# Parallel Motion Planning with Neural Surrogate Models

Leo Bashaw
*Department of Electrical and Computer Engineering*
*Rice University*
Houston, TX
lb73@rice.edu

***Abstract*—Businesses in healthcare, defense, and heavy industry are becoming increasingly reliant on autonomous robots to complete a variety of tasks. To maximize their effectiveness, these robots must be able to navigate in real-time as their environments change. Therefore, a high-speed planning pipeline that exploits modern GPU architectures is desirable. Moreover, in industries that leverage human-in-the-loop robotics, solution paths must be more than smooth or efficient, they must also satisfy nontraditional metrics (e.g. safety).**

**There is a lack of open-source real-time planning libraries that generate solution paths optimized for qualities other than traditional metrics such as smoothness or energy efficiency. We propose a GPU-enabled roadmap-based planner with an embedded neural surrogate model to score graph nodes on non-traditional metrics. Solutions are generated by a search algorithm that incorporates these scores into its heuristic function. We aim to provide open-source software that generates motion plans in real-time that satisfy traditional and abstract metrics.**

***Index Terms*—Perception-aware planning, parallel planning, neural networks.**

## I. Introduction

Motion planning for robotic systems in dynamic environments presents a fundamental challenge that underpins their autonomous operation. As robots become increasingly integrated into industrial environments, the demand for sophisticated planning capabilities has grown rapidly. These industries require algorithms that can run in real time and solutions that are optimized for domain-specific metrics beyond standard efficiency measures.

Many recent publications on real-time motion planning study the problem of autonomous driving [13]. All of these systems leverage modern GPU hardware to achieve real-time performance, and the use of multithreaded computing to accelerate motion planning algorithms has been around since the early 2000s [2], [6], [8], [10]. Three traditional approaches to motion planning– roadmap, tree, and optimization-based algorithms– possess varying degrees of parallelism. The sequential nature of tree construction and iterative optimization processes prevent them from achieving the massive parallelism that is inherent in roadmap construction, although there are successful GPU-based implementations of each type of algorithm [2], [6], [11], [12].

For perception-aware planning, deep learning is used to detect and classify objects, segment environments, and steer planning algorithms toward texture-rich environments in an effort to minimize localization uncertainty [1], [5], [15]. However, we feel that there is a lack of open-source perception-aware real-time motion planners that generate solution paths for maximal visibility of some object, such as an enemy soldier in a combat environment or a delicate body part during surgery.

We address this technological gap and develop a GPU-accelerated planning framework that combines the strengths of probabilistic roadmaps (PRM) with neural surrogate models [7]. We train these models to quantify concepts such as privacy and use them to assign scores to the nodes on the roadmap. These scores are incorporated into the heuristic function during the graph search phase, enabling the generation of solution paths optimized for both traditional metrics (e.g., path length, energy efficiency) and the learned metrics described above.

This report outlines our progress toward creating an open-source planning library that can generate high-quality motion plans optimized for learned metrics at real-time speeds. We discuss our algorithmic implementation, current achievements, and plan to implement the remaining components of our system. The ultimate goal is to provide the robotics community with a versatile tool that can adapt to diverse application requirements while maintaining a performance level suitable for real-world deployment.

## II. Methods

We focus on an indoor mobile robot equipped with a steerable camera. We use Stretch from HelloRobot and plan in a 5-dimensional configuration space $\mathcal{C}$, where each robot state $\mathbf{q} \in \mathcal{C}$ is defined as:

$$\mathbf{q} = [x, y, \theta, \phi, \psi]$$

Here:

- $x, y \in \mathbb{R}$ denote the position of the robot base in the planar workspace,
- $\theta \in [0, 2\pi)$ represents the orientation of the robot base,
- $\phi \in [\phi_{\min}, \phi_{\max}]$ is the pan angle of the camera, and
- $\psi \in [\psi_{\min}, \psi_{\max}]$ is the tilt angle of the camera.

Although the robot's position and camera orientation are represented as a single state $\mathbf{q}$, our collision checking function $\mathrm{isValid}(\mathbf{q})$ depends only on the base configuration.

$$\mathrm{isValid}(\mathbf{q}) = \mathrm{isValid}(x, y, \theta)$$

Fig. 1. Stretch from HelloRobot. Note the steerable camera suspended from the robot's head. We do not use the manipulator attached to the mast and can therefore simplify our problem to planning in a 2D workspace (the robot is represented by a circle that circumscribes the base). Image taken from [17].

This approach allows the robot to visually track objects and compute valid paths simultaneously, as steering the camera does not affect the validity of a state $\mathbf{q}$. This project does not make use of Stretch's manipulator; however, our approach can accommodate higher-dimensional configuration spaces and 3D collision checking.

