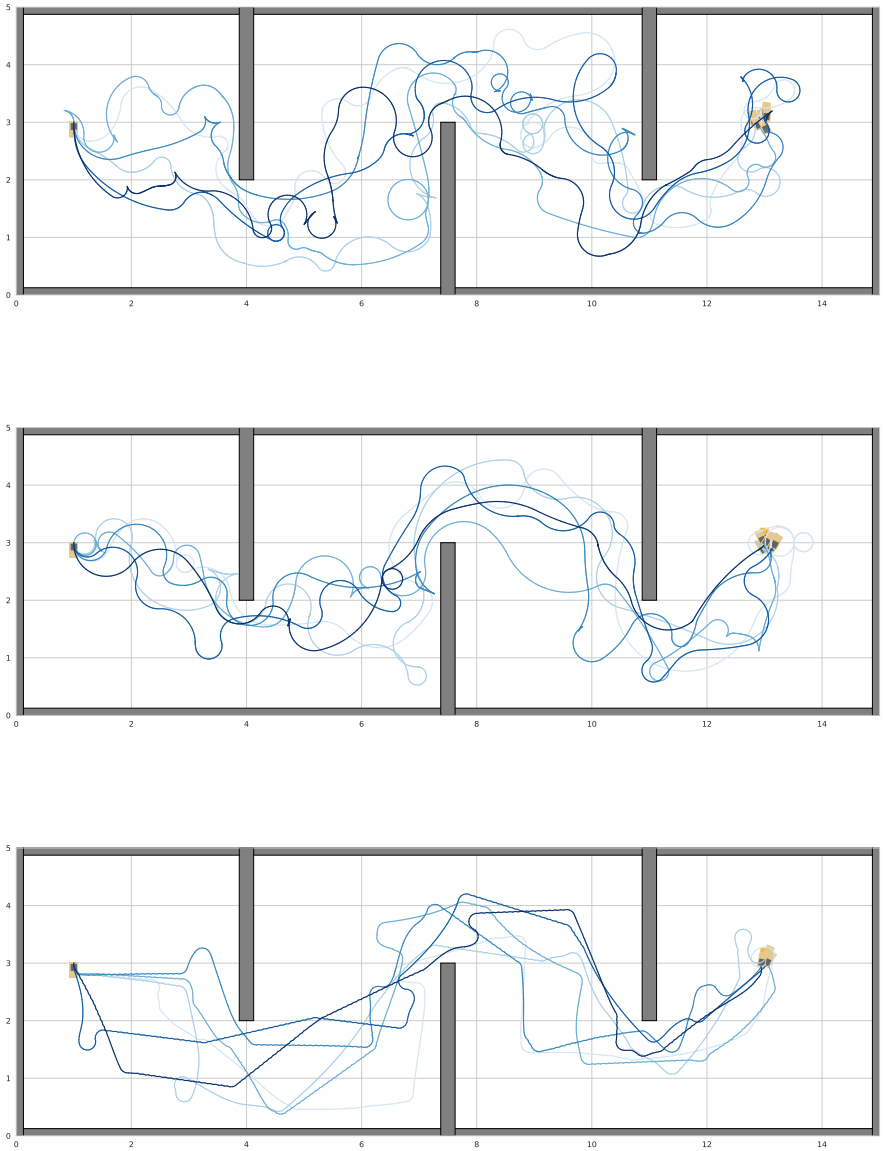


Figure 5.6: Trajectory progression in the *ZigZag* map controlling the first-order car. From top to bottom: *AO-RRT-MCP*, *AO-RRT-MCP-Guided*, *AO-RRT-RL*. Darker color indicates newer trajectory.

