Sampling-based Algorithms for Optimal Motion Planning Using Closed-loop Prediction

Oktay Arslan[1] Karl Berntorp[2] Panagiotis Tsiotras[3]

***Abstract*— Motion planning under differential constraints is one of the canonical problems in robotics. State-of-the-art methods evolve around kinodynamic variants of popular sampling-based algorithms, such as Rapidly-exploring Random Trees (RRTs). However, there are still challenges remaining, for example, how to include complex dynamics while guaranteeing optimality. If the open-loop dynamics are unstable, exploration by random sampling in control space becomes inefficient. We describe CL-RRT$^{\#}$, which leverages ideas from the RRT$^{\#}$ algorithm and a variant of the RRT algorithm, which generates trajectories using closed-loop prediction. Planning with closed-loop prediction allows us to handle complex unstable dynamics and avoids the need to find computationally hard steering procedures. The search technique presented in the RRT$^{\#}$ algorithm allows us to improve the solution quality by searching over alternative reference trajectories. We show the benefits of the proposed approach on an autonomous-driving scenario.**

I. Introduction

Motion planning is ubiquitous in many applications where different levels of autonomy is desired. Given a system that is subject to a set of differential constraints, an initial state, a final state, a set of obstacles, and a goal region, the *motion-planning problem* is to find a control input that drives the system from its initial state to the goal region. This problem is computationally hard to solve [1]. The motion-planning problem is commonly solved using randomized planners such as rapidly-exploring random trees RRT [2] or its asymptotically optimal version RRT* [3]–[5]. RRT relies on random exploration of the state space. RRT-type planners often neglect the differential constraints, or assume the existence of a steering procedure that connects two nodes to each other. However, finding a steering procedure essentially amounts to solving a two-point boundary-value problem. In general, there are no guarantees that a solution exists [6], but the differential constraints restrict the reachable set and should be accounted for in applications [7]–[10].

This paper proposes CL-RRT$^{\#}$, which leverages ideas from the CL-RRT [11] and the RRT$^{\#}$ algorithms [12]–[14]. To handle differential constraints, the proposed approach samples in the output space of the system and incrementally grows trajectories corresponding to the closed-loop dynamics of the system. In other words, we avoid the need for complicated steering procedures, which oftentimes prohibit– or at least complicate– the use of RRT-type planners for systems whose dynamics cannot be neglected. Traditional asymptotically optimal planners, such as RRT$^{\#}$ and RRT*, find optimal paths by searching in a neighborhood of the added node for connections having lowest costs. A result of our algorithm is that we connect and rewire nodes that give lowest costs with respect to the metric defined by the closed-loop system. As a consequence, the proposed algorithm provides the segments of reference trajectories that yield the lowest-cost state trajectory of the *closed-loop* system.

The kinodynamic RRT incrementally grows a tree of suboptimal but dynamically feasible trajectories in the state space by sampling random control inputs and simulating the system with these control inputs [2], [15]. Exploration via control inputs can be inefficient when the dynamics are complex and/or unstable. To remedy this, [11] proposed CL-RRT, which uses closed-loop prediction for trajectory generation. CL-RRT grows a tree in the reference space. Each path of the tree represents a reference trajectory used as an input to the closed-loop system. Each edge of the tree is associated with a segment of a reference trajectory and a state trajectory of the closed-loop system.

Contribution: State-of-the-art sampling-based asymptotically optimal search algorithms such as RRT* and RRT$^{\#}$ rely on a steering procedure that typically satisfies certain conditions to connect the nodes. However, finding a steering procedure typically requires the solution of a two-point boundary value problem and is computationally expensive. Analytic solutions are known only in restrictive special cases. The proposed CL-RRT$^{\#}$ algorithm remedies this computational bottleneck by relying on the closed-loop dynamics instead of steering procedures to connect nodes. The idea is similar to CL-RRT but it differs from it, in that we also ensure asymptotic optimality with respect to the closed-loop dynamics,

II. Problem Formulation

Let $X \subseteq \mathbb{R}^n$, $Y \subseteq \mathbb{R}^p$ and $U \subseteq \mathbb{R}^m$. We assume that the system dynamics can be described by a nonlinear differential equation (i.e., the system dynamics) of the form

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)), \quad x(0) = x_0, \\ y(t) &= h(x(t), u(t)), \end{aligned} \tag{1}$$

where the state $x(t) \in X$, the output $y(t) \in Y$, the control $u(t) \in U$, for all $t \geq 0$ and $x_0 \in X$, and f and h are smooth (continuously differentiable) functions describing the time

[1]Oktay Arslan is a Robotics Technologist with Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA 91109-8099, USA, Email: oktay.arslan@jpl.nasa.gov. He performed this research while at Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, USA.

[2]Karl Berntorp is with Mitsubishi Electric Research Laboratories (MERL), Cambridge, MA 02139, USA, Email: karl.o.berntorp@ieee.org.

[3]Panagiotis Tsiotras is with the faculty of D. Guggenheim School of Aerospace Engineering and the Institute for Robotics and Intelligent Machines at the Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: tsiotras@gatech.edu.

evolution of the system dynamics. Let $\mathcal{X}$ denote the set of all essentially bounded measurable functions mapping $[0, T]$ to X for any $T \in \mathbb{R}_{>0}$, and define $\mathcal{Y}$ and $\mathcal{U}$ similarly. The functions in $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{U}$ are called *state trajectories*, *output trajectories*, and *controls*, respectively.

