State Supervised Steering Function for Sampling-based Kinodynamic Planning

Pranav Atreya
University of Texas at Austin
Austin, TX, United States
pranavatreya@utexas.edu

Joydeep Biswas
University of Texas at Austin
Austin, TX, United States
joydeepb@cs.utexas.edu

ABSTRACT

Sampling-based motion planners such as RRT* and BIT*, when applied to kinodynamic motion planning, rely on steering functions to generate time-optimal solutions connecting sampled states. Implementing exact steering functions requires either analytical solutions to the time-optimal control problem, or nonlinear programming (NLP) solvers to solve the boundary value problem given the system’s kinodynamic equations. Unfortunately, analytical solutions are unavailable for many real-world domains, and NLP solvers are prohibitively computationally expensive, hence fast and optimal kinodynamic motion planning remains an open problem. We provide a solution to this problem by introducing State Supervised Steering Function (S3F), a novel approach to learn time-optimal steering functions. S3F is able to produce near-optimal solutions to the steering function orders of magnitude faster than its NLP counterpart. Experiments conducted on three challenging robot domains show that RRT* using S3F significantly outperforms state-of-the-art planning algorithms on both solution cost and runtime. We further provide a proof of probabilistic completeness of RRT* modified to use S3F.

KEYWORDS

Kinodynamic Motion Planning; Learning Steering Functions; Sampling-based Planning

ACM Reference Format:

1 INTRODUCTION AND RELATED WORK

This work tackles the kinodynamic motion planning (KMP) problem, which is the problem of computing a kinodynamically feasible motion plan that takes a robot from an initial configuration to a goal region. We begin by formally defining the KMP problem and then survey the various approaches to solving it.

Let \( X_C \) be the configuration space of the robot. The state space \( X \) is defined as the Cartesian product of \( X_C \) with \( X_D \), the set of dynamics variables needed to fully describe the dynamics of the robot at any given instance in time. \( X_D \) typically consists of time derivatives of elements of \( X_C \). Let \( U \) be the control space of the robot. The kinodynamic constraints are described by the differential equation \( \dot{x}(t) = f(x(t), u(t)) \), where \( x(t) \in X \) and \( u(t) \in U \). The KMP problem differs from the purely kinematic motion planning (KMP) problem in that the KMP problem operates only on the configuration space \( X_C \). Let \( X_{\text{obs}} \subset X \) be the set of obstacle-colliding states and let \( X_{\text{free}} = X \setminus X_{\text{obs}} \) be the set of valid states. Let \( x_{\text{init}} \in X_{\text{free}} \) be the initial state of the robot and let \( X_{\text{goal}} \subset X_{\text{free}} \) be the goal region. The objective of the KMP problem is to find a collision-free path that takes the robot from \( x_{\text{init}} \) to \( X_{\text{goal}} \) while obeying the kinodynamic constraints. The solution to the KMP problem is a mapping \( c(t) : [0, tf] \to U \) from time to control inputs such that applying \( c(t) \) starting from the state \( x_{\text{init}} \) traces out a path \( x(t) : [0, tf] \to X_{\text{free}} \) such that \( x(t_f) \in X_{\text{goal}} \). A motion plan is considered optimal if it minimizes some cost function \( C(t_f, c, \xi) \). The time-optimal solution minimizes the total time \( t_f \).

We review the state of the art approaches to solving the KMP problem, including search-based planning, sampling-based planning, and learning-based solutions.

**Search-based planning** typically involves constructing a state lattice \( G = (V, E) \) where \( V \subset X_{\text{free}} \) and the edges \( E \) are pre-defined kinodynamically feasible motion primitives [23]. This lattice can then be searched using any graph search algorithm to obtain a solution. Increasing the resolution of the lattice increases the chances that a solution can be found, but comes with an exponential increase in computational cost. Finding a set of motion primitives that work well can also be difficult. Search-based planning algorithms are resolution optimal, in that they can find solutions that are optimal with respect to the discretization used.

