Planning under Uncertainty for Nonholonomic Vehicle in Continuous State Space

William Pope
ASEN 5519: Decision Making under Uncertainty – Final Project
3 May 2022

***Abstract*—Planning for vehicles in the real-world requires accounting for the various uncertainties present in the transition dynamics in order to generate efficient and safe trajectories. Formulating this problem as a Markov decision process allows us to reason about possible risks and rewards, and choose actions to optimize for desired behaviors. Because a continuous environment contains an infinite number of individual states, we need to implement online tree-based solution methods to shrink the problem. And because the system at hand has nonholonomic dynamics, value estimates rely on the Hamilton-Jacobi-Bellman equation to encode those constraints within the environment. Monte Carlo tree search proves to be highly effective at planning paths for a car-like robot, though further work is needed to test the limits of this approach.**

I. Introduction

Markov decision processes (MDPs) can be formulated to solve a wide variety of sequential decision-making problems, one of which is the guidance of an autonomous robot or vehicle. This is presented in its most basic form as the common Grid World environment, where an agent seeks to reach a goal region while avoiding penalties through stochastic state transitions. Using an MDP to plan a path in this scenario allows the agent to consider the risks and rewards associated with different actions, and execute a policy with the highest possible average return.

For a small number of discrete states, this problem can be solved using exact solution methods, such as value iteration or policy iteration. However as the size of the environment grows, it becomes computationally intractable to iterate through all possible pairs of states and actions to calculate their value. This is especially apparent when we move to a continuous state space, where there are an infinite number of states in any given region.

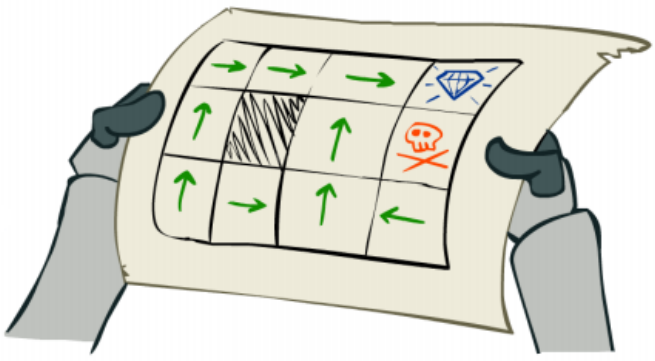

Fig. 1. The classic navigation MDP

One way to address this is to use online planning. Instead of analyzing the entire state space, we only look at the area around the agent's current state, planning one step at a time during execution. Furthermore, sparse sampling techniques only look at a subset of possible states, further lowering our computational cost. Online sampling techniques are even more essential when working with POMDPs, because the growth of the belief space makes all but the smallest problems intractable with exact methods.

One issue in sampling methods is that some trees may not reach the large rewards in an environment, and would therefore not return much information about optimal actions to take. This is especially true when an environment has a sparse reward structure, which is often the case in navigation problems. This is addressed through the use of a value estimate conducted at each leaf node, which can be used in the Bellman backup equation to calculate values back to the root. This estimate can come from some heuristic, such as the Euclidean distance between a state and the goal region. Other approaches involve rollout policies, where the system is simulated forward using some predetermined policy and returns the rewards collected.

A crucial element of online planning is that the algorithm must execute very quickly. If each action is set to execute for a tenth of a second, the MDP solver needs to be able to calculate the next action by the time the previous action terminates. The step size can be lengthened, but this produces jerkier motion and reduces the ability of the agent to react to changes in the environment. The runtime is influenced by a variety of factors, one of which is the choice of value estimate. If the algorithm has to simulate a new rollout trajectory for every new node, it will intuitively require more computation than a simpler approach (like calculating a distance).

One way to alleviate the time requirement of the value estimate is to compute value estimates for all possible states offline and store them, so the solver just needs to look up the estimate for its current node while executing. We will use this hybrid online-offline approach in our formulation, employing the Hamilton-Jacobi-Bellman equation to calculate the optimal value of every state for a given system and environment.

This project is intended as a primer on the use of DMU

tools for path planning applications. The scenario is likely not an exact match for one requiring an MDP, but it stands in for more complex partially-observable scenarios to come in my own research.

II. Background and Related Work

When using traditional motion planning techniques, the simplest way to handle uncertainty is by adding safety margins to obstacles to over-approximate any possible transition uncertainty. This may work for some applications, but it is overly conservative and can miss potential paths. Other approaches allow the agent to execute a trajectory blindly, then stop and replan once it has strayed too far from the original path. The idea of using MDPs in motion planning is mentioned in Lavalle's 2006 *Planning Algorithms* [5], but at that date, methods for large/continuous state spaces had not been widely introduced.

