Proceedings of Machine Learning Research vol 288:1–23, 2025 2nd International Conference on Neuro-symbolic Systems (NeuS)

Real-Time Reachability for Neurosymbolic Reinforcement Learning based Safe Autonomous Navigation

Nicholas Potteiger NICHOLAS.POTTEIGER@VANDERBILT.EDU
Diego Manzanas Lopez DIEGO.MANZANAS.LOPEZ@VANDERBILT.EDU
Taylor T. Johnson TAYLOR.JOHNSON@VANDERBILT.EDU
Xenofon Koutsoukos XENOFON.KOUTSOUKOS@VANDERBILT.EDU
Vanderbilt University, Nashville, TN

Editors: G. Pappas, P. Ravikumar, S. A. Seshia

Abstract

Safety is essential in autonomous navigation, especially as autonomous systems deploy to new environments where collision avoidance is critical. Neurosymbolic reinforcement learning (NeSy RL) approaches show promise for advancing long-term navigation by using symbolic planners to compute high-level waypoints and goal-conditioned RL for low-level control. However, ensuring safety within these frameworks remains a challenge, particularly in new environments that the agent was not optimized for. Current safe RL based navigation techniques offer robust frameworks for ensuring safety. However, these approaches are not adapted for NeSy RL and also present challenges: they can be computationally intensive or constrained by conservative control. To overcome these limitations, we propose a novel approach to safely and efficiently navigate a NeSy RL agent in new environments. The proposed method uses real-time reachability analysis to select subgoals between waypoints and safeguard the actions of a goal-conditioned RL policy. We implement the approach in *Rust* and develop a software package, *RusTReach*, for real-time reachability analysis. We deploy our approach on an embedded device and compare against four approaches in a long-term quadcopter navigation task in a new environment. Our evaluation reveals that our approach is at least 1.7 times faster at navigating than a state-of-the-art alternative while maintaining safety and real-time constraint compliance. Code and videos available at `https://github.com/npotteig/rustreach`.

Keywords: neurosymbolic AI, autonomous navigation, collision avoidance, hierarchical reinforcement learning, real-time reachability, autonomous ground vehicle, unmanned aerial vehicle

1. Introduction

Ensuring safety is critical in autonomous navigation, as these systems are deployed in real-life environments, ranging from delivery services, public transportation, and search and rescue missions. Autonomous agents must operate efficiently to achieve long-term objectives in often unpredictable or new environments, increasing uncertainty and the potential for collision with static and dynamic obstacles. Collision with obstacles can cause significant physical, financial, and reputational losses, underscoring the need for robust safety mechanisms.

Recent autonomous navigation approaches leverage symbolic reasoning and reinforcement learning (RL), each with distinct advantages. Symbolic methods excel in structured decision-making and planning, Zhou et al. (2019); Rocamora et al. (2024), but they rely on accurate domain knowledge, making them less effective in unpredictable settings. RL, in contrast, learns behaviors through trial and error, offering adaptability in Nachum et al. (2018); Kim et al. (2021), yet it struggles with sample inefficiency and lacks interpretability. Neurosymbolic RL (NeSy RL) combines these strengths, integrating RL's adaptability with symbolic systems' logical reasoning , Acharya et al. (2024); Cai et al. (2024), enabling agents to learn flexibly while ensuring interpretability and verifiability.

One category of NeSy RL is *learning for reasoning*, where RL supports symbolic components, Acharya et al. (2024). Toward *learning for reasoning* in navigation, a goal-conditioned RL policy can be integrated with symbolic planning, guiding agents through waypoint subgoals generated by planners like A^* or *RRT*, Xiong et al. (2022). However, these planners ignore system dynamics, and RL lacks safety guarantees. Toward safety, the RL policy learned in a training environment may not be robust to a new environment with new and dynamic obstacles. For reliable deployment in new dynamic environments, goal-conditioned RL must incorporate safety mechanisms that address disturbances and obstacles, ensuring robust, real-time navigation.

Existing approaches for safeguarding RL components focus on minimizing safety violations during runtime or providing theoretical guarantees at design-time. Design-time methods approximate the reachable states for goal navigation, Huang et al. (2022); Lopez et al. (2023); Bogomolov et al. (2019); Wang et al. (2023), but they lack robustness when deployed in unfamiliar environments and suffer from high computational costs. Runtime shielding switches to a safety controller or modifies unsafe actions during deployment, Alshiekh et al. (2018); Thomas et al. (2021); Roza et al. (2023); Musau et al. (2022); Potteiger and Koutsoukos (2024). Musau et al. (2022) and Johnson et al. (2016) use real-time reachability analysis to compute tight over-approximations of reachable states, considering fixed control over a receding finite-time horizon, to decide when to switch to a safe controller. While shielding protects against unsafe states, it becomes challenging to define optimal switching conditions and controllers as system dynamics become more complex. Safe certificate learning learns safety properties alongside the RL policy, Bansal and Tomlin (2021), Xia et al. (2024), Dawson et al. (2023); Peruffo et al. (2021); Singh et al. (2021). Xiong et al. (2022) combines a neural Lyapunov certificate with an RL policy for safe navigation between waypoints, using a runtime monitor to select safe subgoals. While this approach adapts to new environments, its conservative over-approximation can lead to inefficient navigation and there are no safety guarantees around the neural certificate itself. Related work in Appendix A.

Building on insights from previous works, we develop an approach for safe and efficient navigation of a NeSy RL agent in new environments that satisfies real-time constraints and does not require an external verified safe controller. Our approach is an algorithm that selects subgoals to compute safe control using real-time reachability analysis. Given a symbolic planner, a set of obstacles, system dynamics, and model parameters learned via goal-conditioned RL our approach selects safe sugboals in real-time where the set of reachable states does not intersect with obstacles. This reduces the need to switch to an external verified safe controller as safety is ensured through subgoal selection at each timestep. To compute the set of reachable states, we develop an anytime reachability algorithm that will run up until a prescribed runtime deadline. The reachability algorithm, building on our previous work in Johnson et al. (2016), directly incorporates control from the RL policy over a receding finite-time horizon, allowing for more accurate tracking of the agent's future trajectory, compared to using fixed control, improving detection of potential future collisions. The main technical contributions of this work are:

A safe subgoal selection algorithm to construct goal-conditioned RL policies to navigate without collision between waypoints, generated from a symbolic planner, such as $A*$. The subgoals are selected from candidates generated between waypoints using an anytime reachability algorithm, which calculates reachable states within a receding finite-time horizon and meets real-time dead-

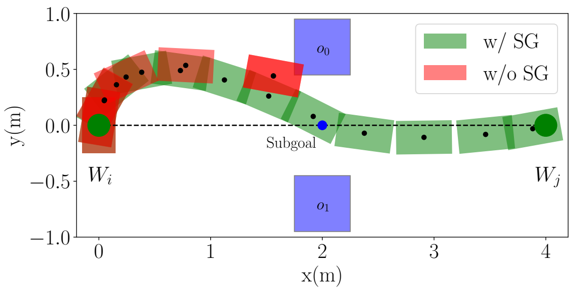


Figure 1: Trajectories for an autonomous navigation scenario between two waypoints with two obstacles. Using a goal-conditioned policy conditioned on the next waypoint W_j leads to a collision, whereas considering a subgoal leads to a safe trajectory.

lines. Safety and efficiency are achieved by selecting candidates with reachable sets that avoid obstacles while minimizing distance to the next waypoint.

We develop a reachability algorithm, which computes a sequence of discrete hyper-rectangles (”boxes”) as the reachable states over a receding finite time horizon using control from an RL policy. The RL policy inputs the centroid of each hyper-rectangle to compute the next hyper-rectangle, enabling more accurate tracking of future trajectories. Additionally, we developed *RusTReach*, a *Rust*-based software package that computes reachable states in real-time, benefiting from *Rust's* memory safety, speed, and embedded systems compatibility.

We evaluated our approach on an NVIDIA Jetson Nano embedded device in a large neighborhood environment, not encountered during training, using a quadcopter model. We conducted two experiments in the neighborhood with static and dynamic obstacles and concluded that our approach navigates to final waypoints at least 1.7 times faster than a state-of-the-art method while minimizing failures (collisions, timeouts, and irrecoverable states) and adhering to real-time constraints. The results further highlight the importance of RL-based control in reachable set computations, as fixed control leads to increased navigation failures. We further evaluate generalization of our approach for vehicles with non-holonomic constraints by testing it on a car-like system in Appendix J. Additionally, we assess both the quadcopter and the car in a corridor environment with a narrow passageway in Appendix I. In the car and corridor experiments, we find that our method continues to outperform the state-of-the-art in navigation efficiency and minimizing navigation failures.