**Sampling-based planning** makes use of a continually improving discretization of the state space through random sampling. One of the most effective sampling-based planning algorithms is the Rapidly Exploring Random Tree (RRT) [17] algorithm. The RRT algorithm works by incrementally sampling the state space and extending the nearest vertex in the tree towards that sample. Because this extension can be made by a random propagation of controls, the RRT algorithm can be applied to kinodynamic systems. RRTs have also been integrated with machine learning approaches to solve the KMP problem. One such work employs the k-nearest-neighbors algorithm within the RRT framework to approximate the cost-to-go function and expand vertices in the tree [29]. It however suffers from lack of optimality of computed trajectories and is only demonstrated to work for simple environments. Reinforcement Learning RRT (RL-RRT) [6] trains an RL agent to do local planning and uses an RRT to guide exploration. The resulting motion plan is suboptimal and since the RL local planner is trained on particular obstacle configurations, may not generalize well to new obstacle environments. Probabilistic Roadmap RL (PRM-RL) [8] also uses RL for local planning but maps sensor observations directly to actions and does not attempt to produce optimal trajectories.
RRT and the aforementioned RRT based algorithms do not produce optimal solutions. An alternative algorithm that produces optimal solutions while maintaining the computational efficiency of RRT is the RRT* algorithm [14]. The RRT* algorithm makes use of a rewiring step to ensure that the path from the root to any vertex in the tree is optimal with respect to the connections in the tree. Because of this, the RRT* algorithm is asymptotically optimal. Many variants of the RRT* algorithm exist that have proven to work well in practice. Informed RRT* [9] improves on RRT* by ensuring that after an initial solution has been found, only states that have the potential to improve the solution are considered as candidate vertices. The BIT* algorithm [10] integrates graph-based and sampling-based planning techniques to more efficiently find and improve on solutions to the planning problem.

One caveat of optimal sampling-based algorithms including RRT* and BIT* is that they all require an optimal steering function to connect states. For any two states \( x_a, x_b \in X \), a steering function \( S(x_a, x_b) \) produces a trajectory \( T : [0, t_f] \rightarrow U \), a mapping from time to control inputs. Integrating \( T \) from \( x_a \) according to the equation of motion \( f \) produces a path \( \Gamma : [0, t_f] \rightarrow X \), a mapping from time to states. An optimal steering function \( S^*(x_a, x_b) \) produces a trajectory \( T^* : [0, t_f] \rightarrow U \) and a path \( \Gamma^* : [0, t_f] \rightarrow X \) that in addition to satisfying the aforementioned constraints, satisfies \( \Gamma^*(t_f) = x_b \) and minimizes some cost function, most commonly time. There exist algorithms like Stable-Sparse RRT (SST) [19] and Asymptotically Optimal RRT (AO-RRT) [12] that do not require a steering function, but in practice they tend to take a significant amount of time to find good quality solutions. Analytical solutions to the steering function exist for some robots, such as those with linear dynamics [28], and so do iterative solutions for specific systems such as omnidirectional robots with bounded acceleration [2], but for most systems computing the optimal steering function requires a call to a computationally expensive nonlinear programming (NLP) solver. There are ways to decrease the computational overhead of NLP solvers to make planning tractable [30], but the NLP solver still remains a significant bottleneck. Previous work has explored whether the steering function can be learned [32]. The learning setup used however was unable to connect arbitrary start and goal states, a necessity if the steering function is to be used in an optimal sampling-based planning algorithm.

Reinforcement learning has also been applied to the KDMP problem. One approach to KDMP for linear systems uses continuous-time Q-learning [16] to deal with dynamics whose differential equations of motion are inaccurate or unreliable. Some have also proposed formulating the KDMP problem entirely as a Markov Decision Process (MDP), where the solution KDMP policy is learned by RL [5].

Optimal control policies is a research area that has also been recently explored. Past works [11] [27] [25] [26] have attempted to train a neural network to learn to produce optimal controls. All of these works however keep the goal state fixed, and so a new policy would need to be learned for every goal state. Optimization-based planning methods rely on numerical optimization to find a solution to the goal that minimizes some cost objective. Example works that fall under this category include GuSTO [4], CHOMP [24], and STOMP [13]. While such optimization-based methods are effective at finding solutions given good initialization, they find difficulty in handling cases where initial solutions are unknown, or when the optimization objective function has local minima (often due to obstacles).

Integrated planning and learning approaches have received significant attention lately. Search on the Replay Buffer (SoRB) [7] demonstrates how the success rate of goal-conditioned RL on long horizon tasks can be improved by adding a planning component. SoRB however is unable to provide theoretical guarantees on completeness and faces difficulty when run on unseen environments. One approach [1] uses precomputation and machine learning to enable real-time kinodynamic planning for quadrotors. It is able to avoid solving two-point boundary value problems directly on quadrotor dynamics by using minimum snap polynomial splines, a technique that only works for a limited class of systems. Model-Predictive Motion Planning Networks (MPC-MNet) [18] proposes the integration of multiple neural components along with Model Predictive Control to solve the kinodynamic motion planning problem. The algorithm is compared with SST and is shown to have faster planning times. It however is unable to produce lower cost paths than SST and drops in performance on unseen environments. While many approaches exist for kinodynamic planning, none so far are able to find low cost solutions in a computationally efficient manner. Approaches either sacrifice low solution cost or performance in pursuit of the other. We propose with this work that both are attainable. In contrast to many learning approaches, our work is also agnostic to obstacle configurations, and so generalizes well to new environments.