We use CUDA, C++, and Python to implement our planner. The graph construction phase is performed almost entirely by our custom CUDA kernels for state generation, edge construction, and collision checking. Surrogate model scoring, state projection, and graph search operations are performed in Python; therefore, we will build Python bindings to expose our CUDA programs to the Python interpreter.

We use the remainder of this section to detail our implementation of each component of the planning pipeline. We recognize that there are inefficiencies in our current approach and defer this discussion to Section IV-B.

### A. State Generation

Our state generation kernel uses one thread to generate each state, with each thread having its own random number generator (RNG) instance. We use the XORWOW RNG included in the CUDA Toolkit, which is inspired by the work of Marsaglia, due to its simplicity and speed [9]. The states are drawn from uniform distributions according to our configuration space bounds with the exception of the camera parameters $\phi$ and $\psi$, which are always set to zero. We hold $\phi$ and $\psi$ at zero in order to streamline the state projection process discussed in Section II-D. Our approach can be modified to use Halton sequences or other pseudorandom approaches to generate states.

### B. Edge Construction

Edge construction involves two steps: a $\mathbf{k}$-nearest-neighbor ($\mathbf{kNN}$) search for the set of nodes and interpolation between each node and its neighbors. We use a brute-force $\mathbf{kNN}$ search that is controlled by varying $\mathbf{k}$ rather than by fixing a search radius. Each thread computes the neighbors for a node $\mathbf{q}_i$ using the $\ell_2$ distance in the $(x, y, \theta)$ subspace. The distance between $\mathbf{q}_i$ and $\mathbf{q}_j$ is therefore:

$$d(\mathbf{q}_i, \mathbf{q}_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (\theta_i - \theta_j)^2}$$

This approach is inefficient in that we calculate some neighbor pairs twice; however, our results show that this inefficiency does not prevent us from achieving real-time performance. See Section III-A for further discussion. After completion of the $\mathbf{kNN}$ search, edges are constructed by linearly interpolating $\mathbf{R}$ points between each node and its neighbors. In all experiments, we hold $\mathbf{R}$ constant at 10.

### C. Collision Checking

We focus on 2D planning problems and therefore perform collision checking between planar primitives. We use circles and rectangles to represent the environment and a circle to represent the robot. This simplification is only possible because the problem space that we are investigating does not necessitate the use of the robot's manipulator. We use one thread to collision check one state (rather than one node or one edge) against all obstacles in the environment. Using one thread per state increases the number of threads launched by the collision checking kernel, and therefore, lowers execution time by increasing throughput. This collision checking method is inspired by the SIMT and SIMD-optimized collision checking algorithms from [6] and [14], respectively. Fig. 2 shows a roadmap in an example environment.

### D. State Projection

We run our state projection functions once the roadmap is populated with nodes and edges. Since all states are sampled with the camera configuration parameters $\phi$ and $\psi$ as zero, the camera pose in the robot frame is the same for every state - this streamlines the forward kinematics computations used to steer the camera towards the object-in-question. Given the camera-to-robot and robot-to-world transformations, and the object pose in the world frame, we can calculate the values of $\phi$ (pan angle) and $\psi$ (tilt angle) that point the camera's optical ($\mathbf{z}$) axis towards the object:

$$\mathbf{T}_{world}^{robot} : \text{Transform from robot base to world} \quad (1)$$

$$\mathbf{T}_{robot}^{camera} : \text{Transform from camera to robot base} \quad (2)$$

$$\mathbf{p}_{obj}^{world} : \text{Object position in world} \quad (3)$$

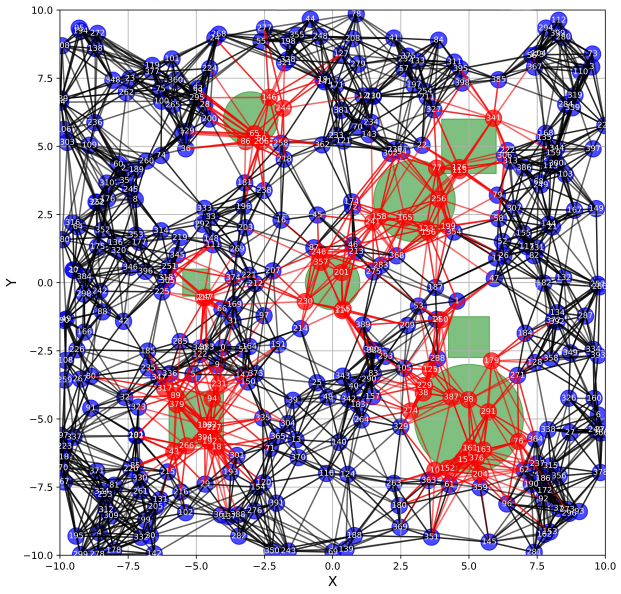


Fig. 2. This image shows an example roadmap that was built with 400 nodes and 10 nearest-neighbor connections. The invalid nodes and edges are shown in red and the green primitives are obstacles in the environment.