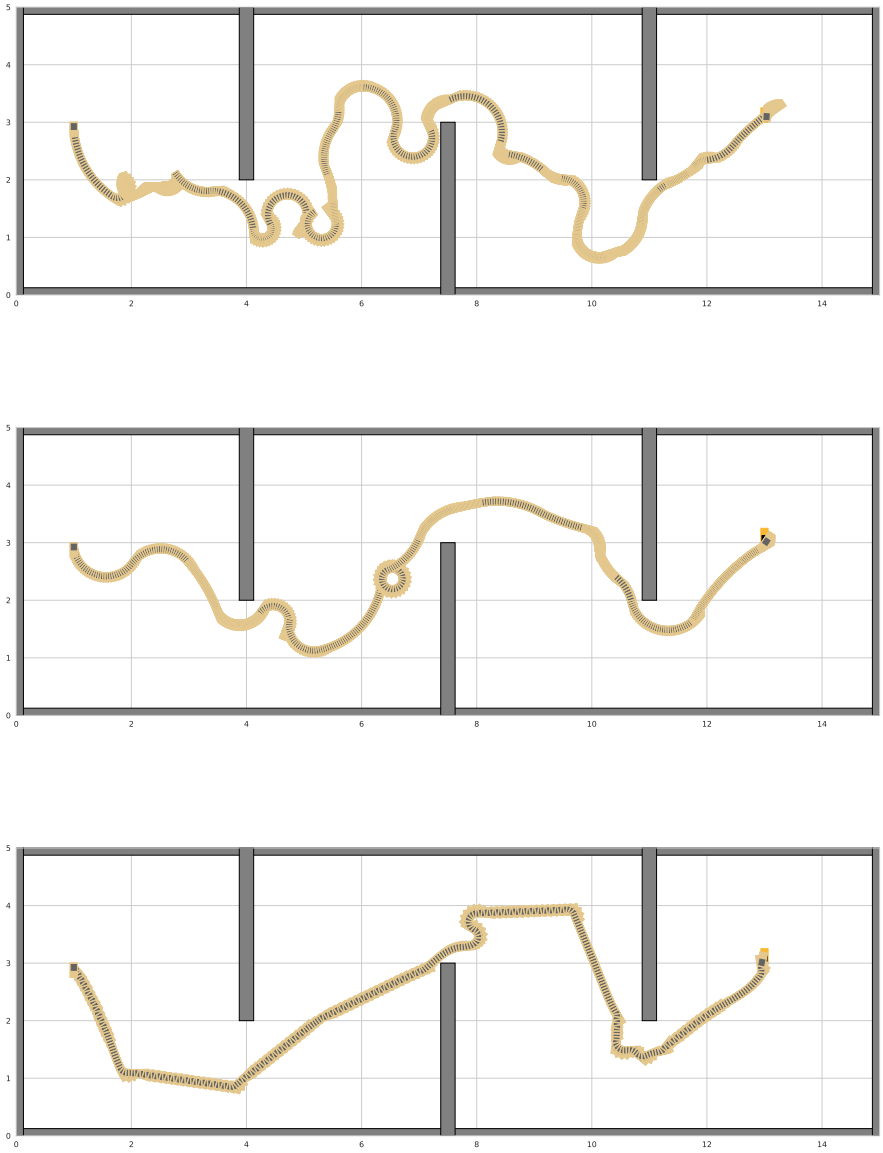


Figure 5.7: Final trajectory in the *ZigZag* map controlling the first-order car. From top to bottom: *AO-RRT-MCP*, *AO-RRT-MCP-Guided*, *AO-RRT-RL*.

5.5 Tree Growth

To get a better picture of the differences and similarities between the three evaluated variants, it can be interesting to analyze how their state-space-tree grows. In the following we present example runs for each variant guiding the first-order-car system through the *Bugtrap* map. To generate the trees, we modified our AO-RRT setup to continue the first round (without cost-boundary) even after a motion plan was found. The trees were plotted after 100, 250 and 500 extension attempts.

5.5.1 MCP

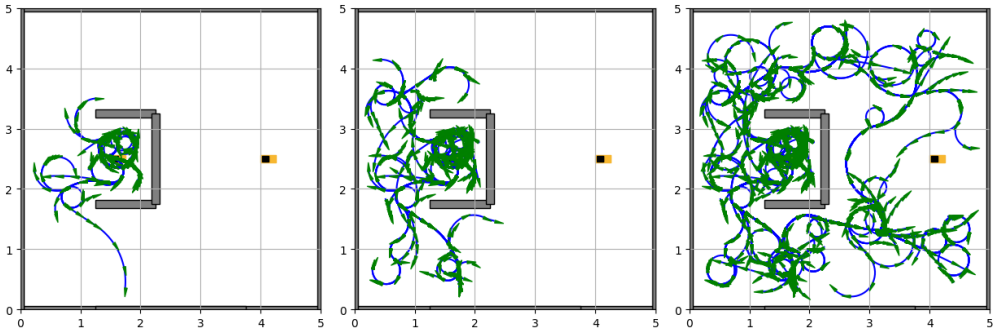


Figure 5.8: RRT-MCP: Tree growth after 100, 250 and 500 samples.

In the MCP version, a significant amount of extension attempts are "wasted" on exploring the inside of the bugtrap structure before managing to "escape". At 100 samples, the first few branches of the tree grow in the open space. At 500 samples, the tree is covering a fair amount of configuration space, but none of the leafs are inside the goal region.