In summary, in this paper we contribute: 1) State Supervised Steering Function (S3F), a learning-based technique to efficiently compute the steering function required by optimal sampling-based planners; 2) S3F-RRT*, a probabilistically complete RRT* algorithm that uses S3F as its steering function; and 3) Empirical results for three kinodynamically-complex robots that demonstrate that S3F-RRT* outperforms state-of-the-art kinodynamic planners.

## 2 Kinodynamic Planning with State Supervised Steering Function

Recall from earlier that given two arbitrary states \( x_a, x_b \in X \) the optimal steering function \( S^*(x_a, x_b) \) produces a trajectory \( T^* \) that optimally connects these two states. We are interested in learning a function \( S \) that approximates \( S^* \) such that \( S(x_a, x_b) \) produces a near-optimal trajectory \( \hat{T} \equiv T^* \). The control trajectory \( \hat{T} \) can be integrated to obtain a path \( \Gamma \).

### 2.1 Steering Function Formulation

Rather than learning \( S \) that produces \( \hat{T} \) directly, we simplify the learning problem by constructing \( \hat{T} \) in an iterative manner. This can be done by learning a policy \( \pi : X \times X \rightarrow U \) where \( \pi \) takes as input the current state of the robot \( x_t \) and the goal state \( x_g \) and produces as output a constant-time control input \( u \) to be executed for a fixed period of time \( t \), resulting in a new state \( x_{t+1} \). Iteratively calling \( \pi \) for a fixed number of iterations \( n \) results in the generation of a piecewise constant control function that we denote \( T_{\max} \). Integrating \( T_{\max} \) from the start state \( x_a \) yields the state function \( \Gamma_{\max} \).
\( \hat{T} \) can be obtained from \( T_{\text{max}} \) by discarding from \( T_{\text{max}} \) all controls past the time when the robot has reached the goal. To be able to do this, \( n, \tau \), the duration of \( T_{\text{max}} \), needs to be greater than the time it takes to connect any two states in \( X \) optimally. The next step is to determine when \( T_{\text{max}} \) actually reaches the goal. The naive approach is to simply select the time at which \( T_{\text{max}} \) is closest to \( x_b \) where closeness is defined using Euclidean distance. The problem with this approach is that our trajectories not only need to reach the goal but also be optimal with respect to the time to goal. Let’s say for one particular trajectory the robot reaches a distance \( d_1 \) from the goal at time \( t_1 \) and a distance \( d_2 \) from the goal at time \( t_2 \). If \( d_2 \) is the closest distance, then we are guaranteed to pick \( t_2 \) as our ending time, even if \( d_2 \) is marginally less than \( d_1 \). However it may be possible that \( t_2 \) is significantly greater than \( t_1 \), and so just to reach a little closer to the goal we’re sacrificing significant time optimality. This type of analysis motivates the solution to this problem. Since there are in essence two objectives that we are optimizing over when selecting the end time – distance to goal and time to reach goal – we should construct a reward function that fairly takes into account both. The following reward function \( R(t) \) does exactly this:

\[
R(t) = \alpha \frac{||x_a - x_b|| - ||T_{\text{max}}(t) - x_b||}{||x_a - x_b||} - t + R_b(T_{\text{max}}(t), x_b)
\]

\[
R_b(T_{\text{max}}(t), x_b) = \begin{cases} 
\beta, & \text{if } ||T_{\text{max}}(t) - x_b|| \leq \mu \\
0, & \text{otherwise}
\end{cases}
\]

The first term is a normalized difference of potential functions, and is maximized when the candidate terminal state is situated at the goal. The use of such potential functions was first introduced as a policy invariant mechanism for reward shaping [21]. The second term, \(-t\), takes into account the second objective: minimizing the time to the goal. Finally the third term provides an additional incentive if the candidate terminal state is very close (\( \leq \mu \) distance away) to the goal. The hyperparameters \( \alpha, \beta, \) and \( \mu \) are positive constants which can be tuned to adjust the relative weights of the three terms.

For all time points which this reward function is calculated, the end time will be the time with the greatest reward. \( \hat{T} \) can then be obtained by discarding all control inputs in \( T_{\text{max}} \) after the end time.

### 2.2 Learning the Policy

The previous section showed how the steering function \( \hat{S} \) can be constructed from a learned policy \( \pi \). We next present how \( \pi \) is learned.

