Safe Planning & Parking for a Wheeled Robot in Unstructured Environments

Chinmay Garg*†, Jaya Aadityaa Ganapathy Subramanian*†, Namya Bagree*†, Sourojit Saha*†

***Abstract*— Autonomous, non-holonomic wheeled robots are being increasingly used in unstructured environments for operations like exploration, search and rescue, etc. Many situations may require robots to park themselves during navigation and operation in these types of environments. These situations can arise due to various reasons such as sensor failure, unfavorable operating conditions, etc., but specifying a safe parking spot and navigating to it safely can be difficult in unstructured environments. Moreover, for certain applications these robots usually operate in groups of heterogeneous robots and have to efficiently plan their routines to prevent interference among themselves. This creates a need for each robot to plan out feasible paths to navigate and also make space for other robots by safely moving aside. In this report, we aim to develop a planner for non-holonomic wheeled robots to reach any given goal position and orientation safely in unstructured environments with multiple obstacles. We also present a novel solution to efficiently find a safe parking location for a vehicle in its surrounding environment. This allows the vehicle to either safely stop when faced with adversarial scenarios or allow other robots in its team to navigate through the environment. We demonstrate the success of our approach through simulated and hardware results on mobile ground robots.**

I. INTRODUCTION

Most autonomous vehicle research is focused on operation in structured environments where vehicles can park in a safe and designated spot. However, there is an increased focus on research in the robotics community on fast moving ground vehicles in unstructured environments. Autonomous systems in unstructured environments are faced with complexities as there are no set rules that can be applied to the planner, making navigation and search for a safe stop/parking spot a non-trivial research problem.

Autonomous mobile robots are used in unstructured environments for multiple use cases, with one of them being search and rescue operations. To explore regions in a time efficient manner, robots work in a group [9]. This leads to scenarios where explorer robots scan the region and provide information to robots with enhanced capabilities to reach a desired location. An important feature in such conditions is to make way for incoming robots. In unstructured environments, this can necessitate the initial robots to shift to a region in a desired orientation while avoiding obstacles.

This gives rise to our target problem, planning for a wheeled robot in an unstructured environment to safely navigate to a location and park in the desired orientation. We develop and run a planner for non-holonomic vehicles in an explored environment. The planner makes use of a ROS framework for easy integration with a real vehicle setup and is implemented on both, simulation and results in multiple environments with no structure. A safe parking planner has also been developed based on potential maps based on the environment to generate a feasible path to park and get out of the way for any other potential hindrances.

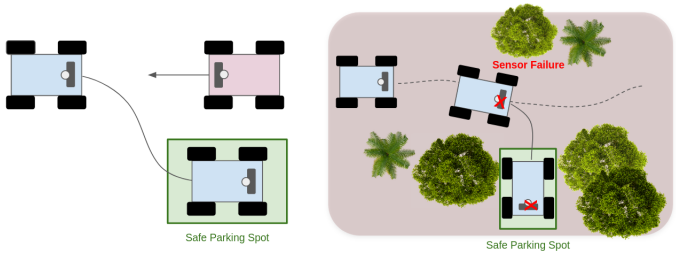

Fig. 1: Safe planning and parking diagram

II. METHOD

A. Map Generation

For map generation, we used LIDAR based SLAM algorithm, Super Odometry [11]. It was used to generate a 3-D point cloud of the environment. The point cloud generated had a resolution of 20cm for each voxel. This point cloud was was taken as the basis for map generation for our planner. The map we used was in the form of occupancy grid. Each grid cell in the occupancy grid could have three values, -1 meaning the state of the grid is unknown, it can be an obstacle or a Open area, 0 meaning the grid is a Open area, and 1 meaning the grid is occupied by obstacle. Fig. 3 and 2 show the maps used in our planner. Conversion of point cloud to occupancy grid was accomplished in the following steps:

- Firstly, the point cloud was taken, and the its range in X and Y direction was calculated. This was then used as the size of the occupancy grid. All the grids in the occupancy grid were initialised with a value of -1.
- For all the points from the ground level up-to the height of the robot, the corresponding grid cells were assigned a value of 1. This was because they were considered as obstacles, as they would cause collision with the robot.
- The points which were above the robot height were considered as obstacle free and the corresponding grid cells were assigned a value of 0.