2. Safe Autonomous Navigation

Autonomous agents must navigate efficiently and safely toward long-term objectives when they deploy to new environments. NeSy RL agents combine symbolic reasoning with reinforcement learning and are effective in generalizable navigation.

One NeSy RL approach is to combine symbolic planning algorithms and hierarchical reinforcement learning (HRL) to navigate to long-term goals. The subset of HRL known as *goal-conditioned reinforcement learning*, Kim et al. (2021); Zhang et al. (2020); Nachum et al. (2018) trains model parameters η to construct a goal-conditioned policy of the form $\pi_g := \pi(x; g, \eta)$ that inputs state x conditioned on a goal g and outputs control u to navigate a system with dynamics $\dot{x} = f(x, u)$ to goal g. The goal g is typically selected as an intermediate short-term goal, also referred to a *subgoal*, helping to break down the long-term goal into adjacent subgoals.

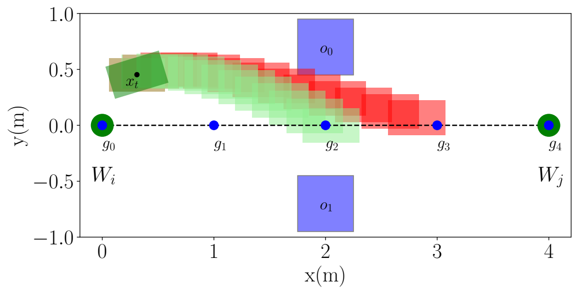


Figure 2: Subgoal selection algorithm. $n_{cand} = 5$ candidates are generated between waypoints W_i and W_j. The subgoal, g_2, selected minimizes the distance to W_j, while satisfying the constraint that the reachable set for a time horizon $2.0s$ from state s_t conditioned on g_2 does not intersect with obstacles o_0 and o_1.

Subgoals are selected using a symbolic planning algorithm like A^* or a sampling-based planner like *RRT*. A^* and *RRT* compute a sequence of p waypoints as subgoals Ω = [W_1, W_2, ..., W_i, W_j, ..., W_{p-1}, W_p] given a set of obstacles Λ, start position, and goal position. The line of sight between each waypoint (W_i, W_j) is guaranteed to not intersect with obstacles at the beginning of simulation with a buffer for the dimensions of the vehicle. $\pi(x; W_i, \eta)$ can then be constructed on each waypoint W_i to traverse to each subsequent waypoints and ultimately the goal. These are efficient algorithms for generating waypoint plans in a new environment without needing to know obstacles Λ during training η.

Challenging factors to consider for subgoal selection are uncertainty in the environment due to disturbances, control from π_g, dynamical constraints, and dynamic obstacles. These factors can cause the system trajectory to deviate from (W_i, W_j) leading to a potential collision. Dynamic obstacles that intersect (W_i, W_j) during a period of simulation need to be considered, else even following the line exactly could cause a collision. Since the control of the vehicle is tied to the conditioned subgoal, we must consider safe subgoals and track the waypoints to the goal based on the current state of the system during deployment. There must be consideration of more points than the next waypoint W_j when computing the next subgoal.

Therefore, the problem is to develop a subgoal selection algorithm that safeguards a NeSy RL agent containing: a symbolic planner to generate a sequence of waypoints Ω and model parameters η optimized using goal-conditioned RL. The subgoal selection algorithm should minimize collisions with obstacles during navigation to a goal given: the NeSy RL agent, vehicle dynamics $\dot{x}$, and a set of obstacles Λ with dynamics $\dot{\Lambda}$. The algorithm must execute in real-time to adapt to the current state of the system during deployment while navigating a sequence of waypoints and avoiding collision. The set of obstacles during deployment will be different from design-time, so the approach must bet generalizable.

3. Safe Subgoal Selection using Real-Time Reachability Analysis

We develop an approach for safe autonomous navigation of NeSy RL agent using reachability analysis for the selection of subgoals. At each control step, a subgoal is selected, and a corresponding policy is constructed to compute a safe control action. The approach chooses subgoals based on the system state, a set of obstacles, system dynamics modeled using ordinary differential equations

(ODEs), waypoints derived from a symbolic planner such as $A*/RRT$, and reachable states modeled as hyper-rectangles ("boxes") to construct goal-conditioned policies, using model parameters optimized via goal-conditioned RL, for safe control.

Before outlining our subgoal selection algorithm, we define key terms:

Definition 1 *REACHTIME, RUNTIME, and REACHABLE SET. The reachtime T_{reach} is the finite time horizon for computing reachable states, while the runtime $T_{runtime}$ is the wall-clock time allotted for this computation. The reachable set, denoted $\mathbf{R}_{[0,T_{reach}]}$, represents the set of states a system can reach within the time interval $[0, T_{reach}]$ under control policy π, given initial state $\mathbf{x}_0$ and input $\mathbf{u}$, satisfying the system dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$:*

$$\mathbf{R}_{[0,T_{reach}]} = \Big\{ q(\mathbf{x}_0, \mathbf{u}_t, t) \mid \mathbf{x}_0, \mathbf{x}_t \in \mathcal{X},\ \mathbf{u}_t = \pi(\mathbf{x}_t),\ \mathbf{u}_t \in \mathcal{U},\ t \in [0, T_{reach}] \Big\}$$

where $q(\mathbf{x}_0, \mathbf{u}_t, t)$ is the solution of the system's ODEs at time t with control $\mathbf{u}_t$.

Definition 2 *SAFETY. A system is considered safe with respect to a set of obstacles Λ_0 (initial positions) and a dynamics function $\dot{\Lambda} = f(\Lambda_0, t)$ over a finite time horizon T_{reach} if:*

$$\forall t \in [0, T_{reach}],\ f(\Lambda_0, t) \cap \mathbf{R}_t = \emptyset$$

where $\mathbf{R}_t$ is the reachable set at discrete time t. This ensures that the reachable set at each discrete time step does not intersect with any obstacle state over the entire time horizon.

During navigation, we assume that the system will always be traversing between two adjacent waypoints, within line-of-sight, in a sequence of p waypoints Ω, $W_i, W_j \in \Omega$. A subgoal g, from a set of goal points $\mathcal{G}$, is selected between W_i, W_j such that the Definition 2 is satisfied and the distance between g and W_j is minimized to encourage time-efficient navigation. This is formulated as the optimizing for the optimal subgoal g^*:

$$g^* = argmin_g\ ||g - W_j||_2 \text{ where, } \forall t \in [0, T_{reach}],\ f(\Lambda_0, t) \cap \mathbf{R}_t = \emptyset, g \in \mathcal{G} \tag{1}$$

Our approach for selecting a subgoal given the system state x_t at time t approximates the optimal subgoal $\hat{g}^*$ within a runtime deadline of T_{sgTime}. First, a set of n_{cand} subgoal candidates $[g_1, g_2, ..., g_{n_{cand}}]$ where $g_i \in \mathcal{G}$ are generated in priority order, where candidates closer in distance to W_j have higher priority. Next, each candidate is assigned an equal budget for runtime $T_{runtime}$ based on n_{cand} and the maximum runtime allowed to compute a subgoal T_{sgMax}. Then, for each candidate g_i a goal-conditioned policy is constructed π_{g_i} given pre-trained model parameters η trained via goal-conditioned RL. π_{g_i} is then used to construct a reachable set $\mathbf{R}_{[0,T_{reach}]}$. If Definition 2 is satisfied then g_i is returned, else the process is repeated for g_{i+1}. This occurs until either a subgoal is returned or no subgoals are deemed safe. In this case, the state is *irrecoverable* and external intervention is needed. An example of the subgoal selection algorithm is illustrated in Figure 2.

The subgoal returned $\hat{g}^*$ is used to construct $\pi_{\hat{g}^*}$ and generate control u to produce the next state x_{t+1}. Subgoal selection and control is repeated between W_i and W_j until the system has navigated within a distance ϵ of W_j. The next adjacent waypoints *e.g.* $W_j, W_k \in \Omega$ will then be selected and repeat until the final waypoint W_p has been reached. Algorithm details are in Appendix B.1.