We employ a supervised learning approach to learn \( \pi \). Since the end goal is to learn the optimal steering function, our dataset consists of solutions to the optimal steering function for a large number of start and goal states. This dataset, generated by an NLP solver, consists of a series of trajectories each described by a tuple \((T^*, \Gamma^*, t_f)\). Here \( T^* : [0, t_f] \rightarrow U \) and \( \Gamma^* : [0, t_f] \rightarrow X \) are the control and state functions introduced earlier. We used the PSOPT [3] optimal control library to generate the trajectories. The start and goal states for each trajectory in the dataset are sampled uniformly at random from the full state space to ensure good state space coverage.

To learn \( \pi \) using this dataset we employ the fact that \( \pi \) is used to generate control and state functions \( \hat{T} \) and \( \hat{\Gamma} \). The arguably simplest approach is to have \( \hat{T} \) imitate \( T^* \) for each start and goal pair in the dataset. The discrepancy in the fact that \( \hat{T} \) is piecewise constant whereas \( T^* \) is continuous can be accounted for by simply averaging controls in \( T^* \) at each length \( \tau \) time interval. \( \pi \) would then be directly supervised by the averaged constant controls in \( T^* \). Although straightforward, this approach fails to learn a well-performing policy. The primary reason for this is that the learning problem involves the approximation of a highly discontinuous function. \( \pi \) is tasked with learning the optimal control function which for many kinodynamic systems is a bang-bang control function. Figure 1a shows an example of this for the quadrotor robot – such discontinuous control functions are hard to represent and learn directly, even by supervised learning.

The solution to this problem is to not use the optimal control function \( T^* \) to supervise the learning, but to instead use the optimal
S3F-RRT(*)

1. $V \leftarrow \{x_{\text{init}}\}, E \leftarrow \emptyset$
2. for $i = 1, \ldots, n$ do
3. $x_{\text{rand}} \leftarrow \text{SampleFree}()$
4. $x_{\text{parent}} \leftarrow \emptyset, x_{\text{ext}} \leftarrow \emptyset, c_{\text{min}} \leftarrow \infty$
5. $x_{\text{near}} \leftarrow \text{NearTo}(G = (V, E), x_{\text{rand}})$
6. for each $x \in X_{\text{near}}$ do
7. $T \leftarrow \text{Steer}(x, x_{\text{rand}})$
8. $x_{\text{new}} \leftarrow \text{EndState}(x, T)$
9. $c_{\text{traj}} \leftarrow \text{SteeringCost}(T)$
10. $b \leftarrow \text{Dist}(x_{\text{new}}, x_{\text{rand}}) < r_{\text{error}} \land \text{ObstacleFree}(x, T)$
11. if $\text{Cost}(x) + c_{\text{traj}} < c_{\text{min}} \land b$ then
12. $x_{\text{parent}} \leftarrow x$
13. $x_{\text{ext}} \leftarrow x_{\text{new}}$
14. $c_{\text{min}} \leftarrow \text{Cost}(x) + c_{\text{traj}}$
15. end if
16. end for
17. if $c_{\text{min}} \neq \infty$ then
18. $V \leftarrow V \cup \{x_{\text{ext}}\}$
19. $E \leftarrow E \cup \{(x_{\text{parent}}, x_{\text{ext}})\}$
20. end if
21. $\text{Rewire}(V, E, x_{\text{ext}})$
22. end for
23. return $G = (V, E)$

Rewire($V, E, x_{\text{ext}}$)

1. $X_{\text{near}} \leftarrow \text{NearFrom}(G = (V, E), x_{\text{ext}})$
2. for each $x \in X_{\text{near}}$ do
3. $T \leftarrow \text{Steer}(x_{\text{ext}}, x)$
4. $x_{\text{new}} \leftarrow \text{EndState}(x_{\text{ext}}, T)$
5. $c_{\text{traj}} \leftarrow \text{SteeringCost}(T)$
6. $b \leftarrow \text{Dist}(x_{\text{new}}, x_{\text{ext}}) < r_{\text{error}} \land \text{ObstacleFree}(x_{\text{ext}}, T)$
7. if $\text{Cost}(x_{\text{ext}}) + c_{\text{traj}} < \text{Cost}(x) \land b$ then
8. $V \leftarrow V \setminus \{x\} \cup \{x_{\text{new}}\}$
9. $E \leftarrow E \cup \{(x_{\text{parent}}, x) \cup \{(x_{\text{ext}}, x_{\text{new}})\}$
10. $\text{PropagateRewiring}(x, x_{\text{new}})$
11. end if
12. end for

PropagateRewiring($x, x_{\text{new}}$)

