# The Value of Planning for Infinite-Horizon Model Predictive Control

Nathan Hatch* and Byron Boots*

***Abstract*— Model Predictive Control (MPC) is a classic tool for optimal control of complex, real-world systems. Although it has been successfully applied to a wide range of challenging tasks in robotics, it is fundamentally limited by the prediction horizon, which, if too short, will result in myopic decisions. Recently, several papers have suggested using a learned value function as the terminal cost for MPC. If the value function is accurate, it effectively allows MPC to reason over an *infinite* horizon. Unfortunately, Reinforcement Learning (RL) solutions to value function approximation can be difficult to realize for robotics tasks. In this paper, we suggest a more efficient method for value function approximation that applies to goal-directed problems, like reaching and navigation. In these problems, MPC is often formulated to track a path or trajectory returned by a planner. However, this strategy is brittle in that unexpected perturbations to the robot will require replanning, which can be costly at runtime. Instead, we show how the intermediate data structures used by modern planners can be interpreted as an approximate *value function*. We show that that this value function can be used by MPC *directly*, resulting in more efficient and resilient behavior at runtime.**

## I. INTRODUCTION

In this paper, we revisit the classic planning and control paradigm ubiquitous in robotics. In this paradigm, behavior is generated *hierarchically*. First, a planner computes a high-level solution to a simplified global problem, typically solved offline and in advance. Then, at runtime, a controller follows that plan locally using a more detailed dynamics model. This strategy arises from the difficulty of solving the entire behavior generation approach at once. Often, this approach works very well [1], [2].

In some applications, this approach can be brittle. Global information is often condensed into a *single* plan and the controller's cost is defined relative to that information. Stochasticity in the environment, or mismatch between the models used for planning and control, can mean that the controller deviates irrecoverably from the plan. When possible, it is common to rerun the planner from the current location. This can be dangerous or expensive if replanning takes too long.

Compare the above approach to reinforcement learning (RL) [3]. RL typically attempts to tackle the entire problem at once, building a *global* value function or policy that encodes the best action at any state in the environment. Compared with plans, value functions are much more informative. A controller with access to a value function could resolve the best immediate action based on value *anywhere* in the state space, resulting in more robust behavior and eliminating the need to replan. While some robotics problems have been successfully solved with RL [4]–[6], learning an accurate value function over the entire space space is typically infeasible due to computational and modeling limitations.

We would like to find an approach that lies between these extremes: more robust than following a single plan, but less expensive than computing a full value function. Recent work on combining Model Predictive Control (MPC) with RL has indeed started to move in this direction. MPC is a classic tool for controlling complex, real-world systems that solves an online optimization problem to choose an action that will minimize predicted future cost. While MPC has been applied to a wide range of challenging tasks in robotics [7]–[9], it is fundamentally limited by the prediction horizon, which, if too short, will result in myopic decisions.

Recently, several papers have suggested using a learned value function as the terminal cost for MPC [10]–[12]. In particular, [11] calculates offline the exact value function along a limit cycle, then uses this to compute a piecewise quadratic approximation to the value function everywhere. Since the resurgence of deep reinforcement learning, other authors have investigated approximating value functions for MPC using neural networks. For example, POLO [13] uses a kind of fitted value iteration where the regression targets are found using trajectory optimization. Another recent example is model-predictive Q-learning (MPQ) [14], which shows a theoretical connection between soft Q-learning [15] and information-theoretic MPC [16]. Like POLO, MPQ learns a value function with the help of trajectory optimization. Other examples include [17]–[20]. All of these approaches incorporate global value function information into MPC, effectively allowing MPC to reason over an *infinite* horizon.

Unfortunately, RL solutions to value function approximation can be infeasible for many robotics tasks. For goal-directed navigation or reaching tasks, using approaches like POLO [13] or MPQ [14] to compute a terminal cost for MPC makes little sense. These methods are time consuming and expensive, even for simple problems. By contrast, planning algorithms can generate a rough trajectory quickly. This leads to a natural question: can planning algorithms be exploited to generate better, more robust terminal cost functions for MPC? This leads to the key insight in this work: modern planning algorithms conveniently already compute an approximate value function as an intermediate step toward generating a solution. Consider, for example, RRT* [21] searching backwards from the goal. This algorithm produces not just a path from the starting location, but also an entire tree of paths to the goal, each node annotated with its distance from the goal—an approximate value function. The algorithm RRT# [22] makes this connection even more explicit by fully solving the Bellman optimality equations at

*University of Washington, Seattle WA 98105 USA. {nhatch2, bboots}@cs.washington.edu

each iteration, to ensure that the value function is the optimal one for the current planning graph.