5.5.2 MCP-Guided

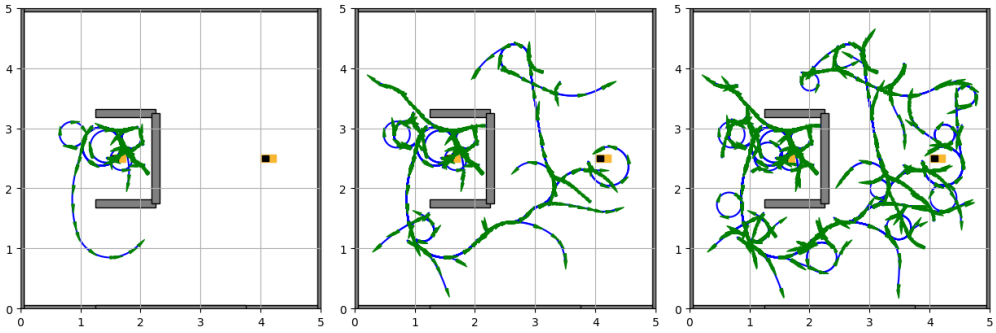


Figure 5.9: RRT-MCP-Guided: Tree growth after 100, 250 and 500 samples.

The guided version of MCP also struggles to escape the trap, with only one branch escaping at 100 extension attempts. Afterwards however, the guided approach coupled with the applied goal bias ($P_{goal} = 0.01$), is clearly growing towards the goal region. At the 500-extension-step-mark, some leafs come fairly close to the target configuration.

5.5.3 RL

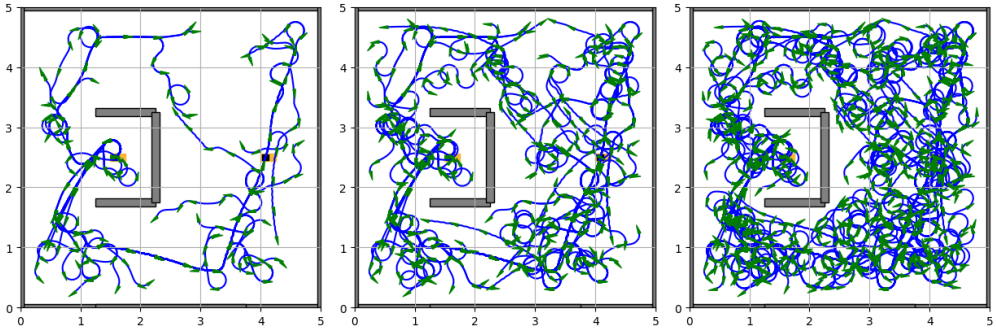


Figure 5.10: RRT-RL: Tree growth after 100, 250 and 500 samples.

Compared to the two trees above, the one built with the RL-extension step manages to escape the bugtrap structure earlier. At 100 steps, the tree is already exploring far into the map, with some leafs coming close to the goal region. At 250 steps, the configuration space is already covered with many states and their connecting trajectories. At 500 steps, we can see, that many leafs are placed close to or inside the goal region. When comparing the motions in between the node-states with the other two approaches, it can be noticed, that the curves of the extension trajectories are a lot sharper and the tree features a noticeable amount of circles.

6 Discussion

In this section we will discuss the implications of the results presented in Chapter 5, as well as the possible avenues for future work.

6.1 Lessons Learned While Training The Policies

The first step of our approach was to train control policies for the robotic systems we target. Using the state-of-the-art algorithm PPO, we were able to train a near-optimal control policy for the first-order car, using a minimalistic reward function design, in a reasonable amount of time (training the model we used for the later experiments took roughly 54 minutes on our CPU-cluster, without much optimization regarding parallelization). The reward function we used does not add additional parameters to the system and therefore does not increase the overall parameter-tuning-complexity.

When applying the same approach to the second order car however, we were met with more difficulties. While this was expected, based on the added dynamic complexity and the additional state-space dimensions, it is interesting that deploying the denser reward function was not fruitful in our case. While we cannot rule out, that there might exist a set of parameters, that enables the design to achieve better results, finding these parameters can be time-intensive, especially considering, that there are already numerous parameters to tune for PPO itself.

This is where the Curriculum Learning approach we used shines. Using simple transformations of random-rollout trajectories, we were able to produce an effective curriculum, enabling the agent to learn the basic dynamics of the environment before graduating to open sampling, i.e. the actual target environment.

When studying the result of the curriculum based training, presented in Subsection 3.4.2.2, it can be noticed, that the agent's performance barely changes when increasing the size of the sampling pool. The first real impact is visible after sampling is switched to fully-random (the original target domain). This finding implies that further optimization of the approach might be possible. Instead of slowly increasing the amount of easier samples the agents sees, and thereby fragmenting the training into 5 different phases, only 2 phases might be sufficient as well.