1. for each $x_{\text{child}} \in \text{Children}(x)$ do
2. $T \leftarrow \text{Trajectories}(x, x_{\text{child}})$
3. if $\text{ObstacleFree}(x_{\text{new}}, T)$ then
4. $x_{\text{next}} \leftarrow \text{EndState}(x_{\text{new}}, T)$
5. $V \leftarrow V \setminus \{x_{\text{child}}\} \cup \{x_{\text{next}}\}$
6. $E \leftarrow E \cup \{(x_{\text{child}}, x_{\text{next}})\}$
7. $\text{PropagateRewiring}(x_{\text{child}}, x_{\text{next}})$
8. else
9. $\text{DeleteSubtree}(x_{\text{child}})$
10. end if
11. end for

Figure 2: S3F-RRT* Algorithm

state function $\Gamma^*$. We term this approach State Supervised Steering Function (S3F). Due to the differential equation $f$ that defines the kinodynamic constraints, state functions are guaranteed to be differentiable (and thus continuous), making learning the optimal state function a feasible problem. Figure 1b shows an example of such a state function for the quadrotor robot—note that despite the associated control function (Figure 1a) being discontinuous, the state function is smooth and continuous. The goal now is to have $\hat{\Gamma}$ imitate $\Gamma^*$ for each trajectory in the dataset. This can be done by ensuring that for various time points $t$ in the range $[0, t_f]$, $\hat{\Gamma}(t) = \Gamma^*(t)$. Recall that $\hat{\Gamma}$ is only obtained by integrating $\hat{T}$. This can be accounted for with the following procedure: sample a series of time points $(t_0, \ldots, t_k)$ in the range $[0, t_f - \tau]$. For each time point $t$, assume that the robot is currently at $\Gamma^*(t)$. If $\hat{\Gamma}$ is to imitate $\Gamma^*$, the robot should be at $\Gamma^*(t + \tau)$ at time $t + \tau$. The actual location of the robot at this time under the current policy $\pi$ can be calculated by evaluating $F(\Gamma^*(t), \pi(\Gamma^*(t), x_{t_f}))$ where $x_{t_f}$ is the goal state of the trajectory and $F : X \times U \rightarrow X$ is an integration function that given a current state and a constant control, integrates the differential equation of motion $f$ to compute the state at units of time later. To get $\hat{\Gamma}$ to imitate $\Gamma^*$ we can thus optimize the following learning objective:

$$
\arg\min_{\theta} \sum_{\Gamma \in D} \sum_{t \in (t_0, \ldots, t_k)} [F(\Gamma^*(t), \pi(\Gamma^*(t), x_{t_f})) - \Gamma^*(t + \tau)]^2
$$

where $\theta$ is the parameter set of $\pi$ and $D$ is the dataset of optimal trajectories. The key takeaway from this learning procedure is that we are learning $\pi$ indirectly. $\pi$ is a component of a state function that we are training to be optimal, and by learning this state function we are indirectly learning the control function $\pi$.

2.3 Sampling-based Planning With Learned Steering Functions: S3F-RRT*

We present S3F-RRT*, a sampling-based planning algorithm that uses the learned steering function to solve the optimal kinodynamic motion planning problem. S3F-RRT* uses S3F as the steering function, and employs a modified rewiring procedure to overcome any potential local inaccuracies in S3F’s trajectories.

Figure 2 presents the algorithmic formulation of S3F-RRT*. Figure 3 shows a visualization of what goes on in each S3F-RRT* iteration. Each iteration begins by sampling a random collision-free state $x_{\text{rand}}$. The NearTo function is then called to obtain the set of all vertices in the current RRT* tree that are near $x_{\text{rand}}$. A state is considered to be near $x_{\text{rand}}$ if the time of the optimal trajectory from that state to $x_{\text{rand}}$ is below some threshold. Each state in $X_{\text{near}}$ is then evaluated as a possible parent to $x_{\text{rand}}$. Steer($x, x_{\text{rand}}$) invokes S3F to compute a control function $T$ that connects $x$ to $x_{\text{rand}}$. To determine $x_{\text{new}}$, where the trajectory actually ends, EndState($x, T$) integrates $T$ from $x$. SteeringCost($T$) returns the cost of the trajectory $T$, which for a time-optimial planning problem is simply the duration of $T$. Cost($x$) returns the cost of going from the start state to $x$ in the current RRT* tree. The Dist function returns the Euclidean distance between two states and is used to ensure that the terminal state of the trajectory is close enough to the target state. ObstacleFree($x, T$) integrates the control function $T$ beginning at $x$ to obtain a state function that maps time to states.
then ensures that every state in this state function does not collide with obstacles.