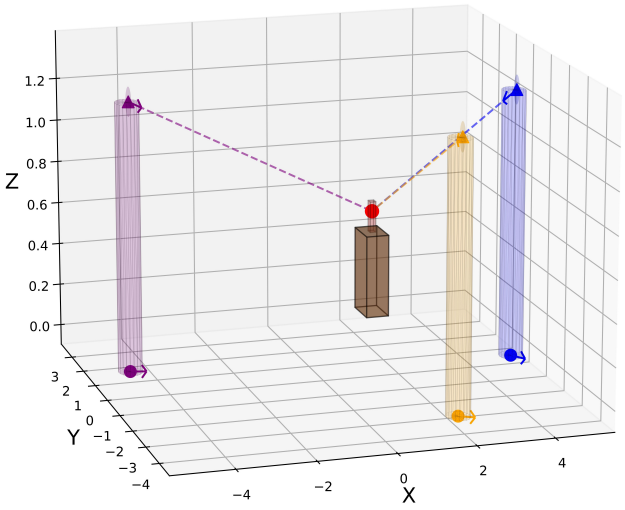


Fig. 3. This figure shows three robots in random locations in an example 3-dimensional environment. All robots have their camera's optical axis aligned to point at the target object. Each robot's base orientation is shown by the vector at its base.

Step 1: Calculate the camera's pose in world and transform the object's position into the camera frame.

$$\mathbf{T}_{world}^{camera} = \mathbf{T}_{world}^{robot} \cdot \mathbf{T}_{robot}^{camera} \tag{4}$$

$$\mathbf{T}_{camera}^{world} = (\mathbf{T}_{world}^{camera})^{-1} \tag{5}$$

$$\mathbf{p}_{obj}^{camera} = \mathbf{T}_{camera}^{world} \cdot \mathbf{p}_{obj}^{world} \tag{6}$$

Step 2: Compute object coordinates in the camera frame and calculate the pan and tilt angles.

$$[d_x, d_y, d_z]^T = \mathbf{p}_{obj}^{camera}[0:3] \tag{7}$$

$$\phi = \arctan 2(d_x, d_z) \tag{8}$$

$$\psi = \arctan 2(-d_y, \sqrt{d_x^2 + d_z^2}) \tag{9}$$

Fig. 3 shows a visualization of three projected states in a 3D environment.

### E. Node Scoring

Once we have the projected nodes, we can calculate the camera's pose in the world frame for each node. We compute the difference between the camera's and object's pose, which encodes the relative spatial configuration between the robot's camera and the object, and use it as the input to our surrogate model. Each pose is represented as a 7D vector:

$$\mathbf{p} = [x, y, z, q_x, q_y, q_z, q_w]$$

where $x, y, z$ specifies the position and $(q_x, q_y, q_z, q_w)$ represents the orientation as a unit quaternion. The surrogate model is a simple multilayer perceptron that uses fully connected layers, dropout for regularization, and the ReLU activation function.

The surrogate model can be swapped according to the use case, that is, we can use different networks that are trained to assign different types of scores. This modularity is highly beneficial because it allows the user to take advantage of our pipeline without being limited to the applications that we focus on.

### F. Graph Search

We plan to use the A-Star search algorithm to compute solution paths and will design a heuristic function that incorporates both surrogate model scores and traditional roadmap edge weights. We have not finalized the details of our implementation for graph search.

## III. Results

In this section, we present performance statistics for our CUDA and Python pipelines in separate subsections. We are still working on Python bindings for our CUDA kernels and therefore cannot perform a detailed analysis of our entire system. CUDA operations are state generation, **kNN** search, edge construction, and collision checking. Python operations are state projection and neural surrogate model inference (node scoring). The statistics presented were collected on a workstation equipped with an Intel i7-12700K CPU and NVIDIA GeForce RTX4090 GPU.

### A. Graph Construction in CUDA

We evaluated our CUDA kernels in 30 hyperparameter configurations: we varied the number of states (5 values) and neighbor connections (3 values) in both 4- and 16-obstacle environments. For each configuration, we run 100 iterations of the integrated kernels and record the average execution time. The specific configurations that we tested are listed below:

- $K = \{5,\ 10,\ 20\}$
- $N = \{1000,\ 2000,\ 5000,\ 10000,\ 20000\}$

Fig. 4 shows the construction times for our 4-obstacle environment. Varying the number of states or neighbor connections alone does not dramatically increase runtime; we even achieve microsecond-level performance for all roadmaps with 2000 nodes. Performance drops sharply as both hyperparameters increase, especially beyond 15,000 states and 12 neighbor connections; however, the system still achieves $<10\,\mathrm{ms}$ runtimes in the worst case.