*Authors contributed equally
†Mechanical Engineering Department, Carnegie Mellon University, Pittsburgh, PA, USA {chinmayg, jganapat, nbagree, sourojis}@andrew.cmu.edu

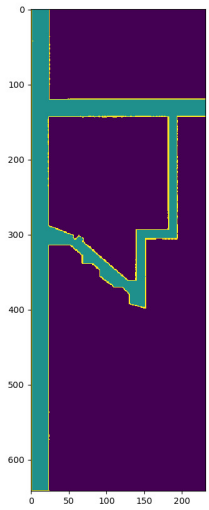

Fig. 2: MIT Tunnel Map (Green: Open area; Yellow: Obstacle; Purple: Unknown)

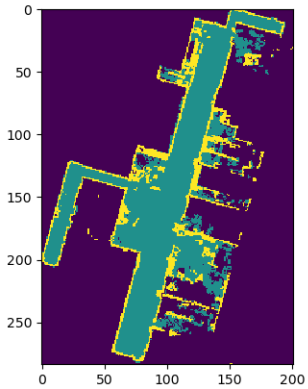

Fig. 3: NSH B-Level Map (Green: Open area; Yellow: Obstacle; Purple: Unknown)

- Finally to account for the scaling between the point cloud and the occupancy grid, the occupancy grid was interpolated by a factor of 5 in both X and Y direction.

B. Lattice State Space Generation

Many robots are non-holonomic and cannot move instantaneously in any direction. Consequently, it is necessary to generate a lattice to implement a search algorithm. In [6], a multi-resolution lattice was generated offline for a given map accounting for the motion constrains of the car, in order to generate a smooth, feasible path. In our implementation, the complete lattice is not generated offline, but rather it is generated online using precomputed motion primitives.

The state space of the non-holonomic robot was discretized in x, y and θ. The grid resolution for x and y was 0.2 metres and for θ was $22.5°$. For each discretized θ, a set of motion primitives were generated which depicted transitions between states. The primary considerations while generating was the feasibility of the motion primitives to be tracked by a car. In addition to this, another key aspect was having each motion primitive action to result in a state part of the discretized state space. During planning, the motion primitives are used to generate the successors of any given node and implicitly construct the lattice for search. In order to reach feasible motion primitives, the following implementations were explored:

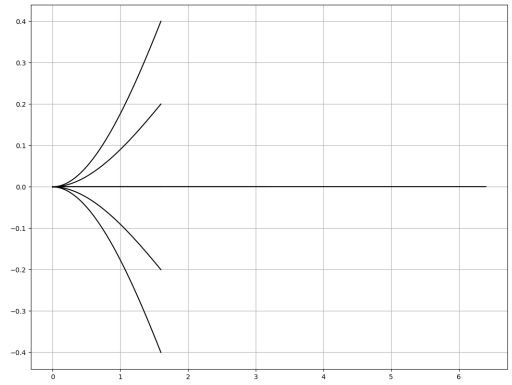

Fig. 4: Spline Based Primitives

1) Dynamics based primitive: A kinematic bicycle model [7] is a common way to model a path that a ground vehicle would actually follow given a set of desired steering angle and velocity. An initial trajectory to be given as input to the controller is needed for this. The initial and final positions were connected by an almost circular trajectory. An LQR controller was run over the initial and final desired states based on the discretization of the state space. Despite setting providing a high goal-position cost, the controller wasn't able to provide paths that exactly matched the given goal position. This could be solved by getting a better initial trajectory, or be left to the controller to solve for in real time when multiple paths append themselves. As we don't account for the steering angle in our state space, multiple primitives from this method connected wouldn't map to exactly feasible paths. We decided to let the controller handle accurate following of the motion primitives that were constrained as discussed ahead.

2) Spline based primitive: A spline can generate path between the start and end location, taking into account the orientations at the start and end points as well. Cubic spline was used to generate the path between two such points. However, no curvature constraint can be put on cubic spline, which was a drawback for the spline based primitive. Five spline based motion primitives were generated for end orientations of $-45°, -22.5°, 0°, 22.5°, 45°$. These are shown in Fig. 4

3) Dubins path based primitive: Dubins path [2] determines the shortest curve connecting any two points in a two-dimensional Euclidean plane (i.e. x-y plane) with a minimum turning radius constraint on the curvature of the path and terminal tangents. Using Dubins path, the motion