In this work, we propose using entire planning trees as approximate value functions that can be used as the terminal cost in MPC. This results in a strategy that is more robust than the classic robotics planning and control paradigm, but much faster than RL approaches to the same problems. Specifically, we focus on the interface between two commonly used algorithms in robotics, the stochastic MPC algorithm Model Predictive Path Integral control (MPPI) [16] and the sampling-based planning algorithm RRT# [22]. We propose treating the terminal point of each sampled MPC rollout as a node of the planning tree and using the corresponding cost-to-go as the terminal cost in MPC. In other words, we use the value function computed as a by-product of planning to extend the horizon of MPC. This information improves the performance of MPC in several ways. First, MPC can be made more *accurate*, as each terminal state can be evaluated using the most relevant planning node. Second, MPC can be made more *robust*. When the controller is perturbed, it simply uses existing nodes in the planning tree—there is no replanning time. Third, the value returned by RRT#, while approximate, lies in a sweet spot: it is fast to compute and useful to MPC in practice.

## II. BACKGROUND

### A. Bellman Backups on Directed Graphs

We begin by providing a brief background discussion on value iteration and its relation to planning algorithms. In RL, Bellman backups are used by value iteration (among other algorithms) to compute a value function. In the simplest setting, value iteration works on a discrete state space $\mathcal{S}$ and action space $\mathcal{A}$. For each state $s \in \mathcal{S}$, each action $a \in \mathcal{A}$ has some cost $c(s,a)$ and deterministically transitions to a new state $f(s,a)$. To compute the value function $V : \mathcal{S} \to \mathbb{R}$, we perform sweeps through the states, computing an (undiscounted) Bellman backup at each state:

$$V(s) \leftarrow \min_a c(s,a) + V(f(s,a)) \tag{1}$$

Equivalently, we can formulate this problem as finding the minimum-cost path along a weighted, directed graph. The vertices of the graph are the states $\mathcal{S}$, and the edges are the pairs $(s, f(s,a))$ for $a \in \mathcal{A}$, with edge weight $\hat{c}(s, f(s,a)) := c(s,a)$. Let $\mathcal{N}(s)$ denote the out-neighbors of $s$. Then the Bellman backup looks like this:

$$V(s) \leftarrow \min_{s' \in \mathcal{N}(s)} \hat{c}(s,s') + V(s') \tag{2}$$

### B. Connection between Bellman Backups and Planning

Finding the minimum-cost path along a directed graph is precisely what planning algorithms do. However, they vary in how they construct this graph. In this paper, we focus on the stochastic graphs constructed by RRT* [23]. Briefly, RRT* iteratively constructs a graph by sampling states in configuration space and connecting them to all existing vertices within some search radius. It then performs a Bellman backup for the new vertex and all of the neighboring vertices.

Much prior work has investigated the connection between graph-based planning and Bellman backups. Like all value iteration methods, RRT* is not guaranteed to produce a globally consistent value function after each Bellman backup. To address this, RRT# [22] incorporates graph search techniques from D* [24] to efficiently update the entire value function after adding each new vertex. This line of reasoning has produced a number of follow-up works; e.g. [25], [26].

In LPA* [27], an incremental heuristic search technique like D*, Bellman backups are instead called "vertex expansions". A vertex is considered "consistent" in LPA* precisely when it satisfies the local Bellman optimality equation—that is, when performing a Bellman backup does not change the value of the vertex. After performing a Bellman backup, LPA* decides which neighboring vertices might now be inconsistent, and adds those vertices to a priority queue.

## III. APPROACH

In this work, we consider behavior generation problems in robotics that are typically solved by invoking a planner to generate a path and a closed-loop controller that follows the path online. This includes many *goal-directed* problems including reaching (in manipulation) and vehicle navigation.

We will frame these problems formally as *reinforcement learning*. Specifically, we consider the problem of learning a policy in an infinite-horizon undiscounted Markov Decision Process (MDP) with terminal states. Such an MDP is defined by a tuple $\mathcal{M} = \{\mathbb{S}, \mathbb{A}, c, f, s_{start}, S_{goal}\}$ where $\mathbb{S}$ is the state space,[1] $\mathbb{A}$ is the action space, $c(s,a)$ is the per-step cost function, $s_{t+1} \sim f(s_t, a_t)$ is the stochastic transition function, $s_{start}$ is the start state, and $S_{goal} \subseteq \mathbb{S}$ is the goal region. A policy $\pi(\cdot|s)$ outputs a distribution over actions given a state. Let $\mu^{\pi}_{\mathcal{M}}$ be the distribution over state-action trajectories obtained by running policy $\pi$ on $\mathcal{M}$. The value function for a given policy $\pi$ is defined as

$$V^{\pi}_{\mathcal{M}}(s) = \mathbb{E}_{\mu^{\pi}_{\mathcal{M}}}\left[\sum_{t=0}^{\infty} c(s_t, a_t) \mid s_0 = s\right]$$

and the action-value function as

$$Q^{\pi}_{\mathcal{M}}(s,a) = \mathbb{E}_{\mu^{\pi}_{\mathcal{M}}}\left[\sum_{t=0}^{\infty} c(s_t, a_t) \mid s_0 = s, a_0 = a\right]$$

Although we have written these as infinite undiscounted sums, the MDP terminates once $s_t \in S_{goal}$ for some $t$, after which time no cost is accumulated. The objective is to find an optimal policy $\pi^* = \arg\min_{\pi} V^{\pi}_{\mathcal{M}}(s_{start})$.