6.2 Investigating the Performance of the RL based Planner

In our experiments, the performance of the AO-RRT planner was significantly increased by using the RL-policy as the local planner. The RL-variant found its initial solution quicker and more reliable than both MCP-based variants while producing noticeably faster motion plans for both dynamical systems in both maps we investigated. The differences were especially pronounced with the more complex dynamics of the second-order car-like system in the larger and more maze-like *ZigZag* map, where the planner variant with the median cost of the planner with the RL-integration was only a third of its next best counter-part (which did not achieve a 100% success rate during the runtime of the experiment). While these results are promising, we take a closer look on some aspects of the experiment results to get a better picture of the overall impact, the RL-extension step has on the AO-RRT planner.

6.2.1 Comparing the Time-To-First-Success

While the plots in Figures 5.2 - 5.5 already visualize the times at which the planners achieve their first success, it is interesting to compare them through a different lens. Instead of plotting the progression of the success-percentage over time, we can compare the distribution of their time-to-first-success $t_{success}$. In Figure 6.2.1 they are visualized as boxplots for all extension variants, with all dynamics and in all maps. In the figure, the boxes signify the Interquartile Range (IQR), meaning 25% of the results lie below and 25% above the boundaries of each box. The whiskers are drawn at 1.5 times the IQR. The line inside each box indicates the median. The plots again emphasize, that in our evaluation experiments, RRT-RL needed significantly less time to find the first feasible solution, while providing a more reliable performance as shown by the significantly smaller spread of the results.

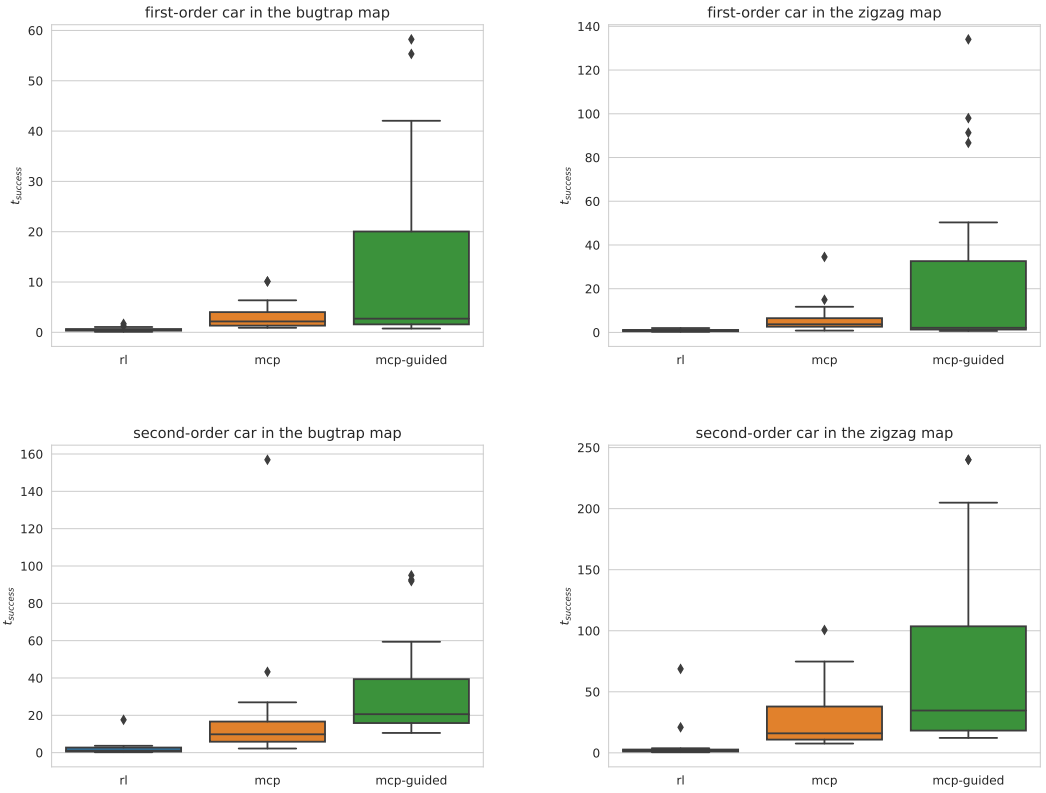


Figure 6.1: Boxplot of the time the planners took to find the initial solution in the different maps and with the different cars.