This project uses the Hamilton-Jacobi-Bellman equation because it can compute an optimal cost function across a state space while considering the dynamics of the system. This equation is born of out optimal control methods, but can be used to plots path for vehicles with dynamic constraints, such as a car. This approach is introduced and refined in [1] and [2]. In these works, they compute the value function for a car-like vehicle in an obstacle-filled environment, navigating to some goal region. This value function is a central part of our approach, since a value estimate is needed to implement a sampling-based MDP solver. However these authors use the value function to directly compute a path through gradient descent. This proves to be less accurate than taking samples from the value function (as done in the MDP), because the central difference calculations create larger boundaries around obstacles.

Current approaches to motion planning under uncertainty include work by H. Kurniawati, who introduced Adaptive Belief Tree in 2016 for planning in dynamics environments [3]. As compared with the HJB method used in this project, ABT allows the planner to adapt to changes in the environment while preserving information from previous solutions.

III. Problem Formulation

The scenario we want to implement is the navigation of a car-like vehicle through a 2-D environment to some goal region, within the shortest amount of time possible. The environment is populated with impassable obstacles that the agent has knowledge of ahead of time. The vehicle is modeled using a curvature-constrained bicycle model, which is a simplified version of the full dynamics of a car, as described in [5].

The vehicle's state vector consists of three elements:

- x, the x-axis position
- y, the y-axis position
- θ, the heading angle with respect to the x-axis

The vehicle's control vector consists of two elements:

- u_v, the forward velocity
- u_ϕ, the steering angle with respect to the car's main axis

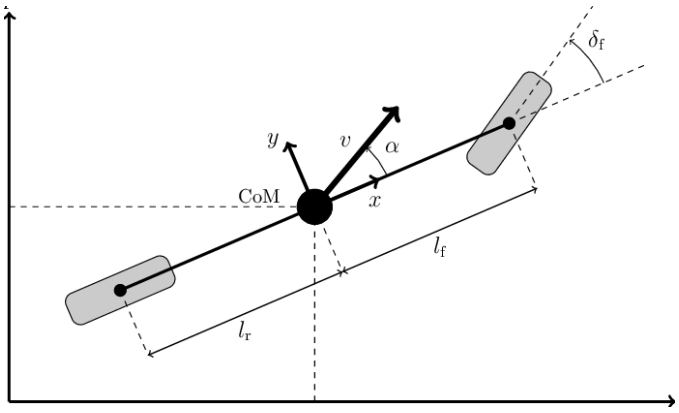

Fig. 2. Bicycle kinematic model

The nonlinear equations of motion are formulated as:

$$f(s,a) = \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}u_v \cos(\theta)\\ u_v \sin(\theta)\\ u_v(1/l)\tan(u_\phi)\end{bmatrix}$$

where l is the length of the wheelbase, which impacts the turning radius.

In order to account for uncertainty in the dynamics, some amount of zero-mean Gaussian noise is added to the velocity and steering angle. This covers variation between the input commanded by the planner and the input executed by the actuators, as well as unmodeled environmental factors like friction and slippage. The equations of motion are updated to include this, where w_v and w_ϕ are randomly sampled from a normal distribution defined by some standard deviation.

$$f(s,a,w) = \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}(u_v + w_v)\cos(\theta)\\ (u_v + w_v)\sin(\theta)\\ (u_v + w_v)(1/l)\tan(u_\phi + w_\phi)\end{bmatrix}$$

The vehicle is modeled as a rectangle for the purpose of collision checking. It is 0.75 m long, 0.25 m wide, and has a wheelbase of 0.5 m that is centered length-wise. The car's state position (x, y) is the position of the center of the rear axle, which is 0.25 m behind the vehicle's geometric center.