### A. An RL Perspective on Model Predictive Control

MPC is widely used in robotics and can be viewed as an online learning approach to synthesizing action sequences for MDPs [28]. Instead of solving for a single globally optimal policy that prescribes the action to take at any state, MPC is a pragmatic approach of optimizing simple, local action sequences at test time. At each timestep, MPC uses an approximate transition model to search for an action sequence that minimizes cost over a finite horizon. The first

[1] We use $\mathbb{S}$ and $\mathbb{A}$ to denote continuous state and action spaces, as opposed to $\mathcal{S}$ and $\mathcal{A}$ for discrete ones.

action from the sequence is executed on the system, and the process is then repeated from the next state.

Following [29], we formalize this process as solving a surrogate MDP whose parameters may differ from the parameters of the true underlying MDP that we wish to solve. The surrogate MDP is $\widehat{\mathcal{M}} = \{\mathbb{S}, \mathbb{A}, c, \hat{f}, s_{start}, S_{goal}, H\}$ with approximate or simplified dynamics $\hat{f}$ and, importantly, a finite horizon $H$. $\widehat{\mathcal{M}}$ is repeatedly solved at every state encountered, resulting in a receding horizon method.

In many problems, solving $\widehat{\mathcal{M}}$ with the limiting horizon $H$ would result in myopic behavior with respect to the original MDP $\mathcal{M}$. This can be especially pronounced in tasks with sparse rewards and the goal-directed tasks considered in this paper, which naturally require reasoning about longer horizons. To contend with the limited horizon, *infinite horizon* MPC [14] sets the terminal cost in MPC as a value function $\widehat{V}$ that adds global information to the problem.

MPC can, therefore, be viewed as iteratively constructing an *estimate* of the Q-function of the original MDP $\mathcal{M}$, under the policy $\pi_\phi$ induced by the action sequence $\phi$ [29]:

$$Q^{\phi}_{H}(s,a) = \mathbb{E}_{\mu^{\pi_\phi}_{\widehat{\mathcal{M}}}}\left[\sum_{i=0}^{H-1} c(s_i,a_i) + \widehat{V}(s_H) \mid s_0 = s, a_0 = a\right] \tag{3}$$

MPC then iteratively optimizes this estimate (at current system state $s_t$) to update the action sequence

$$\phi^*_t = \arg\min_{\phi} Q^{\phi}_{H}(s_t, \pi_\phi(s_t)) \tag{4}$$

Several popular MPC algorithms, including MPPI [29] and receding-horizon linear quadratic regulators [28] can be viewed through this lens.

### B. Adding Value from Planning

The main challenge is obtaining the value function $\widehat{V}$ in Eq. 3. Previous approaches have attempted to learn it from data via Q-learning [13], [14]. This can be costly, as it requires a large number of interactions with the system. For goal-directed problems, it is often significantly cheaper to find a *plan* that MPC can follow to the goal [1], [2]. As discussed in the introduction, if cost-to-go values along this path are provided by the planner, then it implicitly defines an approximation of the value function along the path. If the approximation is "good enough," then we can use it as the terminal cost $\widehat{V}$ in Eq. 3. Our key insight is that many modern planning algorithms construct a *planning tree* over a much larger subset of the state space as an intermediate step toward finding the optimal plan, thereby improving the value function estimate. The connection between planning and RL has been made in many prior works, but none of these have discussed the connection with MPC [22], [25]–[27].

## IV. Algorithm

In our experimental results in Section V, we use MPPI, a stochastic MPC algorithm, as our closed-loop controller [16]. We use RRT# to compute an approximate value function via a planning tree that grows backwards from the desired configuration to the current robot configuration [22]. The implementation details of these algorithms are provided below. Please refer to Fig. 1 for notation.

**Require:** Configuration space $\mathbb{S}$
**Require:** Free space $\mathbb{S}_{free} \subseteq \mathbb{S}$
**Require:** Control space $\mathbb{A}$
**Require:** Horizon $H$
**Require:** Stochastic dynamics $f : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$
**Require:** Model $\hat{f} : \mathbb{S} \times \mathbb{A}^H \to \mathbb{S}^{H+1}$
**Require:** True cost $c : \mathbb{S} \times \mathbb{A} \to [0, \infty)$
**Require:** Kinematic planning cost $\hat{c} : \mathbb{S} \times \mathbb{S} \to [0, \infty)$
**Require:** Maximum steering radius $M$
**Require:** Maximum search radius $R > M$
**Require:** Subroutine `nearest`$(s, \mathcal{S}, \hat{c}, L, r)$: Returns the $L$ points in the set $\mathcal{S}$ that are nearest to $s$ according to the cost function $\hat{c}$. Excludes any points $s'$ for which $\hat{c}(s, s') > r$.