3.1. Generation of Subgoal Candidates

The set of n_{cand} subgoal candidates $[g_1, g_2, ..., g_{n_{cand}}]$ are generated on a segment of the line (W_i, W_j) given: state x_t, W_i, W_j, and n_{cand}. First, x_t is projected onto the line (W_i, W_j) as $proj_d(x_t)$, where the direction vector is $d = W_j - W_i$. Then, the endpoints (e_{prev}, e_{next}) of the segment are calculated from $proj_d(x_t)$. The widths from $proj_d(x_t)$ to e_{prev} and e_{next} are tunable parameters: d_{prev} and d_{next}, where $d_{prev} + d_{next}$ is the distance of (e_{prev}, e_{next}). This is calculated as follows:

$$(u_i, u_j) = \left(\frac{W_i - proj_d(x_t)}{||W_i - proj_d(x_t)||}, \frac{W_j - proj_d(x_t)}{||W_j - proj_d(x_t)||} \right) \tag{2}$$

$$(e_{prev}, e_{next}) = (proj_d(x_t) - d_{prev} \cdot u_i,\ proj_d(x_t) + d_{next} \cdot u_j) \tag{3}$$

(u_i, u_j) are unit vectors towards W_i and W_j from $proj_d(x_t)$. (e_{prev}, e_{next}) are further clipped to be bounded between W_i and W_j. The segment (e_{prev}, e_{next}) is equally divided into n_{cand} candidate points as presented in Appendix B.2. The candidates are ordered by prioritizing candidates closer to W_j first.

3.2. Anytime Computation of Reachable Sets with Reinforcement Learning Control

For each subgoal candidate g_i a respective goal-conditioned policy is constructed $\pi_{g_i} := \pi(x; g_i, \eta)$ given model parameters η pre-trained through goal-conditioned reinforcement learning in an environment with no obstacles. Using π_{g_i}, state x_t, dynamics $\dot{x}$, and a step size h, a tight over-approximation of the set of reachable states $\mathbf{R}_{[0,T_{reach}]}$ over T_{reach} can be computed using a mixed-face lifting approach Johnson et al. (2016) assuming fixed control. We build on this approach to consider RL control, where control is predicted using a RL policy π_{g_i} over T_{reach}. Appendix G demonstrates that considering RL control leads to reachable sets that more accurately track the ground truth future states of a vehicle. Furthermore, h is used to tune the over-approximation error from $\mathbf{R}_{[0,T_{reach}]}$, where smaller h is more accurate but more expensive to compute. To tune h to abide by real-time constraints, an anytime extension was developed in Johnson et al. (2016) to incrementally decrease h and re-compute $\mathbf{R}_{[0,T_{reach}]}$ until a runtime budget of $T_{runtime}$ has elapsed.

3.2.1. REACHABLE SET COMPUTATION USING REINFORCEMENT LEARNING CONTROL

The exact reachable set $\mathbf{R}_{[0,T_{reach}],g}$ cannot be computed for general nonlinear systems. Therefore, methods seek to compute a tight over-approximation of the set of states and set of control such that the ground truth system behavior is contained within the set at time t. However, depending on the complexity of the dynamics and control policy, computing approximations of the control set given a set of states becomes expensive and infeasible in real-time. This can be computed offline during design-time, but design-time approaches may not capture fully the uncertainties faced in an environment unknown during design-time. Therefore, in our approach, to reduce computational complexity, we opt to compute a point estimate of the control set given an over-approximated set of states at time t.

In our approach we consider a mixed-face lifting method from Johnson et al. (2016), that is part of a set of methods for computing the reachable set through flow-pipe construction given: state x_t, RL policy π, dynamics $\dot{x}$, time-horizon T_{reach}, and fixed step size h. The method constructs a sequence of discrete snapshots of the set of reachable states over T_{reach}. We construct each subsequent snapshot by computing point estimates of control using π given the current snapshot. Point estimates allow us to construct a computationally efficient algorithm that can execute in real-time. We observe the point estimate for control tracks reasonably well for a goal-conditioned controller.

The snapshots of the flow-pipe are represented symbolically as hyper-rectangles ("boxes") for efficiency to compute the set of reachable states. For short reach times T_{reach} this representation is effective, but one must consider the over-approximation error as it increases for long reach times if the set of reachable states is not a box.

The initial hyper-rectangle is initialized, using state $\mathbf{x}_0$ with 0 width for each dimension (i.e. a hyper-line) and added to the reach set. Our extension to the original approach is each subsequent hyper-rectangle is calculated first by computing the control given the current hyper-rectangle. In our approach, we compute the centroid of the hyper-rectangle producing a point estimate that is inputted as the state to policy π. The centroid of a D-dimensional hyper-rectangle r is defined as follows:

$$centroid(r) = \left\{ \frac{r_{d,max} + r_{d,min}}{2} \mid 0 < d \leq D \right\} \tag{4}$$

where $r_{d,max}$ and $r_{d,min}$ are the maximum and minimum face values at dimension d. The mixed-face algorithm from Johnson et al. (2016) is continued and repeated using RL control until each hyper-rectangle up to T_{reach} is computed and stored in the reach set. Algorithm details are in Appendix B.3.

3.2.2. EXTENSION TO ANYTIME COMPUTATION

The computation of the reachable set is amended to an anytime algorithm. The algorithm includes an extra parameter for runtime $T_{runtime}$ specifying that $\mathbf{R}_{[0,T_{reach}]}$ must be returned when $T_{runtime}$ has elapsed. We leverage the step size h to make the algorithm amenable to anytime computation. The reachable set $\mathbf{R}_{[0,T_{reach}]}$ is computed for initial step size h. Then an estimate of the remaining runtime $T_{remaining}$ is computed and determined if the step size can be halved allowing $\mathbf{R}_{[0,T_{reach}]}$ to be recomputed to decrease over-approximation error. A decreased over-approximation error can lead to a reduction in scenarios where the over-approximation error is large, where navigation to a goal g is deemed unsafe when in fact it is safe if computed with a smaller step size h (error). $\mathbf{R}_{[0,T_{reach}]}$ is returned when $T_{remaining}$ is less than or equal to zero or the current step size is too small ($h_{cur} < 1\text{e}{-7}$). Algorithm details are in Appendix B.4.

3.3. RusTReach: Reachable Set Computation in Rust

To improve computational efficiency and satisfy real-time constraints, we develop a software package, *RusTReach*, for anytime reachable set computation in Rust. We choose *Rust* due to its memory safety and security benefits, comparable execution time to *C*, package management, and its spread to embedded real-time applications Sharma et al. (2023). In addition, shared libraries can be compiled from *Rust* packages that can seamlessly integrate with existing *C/C++* code. Furthermore, since autonomous vehicle software is typically deployed to an embedded system on a compact hardware testbed, *Rust* is an ideal candidate for development and maintenance. We release our code publicly for repeatability of our results.

4. Evaluation

We perform an evaluation of our safe subgoal selection approach, using *RusTReach* to generate reachable sets, in a *Neighborhood* environment, with *static* and *dynamic* obstacles, using a quadcopter vehicle. We compare safety, navigation, and time efficiency against multiple subgoal selection approaches. Our experiments are conducted on an embedded system, NVIDIA Jetson Nano (Quad-core ARM Cortex-A57, 4 GB 64-bit LPDDR4), to study realistic deployment. An additional experiments with and a *Corridor* environment, described in Appendix C, is in Appendix I for the quadcopter. Further, we evaluate the generalization of our approach to a non-holonomic car vehicle in both environments in Appendix J.

4.1. Experimental Setup

Quadcopter: We consider a quadcopter aerial vehicle. The quadcopter considers realistic model parameters and the dynamics have been demonstrated in previous work, Sabatino (2015), to track well with complex systems. The quadcopter is represented as a two-dimensional rectangle. The quadcopter has a 12-dimensional state (position, orientation, linear and angular velocities) with four-dimensional control (thrust, xyz torques) and measures 0.32 meters on each side. More details on vehicle dimensions, dynamics, and parameters are in Appendix E.

Neighborhood: The *Neighborhood* environment is a two-dimensional representation, shown in Appendix D, of a neighborhood in Microsoft AirSim, Shah et al. (2017): a simulation platform for quadcopter simulation. The grid representation is discrete. The size of the environment is $[-50, 50]^2$ meters, where the origin is in the center $(0, 0)$. There are 1634 of 10K grid cells marked as obstacles.