The MDP is defined by state space S, action space A, transition function T, and reward function R. The state space is defined as a continuous domain over x [m], y [m], and θ [rad]:

$$S = [-10, 10] \times [-10, 10] \times [-\pi, \pi] \subset \mathbb{R}^3$$

The action space is defined as a discrete set of combinations of u_v and u_ϕ. Input u_v has limits of 1.0 m/s and -0.75 m/s, but the time-optimal speed is to choose one of the largest values, so $u_v \in \{-0.75, 1.0\}$ m/s. Input u_ϕ has limits of 0.5 rad and -0.5 rad, but is discretized to keep the action space finite, so $u_\phi \in \{-0.5, 0.0, 0.5\}$ rad. In most cases, the time-optimal steering angle is 0.0 or one of the maximums, but sometimes a slower turn is needed. The resulting action space is:

$$A = \{[-0.75, -0.5], [-0.75, 0.0], ..., [1.0, 0.5]\}$$

$$|A| = 6$$

The transition function is defined according to the system dynamics $f(s, a, w)$. At every time step, the system transitions from s to s' by integrating the state derivative:

$$s' = T(s, a) = s + \int_0^{dt} f(s, a, w) d\tau$$

The reward function is defined over the state and action spaces to reflect our goal of reaching the target region in the shortest amount of time possible. In order to align with the offline value estimate (detailed later), every step taken outside the goal region is given a negative reward equal to the time taken for that step, i.e. $r = -dt = -0.1$. The goal region is a terminal state, so the only way to maximize reward over a trajectory is to get to goal with the fewest amount of steps.

$$R = \begin{array}{cc|c} s & a & R(s,a) \\ \hline \neg \text{ goal} & \text{any} & -dt \end{array}$$

IV. Solution Approach

A. Hamilton-Jacobi-Bellman Computation

We begin by defining a value function v over the 3-dimensional state space, given some target set $\mathcal{T} \subset S$.

$$v(s) = \inf\{t | z \in \mathcal{A}_s, z(t) \in \mathcal{T}\}$$

z is a trajectory starting at state s and $\mathcal{A}_s$ is the set of all possible trajectories from s. In plain language, the value at any given state is the shortest possible time-to-goal. This means the value function is strictly non-negative and has a single global minimum located at the target set. Since the HJB formulation comes from the world of optimal control, it deals in cost instead of reward, so states further from the target have "higher" values. In order to use this with our MDP solver, we just flip the signs at the end of HJB calculation.

For a vehicle with a minimum turning radius, the fastest path between any two points is a series of straight-line segments ($u_\phi = 0$) and tight turning circles ($u_\phi = u_{\phi,max}$) [5]. Speed should also be maximized. Therefore the optimal action at any given point is one in the discrete set A defined previously. These actions will be used when calculating the HJB value at each state.

The value function can be computed using the dynamic programming principle:

$$v(s(t), t) = \min_a \{v(s(t+dt), t+dt) + \int_t^{t+dt} g(s(\tau), a(\tau)) d\tau\}$$

By applying a Taylor series expansion, dividing by dt and taking the limit as $dt \to 0$, we arrive at the final Hamilton-Jacobi-Bellman partial differential equation, which takes the dot product of the value function gradient and the state derivative. Intuitively, this aligns the maximum rate of change of the value function with the direction of the system's evolution, in order to achieve local optimality.

$$-1 = \inf_{u_v, u_\phi} \left\{ \frac{\partial v(s,t)}{\partial s} \cdot f(s, a) \right\}$$

This can be expanded into:

$$-1 = \inf_{u_v, u_\phi} \left\{ \frac{\partial v}{\partial x}\dot{x} + \frac{\partial v}{\partial y}\dot{y} + \frac{\partial v}{\partial \theta}\dot{\theta} \right\}$$

We can now use numerical methods to solve the PDE for $v(s)$. In this project we'll use the finite difference method, which uses a known value and approximations of the derivatives to calculate the value function on a discrete grid. The step size is an important factor in determining the runtime, but the grid must be fine enough that small details in the physical environment are not skipped over. For a 20 m environment, we choose step size h_{xy} = 0.25 m, providing 80 grid points along each axis. A comparable discretization for $\theta \in [-\pi, \pi]$ is $h_\theta = 0.087$ rad, which is about 5 degrees and creates 72 grid points. In total, the PDE solver will have to iterate over $460,800$ grid points in $\mathbb{R}^3$, taking several minutes to complete the procedure. The grid nodes are indexed with i, j, and k along each axis.