Fig. 1. Notation used in algorithms.

**Require:** Start and goal states $s_{start}, s_{goal} \in \mathbb{S}$
**Require:** Distribution $P$ over $\mathbb{S}$ such that $P(s_{start}) > 0$
Initialize vertices $\mathcal{S} = \{s_{goal}\}$
Initialize edges $\mathcal{E} = \{\}$
Initialize value function $\mathcal{V} = \{(s_{goal}, 0)\}$
**while** $s_{start} \notin \mathcal{S}$ **do**
  Sample state $s_{sample} \sim P$
  $s_{near} \leftarrow$ `nearest`$(s_{sample}, \mathcal{S}, \hat{c}, 1, \infty)$
  $s \leftarrow$ `steer`$(s_{sample}, s_{near}, \hat{c}, M)$
  **if** $s \in \mathbb{S}_{free}$ **then**
    $N \leftarrow$ `nearest`$(s, \mathcal{S}, \hat{c}, \infty, M)$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(s, s') : s' \in N\}$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(s', s) : s' \in N\}$
    $\mathcal{V}(x) \leftarrow \infty$
    $\mathcal{V} \leftarrow$ `value_iterate`$(\mathcal{S}, \mathcal{E}, \mathcal{V}, \hat{c})$
  **end if**
**end while**
**return** $\mathcal{S}, \mathcal{V}$

Fig. 2. Function `rrt_sharp`$(s_{start}, s_{goal}, p)$. The goal state $s_{goal}$ can be chosen arbitrarily from the goal region $S_{goal}$ of the original MDP. For notational simplicity, we omit the details of how the steering radius $M$ decreases as a function of the number of vertices $|\mathcal{S}|$.

### A. RRT#

The RRT# planning algorithm as used in this paper is shown in Fig. 2. The pseudocode requires two additional subroutines:

- `steer`$(s_{sample}, s_{near}, \hat{c}, M)$: Projects the point $s_{sample}$ to the ball of radius $M$ around $s_{near}$ according to the distance function $\hat{c}$.
- `value_iterate`$(\mathcal{S}, \mathcal{E}, \mathcal{V}, \hat{c})$: Performs sweeps of Bellman backups (Eq. 2) on the graph $(\mathcal{S}, \mathcal{E})$, with edge weights defined by $\hat{c}$. Returns the optimal value function. The parameter $\mathcal{V}$ is used to initialize the value function for faster convergence.

A few things are noteworthy about our implementation. First, we search backwards from the goal to the start state.

**Require:** Start state $s$
**Require:** Control sequence $a_{0:H-1}$
**Require:** Planning tree $\mathcal{S}$ with value function $\mathcal{V}$
  Roll out $s_{0:H} = \hat{f}(s, a_{0:H-1})$
  Find step costs $c_{step} = \sum_{t=0}^{H-1} c(s_t, a_t)$
  Initialize terminal cost $c_{term} = \infty$
  Find neighbors $N = \texttt{nearest}(s_H, \mathcal{S}, \hat{c}, \infty, R)$
  **if** $N \neq \emptyset$ **then**
    $c_{term} \leftarrow \min_{s' \in N} \hat{c}(s_H, s') + \mathcal{V}(s')$
  **end if**
  **return** $c_{step} + c_{term}$

Fig. 3. Function $\texttt{value}(s, a_{0:H-1}, \mathcal{S}, \mathcal{V})$. For notational simplicity, we assume that collision detection is incorporated into the cost function $c(s, a)$.

This enables us to treat the search tree as a value function for MPC, since the goal state does not change during execution.

Second, rather than using D* to update the value function, we simply perform brute-force sweeps of Bellman backups over the state set $\mathcal{S}$ using `value_iterate`. Practical implementations may wish to implement D* to speed up the planning time, but the end result is the same.

Third, we perform collision checking only on the start and end points of each edge. We assume that the maximum steering radius $M$ is small enough to ensure that no edges travel through thin obstacles. The motivation for this is that we need to perform this operation with an even larger search radius $R > M$ during MPC, so it needs to be fast.

Fourth, we require $P(s_{start}) > 0$ in order to bias the samples towards the start state and to guarantee that the algorithm will terminate (assuming a feasible path exists). Note that probabilistic completeness also requires that every open subset of $\mathbb{S}_{free}$ has nonzero probability.

### B. Extending the Horizon of MPPI

The main technical contribution of our approach is the `value` function shown in Fig. 3, which gives the exact process for interpreting the planning tree as a value function. Given a starting state $s$ and control sequence $a_{0:H-1}$, it essentially temporarily adds the terminal point of the rollout to the planning tree:

1) Collect a list of the planning nodes within radius $R$ of the terminal point $s_H$, according to the distance function $\hat{c}$ used by the planner.
2) For each of these nodes, calculate the exact cost of traveling to that node according to $\hat{c}$,[2] plus the value function $\mathcal{V}$ at that node according to the planner.
3) Then the (approximate) value function $\widehat{V}(s_H)$ is given by the lowest of these summed costs.