Let X_{obs} and X_{goal}, denoting the *obstacle space* and the *goal region*, respectively, be open subsets of X. Let X_{free} also denote the *free space*, given by $X \setminus X_{\text{obs}}$. The function h describes the output y we wish to control. We are interested in the class of control problems in which we wish $y(t)$ to track a time-varying reference trajectory $r(t)$ (*output trajectory generation* problem). We assume the existence of a tracking controller $\phi : (y', y) \mapsto u \in U$ such that given a desired output value $y' \in Y$, and a current output value $y \in Y$ of the system, it computes a control input such that y tracks y'. This means that every reference trajectory that is added to the graph is tracked by the controller ϕ.

A. Problem Statement

Given the state space X, obstacle region X_{obs}, goal region X_{goal}, and functions f and h as in (1), and a tracking controller $\phi : (y', y) \mapsto u \in U$, we wish to find a reference trajectory $r \in \mathcal{Y}$ such that the corresponding unique state trajectory $x \in \mathcal{X}$, output trajectory $y \in \mathcal{Y}$, and control $u \in \mathcal{U}$, computed by ϕ; obey the differential constraints,

$$\dot{x}(t) = f(x(t), u(t)) \quad x(0) = x_0,$$
$$y(t) = h(x(t), u(t)) \text{ for all } t \in [0, T];$$

avoid the obstacles, i.e., $x(t) \in X_{\text{free}}$ for all $t \in [0, T]$; reach the goal region, i.e., $x(T) \in X_{\text{goal}}$; and minimize $J(x, u, r) = \int_0^T g(x(t), u(t), r(t))\, \mathrm{d}t$.

B. Primitive Procedures

We follow the notation in [3] and [12]–[14] and introduce new primitive procedures used in the proposed CL-RRT$^{\#}$.

Closed-loop Prediction: Given a state $x \in X_{\text{free}}$, and an output trajectory $\sigma_y \in \mathcal{Y}$, the function `Propagate` : $(x, \sigma_y) \mapsto \sigma_x \in \mathcal{X}$ returns the state trajectory that is computed by simulating the system dynamics forward in time with the initial state x, and the reference trajectory σ_y.

Queue Operations: Nodes of the computed graphs are associated with keys and priority queues are used to sort these nodes based on the precedence relation between keys. The following functions are implemented to maintain a given priority queue $\mathcal{Q}$:

- $\mathcal{Q}$.`top_key`() returns the highest priority of all nodes in the priority queue $\mathcal{Q}$ with the smallest key value if the queue is not empty. If $\mathcal{Q}$ is empty, then $\mathcal{Q}$.`top_key`() returns a key value of $k = [\infty; \infty]$.
- $\mathcal{Q}$.`pop`() deletes the node with the highest priority in the priority queue $\mathcal{Q}$ and returns a reference to the node.
- $\mathcal{Q}$.`update`(v_y, k) sets the key value of the node v_y to k and reorders the priority queue $\mathcal{Q}$.
- $\mathcal{Q}$.`insert`(v_y, k) inserts the node v_y into the priority queue $\mathcal{Q}$ with the key value k.
- $\mathcal{Q}$.`remove`(v_y) removes the node v_y from the priority queue $\mathcal{Q}$.

Exploration: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, where $\mathcal{G}_y$ and $\mathcal{G}_\sigma$ are graphs whose nodes represent points in Y and trajectories in $\mathcal{X}$, respectively, and $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ are priority queues that are used for ordering of nongoal and goal nodes that represent points in Y, a goal region in the output space $Y_{\text{goal}} \subset Y$, and a point $y \in Y$, the function `Extend` : $(\mathcal{S}, Y_{\text{goal}}, y) \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ includes a new node, multiple edges to $\mathcal{G}_y$ and multiple nodes, edges to $\mathcal{G}_\sigma$, updates the priorities of nodes in $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ and returns an updated tuple $\mathcal{S}'$. The trajectories in the state space and the output space are stored as the edges of $\mathcal{G}_x$ and $\mathcal{G}_y$, respectively.

Exploitation: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$ the function `Replan` : $\mathcal{S} \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ rewires the parent node of the nodes in $\mathcal{G}_y$ based on their cost-to-come values, includes new nodes and edges in $\mathcal{G}_\sigma$, if necessary. This is done by propagating the dynamics of the system to create a new sequence of reference trajectories, and returning the updated tuple $\mathcal{S}'$.

Construction of Solution: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, the function `ConstrSolution` : $\mathcal{S} \mapsto \mathcal{T}_x$ returns a tree whose edges and nodes represent simulated trajectories in $\mathcal{X}$ and the corresponding internal states of the nodes of $\mathcal{G}_y$. These trajectories are computed by propagating the dynamics with reference trajectories that are encoded in a tree of $\mathcal{G}_y$, which is formed by the edges between nodes of $\mathcal{G}_y$ and their parent nodes.

III. The CL-RRT$^{\#}$ Algorithm

To avoid steering procedures that connect two nodes, our approach relies on simultaneous manipulation of two graphs, the output/reference graph $\mathcal{G}_y$ and the state-trajectory graph $\mathcal{G}_\sigma$. The graph $\mathcal{G}_\sigma$ contains the state trajectories resulting from tracking the reference trajectories from $\mathcal{G}_y$. Designing steering procedures is nontrivial for dynamical systems, and usually requires computationally intensive optimization steps. A reference tracking controller is easier to design and requires lightweight computation; this makes the proposed approach promising for real-time motion planning problems. Since the proposed method connects nodes by simulation of the closed-loop simulation, by construction we get dynamic feasibility of the trajectories. To also get optimality, we utilize the asymptotically optimal sampling-based RRT$^{\#}$.