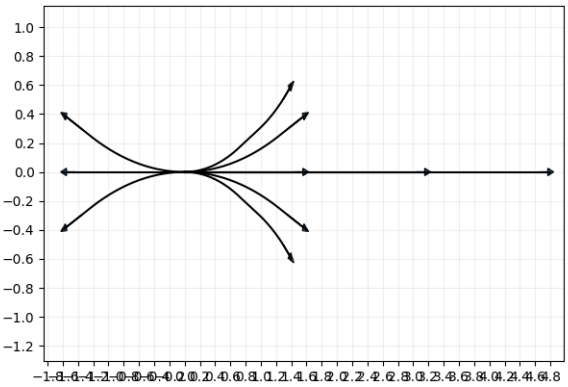

Fig. 5: Dubins Primitives

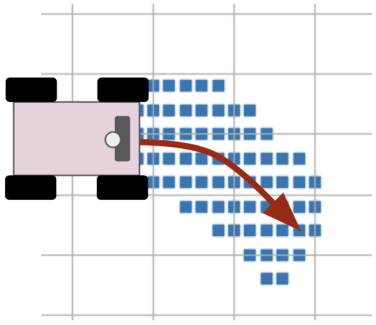

Fig. 6: Grids covered for 45° motion primitive

primitives can be generated such that the terminal state of each Dubins path could be directed to land back on the discretized state space. These factors made this approach extremely feasible for our implementation. Hence Dubins path was used to generate the motion primitives of the robot for any orientation in the discretized state space. A minimum turning radius of 1.5m for the experiment ground robot was used. Two sets of motion primitives were generated - one with only forward primitives, and one with both forward and backward primitives. The forward primitives were generated at $-45^\circ, -22.5^\circ, 0^\circ, 22.5^\circ, 45^\circ$ and backward at $-22.5^\circ, 0^\circ, 22.5^\circ$. Two additional primitives were generated at 0° with twice and four times the length of first primitive. This allows faster convergence to the goal in relatively open environments where the planner can take larger steps towards the goal. The generated Dubins primitives can be seen in Fig. 5.

C. Collision Avoidance

Fast collision avoidance checking is necessary for implicitly generating the lattice while running the planner online. In addition to this, the robot is not a point robot and its form factor needs to be accounted for in collision checking.

In order to tackle this, the grid cells covered by a robot while moving in each motion primitive was calculated for collision checking. At any given state, the robot was approximated as a uniformly distributed set of points describing a rectangle, conservatively approximating the shape of the robot. The grids covered at the given state were stored and this was repeated for all points along each motion primitive. Hence for each primitive, the grids covered by robot traversing that motion primitive were calculated for fast collision avoidance check during the search. An example of the same can be seen in Fig. 6.

D. Planner

Graph representation: Each lattice state in the graph is represented by $< x, y, \theta >$, where x and y are the indices in the discretized grid space and θ is the orientation of the robot. We generate the lattice graph implicitly by generating successive lattice states while running the weighted A* search on the occupancy map. For generating the successors of each node in the lattice graph, we check if the primitive end state is in the open space and if there are no collisions using the precomputed grid cells for each motion primitive.

Lattice Costs and Heuristics: The cost g of each motion primitive is defined by the length of the primitive itself, for both the forward and the backward primitives. For the heuristic h in the weighted A* search, we use a combination of Dubins path length [10] and backward Dijkstra costs[1]. The Dubins path length takes acts as the mechanism constrained heuristic for the non-holonomic wheeled robot and the backward Dijkstra takes the obstacles from the environment into account being the environment constrained heuristic. The backward Dijkstra heuristic costs can be seen in Fig. **??**. Although, Dubins path length is not consistent when including backward primitives, but as both the heuristic functions are admissible, we use a maximum of the two heuristics as our combined heuristic for the weighted A* search.

$$g = \text{Length of Motion Primitive}$$
$$h = \max(h_{Dubins}, h_{BackwardDijkstra})$$

Weighted A-star: We implemented a Weighted A* Search with implicit lattice-graph generation to search from the start point to the goal, and find a path. The heuristic is inflated with a weight to bias the search to move towards the goal and improve the speed. This weight also allows us to emulate different searches, where with a weight of 0 the effect of heuristic is completely removed (Dijkstra's Search) and with a weight of 1 it becomes a simple A* Search.