Note that if there are no planning nodes within radius $R$ of $s_H$, then the cost of that rollout is infinite. Hence, the MPC controller must stay close to the planning tree $\mathcal{S}$ during execution, else it risks becoming irrecoverably lost.

Fig. 4 shows how the `value` function is used by MPPI.

[2]To save computation during MPC, we ignore obstacles when calculating the cost of traveling to a node. We assume that the search radius $R$ is small enough that the set of neighboring nodes does not include nodes on the other side of thin obstacles.

**Require:** Initial state $s_{start}$
**Require:** Goal region $S_{goal}$
**Require:** Planning tree $\mathcal{S}$ with value function $\mathcal{V}$
**Require:** Control covariance $\Sigma$
**Require:** Temperature $\lambda > 0$
**Require:** Number of trajectory samples $Q$
  Initialize $\bar{a}_{0:H-1} = 0, \ldots, 0$
  Initialize $s = s_{start}$
  **while** $s \notin S_{goal}$ **do**
    **for** $i = 1$ **to** $Q$ **do**
      Sample $a^i_{0:H-1} \sim \mathcal{N}(\bar{a}_{0:H-1}, \Sigma)$
      $c^i \leftarrow \texttt{value}(s, a^i_{0:H-1}, \mathcal{S}, \mathcal{V})$
      $w^i \leftarrow \exp(-\frac{1}{\lambda} c^i)$
    **end for**
    $\bar{a}_{0:H-1} \leftarrow \left(\sum_i w^i a^i_{0:H-1}\right) / \left(\sum_i w^i\right)$
    Execute $s \leftarrow f(s, \bar{a}_0)$
    Shift $\bar{a}_{0:H-1} \leftarrow \bar{a}_1, \bar{a}_2, \ldots, \bar{a}_{H-1}, 0$
  **end while**

Fig. 4. Function $\texttt{mppi}(s_{start}, S_{goal}, \mathcal{S}, \mathcal{V}, \Sigma, \lambda, Q)$. The distribution $\mathcal{N}(\mu, \Sigma)$ is a multivariate Gaussian with mean $\mu$ and covariance $\Sigma$.

## V. EXPERIMENTS

### A. Kinematic 2D Environments

We begin with several simple experiments using fully actuated planar robot models. We consider both a simple point robot with state $s = (x, y)$ and a "stick" robot (shown in Fig. 6 in blue) with an additional rotational degree of freedom, $\theta$. Controls are state differences $a \in \mathbb{S}$ with cost

$$c(s, a) = \begin{cases} \mathbb{I}[s \notin S_{goal}] + (a^T W a)^{1/2} & s \in \mathbb{S}_{free} \\ \infty & \text{otherwise} \end{cases} \quad (5)$$

where $\mathbb{I}$ is a 0/1 indicator function, and $W \succ 0$ is a diagonal weight matrix. We use MPPI for the MPC controller (cf. Fig. 4) with the one-step dynamics model $\hat{f}(s, a) = s + a$. The true dynamics $f$ are identical except that they implement noise by adding a small Gaussian perturbation to the control $a$. During MPC execution, actions $a$ are clamped to have small norm $(a^T W a)^{1/2} < \varepsilon$ for some $\varepsilon > 0$.

The kinematic planning cost function is

$$\hat{c}(s_1, s_2) = \left( \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ \theta_2 - \theta_1 \end{bmatrix}^T W \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ \theta_2 - \theta_1 \end{bmatrix} \right)^{1/2} \quad (6)$$

which is similar to the MPC cost $c(s, a)$ except that it does not include the indicator term $\mathbb{I}[s \notin S_{goal}]$. Instead, in the surrogate MDP that RRT# is implicitly solving, progress to the goal is encouraged by the fact that we *must* take a step in the graph at every time step until we reach the goal. Hence, the optimal policy is to take the shortest path to the goal.

We consider three different controllers:

- `min`: MPPI with the subset of the planning tree $\mathcal{S}$ consisting of only the minimum-cost path
- `full`: MPPI with the full planning tree $\mathcal{S}$
- `naive`: A simple waypoint-based controller that simply aims straight at each subsequent node in the minimum-cost path.

For each of the four task environments shown in Fig. 5, we run the RRT# planner 50 times, to obtain a total of 200 planning trees. For each planning tree, we run each controller for five trials. We report three statistics from these trials:

- **Failure %**: We consider a trial to "fail" if the robot does not reach the goal within two minutes. This can happen either because the robot got stuck behind an obstacle, or because it drifted too far away from the planning tree and got lost.
- **Collision %**: Of the successful trials, what percentage collided with an obstacle before reaching the goal?
- **Normalized Cost**: For a given planning tree, we calculate the normalized cost as follows. We compute the average cost accumulated by each controller over the five trials. (Failed or collided trials are not used in these averages. If a controller did not have at least three successful, collision-free trials, we exclude that planning tree from the the computation.) We divide each of these averages by the average for the `min` controller. Thus, the `min` controller always has normalized cost 1.0. We report the mean and standard deviation over the 200 planning trees.