After the best parent has been found and the state has been added to the tree, the rewiring procedure is invoked. Here, the set $X_{\text{near}}$ is constructed by calling $\text{NearFrom}(G = (V, E), X_{\text{goal}})$. The difference between $\text{NearFrom}$ and $\text{NearTo}$ is that $\text{NearFrom}(G = (V, E), X_{\text{goal}})$ considers connections from $X_{\text{goal}}$ to other states as opposed to from other states. $\text{Parent}(x)$ returns the parent of $x$ in the current RRT* tree.

The rewiring procedure internally calls $\text{PropagateRewiring}$. $\text{Children}(x)$ returns the set of all children states to $x$ in the current RRT* tree. $\text{Trajectories}(x, x_{\text{child}})$ returns the control function that was computed earlier by S3F to connect $x$ and $x_{\text{child}}$.

One of the key differences between this algorithm and the original RRT* algorithm is the absence in this algorithmic formulation of finding the nearest state. In the original RRT* algorithm, after a state is randomly sampled, the nearest state in the tree is selected as a source of expansion. A new state is obtained by extending the nearest state towards the randomly sampled state up to a distance $\eta$, and the resultant state is used as the target for the subsequent steering function evaluations. We entirely eliminate this component of the algorithm for simplicity, a modification that was first proposed in Kinodynamic RRT* [28]. This modification is known to not hurt theoretical asymptotic optimality of the RRT* algorithm. The main other difference in this algorithm is a series of modifications that deal with the fact that the learned steering function will reach within an error radius of the goal state. Notable among these is the existence of the $\text{PropagateRewiring}$ procedure.

### 2.4 Correctness of S3F-RRT*

There are two criteria for correctness: solutions returned by S3F-RRT* must satisfy the kinodynamic constraints and must avoid obstacles. Any operation on the S3F-RRT* tree (such as rewiring) can be reformulated as a sequence of state addition and state deletion operations. State deletion by default cannot violate correctness. State addition also satisfies correctness because (1) a state is only added to the tree if the path from the parent to the state is collision free and (2) the path from the parent to the state is generated by integrating the differential equation of motion, implying that the path to the state satisfies kinodynamic constraints. Thus S3F-RRT* is correct.

### 2.5 Probabilistic Completeness Proof of S3F-RRT*

Here we present a summary of the proof of probabilistic completeness (PC) of the S3F-RRT* algorithm. S3F-RRT* is a modification of the original RRT* algorithm [14] designed to make use of a learned steering function. The proof largely follows the structure of the proof of probabilistic completeness of geometric RRT [15], though significant modifications have been made to take into account the presence of kinodynamic constraints and the use of a learned steering function. The full proof can be found in the supplementary materials.

Let $c^* (x_a, x_b)$ denote the cost of the optimal trajectory from $x_a$ to $x_b$ or equivalently the kinodynamic distance from $x_a$ to $x_b$. We assume that $c^*$ obeys the triangle inequality, that is, $c^*(x_a, x_b) \leq c^*(x_a, x_c) + c^*(x_c, x_b)$ for all $x \in X$. Let $S$ be a learned steering function. We assume that with nonzero probability $p$, $S(x_a, x_b)$ yields a state function $\hat{\Gamma}$ that satisfies $c^*(\hat{\Gamma}(t), x_b) \leq c^*(x_a, x_b)$ for all $t \in [0, t_f]$. This assumption in essence states that every state along the path produced by $\hat{S}$ is kinodynamically closer to the goal state than the start state is. For a steering function trained to be optimal, this is a reasonable assumption.

We will use $B(x)$ to denote the subset of the state space $X$ defined by $\{x' \mid c^*(x', x) \leq r\}$. For simplicity, we assume that there exist $\delta_{\text{goal}} > 0, x_{\text{goal}} \in X_{\text{goal}}$ such that $B_{\text{goal}}(x_{\text{goal}}) \subseteq X_{\text{goal}}$. We denote this simplified goal region $B_{\text{goal}}(x_{\text{goal}})$ as $X^*_n$. The goal of the motion planning problem is to find a kinodynamically feasible path $\pi : [0, t_f] \to X_{\text{free}}$ such that $\pi(0) = x_{\text{init}}$ and $\pi(t_f) \in X^*_n$. The clearance of $\pi$ is the maximal $\delta_{\text{clear}}$ such that $B_{\text{clear}}(\pi(t)) \subseteq X_{\text{free}}$ for all $t \in [0, t_f]$.