RRT$^{\#}$ consists of an exploration step and an exploitation step. In the exploration, a graph is extended to a sampled point, followed by a local search on neighboring nodes to update and improve the lowest-cost path information. The exploitation step makes sure that the information available up to that step is fully exploited, to improve convergence speed. The proposed CL-RRT$^{\#}$ can be interpreted as an RRT$^{\#}$ that explores the set of reference paths to the tracking controller and seeks for better reference paths by using the lowest-cost trajectory information computed from closed-loop prediction. Similarly, exploitation improves the reference graph $\mathcal{G}_y$ by utilizing all information of the state-trajectory graph $\mathcal{G}_\sigma$. We will next go through the necessary steps and refer to [16] for a complete description of the implementation.

TABLE I: The node (`OutNode`) and edge (`OutEdge`) data structures for points and trajectories in output space, respectively, and the node (`TrajNode`) and edge (`TrajEdge`) data structures for trajectories in state space.

field	type	description
y	vector $\in \mathbb{R}^p$	output point associated with this node
`g`	real $\in \mathbb{R}$	cost-to-come value
$\bar{\mathtt{g}}$	real $\in \mathbb{R}$	one step look-ahead g-value
`h`	real $\in \mathbb{R}$	heuristic value for the cost between y and $\mathcal{Y}_{\text{goal}}$
$\mathtt{p}_y$	`OutNode`	reference to the parent output node
$\mathtt{p}_\sigma$	`TrajNode`	reference to the parent trajectory node
r	trajectory $\in \mathcal{Y}$	output trajectory associated with this edge
`tail`	`OutNode`	reference to the tail output node
`head`	`OutNode`	reference to the head output node
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this node
e_y	`OutEdge`	reference to the output edge
`outgoing`	`OutEdge` array	list of outgoing output edges
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this edge
`tail`	`TrajNode`	reference to the tail trajectory node
`head`	`TrajNode`	reference to the head trajectory node

A. Details of the Data Structures

Each node v_y in the graph $\mathcal{G}_y$ is associated with a reference point $y \in \mathbb{R}^m$. It is an `OutNode` data structure, summarized in Table I. The data structure v_y contains two estimates of the optimal cost-to-come value between the initial reference point and y: the cost-to-come value `g` and the one step look-ahead g-value $\bar{\mathtt{g}}$. It also keeps a heuristic value `h`, which is an underestimate of the optimal cost value between y and Y_{goal}, to guide and reduce the search effort. When $\bar{\mathtt{g}}$ is updated during replanning, the reference node that yields the corresponding minimum cost-to-come value is stored in the parent reference node $\mathtt{p}_y$. Lastly, $\mathtt{p}_\sigma$ is the trajectory that is computed by closed-loop prediction when the system is simulated with the reference trajectory and the tracking controller ϕ between the nodes $\mathtt{p}_y$ and v_y. Its terminal state represents the internal state associated with v_y.

Each edge e_y in the graph $\mathcal{G}_y$ is an `OutEdge` data structure, summarized in Table I. Each edge e_y is associated with a trajectory $r \in \mathcal{Y}$. It also contains two output nodes, namely, `tail` and `head`, which represent the tail and the head output nodes of e_y, respectively.

Each node v_σ in the graph $\mathcal{G}_\sigma$ is a `TrajNode` data structure, summarized in Table I. Each node v_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains an output edge e_y, which corresponds to the reference trajectory that yields σ as the closed-loop prediction. It also keeps a list of outgoing output edges `outgoing`, and this list is used to compute outgoing trajectory nodes emanating from the terminal state of σ.

Each edge e_σ in the graph $\mathcal{G}_\sigma$ is a `TrajEdge` data structure, summarized in Table I. Each edge e_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains two trajectory nodes, namely, `tail` and `head` which represent the tail and the head trajectory nodes of e_σ, respectively.

B. Details of the Procedures

Algorithm 1 gives the body of the CL-RRT$^\#$ algorithm, which is conceptually similar to RRT$^\#$. First, the algorithm initializes the tuple of data structures $\mathcal{S}$ that is incrementally grown and updated as exploration and exploitation are performed (Line 3). The tuple $\mathcal{S}$ contains the graphs $\mathcal{G}_y$ and $\mathcal{G}_\sigma$, which are used to store output nodes and state trajectory nodes, respectively, and the priority queues $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$. The graph $\mathcal{G}_\sigma$ is created with no edges and v_σ as its only node. This node represents a state trajectory that contains only the initial state $x_{\text{init}} = x_0$. Likewise, the graph $\mathcal{G}_y$ is initialized with no edges and v_y as its only node, which represents $y_{\text{init}} = h(x_{\text{init}}, u_{init})$. The g- and $\bar{g}$-values of v_y are set to zero. The parent trajectory node of v_y is set with the pointer to the node v_σ.