6.2.2 Query Speeds

One of the trade-offs of the RL-based approach compared to MCP and MCP-Guided is the significantly increased time needed for each individual extension. The Boxplot in 6.2 shows the query times of 1000 random pairs of start and target configurations for the first-order car using the RL, MCP and MCP-Guided (with $K = 10$) versions. As can be seen the median query time for the RL-extension step is 6 times longer than the one for MCP-Guided and nearly 70 longer than the one for the one shot version (MCP) in our experiments. While it is not in itself surprising, that the RL-variant, which queries the trained actor-network multiple times, is slower than the other two approaches, these results highlight, that the usefulness of the RL-extensions seem to outweigh the significant performance advantages of the Monte-Carlo approaches.

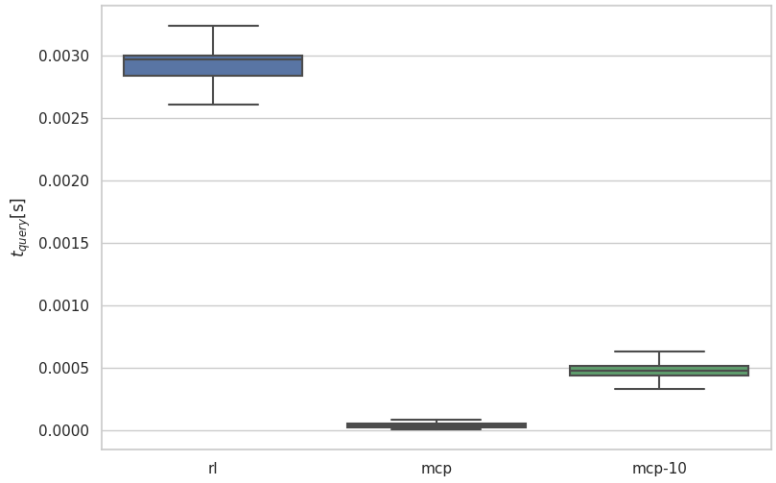


Figure 6.2: Boxplot of the runtime for 1000 random query pairs for the first-order car system

6.2.3 Comparing the Extension Steps

One way of analyzing the differing extension behavior of the RL-local-planner is by comparing the planners regarding the tree growth they promote. The MCP planners sample the control for each step uniformly from the valid control-space. When looking at the tree progression in Figure 5.5.1, the results of this can clearly be seen, as the growing tree features a wide variety of curves produced by diverse combinations of linear velocity and steering angle. As a steering angle very close to 0 is unlikely however, the tree features barely any straight lines. This stands in stark contrast to the tree progression of the RL-integration, seen in Figure 5.5.3, which features numerous straight connections. The variety of curves is also much smaller in the RL-version and a clear bias towards sharp turns can be seen.

These differences in the distribution of the applied controls obviously influence the final trajectory heavily. As the RL-planner produces faster and straighter extension steps, the resulting global motion plan is also generally faster than for the other planners.

6.2.4 Final Connection to the Goal Region

As noted in section 5.5, all planners in our evaluation use a goal-bias of 0.01, which means that the global target-state is sampled as the extension target with a probability of 1%, each iteration. If there are nodes in a suitable distance to the goal region, the planners have a chance to reach it and successfully terminate the motion planning attempt. One of the main advantages of the RL-integration is that it significantly improves the

success chances of this final extension, as compared to the random-control variants, it is able to closely approach given target states with complex maneuvers. This attribute is visible in Figure 5.4. Where both MCP based versions feature complex loops and unnecessary movement around the target, the approach of the RL-system is significantly more direct in this final section of the trajectory.

6.2.5 MCP vs. MCP-Guided

While we mainly use the MCP-variants as a comparison target for the evaluation of our RL-integrated planner, it is nonetheless interesting to also compare the performance of both to each other. In our experiments the MCP was significantly faster to reach the 50% success-rate for both dynamics in both maps, than its guided counter-part. This is interesting, as intuitively it would seem like the selection behavior of the MCP-Guided approach should when paired with the used goal-bias should be more quickly guide the tree growth towards the goal region. We think that one reason for these results is, that while the tree-growth can better targeted with MCP-Guided, the fully random extension actually favors the MCP variant in the early stage of the bugtrap exploration, as well as in the maze-like *ZigZag* structures. In both selecting the extension-trajectory based on the goal bias can actually be counterproductive, as moving sideways often guides the system into collision with the obstacles.