The partial derivatives with respect to x and y are approximated as:

$$(\cos(\theta) v_x)_{ijk} = |\cos(\theta_k)| \frac{v_{i+\xi_k, j, k} - v_{ijk}}{h_{xy}}$$

$$(\sin(\theta) v_y)_{ijk} = |\sin(\theta_k)| \frac{v_{i, j+\nu_k, k} - v_{ijk}}{h_{xy}}$$

The partial derivative with respect to θ is approximated as:

$$(|v_\theta|)_{ijk} = \max\left\{ \frac{v_{ijk} - v_{i,j,k+1}}{h_\theta}, \frac{v_{ijk} - v_{i,j,k-1}}{h_\theta}, 0 \right\}$$

These approximations are plugged into the HJB PDE alongside the system dynamics to solve for the value v_{ijk} as a function of its neighboring values in a sweeping and repeating process, generating unique values $(G^{+1}_{ijk}, F^{+1}_{ijk}, G^{-1}_{ijk}, F^{-1}_{ijk})$ for each action combination. For further implementation information, readers can refer to [1] and [2]. The calculation process begins by initializing the value at every state in the grid according to this rule:

$$v^0_{ijk} = \begin{cases} 0 & \text{if } (x_i, y_j, \theta_k) \in \mathcal{T} \\ \infty & \text{otherwise} \end{cases}$$

Then for all (i, j, k) in the grid, the value v^n_{ijk} is computed with the following update scheme for $n = 1, 2, ...$, where each iteration n visits every node in the grid:

$$v^{n+1}_{ijk} = \min\{G^{+1}_{ijk}, F^{+1}_{ijk}, G^{-1}_{ijk}, F^{-1}_{ijk}, u^n_{ijk}\}$$

Importantly, the value is not updated at nodes that fall within obstacles or on the boundaries of the environment, meaning their value remains fixed at ∞. Iterations n cycle through 8 different index orderings in a Gauss-Seidel sweeping scheme, such that the indices are first iterated as [i-forward, j-forward, k-forward], then as [i-forward, j-forward, k-backward] and so on for all 8 combinations. Through this implementation, the value at each grid node will eventually converge to a final approximation.

The resulting value function is plotted as a heat map in Figs. 3 and 4, showing slices taken at $\theta = \pi/2$ rad and

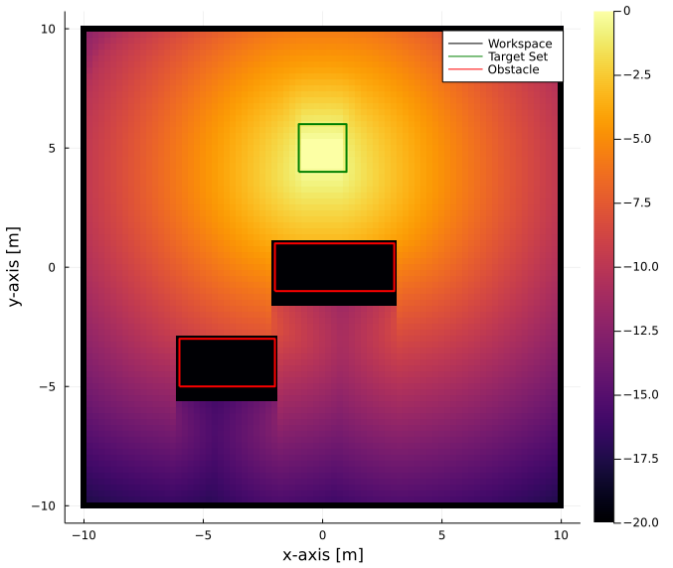

Fig. 3. HJB value function, sliced at $\theta = \pi/2$ (vehicle pointing N). Note that the sign of the values has been flipped for use in MDP/POMDP frameworks

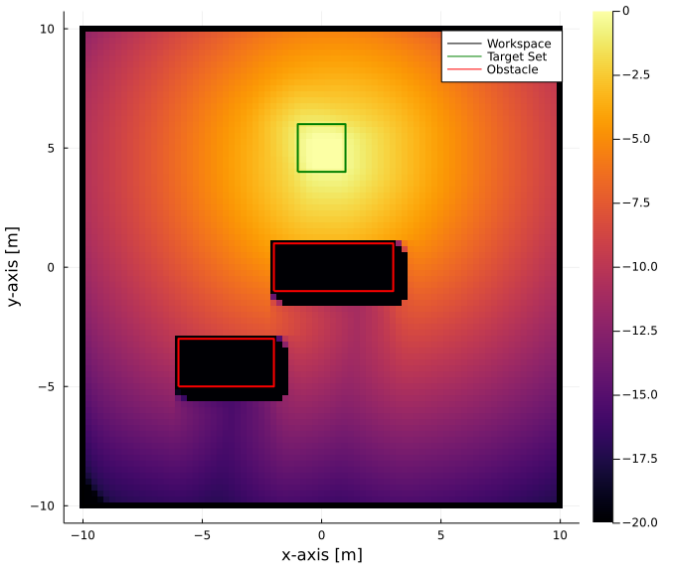

Fig. 4. HJB value function, sliced at $\theta = 3\pi/4$ (vehicle pointing NW)

$\theta = 3\pi/4$ rad. Recall that the value is the optimal time-to-go when the vehicle occupies that state in (x, y, θ). Because of the constraint on turning radius, θ can change the value of the same (x, y) point in different slices, because the vehicle may be pointing away from the target and need to take a longer path to the goal.