Algorithm 1: The CL-RRT$^\#$ Algorithm

```
1 CL-RRT#(x_init, X_goal, X)
2     Y_goal := OutputMap(X_goal);
3     S ← Initialize(x_init, Y_goal);
4     for k = 1 to N do
5         y_rand ← Sample(k);
6         S ← Extend(S, Y_goal, y_rand);
7         S ← Replan(S);
8     T_x ← ConstrSolution(S);
9     return T_x;
```

The algorithm iteratively builds a graph of collision-free reference trajectories $\mathcal{G}_y$ by first sampling an output point y_{rand} from the obstacle-free output space Y_{free} (Line 5) and then extending the graph towards this sample (Line 6), at each iteration. The cost of the unique trajectory from the root node to a given node v_y is denoted as $\mathtt{Cost}(v_y)$. It also builds another graph $\mathcal{G}_\sigma$, to store the state trajectories computed by simulation of the closed-loop dynamics when a reference trajectory is tracked. Once a new node is added to $\mathcal{G}_y$ after `Extend`, `Replan` is called to improve the existing solution by propagating the new information (Line 7). The dynamic system is simulated for different reference trajectories as needed during the search process. The computed state trajectories are added to the graph $\mathcal{G}_\sigma$ as new nodes along with the corresponding controls.

Finally, when a predetermined maximum number of iterations is reached, `ConstrSolution` extracts the spanning tree of $\mathcal{G}_y$ that contains the lowest-cost reference trajectories (Line 8). Algorithm 2 gives the details of `ConstrSolution`.

1) The `Extend` *Procedure:* The `Extend` procedure is given in Algorithm 3. It first extends the nearest output node $v_{y,\text{nearest}}$ to the output sample y (Lines 4-5). The output trajectory that extends the nearest output node $v_{y,\text{nearest}}$ towards the output sample y is denoted as r_{new}. The final output point on the output trajectory r_{new} is denoted as y_{new}. If r_{new} is collision-free, then a new output node $v_{y,\text{new}}$ is created to represent the new output point y_{new} (Line 8).

The members of the node $v_{y,\text{new}}$ are set as follows. First, `Near` finds the set of neighbor output nodes V_{near} in the neighborhood of the new output point y_{new} (Line 9). Then, the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ of the new output node $v_{y,\text{new}}$ are computed by using the information of the neighbor output nodes (Lines 10-19).

Once the new output node $v_{y,\text{new}}$ is created together with

Algorithm 2: The `ConstrSolution` Solution Procedure

1 **ConstrSolution**($\mathcal{S}$)
2 $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$;
3 $(V_y, E_y) \leftarrow \mathcal{G}_y$; $X \leftarrow \varnothing$;
4 **foreach** $v_y \in V_y$ **do**
5 $\sigma \leftarrow v_y.\mathtt{p}_\sigma.\sigma$;
6 $v_x \leftarrow$ `StateNode(`σ`.back())`;
7 $V_x \leftarrow V_x \cup \{v_x\}$;
8 $v_{x,\text{parent}} \leftarrow$ `find(`V_x`,`σ`.front())`;
9 **if** $v_{x,\text{parent}} = \varnothing$ **then**
10 $v_{x,\text{parent}} \leftarrow$ `StateNode(`σ`.front())`;
11 $V_x \leftarrow V_x \cup \{v_{x,\text{parent}}\}$;
12 $e_x \leftarrow$ `StateEdge(`$v_{x,\text{parent}}$, v_x, σ`)`;
13 $E_x \leftarrow E_x \cup \{e_x\}$;
14 $X \leftarrow X \cup \{\sigma$`.back()`$\}$;
15 **return** $\mathcal{T}_x = (V_x, E_x)$;