We conduct two experiments to evaluate long-term navigation in the *Neighborhood* environment: one with *static* obstacles and another with *dynamic* obstacles. For both, we generate 1000 start positions p_0 and goal positions p_f with respective paths represented as sequences of waypoints computed using the A^*/RRT planning algorithm, considering a obstacle buffer of 1 meter for vehicle feasibility. The vehicle start location is p_0 as well as the initial waypoint. The distance threshold to reach each waypoint in the sequence is $\epsilon = 1.0$. For the *static* obstacle experiment, the obstacles are fixed for the duration of simulation. For the *dynamic* obstacle experiment, obstacles are spawned between two waypoints in the path, if the distance between the two is sufficiently large > 2.9 meters. The obstacle dynamics translate the obstacle position perpendicular to the path at 0.5 m/s, creating a challenging navigation scenario.

Simulation Parameters: During each timestep, a policy action is sampled and subgoal is selected. The policy action is a desired velocity inputted to a PID controller. The policy selects an action every $dt = 0.1$ seconds, with a maximum of $T_{max} = 1000$ timesteps. The PID control sampling rate is 5000Hz. A subgoal is selected every $T_{sgMax} = 100$ms, and the reach time is $T_{reach} = 1.0$s. The initial step size is $h = 0.1$. We use $n_{cand} = 5$ subgoal candidates with distances $d_{prev} = 5$ and $d_{next} = 5$ meters. The simulation terminates when the target is reached, a collision occurs, T_{max} timesteps are reached, or no safe subgoal is found.

Optimizing Goal-Conditioned Policies: To avoid confounding factors related to differences in training algorithms, we use the same goal-conditioned RL algorithm to optimize a single set of model parameters, η_i, for each vehicle i. We use the training algorithm Model-Free Neural Lyapunov Control (MFNLC) from Xiong et al. (2022) to stably and efficiently navigate to a short-term subgoal through the joint optimization of a neural Lyapunov function and a neural network controller in a reinforcement learning update loop. This learning approach was shown to be more efficient in learning to navigate to a goal in an environment without obstacles than standard reinforcement techniques such as PPO and TD3. Training occurs in *Python* using *PyTorch*, Paszke et al. (2019), then model parameters are transferred to *Rust* via ONNX (2024) for evaluation. More details on the architecture and learning algorithm can be found in F.

Subgoal Selection Methods: Waypoints are generated using either A^* or RRT. Unless otherwise specified, A^* is used for waypoint generation. The approaches share the same set of model parameters to construct goal-conditioned RL policies. The approaches are described as follows: The **Waypoint Only (WO)** method does not consider safe navigation during runtime. The subgoal selected is fixed to the next waypoint W_j and repeated for subsequent waypoints. We consider two variants: one that follows the $A*$ waypoints and another that follows RRT waypoints. The **Model-Free Neural Lyapunov Control (MFNLC)** approach, based on the state-of-the-art method from Xiong et al. (2022), uses a neural Lyapunov function to build a runtime monitor that generates infinite-time reachable sets. These reachable sets are over-approximated as circles, and a safe subgoal is selected if its reachable set (circle) does not intersect with obstacles. The **RusTReach Fixed Control (RR FC)** method selects safe subgoals using anytime reachability analysis, where reachable sets are computed using fixed control over a finite-time horizon T_{reach}. The fixed control value u is generated using π, based on the current state x_t, and is inputted to Algorithm 2, which has been modified to replace centroid calculation with a constant control output $u := \pi(x_t)$. Finally, the **RusTReach RL Control (RR RLC) (Ours)** method, which we propose, selects safe subgoals using anytime reachability analysis, where reachable sets are computed using control from an RL policy π over a finite-time horizon T_{reach}.

Metrics: The approaches are evaluated for success using the following metrics computed over 1000 simulations (episodes): *Mean Time-to-goal (TTG)*, representing the average time (in seconds) for the vehicle to reach within a distance ϵ of the final waypoint, considering only simulations that end upon reaching the final waypoint; *Failure ratio*, measuring instances where the simulation either reaches T_{max}, terminates early due to failure to identify a safe subgoal (*irrecoverable*), or collides with an obstacle (*unsafe*), limited to one per simulation and calculated as $($*timeouts* + *no sg* + *collisions*$)/1000$; *Mean Subgoal (SG) Selection Time*, the average time (in milliseconds) to select the next subgoal at each timestep; and *Missed Deadline Ratio*, which is the ratio of timesteps where the SG selection time exceeds $T_{sgMax} = 100$ ms, out of a total of over 30K timesteps.

4.2. Experimental Results

The results in Table 1 show that the WO (A^*) approach achieves the fastest mean TTG but suffers from a high failure rate, primarily due to collisions. For WO (RRT), reducing waypoint spacing decreases failures, but still results in a significant number, while also leading to more conservative travel. Our approach, RR RLC, minimizes failures while achieving a mean TTG that is > 1.7 times faster than MFNLC and WO (RRT). Both RR RLC and MFNLC experience increased failures in the presence of dynamic obstacles; however, MFNLC exhibits more than twice the number of failures compared to RR RLC. We attribute this to MFNLC's conservative nature, which reduces navigation efficiency and often results in timeouts or oscillations near obstacles. This issue is further highlighted in Appendix C, where MFNLC struggles to navigate paths in close proximity to obstacles. We seek to mitigate the number of failures from RR RLC considering dynamic obstacles in future work for increased safety. Additionally, RR RLC exhibits fewer failures than RR FC with dynamic obstacles, likely due to its improved ability to track future states accurately. This is

Obstacles	Method	Mean TTG (s)	Failure Ratio	Mean SG Selection Time (ms)	Missed Deadline Ratio
Static	WO (A^*)	**13.531**	0.338	-	-
	WO (RRT)	33.731	0.133	-	-
	MFNLC	30.538	0.004	-	-
	RR FC	18.001	0.001	10.403	**0.000**
	RR RLC (Ours)	18.013	**0.000**	**8.391**	**0.000**
Dynamic	WO (A^*)	**12.876**	0.570	-	-
	WO (RRT)	33.655	0.155	-	-
	MFNLC	40.232	0.163	-	-
	RR FC	18.895	0.149	12.019	**0.000**
	RR RLC (Ours)	19.718	**0.066**	**11.363**	**0.000**

Table 1: Quadcopter-Neighborhood Experiment Results. We evaluate 1000 start and goal position pairs each with a sequence of waypoints. The task is to navigate starting from the first waypoint (start position) through the sequence of waypoints to the final waypoint (goal position), while avoiding static or dynamic obstacles. The best value for each metric is highlighted in bold.

further exacerbated in the *Corridor* environment in Appendix C. Notably, RR RLC has a slightly lower mean subgoal selection time compared to RR FC. This may be due to RR RLC computing reachable sets for fewer subgoal candidates in cases where RR FC inaccurately predicts a subgoal as unsafe, leading to extra subgoal candidate computations. Finally, we note that both RR RLC and RR FC minimize deadline violations to satisfy real-time constraints.

5. Conclusion

We propose an approach for safe autonomous navigation of a NeSy RL agent between waypoints in new environments with obstacles. Our method employs a safe subgoal selection algorithm using reachability analysis to construct goal-conditioned RL policies for collision-free control. The algorithm considers waypoints from a symbolic planner, system dynamics, and hyper-rectangles for reachable states—alongside a RL component that learns model parameters through goal-conditioned RL. Subgoals are selected from candidates whose reachable sets are computed, ensuring obstacle-free navigation while minimizing distance to the next waypoint. Reachable states over a finite-time horizon are computed in an anytime manner using RL-based control for real-time compliance. We implement our algorithm in *Rust* and develop *RusTReach*, an open-source software package for real-time reachability analysis. We evaluate against four approaches in a long-term quadcopter navigation task, our method is > 1.7 faster at traversing through waypoints on average while being more robust to static and dynamic obstacles than a state-of-the-art safe navigation approach while reducing navigation failures (collisions, timeouts, and irrecoverable states) and minimizing deadline violations. However, there are still a high amount of failures when considering dynamic obstacles that we seek to mitigate in future work. Results further highlight the necessity of RL-based control, as fixed control increases navigation failures.

This material is based upon work sponsored by the Air Force Research Lab (AFRL) and DARPA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFRL or DARPA.

References

Kamal Acharya, Waleed Raza, Carlos Dourado, Alvaro Velasquez, and Houbing Herbert Song. Neurosymbolic reinforcement learning and planning: A survey. *IEEE Transactions on Artificial Intelligence*, 5(5):1939–1953, 2024. doi: 10.1109/TAI.2023.3311428.

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18, New Orleans, Louisiana, USA, 2018. AAAI Press. ISBN 978-1-57735-800-8.