Since our vehicle has finite dimensions, any state that causes part of the vehicle to overlap with an obstacle is considered part of the obstacle set and left at $v_{ijk} = \infty$. The vehicle is modeled as a rectangle, so the obstacle region differs between different orientations θ. The gap between the two obstacles is only 2 m, while the vehicle is 0.25 m wide and the HJB step size is $h_{xy} = 0.25$ m. In theory the vehicle can pass through a gap as small as 0.25 m, but if the HJB step size were any larger, that gap may be missed in the grid.

In order to better illustrate the properties of this value function, we can make the scenario "harder" for the vehicle by increasing its minimum turn radius, reducing its maximum reverse speed, and constraining allowable orientations in the target set. The value function plotted in Fig. 5 takes on a more interesting appearance under these new constraints.

It is possible to plan optimal trajectories to goal by simply following the gradient of the HJB value function. This can be a good option in fully/mostly deterministic motion planning settings, since this is essentially equivalent to the fast marching method. However for systems with uncertainty in their transitions and/or states, MDP and POMDP approaches are a more robust way to tackle the planning problem. Here, we use the HJB value function to determine how "good" a potential state is, then use further machinery to choose actions. Because the value approximation is only defined at the nodes of the grid, we use a multivariate linear interpolation scheme to calculate the value at off-grid points for use in MCTS value estimates.

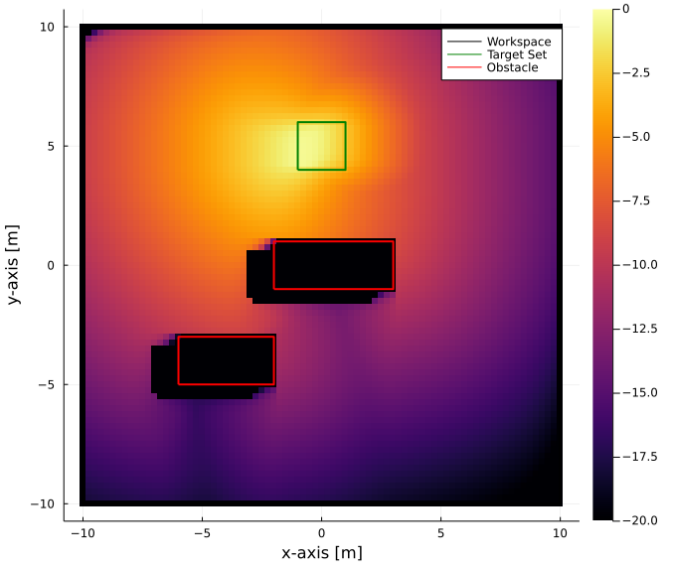

Fig. 5. HJB value function, sliced just above $\theta = 0$ (vehicle pointing ENE). The turning radius and reverse speed have been downgraded, and the target set only accepts orientations in $\theta_{\mathcal{T}} \in [-\pi/16, \pi/16]$

B. Monte Carlo Tree Search

Continuous state MDPs have an infinite number of possible states, so you cannot use traditional solvers that need to calculate a value for every state. One approach is to use an online tree search algorithm that samples states from the infinite state space. This approach requires tracking the value at only a small number of states, which we can use rollout estimates for.

Monte Carlo tree search is one of these online sampling-based methods, and it uses a finite number of forward sim-

ulations to grow a state-action tree from the current state (s_0). There are four key steps: search, expansion, rollout, and backup. We keep track of count $N(s,a)$ and value estimate $Q(s,a)$ as new branches are added, ultimately executing the highest-value action at s_0 and restarting the tree from s_0'. For an infinite state MDP, we need to define a maximum number of child nodes for each state-action pair, otherwise the tree will never grow deeper than the first layer. The full procedure for each simulation is detailed here [4]:

1) Search: Each simulation begins at the root node (s_0) and chooses an action (a_{UCB}) according to the UCB heuristic. If the $s_0 - a_{UCB}$ pair has already filled its child layer, then one of those states is chosen at random, and the search continues from that node. Eventually we will reach a state-action pair that has not filled its child layer, at which point we have an opportunity to expand the tree.