Results are shown in Table I. As expected with randomized planning and control algorithms, there is a wide variance in normalized cost for the `full` controller. However, overall it seems that in these very simple environments, the `full` controller slightly outperforms the `min` controller. The `naive` controller collides with static obstacles on about one third of the trials, because it does not account for noise. Interestingly, when it doesn't collide, it tends to find shorter paths than the MPC-based controllers. Even though it cannot take shortcuts along the planned path like MPPI can, it seems like the stochasticity of MPPI (or its ability to account for and avoid obstacles when the planned path goes near them) makes MPPI's paths longer.

Computationally, the `full` controller requires roughly 15 milliseconds (ms) of wall clock time per iteration. This is only about twice as expensive as the `min` controller (7 ms).

TABLE I
STATIC EXPERIMENT RESULTS

| | Failure % | Collision % | Normalized Cost |
|---|---|---|---|
| Point | `naive`: 0.0<br>`min`: 0.0<br>`full`: 0.0 | `naive`: 34.6<br>`min`: 0.0<br>`full`: 0.0 | `naive`: $0.950 \pm 0.023$<br>`min`: —<br>`full`: $0.987 \pm 0.029$ |
| Stick | `naive`: 0.0<br>`min`: 2.2<br>`full`: 1.8 | `naive`: 34.0<br>`min`: 1.0<br>`full`: 1.0 | `naive`: $0.917 \pm 0.090$<br>`min`: —<br>`full`: $0.983 \pm 0.084$ |

### B. Dynamic 2D Environments

The usefulness of a value function becomes more obvious in environments where it is infeasible to stop and replan in the case of unforeseen circumstances. To demonstrate this, we modify the previous experiments to incorporate second-order dynamics. We use state state $s = (p, \dot{p})$ so that instead

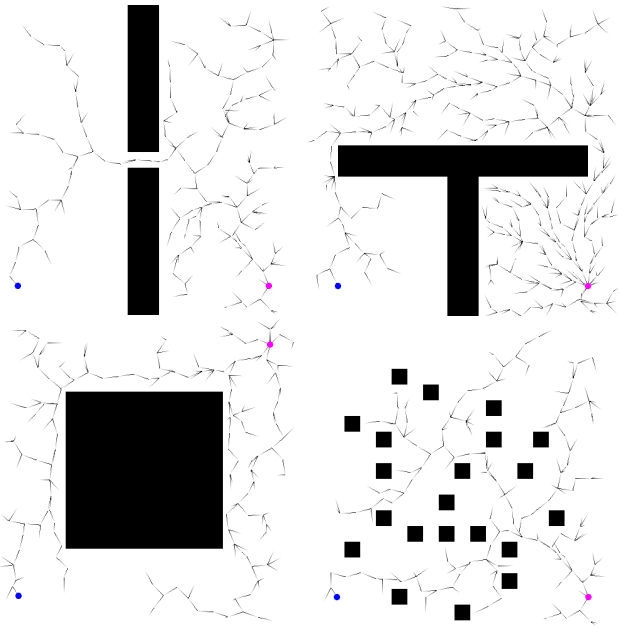

Fig. 5. Planar planning environments. Clockwise from upper left: `gate`, `bugtrap`, `forest`, `blob`. The goal configuration is shown in magenta on the right side of each image. Also shown is the final RRT# search tree for the point robot at the moment when a solution is found.

of the kinematic transition $f(s, a) = s + a$ we have

$$f\left(\begin{bmatrix} p \\ \dot{p} \end{bmatrix}, a\right) = \begin{bmatrix} p + \dot{p}\Delta t \\ \dot{p} + a \end{bmatrix} \tag{7}$$

where $\Delta t$ is the control timestep. After applying the update in Eq. (7), the velocity $\dot{p}$ is clipped to have norm $(\dot{p}^T W \dot{p})^{1/2} < \varepsilon$, so that we retain the same velocity limit as in the kinematic experiments. We again implement noise by perturbing the action $a$, and we use the same cost function $c(s, a)$ with the same weights $W$ as in Eq. (5) except that the action $a$ is now in velocity space. The planner still uses the kinematic model and cost function $\hat{c}$ from Eq. (6), so in this case $c$ and $\hat{c}$ are only distantly related.

We further experiment with dynamic (i.e. moving / non-static) spherical obstacles which were not considered during planning, shown in Fig. 6 as red circles. At each timestep, we perturb their velocity according to a uniform distribution and then clamp it to a maximum speed.