We assume for this proof that there exists a valid trajectory $\pi : [0, t_f] \to X_{\text{free}}$ with clearance $\delta_{\text{clear}} > 0$. Without loss of generality, assume that $\pi(t_f) = x_{\text{goal}}$, i.e., the trajectory terminates at the center of the goal region. Let $L$ be the total cost of $\pi$, and let $v = \min(\delta_{\text{clear}}, \delta_{\text{goal}})$. Let $m = \frac{L}{v}$. Define a sequence of $m + 1$ points $x_0 = x_{\text{init}}, \ldots, x_m = x_{\text{goal}}$ along $\pi$ such that the cost of traversal from one point to the next is $\frac{v}{m}$. Therefore, $c^*(x_i, x_{i+1}) \leq \frac{v}{m}$ for every $0 \leq i < m$. We will now prove that as the number of iterations increases, the S3F-RRT* algorithm will generate a path passing through the vicinity of these $m + 1$ points with probability asymptotically approaching one.
We compared S3F to the current state of the art on three challenging problems: Dubin’s car with acceleration, tractor trailer, and quadrotor robots. For each problem space, we solve a series of minimum-time motion planning problems using S3F-RRT*, RRT*, and SST. The BARN dataset [22] was used to obtain realistic, obstacle dense maps to run the comparisons on. Figure 4 depicts sample solutions and their planning trees found by S3F-RRT* on the three problem spaces.

3.1 Robot Kinodynamics

The three robot models used in this paper are the Dubin’s car with acceleration, tractor trailer, and quadrotor robots. Here we introduce these robot domains in more detail along with their equations of motion.

**Dubin’s Car with Acceleration:** $X = [x, y, \theta, \alpha], U = [a, k]$

\[
\begin{align*}
\dot{x} &= v \cos(\theta) \\
\dot{y} &= v \sin(\theta) \\
\dot{\theta} &= ak
\end{align*}
\]  

(3)

The Dubin’s car with acceleration is a curvature constrained robot car. $x$, $y$, $\theta$, and $\alpha$ are the $x$-position, $y$-position, orientation, and velocity of the car, and $a$ and $k$ are the acceleration and curvature control inputs. The motion of the car is subject to the curvature constraint $|k| \leq \frac{1}{r_{\text{min}}}$ where $r_{\text{min}}$ is the minimum radius of turning.

**Tractor Trailer:** $X = [x, y, \theta, \phi, a], U = [a, \phi]$

\[
\begin{align*}
\dot{x} &= v \cos(\theta) \\
\dot{y} &= v \sin(\theta) \\
\dot{\theta} &= a/k \\
\dot{\phi} &= \left(\frac{v}{L}\right) \tan(\phi)
\end{align*}
\]  

(4)

The tractor trailer robot consists of a four wheeled robot car pulling a two wheeled trailer. The robot car in isolation has the same dynamics as the Dubin’s car with acceleration. $x$, $y$, $\theta$, $\phi$, and $a$ are the $x$-position, $y$-position, orientation, of the car, velocity of the car, and orientation of the trailer, respectively. The control inputs are $a$ and $\phi$ which represent the acceleration and heading. $L$ is the distance between the front and rear axles of the robot car, and $D$ is the length of the rod connecting the trailer with the car.

**Quadrotor:** $X = [x, y, z, \dot{x}, \dot{\dot{x}}, \dot{y}, \dot{\dot{y}}, \dot{\theta}, \dot{\phi}, \dot{\psi}], U = [r_1, r_2, r_3, r_4]$
We evaluate the learned steering function for each of the three robot domains. Results are depicted for 1500 data points. Figures 5a, 5b, and 5c plot the average cost of best solution found by each of the algorithms against wall-clock time for the different problem spaces, 85% of the trajectories have costs that are less than 10% of the optimal cost; and for the quadrotor problem space, 85% of the trajectories are within 10% of the optimal cost. These results indicate that on average, S3F is able to reach very close to the desired goal.

Measuring the quality of the solutions produced by S3F in terms of time optimality can easily be done by comparing S3F’s trajectory costs with the optimal costs as determined by the NLP solver. Figure 5b shows the CDF plots of the ratios of the cost of solutions of trajectories produced by S3F with the cost of solutions of trajectories produced by the NLP solver. An ideal value of the ratio is close to 1. Results are depicted for 1500 trajectories. We can see that for all three problem spaces the trajectories are very close to optimal. Specifically, for the Dubin’s car with acceleration problem space, 90% of S3F’s trajectories have costs that are less than 1.25 times as suboptimal as the optimal cost; for the tractor trailer problem space, 90% of S3F’s trajectories have costs that are less than 1.25 times as suboptimal as the optimal cost; and for the quadrotor problem space, 80% of S3F’s trajectories have costs that are less than 1.25 times as suboptimal as the optimal cost.

### 3.3 Planning Comparisons

Here we compare planning using the S3F-RRT* algorithm against RRT* with NLP steering, RRT, and SST. By comparing against SST, we can omit a comparison against AO-RRT since previous work [20][31] has shown that empirically SST outperforms AO-RRT. Comparisons are done on all three problem spaces. Starting and ending points for each planning query are sampled randomly across five different maps.