E. Traversability Map and Safe Parking

While a robot is navigating a path it is possible it encounters a situation where the robot should safely park itself at a location. But, in unstructured environments there are no well-defined parking spots. Therefore, to find safe parking locations in an unstructured map, we compute a parking cost Ψ for the environment in a known map. The cells in the grid space with the lowest Ψ are then considered to be the safe

parking spots and in case of any adversity the robot safely plans a path to the safe parking spot.

The parking cost Ψ is defined as a weighted combination of the Safe Parking cost which indicates the overall safety of the parking spot, and the Forward Dijkstra cost from the current location which indicates the cost of reaching the parking spot. The weights of these costs can be modified based on the situation that the robot is facing. For instance, if the robot should go to a safer parking location without accounting for the time it takes to reach the parking spot, then λ_{sp} can be increased and λ_{fd} can be decreased.

$$\Psi = \lambda_{sp} \times C_{SafeParking} + \lambda_{fd} \times C_{ForwardDijkstra}$$

The forward Dijkstra Cost $C_{ForwardDijkstra}$ is essentially the cost of the shortest path from the robot's current location to the safe parking spots. We define the Safe Parking Cost $C_{SafeParking}$ as a combination of three different costs, Traversability Cost, Obstacles Cost and Backward Dijkstra Cost i.e. Cost to goal. The **Traversability Cost** takes into account the overall terrain of the environment, such as the roughness, slope and step in each grid cell as presented in [8]. This is useful in unstructured outdoor environments like forests, hills, etc. where it would be safer to park the vehicle on smooth and flat areas. The **Obstacles Cost** is useful in ensuring that the vehicle moves away from the open areas and closer to the obstacles, to avoid blocking the free spaces which may be used by other vehicles and robots for moving around in the environment. Lastly, the **Backward Dijkstra Cost** keeps track of the cost to goal from the each cell in the environment, so that when finding a safe parking spot the robot doesn't go away from the goal to park safely.

$$C_{SafeParking} = f(C_T, C_O, C_H)$$
$$\text{where} \quad C_T = \text{Traversability Cost}$$
$$C_O = \text{Obstacles Cost}$$
$$C_H = \text{Backward Dijkstra Cost}$$

These costs can be combined together by a simply adding the costs together to get a safe parking cost. But, with more experiments and testing with different environments (both indoor and outdoor), these costs can be combined together with a better function that results in a more informative safe parking cost. These costs can be computed offline for a known map to save on the online computation.

In our experiments, we worked with indoor maps where the traversability cost don't exactly provide any information as the entire floor surface is flat. Therefore, we ignored the traversability cost in our $C_{SafeParking}$ computation. Secondly, for $C_{ForwardDijkstra}$ since the forward Dijkstra computation adds significant overhead while running the robot in real-time, for our experiments we also decided to only emulate adversarial situations where the cost of reaching the parking spot doesn't matter. Therefore, for our experiments we use a simplified overall parking cost $\Psi_{simplified}$ with $\lambda_{sp} = 1$. The generated costmaps can be seen in Fig. 7 and 8.

$$\Psi_{simplified} = \lambda_{sp} \times (C_O + C_H)$$

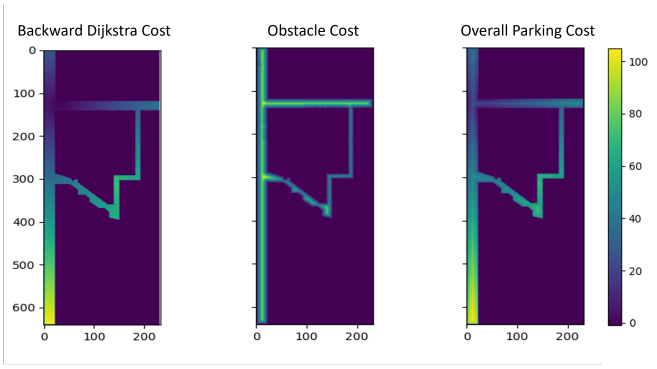

Fig. 7: (Left to Right) Backward Dijkstra Costmap, Obstacle Costmap and the Overall Parking Costmap for MIT Tunnel Map with an arbitrary goal