6.2.6 Convergence Rate

While the integrated AO-RRT-RL planner finds its initial solutions quick and is able to improve the solution quality in subsequent iterations, the rate of improvement slows down fast. The plots presented in Section 5.2 and 5.3 show that most of computational time is used for rather small improvements of the quality. The dynamic is also visible in both alternative planner variants. This behavior is not atypical for *AO-x* and in fact many asymptotically optimal sampling motion planners, as the benchmarking done by Hönig et al. [47] shows. In practice, this attribute motivates the combination of AO-SBMP with other planning approaches, like numerical optimization. Using AO-SBMP to find a high-quality guess which can then be further improved by an optimization based planner can be an effective strategy to use the qualities of both planning-methods while compensating each other's shortcomings.

6.3 Future Work

The results of the thesis open up various avenues for future research. In the following we will discuss the ones we find most promising.

6.3.1 Integrating The Critic Network Into The Nearest-Neighbor Search

As described in Section 3 we were able to train a near-optimal control policy for both the first- and second-order car-like system using the minimal reward function depicted in Equation 3.21. In this version of the reward function, the reward is solely based on the number of steps the agent needs to take to go from the initial to the target state. Using $\gamma = 1$, the value of any state $\mathbf{x}$, when following the optimal policy π_* would be

$$v_*(\mathbf{x}) = 1 - \frac{n}{N_{max}} \tag{6.1}$$

with n describing the minimum-number of steps needed to reach the target. We can reformulate this equation as

$$n = (1 - v_*(\mathbf{x})) \cdot N_{max} \tag{6.2}$$

If we would have a good estimation of the value-function $V(\mathbf{x})$, we could calculate a usable estimation of the steps-to-target with

$$\hat{n} = (1 - V(\mathbf{x})) \cdot N_{max} \tag{6.3}$$

This estimation would in turn provide a good metric for the nearest-neighbor search for sampling-based planners. In future research it could be explored if the critic-network trained as part of an actor-critic based training of control policies can be integrated into a distance estimator, like Chiang et al. suggested in their Discussion of RL-RRT [39]. To compensate for the additional runtime compared to the much less accurate, but also much faster weighted euclidean distance approximation commonly deployed, one could utilize a hierarchical approach similar to the one the authors proposed for their (separately trained) distance-approximator.

6.3.2 Integration Into Other Probabilistically Optimal Planning Frameworks

While the AO-x framework we explored provides straightforward integration and competitive performance, it would be interesting to explore and compare the performance impact of the trained RL-control policies into other state-of-the-art asymptotically optimal motion planners. A great candidate for this could be *Stable Sparse RRT**, originally proposed by Li et al. [27].

6.3.3 Investigating Obstacle-Aware Policies

While the use of obstacle-unaware control policies proved to be an efficient solution for our target dynamics in the evaluated maps, they can significantly hinder the system's ability to find solutions in some scenarios, as outlined in Section 4.4. While integrating a small amount of randomness in the extension step can combat these issues, it might hinder the performance. Another approach would be to introduce obstacle-awareness for the RL-agent, like proposed by Chiang et al. [39] and Faust et al. [36]. While this approach significantly inflates the agent's observation space and complicates the local-planner integration, the chance of producing feasible trajectories should be improved. In future research it would be interesting to compare the performance of these two strategies directly.

7 Conclusion

The problem of generating motion plans to guide complex robotic systems through their environment without collisions or constraint-violations has inspired many different method-classes in the field of motion planning, like search- or numerical-optimization-based planning. One of these, focused on fast probabilistically complete or even optimal planning in high-dimensional state-spaces, is Sampling-Based Motion Planning.

Planners of this class utilize modern high-performant collision checking modules to build a tree- or graph-based representation of the state-space by sampling collision-free states, selecting a suitable node and extending this previously discovered free state towards the sampled one with a collision-free trajectory.

The implementation of the extension primitive of this approach can be challenging for kinodynamic systems without a known efficient analytical steering function for optimal or near-optimal connections. This motivated the use of Reinforcement Learning to train a neural network based control-agent to guide the extensions instead.

To evaluate this approach, we designed training environments for two car-like robotic systems. The first is controlled via linear-velocity and steering-angle, while the second is controlled via linear-acceleration and the angular velocity of the steering angle. The control-agents were trained in obstacle-free and smaller-scale versions of the target workspace. For the training setup, we utilized Proximal Policy Optimization, which we supplemented with a custom Curriculum Learning approach in the case of the second order car. Based on the translational- and rotational invariances inherent in our target-dynamics we designed an automatic way of generating training sets of starting configurations with a known maximum solution cost. Using these training sets as our curriculum, we managed to significantly speed up the training process.

After successfully training policies for both target systems, they were subsequently used to build a local-planner module for the extension step of one of the most prominent sampling-based motion planners, RRT. The resulting RL-RRT planner, was then integrated into the asymptotically optimal motion planning framework *AO-x* and evaluated for the two target systems, in two different scenarios, comparing them with two Monte-Carlo based alternatives for the extension step.