Algorithm 3: The `Extend` Procedure

1 **Extend**($\mathcal{S}, X_{\text{goal}}, y$)
2 $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$;
3 $(V_y, E_y) \leftarrow \mathcal{G}_y$; $(V_\sigma, E_\sigma) \leftarrow \mathcal{G}_\sigma$;
4 $v_{y,\text{nearest}} \leftarrow$ `Nearest(`$\mathcal{G}_y$,y`)`;
5 $r_{\text{new}} \leftarrow$ `Steer(`$v_{y,\text{nearest}}.y$,y`)`;
6 **if** `ObstacleFree(`r_{new}`)` **then**
7 $y_{\text{new}} \leftarrow r_{\text{new}}$`.back()`;
8 $v_{y,\text{new}} \leftarrow$ `OutNode(`y_{new}`)`;
9 $v_{y,\text{new}}.\mathtt{h} \leftarrow$ `ComputeHeuristic(`y_{new},Y_{goal}`)`;
10 $V_{\text{near}} \leftarrow$ `Near(`$\mathcal{G}_y$,y_{new},$|V_y|$`)` $\cup \{v_{y,\text{nearest}}\}$;
11 $E_{y,\text{succ}} \leftarrow \varnothing$; $E_{y,\text{pred}} \leftarrow \varnothing$;
12 **foreach** $v_{y,\text{near}} \in V_{\text{near}}$ **do**
13 $r \leftarrow$ `Steer(`y_{new},$v_{y,\text{near}}.y$`)`;
14 **if** `ObstacleFree(`r`)` **then**
15 $e_y \leftarrow$ `OutEdge(`$v_{y,\text{new}}$,$v_{y,\text{near}}$,r`)`;
16 $E_{y,\text{succ}} \leftarrow E_{y,\text{succ}} \cup \{e_y\}$;
17 $r \leftarrow$ `Steer(`$v_{y,\text{near}}.y$,y_{new}`)`;
18 **if** `ObstacleFree(`r`)` **then**
19 $e_y \leftarrow$ `OutEdge(`$v_{y,\text{near}}$,$v_{y,\text{new}}$,r`)`;
20 $E_{y,\text{pred}} \leftarrow E_{y,\text{pred}} \cup \{e_y\}$;
21 $V'_\sigma \leftarrow \varnothing$; $E'_\sigma \leftarrow \varnothing$;
22 **foreach** $e_y \in E_{y,\text{pred}}$ **do**
23 $v_{y,\text{pred}} \leftarrow e_y$.`tail`;
24 $v_{\sigma,\text{pred}} \leftarrow v_{y,\text{pred}}.\mathtt{p}_\sigma$;
25 $x_{\text{pred}} \leftarrow v_{\sigma,\text{pred}}.\sigma$`.back()`;
26 $\sigma \leftarrow$ `Propagate(`x_{pred},e_y.r`)`;
27 **if** `ObstacleFree(`σ`)` **then**
28 $v_{\sigma,\text{new}} \leftarrow$ `TrajNode(`σ,e_y,$E_{y,\text{succ}}$`)`;
29 $e_\sigma \leftarrow$ `TrajEdge(`$v_{\sigma,\text{pred}}$,$v_{\sigma,\text{new}}$,σ`)`;
30 $V'_\sigma \leftarrow V'_\sigma \cup \{v_{\sigma,\text{new}}\}$;
31 $E'_\sigma \leftarrow E'_\sigma \cup \{e_\sigma\}$;
32 **if** $v_{y,\text{new}}.\bar{\mathtt{g}} > v_{y,\text{pred}}.\mathtt{g} +$ `Cost(`σ`)` **then**
33 $v_{y,\text{new}}.\bar{\mathtt{g}} \leftarrow v_{y,\text{pred}}.\mathtt{g} +$ `Cost(`σ`)`;
34 $v_{y,\text{new}}.\mathtt{p}_y \leftarrow v_{y,\text{pred}}$;
35 $v_{y,\text{new}}.\mathtt{p}_\sigma \leftarrow v_{\sigma,\text{new}}$;
36 $V_y \leftarrow V_y \cup \{v_{y,\text{new}}\}$;
$E_y \leftarrow E_y \cup E_{y,\text{succ}} \cup E_{y,\text{pred}}$;
37 $V_\sigma \leftarrow V_\sigma \cup V'_\sigma$; $E_\sigma \leftarrow E_\sigma \cup E'_\sigma$;
38 $\mathcal{G}_y \leftarrow (V_y, E_y)$; $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$;
39 $\mathcal{Q} \leftarrow$ `UpdateQueue(`$\mathcal{Q}$,$v_{y,\text{new}}$`)`;
40 $\mathcal{Q}_{\text{goal}} \leftarrow$ `UpdateGoal(`$\mathcal{Q}_{\text{goal}}$,$v_{y,\text{new}}$,$X_{\text{goal}}$`)`;
41 **return** $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$;

the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ connecting it to its neighbor output nodes V_{near}, `Extend` attempts to find the best incoming edge that yields a segment of a reference trajectory which incurs minimum cost to get to $v_{y,\text{new}}$ among all incoming edges in $E_{y,\text{pred}}$ (Lines 20-34).

When a trajectory node $v_{\sigma,\text{new}}$ is created, the outgoing state trajectories emanating from the final state of the state trajectory $v_{\sigma,\text{new}}.\sigma$ are not immediately computed, for the sake of efficiency. Instead, the algorithm keeps the set of candidate outgoing output trajectories, that is, the edges in $E_{y,\text{succ}}$, in a list $v_{\sigma,\text{new}}$.`outgoing`, and the simulation of the system for these output trajectories is postponed until the head output node of the output edge $v_{\sigma,\text{new}}.e_y$ is selected for the Bellman update during the `Replan` procedure. Once the new state trajectory node $v_{\sigma,\text{new}}$ and the edge between the predecessor state trajectory node $v_{\sigma,\text{pred}}$ and itself are created (Lines 27-28), they are added to the set of nodes and edges of the graph $\mathcal{G}_\sigma$, respectively (Lines 29-30). If the incoming output edge e_y between the predecessor output node $v_{y,\text{pred}}$ and the new output node $v_{y,\text{new}}$ yields a collision-free state trajectory σ that incurs a cost less than the current cost of $v_{y,\text{new}}$, then, the $\bar{\mathtt{g}}$-value of $v_{y,\text{new}}$ is set with new lower cost, $v_{y,\text{pred}}$ and $v_{\sigma,\text{new}}$ are made the new parent output node and the new parent state trajectory node of $v_{y,\text{new}}$ (Lines 31-34).

After the creation of the new output node $v_{y,\text{new}}$, it is added to the graph $\mathcal{G}_y$ together with all of its collision-free output edges, and all trajectory nodes and edges created during the simulation of the system dynamics are added to $\mathcal{G}_\sigma$. Lastly, the priority queues, $\mathcal{Q}$ and $\mathcal{Q}_{\text{goal}}$ are updated accordingly by using the information of the new output node $v_{y,\text{new}}$, that is, reordering of the priorities after insertion of $v_{y,\text{new}}$ to the queue $\mathcal{Q}$ and reordering the goal output nodes in $\mathcal{Q}_{\text{goal}}$ if $v_{y,\text{new}}$ is a goal output node (Lines 38-39).