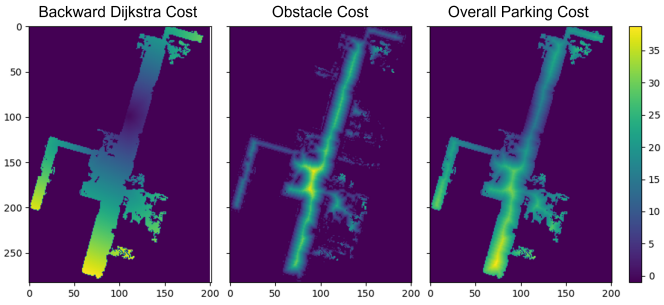

Fig. 8: (Left to Right) Backward Dijkstra Costmap, Obstacle Costmap and the Overall Parking Costmap for NSH B-Level Map with an arbitrary goal

III. EXPERIMENTAL SETUP

A. Software Implementation

The offline scripts to generate primitives, collision checking, creating maps etc. were written in Python, and the required outputs were saved as JSON configuration files. All onboard code pertaining to the motion planner was developed in C++, which read the pre-generated configuration files. This motion planner was further integrated with a ROS framework to run simulations and operate the wheeled mobile robot. The complete code base can be found in the GitHub repository here.

1) ROS Framework: Our objective was to use our planner on a ground vehicle to run it in real-time during the planning process. A ROS node allows easy transfer of data between all systems running on a ground vehicle, such as perception, planning, controls, etc. Therefore, we implemented our planner in a ROS framework, taking in the current ground vehicle's location and desired end position and orientation and sending out a feasible set of points to be followed by the ground vehicle to reach the goal. This ROS node was run at a rate of 1 Hz and only ran the planner when a new goal was published to it.

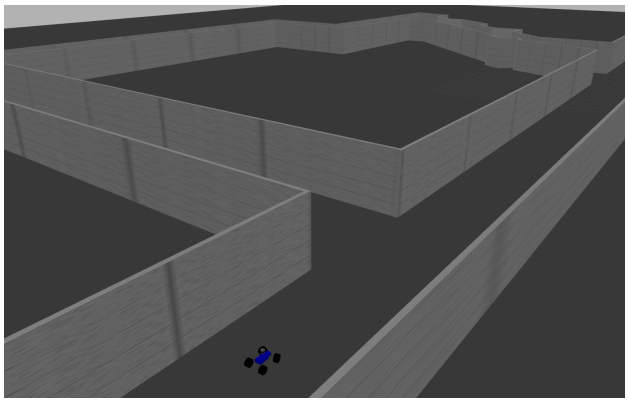

Fig. 9: Simulated environment in Gazebo

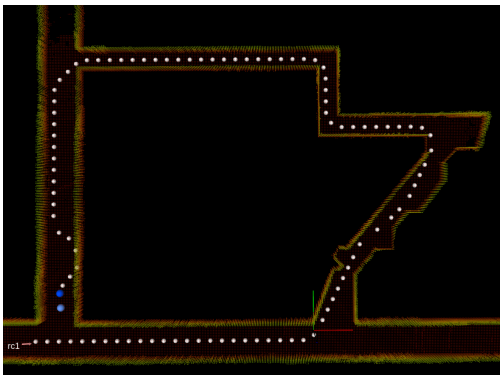

Fig. 11: Planner run on a simulated environment, white points depict the planned route

2) Controller integration: The ground vehicle has to follow the output trajectory from the planner. As we don't re-plan at every instant when the ground vehicle travels to the goal, the trajectory is generated only once by the planner. To ensure continuous tracking of the desired plan, an iLQR controller [5] was setup which clipped the current position to the closest point on the trajectory and gave the desired velocity and steering commands to the ground vehicle.

3) Simulator setup: To test the performance of our planner, we implemented our planner on maps created from real data and visualized it on matplotlib [3]. To further test out our ROS framework and controller integration, we simulated indoor environments without a defined structure on Gazebo [4]. This was selected for easy integration with ROS and provided an initial test for our planner-controller performance.

B. Hardware Implementation

Our hardware setup included a Traxxas remote controlled truck fitted with a communication node, LiDAR and IMU sensors, a Jetson AGX Xavier and a motor controller.

For our perception framework, we make use of Super Odometry [11], which is an IMU-centric SLAM pipeline that provides estimates of each agent's odometry. We get the global LIDAR points are added at each step to help make our occupancy map as defined in Section II-A.

A base station is used to provide goal positions through a way point sharing interface on RViz. All systems are run over ROS and use the DDS protocol for real-time communication.