2) Expand: Expansion occurs when we add a new state node to the tree. In the *Search* step we followed the UCB heuristic to navigate to the most productive region of the tree (with respect to the exploration-vs-exploitation problem). With our unfilled state-action pair, we feed s_{edge} and a_{UCB} into the MDP's transition function $T(s,a)$ to propagate the system one time step ahead to state s', which is added to the search tree.

3) Rollout: Depending on the problem, we can either simulate a rollout policy or calculate a value estimate directly. For this problem, a rollout would entail planning a path from s' to the target set using motion planning techniques. This is likely not feasible due to the time it would take to calculate. A direct estimate such as the distance between state (x,y) and the target set is not practical either because there could be obstacles in the way and the vehicle has nonholonomic dynamics which constrain its movements. This is why we use the Hamilton-Jacobi-Bellman equation to calculate time-to-go over the entire state space before beginning MCTS. By simply interpolating the grid, we can instantly return a nearly exact estimate of the new state's value, allowing us to move onto the backup step.

4) Backup: The purpose of running multiple simulations from the same root node is that we can collect more value estimates from the downstream environment that are then used to refine estimates of $Q(s_0,a)$. This happens by working back up the tree to the root, updating the value $Q(s,a)$ at each node along the way with a backup equation and an averaging equation:

$$Q_{new}(s,a) = R(s,a) + \gamma Q_{new}(s',a)$$

$$Q'_{est}(s,a) = Q_{est}(s,a) + \frac{Q_{new}(s,a) - Q_{est}(s,a)}{N(s,a)}$$

The counter $N(s,a)$ is also incremented for every state-action pair that was visited during the simulation. With $Q(s,a)$ and $N(s,a)$ updated along the node path of the simulation, the process begins again with *Expand*.

Once the time limit for each online planning step is reached, the agent chooses whichever action $a \in A$ that has the highest value $Q(s_0,a)$, then executes it for one time step. While the action is executing, MCTS restarts with an empty tree to calculate the next action.

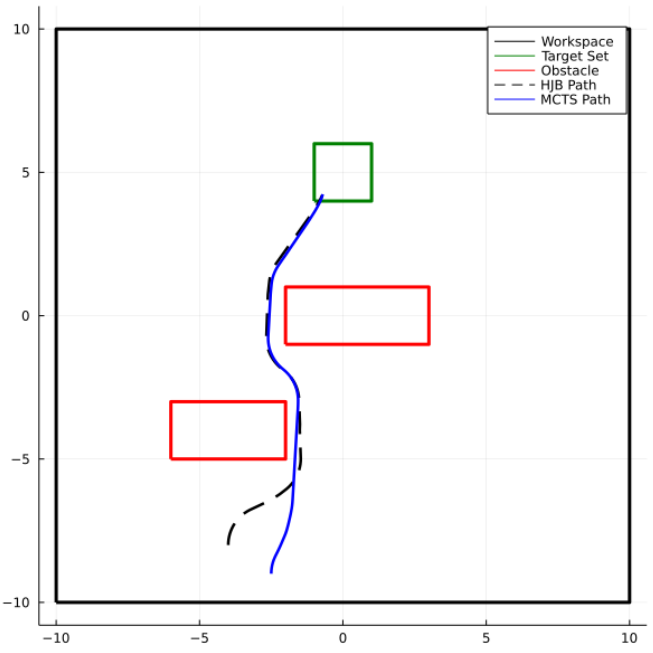

Fig. 6. Paths generated by MCTS and HJB gradient over deterministic transitions

V. Results

A. Basic Path Planning

The first result to look at is the MCTS planner's behavior when there is zero uncertainty. If every transition is deterministic, then the HJB value estimates will be the exact value of each state, and MCTS should be able to plan a perfectly optimal path. This will verify that the planner's mechanics generally work as expected. We can use the HJB value function's gradient as a separate method of generating the optimal path, then compare the two. These two paths are plotted in Fig. 6. Starting from similar areas, both planners choose an appropriate route between the obstacles. MCTS actually produces a slightly better route, because the HJB gradient method is less accurate and thus takes a wider path around obstacles. This result confirms that MCTS is behaving as expected, at least under no uncertainty.

B. Planning under Transition Uncertainty

We now apply noise to the control inputs, as detailed in the problem formulation. This small amount of Gaussian noise creates stochastic transitions, throwing the vehicle slightly off course every step. Three different levels of noise are tested and plotted in Fig. 7. Recall that w_v and w_ϕ are sampled from normal distributions and added on to the commanded input vector at every step.