In our evaluation, we observed, that the version with the RL extension step significantly outperformed the two alternatives in both scenarios and for both robot systems. RL-RRT found its solutions faster and more reliably than both *MCP* and *MCP-Guided*. Both the initial and the final trajectories of AO-RL-RRT had a significantly smaller

path-execution-time than the MCP based alternatives. Guiding the second-order car in the *ZigZag* map, its median path-execution-time was less than half compared to *MCP-Guided*.

Our visual analysis of the tree growth promoted by the RL-integration suggests that RL-RRT integration explores the state space faster than *MCP* and *MCP-Guided*.

Bibliography

[1] Statista. "Revenue of the warehouse robotics market worldwide from 2021 to 2030 (in billion u.s. dollars) [chart]." (2021), [Online]. Available: https://www.statista.com/statistics/1250585/warehouse-robotics-market-revenue-worldwide/ (visited on 08/11/2023).

[2] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, Mar. 1994.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing Atari with Deep Reinforcement Learning*, Dec. 2013. arXiv: 1312.5602 [cs].

[4] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[5] Google DeepMind. "The Challenge Match," [Online]. Available: https://www.deepmind.com/research/highlighted-research/alphago/the-challenge-match (visited on 08/12/2023).

[6] J. D. Gammell and M. P. Strub, "Asymptotically Optimal Sampling-Based Motion Planning Methods," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, no. 1, pp. 295–318, May 2021. arXiv: 2009.10484 [cs].

[7] S. M. LaValle, *Planning Algorithms*, 1st ed. Cambridge University Press, May 2006.

[8] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning, and Control*. Cambridge, UK: Cambridge University Press, 2017.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[10] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2520–2525.

[11] P Rouchen, M Fliess, J Levine, and P. Martin, "Flatness and motion planning: The car with n trailers," in *European Control Conference*, 1992, pp. 1518–1522.

[12] D. Malyuta, T. P. Reynolds, M. Szmuk, *et al.*, "Convex Optimization for Trajectory Generation: A Tutorial on Generating Dynamically Feasible Trajectories Reliably and Efficiently," *IEEE Control Systems Magazine*, vol. 42, no. 5, pp. 40–113, Oct. 2022.

[13] Y. Mao, M. Szmuk, X. Xu, and B. Acikmese, *Successive Convexification: A Superlinearly Convergent Algorithm for Non-convex Optimal Control Problems*, Feb. 2019. arXiv: 1804.06539 [math].

[14] R. Bonalli, A. Cauligi, A. Bylard, and M. Pavone, *GuSTO: Guaranteed Sequential Trajectory Optimization via Sequential Convex Programming*, Feb. 2019. arXiv: 1903.00155 [cs, math].

[15] M. Toussaint, *Newton methods for k-order Markov Constrained Motion Problems*, Jul. 2014. arXiv: 1407.0414 [cs].

[16] J. Kamat, J. Ortiz-Haro, M. Toussaint, F. T. Pokorny, and A. Orthey, *BITKOMO: Combining Sampling and Optimization for Fast Convergence in Optimal Motion Planning*, Sep. 2022. arXiv: 2203.01751 [cs].

[17] S. M. LaValle and J. J. Kuffner, "Randomized Kinodynamic Planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[18] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Department of Computer Science, Iowa State University, Technical Report 9811, 1998.

[19] T. Kunz and M. Stilman, "Kinodynamic RRTs with Fixed Time Step and Best-Input Extension Are Not Probabilistically Complete," in *Algorithmic Foundations of Robotics XI*, H. L. Akin, N. M. Amato, V. Isler, and A. F. Van Der Stappen, Eds., vol. 107, Cham: Springer International Publishing, 2015, pp. 233–244.

[20] A. Settimi, D. Caporale, P. Kryczka, M. Ferrati, and L. Pallottino, "Motion primitive based random planning for loco-manipulation tasks," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, Cancun, Mexico: IEEE, Nov. 2016, pp. 1059–1066.

[21] A. H. Qureshi and Y. Ayaz, "Potential functions based sampling heuristic for optimal path planning," *Autonomous Robots*, vol. 40, no. 6, pp. 1079–1093, Aug. 2016.

[22] M. Kang, D. Kim, and S.-E. Yoon, *Harmonious Sampling for Mobile Manipulation Planning*, Sep. 2019. arXiv: 1809.07497 [cs].