Figures 6a, 6b, and 6c plot the average cost of best solution found by each of the algorithms against wall-clock time for the different robot domains. Results of 25 planning problems are depicted in each plot. In many cases, it takes the algorithms quite a long time to find their first solution. This causes the graphs to not be monotonically

\[ x = \frac{1}{w} (\cos \theta \sin \phi \sin y + \sin \theta \sin y)(\tau_1 + \tau_2 + \tau_3 + \tau_4) \]
\[ y = \frac{1}{w} (\cos \theta \sin \phi \sin y - \sin \theta \cos y)(\tau_1 + \tau_2 + \tau_3 + \tau_4) \]
\[ z = \frac{1}{w} (\cos \theta \cos \phi)(\tau_1 + \tau_2 + \tau_3 + \tau_4) \]
\[ \dot{\theta} = \frac{L(\tau_1 - \tau_3) - 2wL^2 \phi \dot{\phi}}{2wr^2/5 + 2wL^2} \]
\[ \dot{\phi} = \frac{L(\tau_2 - \tau_4) + 2wL^2 \dot{\theta}}{2wr^2/5 + 2wL^2} \]
\[ \dot{\gamma} = \frac{b(\tau_1 - \tau_2 + \tau_3 - \tau_4)}{2wr^2/5 + 4wL^2} \]

The quadrotor is a lightweight, agile robot heavily used in research and industrial applications. \( x, y, \) and \( z \) represent the Cartesian coordinates of the quadrotor. \( \theta, \phi, \) and \( y \) represent the pitch, roll, and yaw, respectively. \( w \) is the weight of the quadrotor, \( L \) is the length of an arm, \( r \) is the radius of the sphere representing the center blob of the quadrotor, \( g \) is the gravitational acceleration, and \( b \) is a constant. \( \tau_1 \) through \( \tau_4 \) represent the thrusts generated by each of the four motors and are the control inputs for the quadrotor.
Figure 6: Comparison of the planning results of the S3F-RRT*, NLP-RRT*, RRT and SST planning algorithms on the three robot domains. Planning time is plotted against the cost of the best solution found thus far, averaged across 25 planning trials.

Figure 7: Failure rate (f) and time to first solution (t) of different planners

decreasing, since the cost of best solution before a solution is found cannot be plotted. We observe in the graphs that S3F-RRT* is able to find solutions very quickly, and is able to find better solutions than the baseline algorithms irrespective of the amount of computation time given. One of the key reasons why this occurs is that due to the speed of evaluation of the learned steering function, many more RRT* iterations can be completed in a unit time as opposed to NLP-RRT*, enabling the more rapid exploration of the state space by the sampling-based planning algorithm. Furthermore, because S3F does a good job at approximating the optimal steering function, waypoints in the final planned path are connected in a near-optimal fashion. This is something that the baseline algorithms like SST and RRT are unable to do, because in these algorithms waypoints are connected by randomly sampled trajectories, resulting in significant suboptimalism.

Figure 7 depicts the rate of failure and average time to first solution of the different algorithms. The time to first solution differs from the cost of best solution in Figure 6 in that the former only considers how long it takes to find the first feasible solution. We can see that across the different problem spaces, S3F-RRT* has lower rates of failure than SST and NLP-RRT*. Figure 6c seems to show that S3F-RRT* and NLP-RRT* have similar performance on the quadrotor domain, but the data in the table shows that S3F-RRT* has a much lower rate of failure and finds its first solution far more quickly, demonstrating that S3F-RRT* indeed has better performance. S3F-RRT* on average is able to find its first solution almost as quickly as RRT. It takes on average an order of magnitude more time for SST and NLP-RRT* to find their first solutions.

4 CONCLUSION

We introduced State Supervised Steering Function, a learning based approximation of the optimal steering function for complex kinodynamic systems. We demonstrate that the learned steering function can be used in sampling-based planners to achieve superior planning results. This superiority is assessed on metrics of time to find solution and quality of solution for three challenging robot domains. Finally, we present a proof of probabilistic completeness of RRT* using S3F, demonstrating its theoretical soundness.

ACKNOWLEDGMENTS

This work has taken place in the Autonomous Mobile Robotics Laboratory (AMRL) at UT Austin. AMRL research is supported in part by NSF (CAREER-2046955, IIS-1954778, SHF-2006404), ARO (W911NF-19-2-0333,W911NF-21-2017), DARPA (HR001120C0031), Amazon, JP Morgan, and Northrop Grumman Mission Systems. The views and conclusions contained in this document are those of the authors alone.
REFERENCES