Results are shown in Table II. The results from Table I are summarized in the row "First / Static" (meaning first-order dynamics / static obstacles). Since the `naive` controller cannot handle dynamic obstacles or second-order dynamics, we report results only for `min` and `full`. Interestingly, we see that both controllers are more successful and collide less frequently when the dynamics are second-order. This may be because our implementation of noise in the second-order simulation is less aggressive, or because MPC execution tends to take fewer time steps for the second-order system. Perhaps the MPPI shift operator (cf. Fig. 4) is more accurate for second-order systems.

The results with dynamic obstacles are very different. For both first- and second-order systems, dynamic obstacles cause a dramatic increase in failure rate for the `min` con-

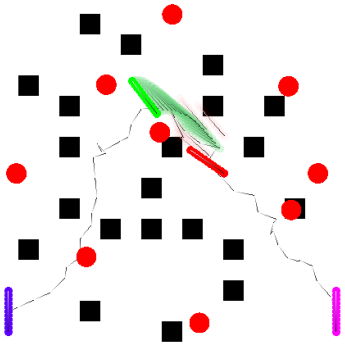

Fig. 6. Visualization of MPC execution for the stick robot in the `forest` environment. The start and goal configurations are shown in blue and magenta, respectively. The thin, jagged line is a 2D projection (ignoring rotational pose) of the plan returned by the planner. Note that it is *not* the full planning tree; rather, it is the shortest path from start to goal within that tree. The red circles are dynamic obstacles. The nominal MPPI trajectory is shown in in green. The planning node that gives the optimal terminal value for the nominal trajectory (cf. Fig. 3) is shown in red. Finally, the very light lines near the end of the nominal trajectory are examples of terminal points $x_H$ from the most recent set of MPPI samples (cf. Figs. 3, 4).

troller, but only a small increase in failure rate for the `full` controller. Because we calculate normalized cost based on successful, collision-free trials, the normalized cost of the `full` controller with dynamic obstacles varies considerably. In fact, in the case of the second-order system, `full` finds slightly higher-cost trajectories than `min` on average. Qualitatively, this is because during unlucky encounters with dynamic obstacles, `full` is able to find (high-cost) alternative paths to avoid the dynamic obstacles, while `min` simply collides or flies off of the planned path and cannot recover. Failure can be interpreted as infinite cost, and is not included in normalized cost averages.

Qualitative results are here and source code is here.

TABLE II

DYNAMIC EXPERIMENT RESULTS

| | Failure % | Collision % | Normalized Cost |
|---|---|---|---|
| First / Static | `min:` 1.1<br>`full:` 0.9 | `min:` 0.5<br>`full:` 0.5 | `min:` —<br>`full:` $0.985 \pm 0.063$ |
| First / Dynamic | `min:` 6.8<br>`full:` 1.4 | `min:` 2.0<br>`full:` 3.3 | `min:` —<br>`full:` $0.947 \pm 0.273$ |
| Second / Static | `min:` 0.5<br>`full:` 0.5 | `min:` 0.0<br>`full:` 0.0 | `min:` —<br>`full:` $0.973 \pm 0.049$ |
| Second / Dynamic | `min:` 3.0<br>`full:` 0.6 | `min:` 0.5<br>`full:` 0.4 | `min:` —<br>`full:` $1.009 \pm 0.274$ |

### C. 3D Skid Steering Environment

Finally, we demonstrate the generality of this approach by showing a few examples in a 3D environment designed to simulate a real-world robot. The Clearpath Warthog (see Fig 7, left) is a four-wheeled, skid-steered robot designed for off-road navigation. The dynamics of this robot are difficult to model accurately, so we used a simple first-order kinematic model for both the RRT# planner and the MPPI controller.

Fig. 7. **Left:** The University of Washington Clearpath Warthog robot. **Right:** The 3D skid steering environment that simulates the Warthog. A video of this environment can be found here. The source code for this environment is not publicly available.

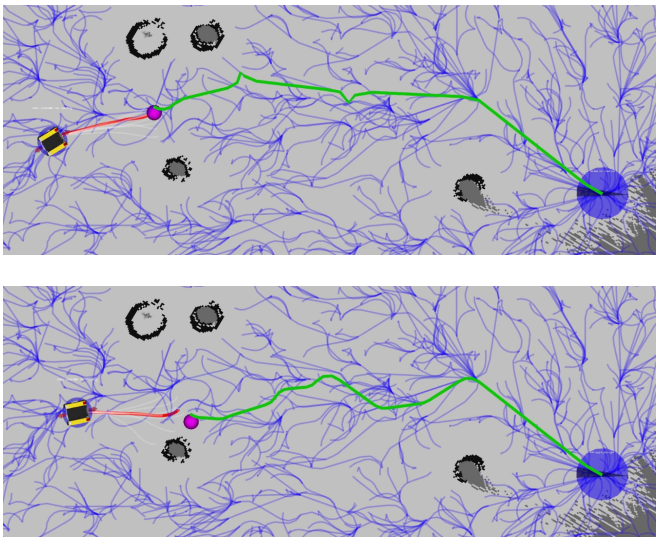

Fig. 8. Qualitative results on the 3D environment. The optimized MPPI rollout is shown in red. The planning node used to compute the terminal cost for that rollout is shown in pink, and the corresponding path from that node to the goal is shown in green. The thin blue lines show other branches of the planning tree. See this video for an animated version.