[23] L. Janson, B. Ichter, and M. Pavone, "Deterministic sampling-based motion planning: Optimality, complexity, and performance," *The International Journal of Robotics Research*, vol. 37, no. 1, pp. 46–61, Jan. 2018.

[24] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[25] F. Staals, *Computational Geometry - Lecture 9 : Range searching and kd-trees*, Utrecht University, Dec. 2021.

[26] M. C. Lin, D. Manocha, and Y. J. Kim, "Collision and proximity queries," in *Handbook of discrete and computational geometry*, Chapman and Hall/CRC, 2017, pp. 1029–1056.

[27] Y. Li, Z. Littlefield, and K. E. Bekris, "Asymptotically optimal sampling-based kinodynamic planning," *The International Journal of Robotics Research*, vol. 35, no. 5, pp. 528–564, Apr. 2016.

[28] K. Hauser and Y. Zhou, "Asymptotically Optimal Planning by Feasible Kinodynamic Planning in a State–Cost Space," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1431–1443, Dec. 2016.

[29] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (Adaptive Computation and Machine Learning Series), Second edition. Cambridge, Massachusetts: The MIT Press, 2018.

[30] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[31] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal Policy Optimization Algorithms*, Aug. 2017. arXiv: 1707.06347 [cs].

[33] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, PMLR, 2015, pp. 1889–1897.

[34] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, Nov. 2012.

[35] T. McMahon, A. Sivaramakrishnan, E. Granados, and K. E. Bekris, "A Survey on the Integration of Machine Learning with Sampling-based Motion Planning," *Foundations and Trends® in Robotics*, vol. 9, no. 4, pp. 266–327, 2022. arXiv: 2211.08368 [cs].

[36] A. Faust, O. Ramirez, M. Fiser, *et al.*, *PRM-RL: Long-range Robotic Navigation Tasks by Combining Reinforcement Learning and Sampling-based Planning*, May 2018. arXiv: 1710.03937 [cs].

[37] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, *Continuous control with deep reinforcement learning*, Jul. 2019. arXiv: 1509.02971 [cs, stat].

[38] A. Faust, P. Ruymgaart, M. Salman, and R. Fierro, "Continuous action reinforcement learning for control-affine systems with unknown dynamics," *IEEE/CAA Journal of Automatica Sinica*, vol. 1, no. 3, pp. 323–336, Jul. 2014.

[39] H.-T. L. Chiang, J. Hsu, M. Fiser, L. Tapia, and A. Faust, *RL-RRT: Kinodynamic Motion Planning via Learning Reachability Estimators from RL Policies*, Jul. 2019. arXiv: 1907.04799 [cs].

[40] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis, "Learning Navigation Behaviors End-to-End With AutoRL," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 2007–2014, Apr. 2019.

[41] A. Sivaramakrishnan, E. Granados, S. Karten, T. McMahon, and K. E. Bekris, *Improving Kinodynamic Planners for Vehicular Navigation with Learned Goal-Reaching Controllers*, Oct. 2021. arXiv: 2110.04238 [cs].

[42] Z. Littlefield and K. E. Bekris, "Efficient and Asymptotically Optimal Kinodynamic Motion Planning via Dominance-Informed Regions," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid: IEEE, Oct. 2018, pp. 1–9.

[43] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7382–7431, 2020.

[44] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[45] O. R. developers, *Onnx runtime*, `https://onnxruntime.ai/`, Version: 1.15.0, 2023.

[46] M. Kleinbort, E. Granados, K. Solovey, R. Bonalli, K. E. Bekris, and D. Halperin, *Refined Analysis of Asymptotically-Optimal Kinodynamic Planning in the State-Cost Space*, Mar. 2020. arXiv: `1909.05569 [cs]`.

[47] W. Hönig, *Motion Planning - Sampling-Based Geometric Motion Planning: PRMs (Lecture Slides)*, Lecture at TU Berlin, May 2023.

Appendix A

Reinforcement Learning Parameters

Parameter Name	Value
learning_rate	7.77e-05
n_steps	2048
batch_size	64
n_epochs	10
gamma	0.999
gae_lambda	0.95
clip_range	0.2
clip_range_vf	0.5
normalize_advantage	True
ent_coef	0.01
vf_coef	0.5
max_grad_norm	0.5
use_sde	False
sde_sample_freq	-1
target_kl	None
device	auto

Table 7.1: PPO parameters used for the training of both dynamical systems. These parameters are based on the PPO defaults in stable-baselines 3 [44], which we modified until we got useful results. While these were sufficient in our case (in the second order car system in combination with our curriculum approach), they are probably not the best possible combination and sophisticated automated parameter tuning approaches might find sets which work even better.