Somil Bansal and Claire J. Tomlin. Deepreach: A deep learning approach to high-dimensional reachability. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1817–1824, 2021. doi: 10.1109/ICRA48506.2021.9561949.

John Bobzwik. Quadcopter simcon, 2024. URL https://github.com/bobzwik/Quadcopter_SimCon.

Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. Juliareach: a toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 39–44, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362825. doi: 10.1145/3302504.3311804. URL https://doi.org/10.1145/3302504.3311804.

Zhixi Cai, Cristian Rojas Cardenas, Kevin Leo, Chenyuan Zhang, Kal Backman, Hanbing Li, Boying Li, Mahsa Ghorbanali, Stavya Datta, Lizhen Qu, et al. Neusis: A compositional neuro-symbolic framework for autonomous perception, reasoning, and planning in complex uav search missions. *arXiv preprint arXiv:2409.10196*, 2024.

Charles Dawson, Sicun Gao, and Chuchu Fan. Safe control with learned certificates: A survey of neural lyapunov, barrier, and contraction methods for robotics and control. *IEEE Transactions on Robotics*, 39(3):1749–1767, 2023.

Chao Huang, Jiameng Fan, Xin Chen, Wenchao Li, and Qi Zhu. Polar: A polynomial arithmetic framework for verifying neural-network controlled systems. In *Automated Technology for Verification and Analysis: 20th International Symposium, ATVA 2022, Virtual Event, October 25–28, 2022, Proceedings*, page 414–430, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-19991-2. doi: 10.1007/978-3-031-19992-9_27. URL https://doi.org/10.1007/978-3-031-19992-9_27.

Taylor T Johnson, Stanley Bak, Marco Caccamo, and Lui Sha. Real-time reachability for verified simplex design. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(2):1–27, 2016.

Junsu Kim, Younggyo Seo, and Jinwoo Shin. Landmark-guided subgoal generation in hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 34:28336–28349, 12 2021.

Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson. Nnv 2.0: The neural network verification tool. In *Computer Aided Verification: 35th International Conference,* *CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*, page 397–412, Berlin, Heidelberg, 2023. Springer-Verlag. ISBN 978-3-031-37702-0. doi: 10.1007/978-3-031-37703-7_19. URL https://doi.org/10.1007/978-3-031-37703-7_19.

Patrick Musau, Nathaniel Hamilton, Diego Manzanas Lopez, Preston Robinette, and Taylor T Johnson. On using real-time reachability for the safety assurance of machine learning controllers. In *2022 IEEE International Conference on Assured Autonomy (ICAA)*, pages 1–10, Red Hook, NY, USA, 2022. Curran Associates Inc.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3307–3317, Red Hook, NY, USA, 2018. Curran Associates Inc.

Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, et al. F1/10: An open-source autonomous cyber-physical platform. *arXiv preprint arXiv:1901.08567*, 1(1):1–10, 2019.

ONNX. Open neural network exchange, 2024. URL https://github.com/onnx/onnx.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, and et al. Bradbury. *PyTorch: an imperative style, high-performance deep learning library*, page 8026–8037. Curran Associates Inc., Red Hook, NY, USA, 2019.

Andrea Peruffo, Daniele Ahmed, and Alessandro Abate. Automated and formal synthesis of neural barrier certificates for dynamical models. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 370–388, Berlin, Heidelberg, 2021. Springer.

Nicholas Potteiger and Xenofon Koutsoukos. Safeguarding autonomous uav navigation: Agent design using evolving behavior trees. In *2024 IEEE International Systems Conference (SysCon)*, pages 1–8, Red Hook, NY, USA, 2024. Curran Associates Inc.

Bernardo Martinez Rocamora, Paulo V. G. Simplicio, and Guilherme A. S. Pereira. A behavior tree approach for battery-aware inspection of large structures using drones. In *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 234–240, 2024. doi: 10.1109/ICUAS60882.2024.10557083.

Felippe Schmoeller Roza, Karsten Roscher, and Stephan Günnemann. Safe and efficient operation with constrained hierarchical reinforcement learning. In *Sixteenth European Workshop on Reinforcement Learning*, volume 16, pages 1–10, Munich, Germany, 09 2023. Fraunhofer-Publica.

Francesco Sabatino. Quadrotor control: modeling, nonlinearcontrol design, and simulation. Master's thesis, KTH Royal Institute of Technology, 2015.

Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, volume 5, pages 621–635, Cham, Switzerland, 2017. Springer, Cham.

Ayushi Sharma, Shashank Sharma, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state, challenges and open problems. *arXiv preprint arXiv:2311.05063*, 1:1–31, 2023.

Sumeet Singh, Spencer M Richards, Vikas Sindhwani, Jean-Jacques E Slotine, and Marco Pavone. Learning stabilizable nonlinear dynamics with contraction-based regularization. *The International Journal of Robotics Research*, 40(10-11):1123–1150, 2021.

Garrett Thomas, Yuping Luo, and Tengyu Ma. Safe reinforcement learning by imagining the near future. *Advances in Neural Information Processing Systems*, 34:13859–13869, 2021.

Yixuan Wang, Weichao Zhou, Jiameng Fan, Zhilu Wang, Jiajun Li, Xin Chen, Chao Huang, Wenchao Li, and Qi Zhu. Polar-express: Efficient and precise formal reachability analysis of neural-network controlled systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 43(3): 994–1007, November 2023. ISSN 0278-0070. doi: 10.1109/TCAD.2023.3331215. URL https://doi.org/10.1109/TCAD.2023.3331215.

Xingpeng Xia, Jason J Choi, Ayush Agrawal, Koushil Sreenath, Claire J Tomlin, and Somil Bansal. Gait switching and enhanced stabilization of walking robots with deep learning-based reachability: A case study on two-link walker. *arXiv preprint arXiv:2409.16301*, 2024.

Zikang Xiong, Joe Eappen, Ahmed H. Qureshi, and Suresh Jagannathan. Model-free neural lyapunov control for safe robot navigation. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5572–5579, Red Hook, NY, USA, 2022. Curran Associates Inc. doi: 10.1109/IROS47612.2022.9981632.

Tianren Zhang, Shangqi Guo, Tian Tan, Xiaolin Hu, and Feng Chen. Generating adjacency-constrained subgoals in hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21579–21590, 2020.

Boyu Zhou, Fei Gao, Luqi Wang, Chuhao Liu, and Shaojie Shen. Robust and efficient quadrotor trajectory generation for fast autonomous flight. *IEEE Robotics and Automation Letters*, 4(4): 3529–3536, 2019. doi: 10.1109/LRA.2019.2927938.

Appendix A. Related Work

Several existing approaches for autonomous agents with RL components have been developed to satisfy safety constraints during deployment.

Design-time approaches seek to provide strong theoretical guarantees to the system, so the system performs as expected during deployment. Methods seek to compute over-approximations of the control output from a neural network controller to construct a set of reachable states. The set of reachable states can then be used to verify that safety violations do not occur. There are two types of over-approximations: output range approximations and functional over-approximations. In Lopez et al. (2023) and Bogomolov et al. (2019) they estimate the output of a neural network using an approximated range and, while efficient, can lead to large over-approximation errors in complex nonlinear systems. Huang et al. (2022) demonstrate that a functional approximation more accurately models the full input-output dependencies by using Taylor models as function approximations. Functional approximations can successfully construct the reachable set for complex tasks with nonlinear dynamics, but there is a tradeoff of higher computational cost. This computational cost was reduced in Wang et al. (2023) through parallelization, increasing time efficiency, but still has relatively high computational demand making it infeasible to compute in real-time.

Towards safety during deployment, a class of methods known as shielding or intervention can be employed to guard the system from making unsafe actions. Some methods such as Johnson et al. (2016) and Musau et al. (2022) predict potentially unsafe control output from a neural network controller and will switch to a verified safe controller if needed. Both approaches use real-time reachability to determine the switching condition. Instead of switching to a safe controller, other methods, such as Roza et al. (2023) and Potteiger and Koutsoukos (2024), optimize an auxiliary policy to perturb the raw control output to satisfy safety constraints. These two approaches also use goal-conditioned reinforcement learning to optimize for safe navigation to long-term goals. Shielding is effective when designed carefully but it becomes increasingly difficult to create a verified safe controller or optimally perturb control in complex nonlinear systems.