2) The `Replan` *Procedure:* The `Replan` procedure is given in Algorithm 4 (see [12]). It improves the cost-to-come values of the output nodes by operating on the nonstationary and promising nodes of the graph $\mathcal{G}_y$. It pops the most promising nonstationary node from the priority queue $\mathcal{Q}$, if there are any, and this node is made stationary by assigning its $\bar{\mathtt{g}}$-value to its g-value (Lines 5-6). Then, the g-value of the output node v_y is used to improve the $\bar{\mathtt{g}}$-values of its neighbor output nodes. To do this, `Replan` computes all outgoing state trajectories emanating from internal state of the output node v (Lines 9–16). All newly computed, obstacle-free, state trajectory nodes and edges are added to $\mathcal{G}_\sigma$ (Line 18).

In Lines 19–27, for each outgoing state trajectory σ, `Replan` adds up its cost, incurred by reaching to the successor output node $v_{y,\text{succ}}$ to the g-value of v_y, compare it with the current $\bar{\mathtt{g}}$-value of $v_{y,\text{succ}}$, and if the outgoing state trajectory edge σ yields a lower cost than $v_{y,\text{succ}}$, the $\bar{\mathtt{g}}$-value of $v_{y,\text{succ}}$ is set with new lower cost, and v_y and $v_{\sigma,\text{succ}}$ are made the new parent output node and the new parent state trajectory node of $v_{y,\text{succ}}$, respectively (Lines 23-25).

The auxiliary procedures in `Extend` and `Replan` can be found in [16].

Algorithm 4: `Replan` Procedure

Replan$(\mathcal{S}, X_{\text{goal}})$
  $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$;
  $(V_\sigma, E_\sigma) \leftarrow \mathcal{G}_\sigma$;
  while $\mathcal{Q}$.`top_key()` $\prec \mathcal{Q}_{\text{goal}}$.`top_key()` **do**
    $v_y \leftarrow \mathcal{Q}$.`pop()`;
    v_y.g $\leftarrow v_y$.$\bar{\text{g}}$;
    $v_\sigma \leftarrow v_y$.p_σ;
    $x \leftarrow v_\sigma.\sigma$.`back()`;
    foreach $e_y \in v_\sigma$.`outgoing` **do**
      $v_{y,\text{succ}} \leftarrow e_y$.`head`;
      $\sigma \leftarrow$ `Propagate`$(x, e_y.r)$;
      if `ObstacleFree`(σ) **then**
        $v_{\sigma,\text{succ}} \leftarrow$ `TrajNode`$(\sigma, e_y,$`outgoing`$(\mathcal{G}_y, v_{y,\text{succ}}))$;
        $e_\sigma \leftarrow$ `TrajEdge`$(v_\sigma, v_{\sigma,\text{succ}}, \sigma)$;
        $V_\sigma \leftarrow V_\sigma \cup \{v_{\sigma,\text{succ}}\}$;
        $E_\sigma \leftarrow E_\sigma \cup \{e_\sigma\}$;
    v_σ.`outgoing` $\leftarrow \varnothing$;
    $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$;
    foreach $v_{\sigma,\text{succ}} \in$ `succ`$(\mathcal{G}_\sigma, v_\sigma)$ **do**
      $\sigma \leftarrow v_{\sigma,\text{succ}}.\sigma$;
      $v_{y,\text{succ}} \leftarrow v_{\sigma,\text{succ}}.e_y$.`head`;
      if $v_{y,\text{succ}}.\bar{\text{g}} > v_y$.g + `Cost`$(\sigma)$ **then**
        $v_{y,\text{succ}}.\bar{\text{g}} \leftarrow v_y$.g + `Cost`$(\sigma)$;
        $v_{y,\text{succ}}.\text{p}_y \leftarrow v_y$;
        $v_{y,\text{succ}}.\text{p}_\sigma \leftarrow v_{\sigma,\text{succ}}$;
        $\mathcal{Q} \leftarrow$ `UpdateQueue`$(\mathcal{Q}, v_{y,\text{succ}})$;
        $\mathcal{Q}_{\text{goal}} \leftarrow$ `UpdateGoal`$(\mathcal{Q}_{\text{goal}}, v_{y,\text{succ}}, X_{\text{goal}})$;
  return $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$;

C. Properties of the Algorithm

CL-RRT$^\#$ provides both dynamic feasibility and asymptotic optimality guarantees, that is, the lowest-cost reference trajectory computed by the algorithm converges to the optimal reference trajectory, almost surely. The former property is an immediate result of using closed-loop prediction during the search phase. During the extension of the graph $\mathcal{G}_y$, if some segments of a reference trajectory cannot be tracked, the corresponding state trajectory is not stored in the graph $\mathcal{G}_\sigma$. Optimality is due to the property of the RRT$^\#$ algorithm [12]. The proposed algorithm incrementally grows a graph $\mathcal{G}_y$ in the output space in a similar fashion as the RRG algorithm [5]. Therefore, the lowest-cost path encoded in $\mathcal{G}_y$ converges to the optimal output trajectory in the output space almost surely. In addition, the lowest-cost output trajectory encoded in the graph $\mathcal{G}_y$ is extracted at the end of each iteration in a similar fashion as the RRT$^\#$ algorithm. Given the cost function that associates each edge in $\mathcal{G}_y$ with a nonnegative cost value being *monotonic* and *bounded*, the proposed algorithm is asymptotically optimal [3].