Fig. 10: Hardware setup and test location

IV. RESULTS

1) Simulation results: We tested our planner in simulation in Gazebo for various pairs of start and goal states. The environment maps used were NSH B-level and a tunnel environment in Gazebo. Two sets of primitives were used for testing the planner, forward-only and forward-backward. Both planner tests, were able to find paths from any start and goal configuration. For the given primitives and lattice, we arrived at the optimal path. During pseudo adversarial situations, the safe parking planner is called and the robot successfully finds a planned trajectory to the nearest safe parking spot.

2) Hardware results: Our hardware testing began with mapping the environment, converting it to a cost map and then testing the hardware on the generated trajectory. Start and goal pairs were given for making a u-turn, traversing a tunnel and moving into a room. The robot tracked the generated paths sufficiently well. The robot was also able to track the re-planned path to the nearest safe parking spot well. The obstacles faced during hardware testing were primarily in domains outside planning - mapping and controls. Pre-processing of generated maps was necessary to remove

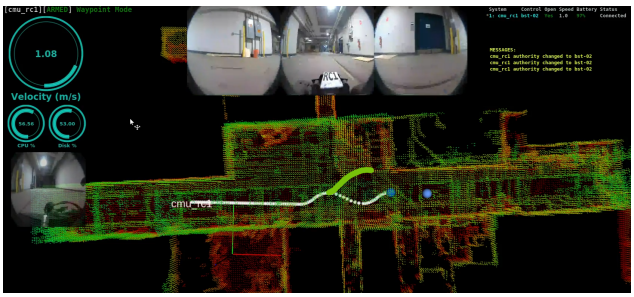

Fig. 12: Planner run on an RC truck, white points indicate the plan and green line indicated the safe parking maneuver

outliers and reduce noise to allow finding feasible paths. Additionally, controller improvements can aid the robot in tracking the generated path better and reduce drift. Our hardware implementations are shown in this YouTube video.

V. CONCLUSIONS

In this project, we have presented a planner for navigation and adversarial safe parking in any given environment. Our approach involves a weighted A* search with implicit lattice generation accounting for robot feasible motions. Further, we propose a safe parking approach which relies on a parking cost map, combining multiple factors from the environment like traversability, obstacles and the cost to goal. This is then used determine the safe parking regions in a given map that the robot can move to an adversity. The resulting approach has been validated in simulation and hardware for navigation and safe parking for a non-holonomic wheeled robot with positive results. In the future, we can further extend this work by augmenting our planner for partially known environments, and developing a more informative safe parking function for unstructured environments. Additionally, we would also like to use an anytime approach, wherein we iteratively improve the paths found in real-time.

ACKNOWLEDGMENT

We thank Dr. Maxim Likhachev and his teaching assistants, Rishi Veerapaneni and Muhammad Suhail Saleem, of the Planning and Decision-Making for Robotics (16-782) course at Carnegie Mellon University for guiding us through this project and helping us out overcome difficulties faced while implementing various planning algorithms in our project.

REFERENCES

[1] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[2] Lester E. Dubins. "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". In: *American Journal of Mathematics* 79 (1957), p. 497.

[3] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[4] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3.

[5] Weiwei Li and Emanuel Todorov. "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems." In: vol. 1. Jan. 2004, pp. 222–229.

[6] Maxim Likhachev and Dave Ferguson. "Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles". In: *The International Journal of Robotics Research* 28.8 (2009), pp. 933–945.

[7] Philip Polack et al. "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[8] David Jacob Russell et al. "UAV Mapping with Semantic and Traversability Metrics for Forest Fire Mitigation". In: *ICRA 2022 Workshop in Innovation in Forestry Robotics: Research and Industry Adoption*. 2022. URL: https://openreview.net/forum?id=Bbx8xClhG9.

[9] Sebastian Scherer et al. "Resilient and Modular Subterranean Exploration with a Team of Roving and Flying Robots". In: *Field Robotics Journal* (May 2022), pp. 678–734.

[10] Andrew Walker. *Dubins-Curves: an open implementation of shortest paths for the forward only car*. 2008–. URL: https://github.com/AndrewWalker/Dubins-Curves.

[11] Shibo Zhao et al. "Super Odometry: IMU-centric LiDAR-Visual-Inertial Estimator for Challenging Environments". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2021), pp. 8729–8736.