- σ_1: $w_v \sim \mathcal{N}(0, 0.02)$ m/s, $w_\phi \sim \mathcal{N}(0, 0.01)$ rad
- σ_2: $w_v \sim \mathcal{N}(0, 0.10)$ m/s, $w_\phi \sim \mathcal{N}(0, 0.05)$ rad
- σ_3: $w_v \sim \mathcal{N}(0, 0.50)$ m/s, $w_\phi \sim \mathcal{N}(0, 0.25)$ rad

The path for σ_3 is a bit jerkier than the others, but generally the system behaved essentially the same with each amount of uncertainty. The maximum forward speed is only 1.0 m/s, so adding noise with a deviation of 0.5 m/s represents a pretty significant amount of volatility. However, several factors contribute to the behavior observed.

First, the scaling between the size of the environment, the size of the vehicle, and the step size is very favorable to the agent. For a step size of $dt = 0.1$ sec and a top speed of 1.0 m/s, the vehicle hardly covers any ground in each step. The paths shown are made up of over 140 individual individual steps. This means that the system can quickly recover from any adverse transitions, and it would be very difficult for the system to end up somewhere it didn't want to be. Additionally, the nonholonomic dynamics of the vehicle mean that over very short time periods, its motion is almost entirely parallel with its heading angle. There are no stochastic transitions that cause the car to jump to the side, so it can navigate close to the side of obstacles with little risk of collision, which can be observed as it passes through the gap in our environment. Finally, despite the noise added to the input vector, each of the actions in A is so different from the others it would be difficult to misattribute a value estimate to the wrong $Q(s, a)$. For example in the grid world environment, if we commanded :up and the system transitioned to the right, the value backup would return the value for the right cell, and we would need more simulations to approach a value estimate more representative of the proobability of transitioning to the expected cell. In this system, the state transitions for a given action will rarely overlap with the state transitions of another action, meaning we can reliably estimate values for $Q(s, a)$ with relatively few simulations.

In a sparse sampling implementation, there needs to be a bound on the number of child nodes a given state-action pair can grow. For simplicity, this project set a constant value of $w_max = 3$, although [4] discusses setting it with hyperparameters and $N(s, a)$. This means that for every single state node with 6 possible actions, each of those actions was allowed to evolve into 3 new state nodes, creating 18 new nodes from every original parent. At this rate, the tree becomes fairly wide, quickly. While just 2 layers deep, the tree contains a total of $1 + 18 + 18^2 = 343$ state nodes, also written as:

$$n = w|A|^0 + w|A|^1 + w|A|^2$$

Going past 2 layers leads to a significant expansion of the tree, and each of these nodes requires a new simulation to be created. Setting a larger number of children for each state-action pair can be beneficial because it allows for further sampling of $s' = T(s, a)$, however the size of the tree quickly becomes an inhibitor. Anecdotal timing of the each search tree showed that 250 simulations could execute in about 0.02 seconds, so there is still some margin to remain under the $dt = 0.1$ sec cap. This is also likely the most basic form of a sparse sampling tree, where more advanced methods help address the depth vs width issue.

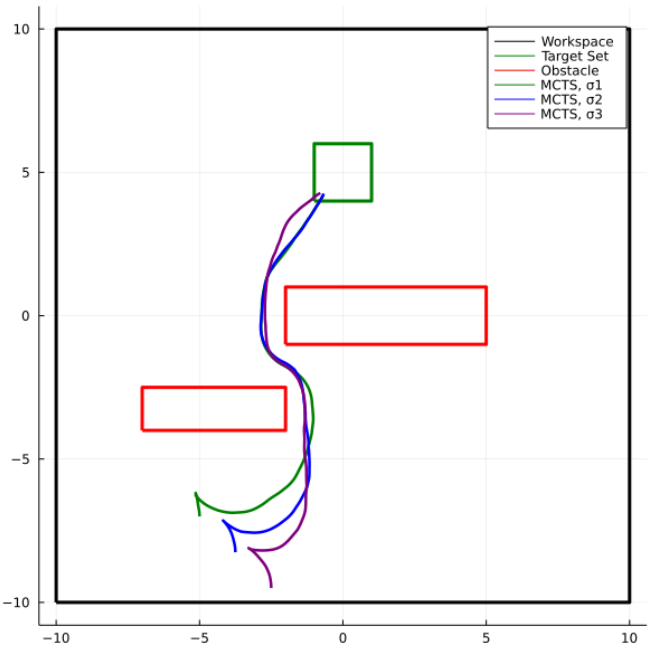

Fig. 7. Paths generated by MCTS corresponding to different levels of transition uncertainty

C. Impact of HJB Value Function

The most consequential element of this system is the HJB value function in use. Computing this function offline gives the solver access to a near perfect value estimate at every state in the state space. Because the value function is globally ascending and the time step is short, there is little reason to search more than 1-2 layers deep in our tree. Whichever action has the best value in one step essentially denotes the action that maximizes the gradient at the current state, so choosing $\max_a Q(s_0, a)$ is essentially just a gradient ascent scheme.