Furthermore, safety can be incorporated into the training algorithm of the neural network through learning safety certificates. Safety certificates such as Lyapunov, Hamilton-Jacobi reachability analysis, barrier, and contraction functions are approximated and optimized with a neural network controller to learn safety properties towards stability, safety, and trajectory tracking. Neural Lyapunov functions, such as in Xiong et al. (2022), can be jointly optimized with learning-enabling the controller to certify stability properties are satisfied. DeepReach, Bansal and Tomlin (2021), proposes a neural PDE solver, to address scalability issues with traditional methods, in Hamilton-Jacobi reachability analysis for analyzing the stability of high-dimensional systems. Xia et al. (2024) extend DeepReach to stabilize gait switching in a bipedal locomotion robot. Neural barrier functions in Dawson et al. (2023) and Peruffo et al. (2021) certify that a system remains within a specified safe region. Contraction methods, such as in Dawson et al. (2023) and Singh et al. (2021), certify the ability of a system to track a given trajectory. These methods are beneficial for efficiently learning properties to minimize safety violations for robustness at deployment but lack the guarantees of design-time approaches.

Xiong et al. (2022) is the most similar to our approach. The authors jointly optimize, via reinforcement learning, a neural Lyapunov function with a goal-conditioned reinforcement learning policies to navigate to short-term subgoals. The neural Lyapunov function is used in a runtime monitor to over-approximate the set of reachable states, over infinite-time, as a circle conditioned on an intermediate subgoal as a sink. The monitor selects a subgoal on an A^* path at each timestep, where the reachable set (circle) does not intersect with obstacles. The evaluation of their method demonstrated safety and effectiveness in navigating to the target goal. However, the representation of the set of reachable states as a circle may lead to a large over-approximation error leading to conservative navigation. We seek to make our navigation less conservative, while still maintaining safety in our approach.

Appendix B. Algorithm Details

In our algorithm, safe subgoals are selected in real-time by using an anytime reachability algorithm to determine which subgoals will lead to future states that do not intersect with obstacles and minimize the distance to desired waypoints. The anytime reachability algorithm is based on the algorithm in Johnson et al. (2016), that constructs the set of reachable states, represented as hyper-rectangles, over a finite-time horizon in real-time assuming fixed control. The anytime property bounds the reachability computation to execute below a specified runtime budget aiding in constraining subgoal selection execution to real-time. Furthermore, to more accurately model the set of reachable states, we extend their algorithm to use control from a RL policy over a receding finite-time horizon.

B.1. Safe Subgoal Selection Algorithm

Algorithm 1: Safe Subgoal Selection using Real-Time Reachability

```
Inputs  : state $x_t$, start $W_i$, end $W_j$, subgoal candidate number $n_{cand}$, model parameters $\eta$,
          vehicle dynamics $\dot{x} = f(x,u)$, reachtime $T_{reach}$, step size $h$, maximum runtime
          $T_{sgMax}$, obstacles $\Lambda_0$, obstacle dynamics $\dot{\Lambda} = f(\Lambda_0, t)$
Outputs: Bool subgoalFound, Goal subgoal
subgoal ← null;
subgoalFound ← false;
Goal[] sgs ← generateSubgoalCands($x_t, W_i, W_j, n_{cand}$);   // Generate in priority
 order, closest to $W_j$ first
$T_{runtime} \leftarrow T_{sgMax}/n_{cand}$;
for $g \in sgs$ do
    $\pi_g \leftarrow \pi(x; g, \eta)$;                // Construct policy $\pi_g$ from $g$ and $\eta$
    $\mathbf{R}_{[0,T_{reach}]} \leftarrow$ anytimeReach($x_t, \pi_g, \dot{x}, T_{reach}, h, T_{runtime}$);      // Execute
     Algorithm 3
    if $\forall t \in [0, T_{reach}],\ f(\Lambda_0, t) \cap \mathbf{R}_t == \emptyset$ then
        subgoal ← $g$;
        subgoalFound ← true;
        return subgoalFound, subgoal;
    end
end
return subgoalFound, subgoal;
```

B.2. Generation of Subgoal Candidates

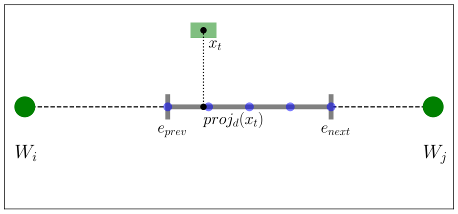


Figure 3: $n_{cand} = 5$ subgoal candidates (blue) generated and divided equally on the segment (gray) on the line (W_i, W_j). The endpoints (e_{prev}, e_{next}) of the segment are distances d_{prev} and d_{next} from the projection, $proj_d(x_t)$, of state x_t onto the vector with direction $d = W_j - W_i$.

B.3. Reachable Set Computation Algorithm

Algorithm 2: Reachable Set Computation using RL Control

```
Inputs  : state $x_t$, policy $\pi$, dynamics $\dot{x} = f(x,u)$, reachtime $T_{reach}$, step size $h$
Outputs: Box[] reachSet
Box[] reachSet;
$D \leftarrow$ length($x_t$);
Box currentRect ← initialRect($x_t$);                    // Initialize with 0 width
reachSet.add(currentRect);
$T_{reachRemain} \leftarrow T_{reach}$;
while $T_{reachRemain} > 0$ do
    $u \leftarrow \pi$(centroid(currentRect));
    Box[$D \times 2$] nebs ← nbds(currentRect, $h, d(r, i \mid u, \dot{x})$);
    crt ← minCrossReachTime(nebs, $d(r, i \mid u, \dot{x})$);
    advReachTime ← min(crt, $T_{reachRemain}$);
    currentRect ← advR(nebs, advReachTime, $d(r, i \mid u, \dot{x})$);
    $T_{reachRemain} \leftarrow T_{reachRemain}$ − advReachTime;
    reachSet.add(currentRect);
end
return reachSet;
```

We follow the algorithm from Johnson et al. (2016), by constructing neighborhoods, in the *nbds* function, for each face in *currentRect* considering the expansion and translation of *currentRect* based on the derivative . For a hyper-rectangle, there are two faces per dimension and thus there are two neighborhoods that are constructed based on the reach timestep h and derivative bounds along each face index i, given the current hyper-rectangle *currentRect* and control u. The derivative bounds are computed using a user-defined function $der(r, i \mid u, \dot{x})$ that inputs a hyper-rectangle r and face index i given control u and dynamics $\dot{x}$. The function outputs the derivative scaler-value for face index i.

Then the minimum reach-time, *crt*, for any point to travel across any of the neighborhoods in the corresponding direction is computed in the *minCrossReachTime* function. This is computed by looking at the minimum and maximum derivative within the hyper-rectangle for each neighborhood, utilizing the user-defined derivative bounds function and the width of the neighborhood at the corresponding dimension.

Finally, the next *currentRect* is computed based on the neighborhoods and computed reach-time to advance in the *advR* function, calculated as *advReachTime* using *crt* but may be reduced if it exceeds the remaining reach-time $T_{reachRemain}$. This is computed by advancing each face by the maximum derivative in the outward direction (expansion of the hyper-rectangle) in its neighborhood, using the user-defined derivative bounds function, then multiplying by *advReachTime*. The next *currentRect* is computed and added to the existing sequence of hyper-rectangles *reachSet* until the desired reach-time T_{reach} has been advanced ($T_{reachRemain} < 0$).

B.4. Anytime Reachable Set Computation Algorithm

Algorithm 3: Anytime Reachable Set Computation

```
Inputs  : state $x_t$, policy $\pi$, dynamics $\dot{x} = f(x, u)$, reachtime $T_{reach}$, step size $h$, runtime
          $T_{runtime}$
Outputs: $\mathbf{R}_{[0,T_{reach}]}$
$\mathbf{R}_{[0,T_{reach}]} \leftarrow \emptyset$;
elapsedTime $\leftarrow 0$;
$T_{remaining} \leftarrow T_{runtime}$;
startTime $\leftarrow$ now();
$h_{cur} \leftarrow h$;
nextIterEstimate $\leftarrow 0$;
while $T_{remaining} > 0$ or $h_{cur} < 1\mathrm{e}{-7}$ do
    $\mathbf{R}_{[0,T_{reach}]} \leftarrow$ reachSet($x_t, \pi, \dot{x}, T_{reach}, h$);          // Execute Algorithm 2
    elapsedPrev $\leftarrow$ elapsedTime;
    elapsedTime $\leftarrow$ now() $-$ startTime;
    prevIterEstimate $\leftarrow$ elapsedTime $-$ elapsedPrev;
    if prevIterEstimate $\times 2 + 1 <$ nextIterEstimate then
        nextIterEstimate $\leftarrow$ nextIterEstimate $\times 2$;
    end
    else
        nextIterEstimate $\leftarrow$ prevIterEstimate $\times 2 + 1$;
    end
    $T_{remaining} \leftarrow T_{runtime}$ $-$ elapsedTime $-$ nextIterEstimate;
    $h_{cur} \leftarrow h_{cur}/2$;
end
return $\mathbf{R}_{[0,T_{reach}]}$;
```