IV. Numerical Study

We evaluate the proposed algorithm on a simulated autonomous-driving example. The goal is to traverse a circuit while minimizing the Euclidean trajectory length. We compare the proposed CL-RRT$^\#$ algorithm against CL-RRT [11] and CL-RRT*, which can be found as a special case of the proposed algorithm, CL-RRT$^\#$, similarly to the differences between RRT*and RRT$^\#$ [12].

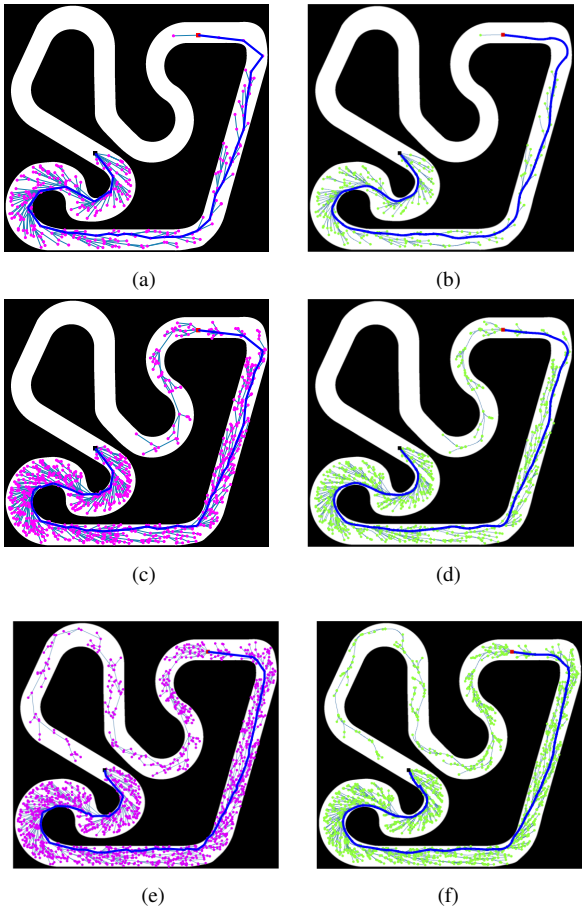

Fig. 1: The evolution of the solution trees for reference paths (a), (c), (e) and state trajectories (b), (d), and (f), computed by CL-RRT$^\#$. The trees are at 1000, 2000, and 3000 iterations, respectively.

The vehicle is described by a kinematic single-track model. Note that there is no analytic solution to the steering procedure assumed in RRT* and RRT$^\#$ for this vehicle model. The system dynamics are given by

$$\begin{aligned} \dot{p}_x &= v\cos(\psi+\beta)/\cos(\beta), \\ \dot{p}_y &= v\sin(\psi+\beta)/\cos(\beta), \\ \dot{\psi} &= v\tan(u_1)/L, \quad \dot{v} = u_2, \end{aligned}$$

where L is the wheel base, β is the body slip angle of the velocity vector relative to the vehicle-fixed frame, and u_1, u_2 are the controls for the steering angle and translational velocity, respectively. Each input takes values in an interval, that is, $u_i \in [u_i^l, u_i^u]$. A pure-pursuit controller tracks a given reference path [17]. The heading command is generated by following a look-ahead point on a given reference path. The speed command is assumed given as a desired speed v_{crs}, which is tracked by a proportional controller.

Fig. 1 shows the trees at different stages when using the proposed algorithm. The vehicle is initially located at the black square with zero heading angle and zero speed. The task is to move to the red square. From Figs. 1(a), 1(c), and

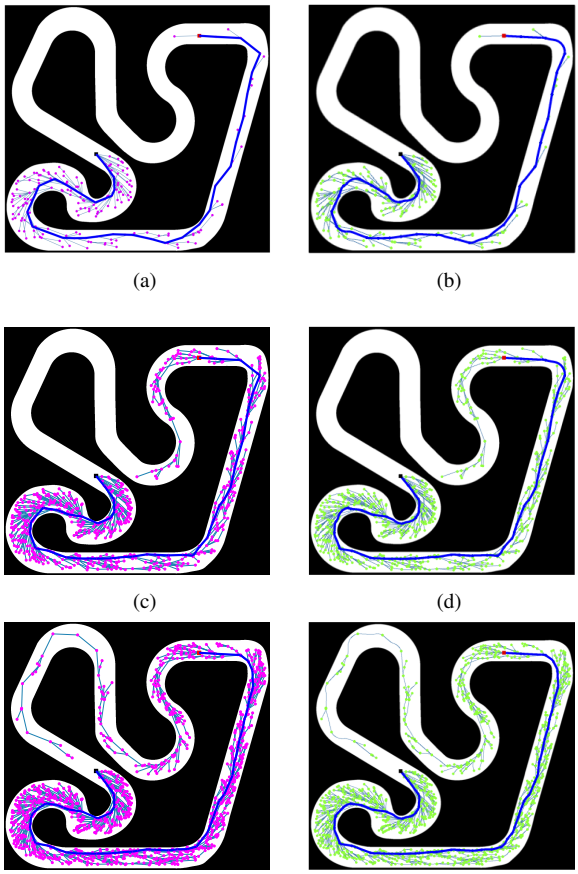

Fig. 2: The evolution of the solution trees for reference paths and state trajectories computed by CL-RRT*, with same notation as in Fig. 1.