However, the HJB scheme has its drawbacks for implementation in this kind of scenario. The main issue is that it is computationally expensive to calculate, and is not tolerant to changes in the static environment in its current form. For example if a new obstacle was introduced that blocked the gap in the environment, the HJB value function would still indicate that that region offered the shortest path to goal. This will leave the vehicle stuck in a newly formed local minima unless a search tree is deep enough to find a new route past the obstacle. Since our current search trees are only 2-3 steps deep, that doesn't seem likely without modification. One approach to explore in future work is to recalculate all or parts of the HJB value function online, such that new obstaceles can be accounted for. Any partial calculation would still require starting iteration at/near the target set, since the HJB formulation relies on dynamic programming based on a known true value ($v_{\mathcal{T}} = 0$). Even a coarse version of the value function takes several seconds to compute, as shown in Fig. 8, a plot of several collected computation runtimes.

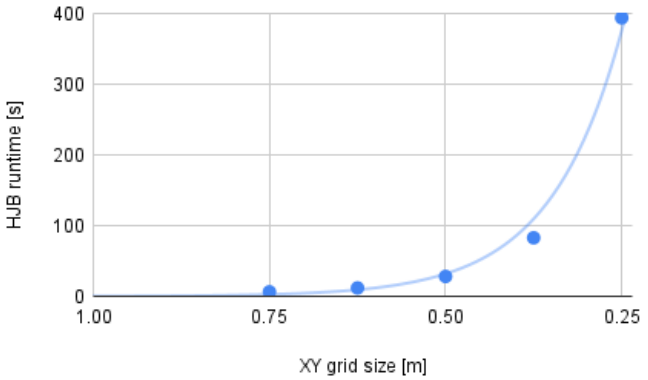

Fig. 8. Time required to calculate HJB value function as grid size is refined

VI. Conclusion

Ultimately, the MCTS planner performed well for the task at hand and had no issues even when significant amounts of uncertainty were added to the transition function. The planner was greatly helped by the rtainty were added to the transition function. The planner was greatly helped by the use of the Hamilton-Jacobi-Bellman equation for computation of a global, near-exact optimal value function for use in rollout estimates. This allowed the planner to take locally optimal actions, even though the search tree itself only extended about 2-3 steps into the future. For all its usefulness, the HJB value function is cumbersome to initially calculate, and is not easily adaptable to new obstacles in the environment. Future expansions of this project might introduce a fully online value estimate, which will likely come with its own computational constraints due to the nonholonomic dynamics of the vehicle. MCTS has shown to be as effective as expected in an infinite state space, but further refinement is needed to make this implementation ready for more challenging conditions.

VII. Contributions and Release

This project was conducted by William Pope, who grants permission for this report to be posted publicly.

References

[1] Takei, R., Tsai, R. "Optimal Trajectories of Curvature Constrained Motion in the Hamilton–Jacobi Formulation," J Sci Comput 54, 622–644 (2013). https://doi.org/10.1007/s10915-012-9671-y

[2] C. Parkinson, A. L. Bertozzi and S. J. Osher, "A Hamilton-Jacobi Formulation for Time-Optimal Paths of Rectangular Nonholonomic Vehicles," 2020 59th IEEE Conference on Decision and Control (CDC), 2020, pp. 4073-4078, doi: 10.1109/CDC42340.2020.9303861.

[3] Kurniawati, H., Yadav, V. (2016). An Online POMDP Solver for Uncertainty Planning in Dynamic Environment. In: Inaba, M., Corke, P. (eds) Robotics Research. Springer Tracts in Advanced Robotics, vol 114. Springer, Cham. https://doi.org/10.1007/978-3-319-28872-7_35

[4] M. J. Kochenderfer, *Algorithms for Decision Making*. Cambridge, MA: MIT Press, 2022.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge, MA: Cambridge University Press, 2006.