B.5. Collision Avoidance & Safe Subgoal Selection

Safety is achieved by ensuring that the system will not collide with a set of static obstacles Λ. Static obstacles are represented as rectangles to trivially check for collisions. Avoiding collisions requires that the reachable set $\mathbf{R}_{[0,T_{reach}]}$ for the subgoal selected $\hat{g}^*$ at each timestep must not intersect with obstacles to satisfy Definition 2. To select the safe subgoal, in Algorithm 1, the set of n_{cand} subgoal candidates are iterated through. For each subgoal candidate g_i, the reachable set

$\mathbf{R}_{[0,T_{reach}]}$ is computed as a sequence hyper-rectangles. Each hyper-rectangle r_i in $\mathbf{R}_{[0,T_{reach}]}$ is checked for intersection with any obstacle in Λ. If no hyper-rectangle intersects with the obstacles, then Definition 2 is satisfied and the subgoal g_i is returned. If no subgoal satisfies the condition, then the state is *irrecoverable* and the algorithm returns that a subgoal was not found and external control should take over.

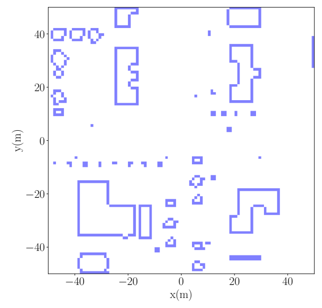
Appendix C. Corridor Environment

The *Corridor* environment is designed to evaluate traversal through a narrow space between obstacles. The two-dimensional environment in Figure 4(*b*) contains four obstacles at $[2, 0.7]$, $[2, -0.7]$, $[2, 1.4]$, $[2, -1.4]$. A vehicle starts on the left side of the obstacles and the objective is to navigate to a waypoint on the right side.

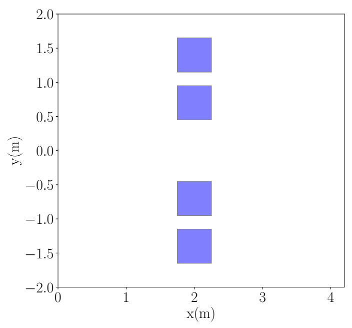
The *Corridor* experiments are designed to evaluate navigation through narrow passageways in the *Corridor* environment considering static and dynamic obstacles. We generate 1000 start waypoint, end waypoint, and vehicle start position triplets $(W_{start}, W_{end}, p_0)$. p_0 can be different from W_{start} to consider the position error from navigating to W_{start} from a previous waypoint. W_{start} is selected on the left-side bounded by $[-0.5, 0.5]^2$, W_{end} is selected on the right side bounded by $[3.5, 4.5]^2$, and p_0 is selected on the left-side bounded by $[-0.75, 0.75]^2$. We estimate if the path is feasible for the vehicle to navigate without collision by checking that the line (W_{start}, W_{end}) with a buffer for the vehicle size does not intersect with the two obstacles. If the line intersects with obstacles then we discard this triplet and re-compute until 1000 triplets have been generated. The distance threshold to reach the waypoint W_{end} is $\epsilon = 0.2$ meters.

In the *static* obstacle experiment the obstacle are fixed throughout simulation. In the dynamic obstacle experiment, the interior obstacles ($[2, 0.7]$, $[2, -0.7]$) swap positions by moving inward at 0.5m/s until they switch positions.

Appendix D. Obstacle Maps



(*a*) Neighborhood Map



(*b*) Corridor Map

Figure 4: Two environment maps for experiments, each with square obstacles (0.5 meters wide, in blue). The first is a *Neighborhood* environment, a 100×100 grid with 685 obstacle cells. The second is a *Corridor* environment with four obstacles forming a narrow passage.

Appendix E. Quadcopter Dynamical Model

The aerial vehicle is a linearized quadcopter model from Sabatino (2015). The quadcopter model was linearized, but still shown to track well with the full nonlinear dynamics. The size of the vehicle is 0.32 meters in width and length. The 12-dimensional state x contains the linear and angular positions
$\begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}$ and the linear and angular velocities
$\begin{bmatrix} u & v & w & p & q & r \end{bmatrix}$. When conditioned on a goal g, the state is transformed to $x = \begin{bmatrix} g - x_{0:3}, x_{3:12} \end{bmatrix}$. The quadcopter is controlled using vertical thrust and linear torques $\begin{bmatrix} f_t & \tau_x & \tau_y & \tau_z \end{bmatrix}$. The system is simulated using the following linear dynamics: Parameters for the quadcopter are: $g =$

$$\begin{aligned} \dot{x} &= u & \dot{u} &= -g\theta \\ \dot{y} &= v & \dot{v} &= g\phi \\ \dot{z} &= w & \dot{w} &= \tfrac{-f_t}{m} \\ \dot{\phi} &= p & \dot{p} &= \tfrac{\tau_x}{I_x} \\ \dot{\theta} &= q & \dot{q} &= \tfrac{\tau_y}{I_y} \\ \dot{\psi} &= r & \dot{r} &= \tfrac{\tau_z}{I_z} \end{aligned}$$

$9.81\ m/s^2$, $m = 1.2\ kg$, $I_x = 0.0123\ kgm^2$, $I_y = 0.0123\ kgm^2$, $I_z = 0.0224\ kgm^2$. The parameters were set to match a rough estimate of a DJI F450 quadcopter in Bobzwik (2024).

Appendix F. Optimizing Goal-Conditioned Policies

The goal-conditioned reinforcement learning approach we use is Model-Free Neural Lyapunov Control (MFNLC) from Xiong et al. (2022) that jointly optmizes a neural Lyapunov function and neural network controller. The controller is a neural network is defined with parameters η to construct a goal-conditioned policy $\pi(x; g, \eta)$. The state x is transformed to be conditioned on goal g. The controller has two hidden layer of size 32 with *ReLU* activation functions. The input size is the state size. The output size is 2 for the desired velocity a_t, bounded between $[-0.5, 0.5]^2$m/s by scaling the output of a *TanH* activation function. a_t is then inputted to a pre-configured PID controller to compute the control. In the case of the car, this is steering angle δ and thrust input u. Using an abstraction of the control such as desired velocity allows the neural network to be compatible with external interfaces in realistic simulation such as Microsoft AirSim Shah et al. (2017) that supports desired velocity input, but may not directly support our particular control input.

The Lyapunov function $V(x)$ is also a neural network where the input size is the state size, output size is 1, and two hidden layers of size 64. The Lyapunov function is optimized towards properties that when incorporated into the update step in the loss computation of π_{gc} increases stabilization and training efficiency for navigation to a goal position. The properties are as follows:

$$\begin{aligned} &V(x_o) = 0 \\ &V(x_t) > 0 \\ &V(x_{t+1}) - V(x_t) < 0 \end{aligned}$$

The properties ensure that at a sink x_o, in this case the goal position, the output of the Lyapunov funciton is zero and for all other states the value is strictly positive. The third property ensures that the values decreases between two timesteps encouraging the system to stabilize at the sink.

Intuitively, this means the vehicle would travel closer to the goal (sink) for each subsequent timestep eventually converging to the goal (sink).

For each vehicle i, a set of parameters η_i is trained for a neural network controller. The parameters are optimized for 1000 simulation episodes, where there are 100 timesteps per episode. Each episode is initialized with a randomized start and fixed goal in a 6×6 meter space with no obstacles. η_i is trained to minimize the distance to the target and ensure that the desired velocity $a_t := \pi(x_t; g_t, \eta_i)$ aligns with the goal g_t relative to the position of vehicle i (p_t) from the state x_t. More formally, the reward function is:

$$rew(p_t, a_t) = ||a_t - (g_t - p_t)||_2 - ||g_t - p_t||_2$$

The joint training of the Lyapunov function $V(s_t)$ is incorporated into the loss function to update η_i to provide stability.