We implemented the algorithm proposed in this paper in the navigation stack for this robot, and we tested it in simulation (see Fig 7, right).

Fig. 8 shows qualitative results. The lower screenshot was taken approximately one second after the upper one. We see that, due to stochasticity and/or modeling errors, the robot's position has changed such that the optimized MPC rollout now uses a different branch of the planning tree. Without access to the full planning tree, the MPC controller would not be able to make such optimizations.

## VI. CONCLUSION

In this paper, we have drawn a connection between reinforcement learning, control, and planning by interpreting them all as searching for a value function. The concept of a Bellman backup is central to all three, although historically they have each used idiosyncratic names—e.g. dynamic programming, vertex expansion—for this concept. This fundamental connection allows us to recognize *planning trees* as approximate value functions and reuse them as the terminal value function for MPC at no additional computational cost.

## REFERENCES


[1] D. Ferguson, T. M. Howard, and M. Likhachev, "Motion planning in urban environments," *Journal of Field Robotics*, vol. 25, no. 11-12, pp. 939–960, 2008.

[2] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on control systems technology*, vol. 17, no. 5, pp. 1105–1118, 2009.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[5] M. Riedmiller, M. Montemerlo, and H. Dahlkamp, "Learning to drive a real car in 20 minutes," in *2007 Frontiers in the Convergence of Bioscience and Information Technologies*. IEEE, 2007, pp. 645–650.

[6] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami *et al.*, "Emergence of locomotion behaviours in rich environments," *arXiv preprint arXiv:1707.02286*, 2017.

[7] P. Abbeel, A. Coates, and A. Y. Ng, "Autonomous helicopter aerobatics through apprenticeship learning," *The International Journal of Robotics Research*, vol. 29, no. 13, pp. 1608–1639, 2010.

[8] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1433–1440.

[9] T. Erez, K. Lowrey, Y. Tassa, V. Kumar, S. Kolev, and E. Todorov, "An integrated system for real-time model predictive control of humanoid robots," in *2013 13th IEEE-RAS International conference on humanoid robots (Humanoids)*. IEEE, 2013, pp. 292–299.

[10] H. Chen and F. Allgöwer, "A quasi-infinite horizon nonlinear model predictive control scheme with guaranteed stability," *Automatica*, vol. 34, no. 10, pp. 1205–1217, 1998.

[11] T. Erez, Y. Tassa, and E. Todorov, "Infinite-horizon model predictive control for periodic tasks with contacts," *Robotics: Science and systems VII*, p. 73, 2012.

[12] F. de Almeida, "Waypoint navigation using constrained infinite horizon model predictive control," in *AIAA Guidance, Navigation and Control Conference and Exhibit*, 2008, p. 6462.

[13] K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch, "Plan online, learn offline: Efficient learning and exploration via model-based control," in *International Conference on Learning Representations*, 2018.

[14] M. Bhardwaj, A. Handa, D. Fox, and B. Boots, "Information theoretic model predictive Q-learning," in *Learning for Dynamics and Control (L4DC) Conference*, 2020.

[15] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, "Reinforcement learning with deep energy-based policies," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2017, pp. 1352–1361.

[16] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic MPC for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 1714–1721.

[17] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Advances in Neural Information Processing Systems*, 2017, pp. 5360–5370.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[19] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[20] W. Sun, J. A. Bagnell, and B. Boots, "Truncated horizon policy search: Combining reinforcement learning & imitation learning," in *International Conference on Learning Representations*, 2018.

[21] S. Karaman and E. Frazzoli, "Optimal kinodynamic motion planning using incremental sampling-based methods," in *49th IEEE conference on decision and control (CDC)*. IEEE, 2010, pp. 7681–7687.

[22] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 2421–2428.

[23] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[24] A. Stentz, "Optimal and efficient path planning for partially known environments," in *Intelligent unmanned ground vehicles*. Springer, 1997, pp. 203–220.

[25] O. Arslan and P. Tsiotras, "Incremental sampling-based motion planners using policy iteration methods," in *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 2016, pp. 5004–5009.

[26] O. Arslan, K. Berntorp, and P. Tsiotras, "Sampling-based algorithms for optimal motion planning using closed-loop prediction," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 4991–4996.

[27] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[28] N. Wagener, C.-A. Cheng, J. Sacks, and B. Boots, "An online learning approach to model predictive control," in *Proceedings of Robotics: Science and Systems (RSS)*, 2019.

[29] Anonymous, "Blending MPC & value function approximation for efficient reinforcement learning," in *Submitted to International Conference on Learning Representations*, 2021, under review. [Online]. Available: https://openreview.net/forum?id=RqCC_00Bg7V