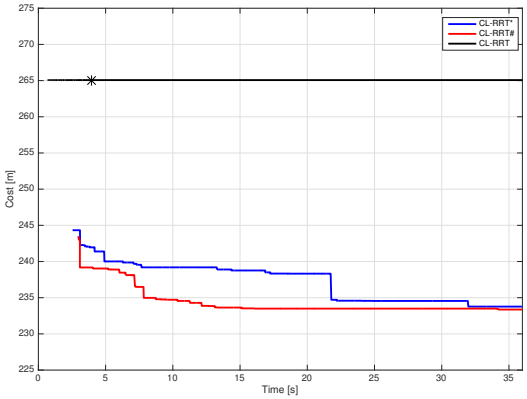

Fig. 3: Value of the cost function over the computation time. The black marker indicates the end of the 3000 iterations for CL-RRT.

1(e), CL-RRT$^{\#}$ grows a graph in the output space. The path corresponding to the lowest-cost trajectory is shown in blue. Figs. 1(b), 1(d), and 1(f), show the corresponding state. Fig. 2 shows the reference and state trajectories for CL-RRT*. CL-RRT$^{\#}$ reaches a lower-cost trajectory than CL-RRT* for the same number of iterations. This is most clearly seen in the turns in (b), (d), and (f) figures, where CL-RRT$^{\#}$ takes sharper turns. Fig. 3 shows path length over computation time for the different algorithms. The proposed CL-RRT$^{\#}$ returns a lower-cost solution for the same computation time. CL-RRT gets stuck at its first solution.

V. Conclusion

We presented CL-RRT$^{\#}$, a new asymptotically optimal motion-planning algorithm that uses closed-loop prediction for trajectory generation. The approach is a hybrid of CL-RRT and RRT$^{\#}$. It grows a graph of reference trajectories, used as inputs to a low-level tracking controller, and chooses the one that yields the lowest-cost state trajectory of the closed-loop system. CL-RRT$^{\#}$ provides dynamic feasibility by construction and ensures asymptotic optimality. CL-RRT$^{\#}$ avoids the need for steering procedures to connect nodes in the graph, instead relying on the closed-loop dynamics to provide the necessary state connections. Our approach is therefore applicable to a range of dynamical systems where the dynamics and/or kinematics is important to consider, such as vehicles or underactuated robots.

Acknowledgment: The portion of the work not done at Mitsubishi Electric Research Labs has been supported, in part, by ARO MURI award W911NF-11-1-0046 and ONR award N00014-13-1-0563.

References

[1] J. H. Reif, "Complexity of the movers problem and generalizations," in *IEEE Conf. Foundations of Computer Science*, 1979, pp. 421–427.

[2] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[3] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[4] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the RRT*," in *IEEE Int. Conf. Robotics and Automation*, Shanghai, China, May 2011.

[5] S. Karaman and E. Frazzoli, "Optimal kinodynamic motion planning using incremental sampling-based methods," in *49th IEEE Conf. Decision and Control*, Atlanta, GA, Dec. 2010.

[6] R. Vinter, *Optimal Control*. Boston, MA: Birkhäuser, 2010.

[7] O. Arslan, E. A. Theodorou, and P. Tsiotras, "Information-theoretic stochastic optimal control via incremental sampling-based algorithms," in *IEEE Symp. Adaptive Dynamic Programming and Reinforcement Learning*, Orlando, FL, Dec. 2014.

[8] J. Leonard *et al.*, "A perception-driven autonomous urban vehicle," *J. Field Robotics*, vol. 25, no. 10, pp. 727–774, 2008.

[9] J. J. Kuffner, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue, "Dynamically-stable motion planning for humanoid robots," *Autonomous Robots*, vol. 12, no. 1, pp. 105–118, 2002.

[10] K. Berntorp and S. D. Cairano, "Joint decision making and motion planning for road vehicles using particle filtering," in *IFAC Symp. Advances in Automotive Control*, Kolmården, Sweden, June 2016.

[11] Y. Kuwata, J. Teo, S. Karaman, G. Fiore, E. Frazzoli, and J. P. How, "Motion planning in complex environments using closed-loop prediction," in *AIAA Guidance, Navigation, and Control Conf.*, 2008.

[12] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *IEEE Int. Conf. Robotics and Automation*, Karlsrühe, Germany, May 2013.

[13] ——, "Dynamic programming guided exploration for sampling-based motion planning algorithms," in *IEEE Int. Conf. Robotics and Automation*, Seattle, WA, May 2015.

[14] ——, "Dynamic programming principles for sampling-based motion planners," in *ICRA Optimal Robot Motion Planning Workshop*, Seattle, WA, May 2015.

[15] S. M. LaValle and J. J. Kuffner, Jr., "Randomized kinodynamic planning," *Int. J. Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[16] O. Arslan, K. Berntorp, and P. Tsiotras, "Sampling-based Algorithms for Optimal Motion Planning Using Closed-loop Prediction," *ArXiv e-print: 1601.06326*, Jan. 2016.

[17] O. Amidi, "Integrated mobile robot control," Carnegie Mellon University, Robotics Institute, Tech. Rep. CMU-RI-TR-90-17, May 1990.