Appendix G. Improving tracking of reachable states using RL control

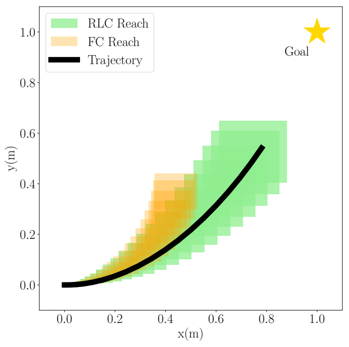


Figure 5: Set of reachable states over a finite-time horizon $T_{reach} = 2.0s$ and step size $h = 0.1$ for the autonomous car, given a goal-conditioned policy π_g where $g = [1, 1]$. Two control methods for the reachable set computation algorithm are compared, fixed control (FC) in orange where control is constant (original approach, Johnson et al. (2016)) and RL control (RLC) in green, our extension, where the control is predicted over the finite-time horizon. We find FC stops tracking the ground truth trajectory a short time after the initial state, while RLC tracks for the entire time-horizon.

Appendix H. Simulation Parameters

The control period for each vehicle is a *timestep* of $dt = 0.1$ seconds (100 milliseconds), during which a subgoal is selected, and control inputs are generated. There is a maximum of $T_{max} = 200$ timesteps (20s) in *Corridor* and $T_{max} = 1000$ timesteps (100s) in *Neighborhood*.

At each timestep, there is a deadline of $T_{sgMax} = 100$ms to compute the next subgoal. We set the reach time $T_{reach} = 2.0$s for *Corridor* and $T_{reach} = 1.0$s for *Neighborhood* to allow sufficient time to correct potentially unsafe actions before reaching an irrecoverable state, where a collision is unavoidable. The initial step size is $h = 0.1$ to reflect the control period, but can decrease if the anytime reachability algorithm has excess runtime to re-compute the reachable set. The number of subgoal candidates we choose is $n_{cand} = 5$ for *Corridor*, where distances for constructing the line segment for selection are $d_{prev} = 4, d_{next} = 4$ meters, and $n_{cand} = 5$ for *Neighborhood*, where distances $d_{prev} = 5, d_{next} = 5$ meters.

A simulation (episode) terminates if the target/final waypoint is reached, a collision has occurred, T_{max} timesteps have elapsed, or the subgoal selection algorithm cannot find a safe subgoal due to all of the n_{cand} sugboals violating the safety condition in Definition 2.

Appendix I. Quadcopter Corridor Experiment

In the results in Table 2 for the quadcopter, the Waypoint-Only (WO) approach has the fastest TTG, but suffers from an excessive number of failures. Our approach (RR RLC) minimizes navigation failures and has a mean TTG that is more than two times faster than MFNLC. The conservative over-approximation of the reachable set in MFNLC leads to a degradation in navigation efficiency. Also, we find more failures from MFNLC in *Corridor* than the *Neighborhood* because of the close proximity to obstacles. The vehicle slows to a halt resulting in a timeout or drift near the passageway between the two obstacles leading to a low-speed collision. We find that RR RLC also exhibits more failures than the *Neighborhood*, but not to the same extent. Future work should investigate improvements to further reduce navigation failures in tight navigation spaces. Additionally, the inclusion of RL control in RR RLC was shown to have fewer navigation failures than fixed control where RR FC tends to raise false negatives that there is no safe subgoal. The reduced conservatism and increased accuracy of RR RLC reachable sets results in safer and faster navigation in this narrow passageway environment.

Obstacles	Method	Mean TTG (s)	Failure Ratio	Mean SG Selection Time (ms)	Missed Deadline Ratio
Static	WO	**3.574**	0.264	-	-
	MFNLC	17.721	0.820	-	-
	RR FC	3.473	0.539	31.694	**0.000**
	RR RLC (Ours)	4.064	**0.118**	**24.278**	<0.001
Dynamic	WO	**2.700**	0.999	-	-
	MFNLC	17.725	0.819	-	-
	RR FC	5.741	0.988	69.764	**0.000**
	RR RLC (Ours)	5.429	**0.183**	**40.133**	<0.001

Table 2: Quadcopter-Corridor Experiment Results. 1000 triplets (start waypoint, end waypoint, vehicle start position) are generated and checked for feasibility. The task is to navigate to the end waypoint from the vehicle start position, while avoiding static or dynamic obstacles. The best value for each metric is highlighted in bold.

Appendix J. Experiments with Car Vehicle

We conduct additional experiments with a ground vehicle that has nonholonomic constraints to evaluate the generalization of our approach to different types of vehicles. The vehicle is an autonomous car modeled with bicycle dynamics.

J.1. Car Dynamical Model

The ground vehicle is a kinematic bicycle model set to track an autonomous car in the F1/10 platform O'Kelly et al. (2019): an autonomous vehicle simulation platform. The size of the vehicle is

0.5 meters in length and 0.25 in width. The four-dimensional state x contains the position, speed, and heading angle $\begin{bmatrix} x & y & v & \theta \end{bmatrix}$. When conditioned on a goal g, the state is transformed to $x = \begin{bmatrix} g - x_{0:2}, x_{2:4} \end{bmatrix}$ to consider the goal g relative to the global position $x_{0:2}$. The F1/10 car is controlled using a steering and thrust input $\begin{bmatrix} \delta & u \end{bmatrix}$ The system is simulated using the following nonlinear dynamics: Parameters for the F1/10 car are: $c_a = 1.9569$, $c_m = 0.0342$, $c_h = -37.1967$, $l_f = 0.225$,

$$\begin{aligned} \dot{x} &= vcos(\theta), & \dot{\theta} &= \tfrac{v}{l_f + l_r} tan(\delta), \\ \dot{y} &= vsin(\theta), & \dot{v} &= -c_a v + c_a c_m (u - c_h), \end{aligned}$$

$l_r = 0.225$. The parameters were estimated using MATLAB's Grey-Box System Identification Tool in Musau et al. (2022).

J.2. Neighborhood Experiment

The results in Table 3 are similar to the Quadcopter-Neighborhood experiment. We find that RR RLC minimizes navigation failures with low subgoal selection time. We note that while MFNLC and RR RLC increase in failures for dynamic obstacles, the MFNLC has more failures with the car more than the quadcopter. We suspect this is due to the slow turning rate of the car vehicle making it more challenging to compute adaptive control near obstacles.

Obstacles	Method	Mean TTG (s)	Failure Ratio	Mean SG Selection Time (ms)	Missed Deadline Ratio
Static	WO (A^*)	**9.566**	0.361	-	-
	WO (RRT)	18.598	0.161	-	-
	MFNLC	23.627	0.018	-	-
	RR FC	13.172	0.020	10.796	**0.000**
	RR RLC (Ours)	12.491	**0.000**	**9.373**	**0.000**
Dynamic	WO (A^*)	**9.306**	0.579	-	-
	WO (RRT)	18.342	0.366	-	-
	MFNLC	20.816	0.794	-	-
	RR FC	13.839	0.215	13.153	**0.000**
	RR RLC (Ours)	13.737	**0.043**	**9.768**	**0.000**

Table 3: Car-Neighborhood Experiment Results. We evaluate 1000 start and goal position pairs each with a sequence of waypoints generated from A^*. The task is to navigate starting from the first waypoint (start position) through the sequence of waypoints to the final waypoint (goal position), while avoiding static or dynamic obstacles. The best value for each metric is highlighted in bold.

J.3. Corridor Experiment

The success of RR RLC is shown in the car vehicle, similar to the quadcopter, in Table 4. We observe a faster mean TTG and fewer failures across all approaches, suggesting that navigating the narrow passageway was easier with the car than with the quadcopter. The quadcopter's rapid movement and lack of friction may introduce challenges when maneuvering through tight spaces. From this experiment, we conclude that RR RLC is both efficient and safe for traversing narrow corridors.

Obstacles	Method	Mean TTG (s)	Failure Ratio	Mean SG Selection Time (ms)	Missed Deadline Ratio
Static	WO	**2.200**	0.228	-	-
	MFNLC	5.915	0.570	-	-
	RR FC	2.564	0.111	16.169	**0.000**
	RR RLC (Ours)	2.498	**0.045**	**12.647**	$<$0.001
Dynamic	WO	**2.200**	0.999	-	-
	MFNLC	6.042	0.609	-	-
	RR FC	5.191	0.483	32.462	$<$**0.001**
	RR RLC (Ours)	5.014	**0.056**	**29.818**	$<$**0.001**

Table 4: Car-Corridor Experiment Results. 1000 triplets (start waypoint, end waypoint, vehicle start position) are generated and checked for feasibility. The task is to navigate to the end waypoint from the vehicle start position, while avoiding static or dynamic obstacles. The best value for each metric is highlighted in bold.