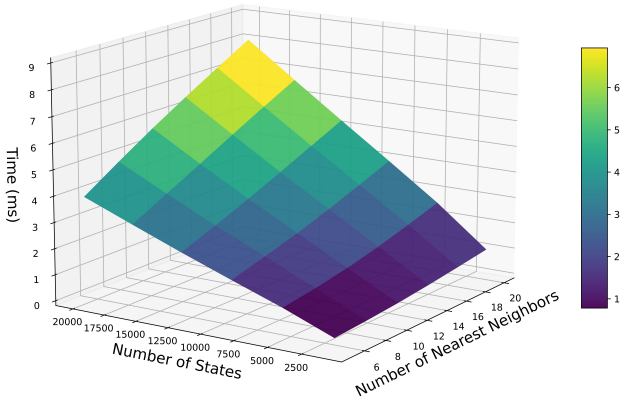


Fig. 4. This surface plot shows the time required to generate a roadmap in an environment with 4 obstacles. Times are shown for several configurations: the number of nodes and neighbor connections are varied. This plot is lightly interpolated to aid the reader's visual understanding.

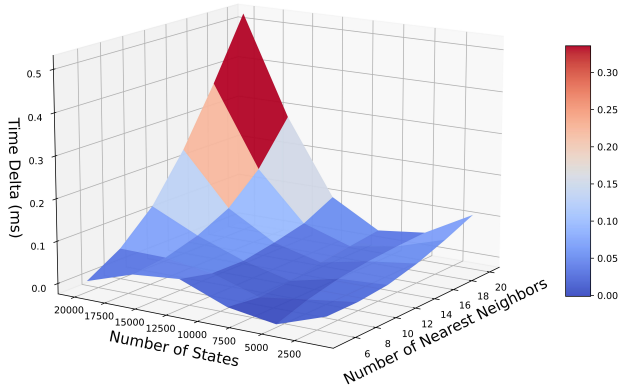


Fig. 5. This surface plot shows the difference in roadmap construction time between a 16- and 4-obstacle environment across a range of roadmap configurations. This plot is lightly interpolated to aid the reader's visual understanding.

We show a breakdown of the graph construction pipeline in Fig. 6. The cases increase in complexity along the x-axis and are defined as follows:

- Case 1: $K = 5$, $N = 2000$, $Obs = 4$
- Case 2: $K = 10$, $N = 5000$, $Obs = 16$
- Case 3: $K = 20$, $N = 20000$, $Obs = 16$

The **kNN** computation dominates the runtime in all three cases, and its dominance becomes exponentially more pronounced as the complexity of the roadmap increases. In Case 3, the neighbor search accounts for more than 90% of the total runtime and is the primary driver of exponential increases in runtime. Edge construction time (i.e., the linear interpolation between neighbors) also grows with the complexity of the roadmap, albeit at a much slower pace. The cost of collision checks grows almost imperceptibly, while the cost of state generation remains roughly constant.

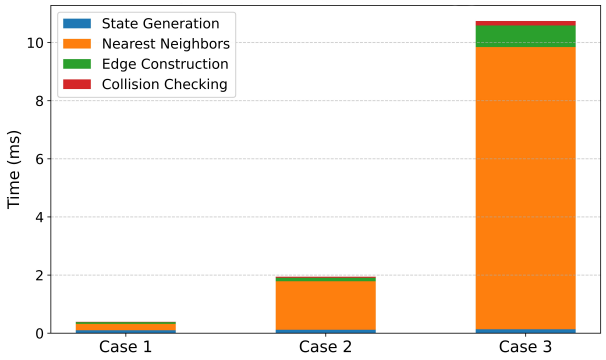


Fig. 6. This bar graph shows each phase's contribution to the overall roadmap construction time. The cases increase in complexity along the x-axis. See the text for a detailed description of each roadmap construction case. These construction times include some overhead incurred by the Nvidia NCU profiler, which limits caching and dynamic clocking.

### *B. Node Scoring and State Projection in Python*

We perform simpler benchmarking for our Python pipeline because batch size is the only hyperparameter to vary, and we use third-party libraries known for their stability and speed [16]. We benchmark node scoring and state projection as one integrated module and run 100 samples per batch size. We present the average times for each batch size in Fig. 7.

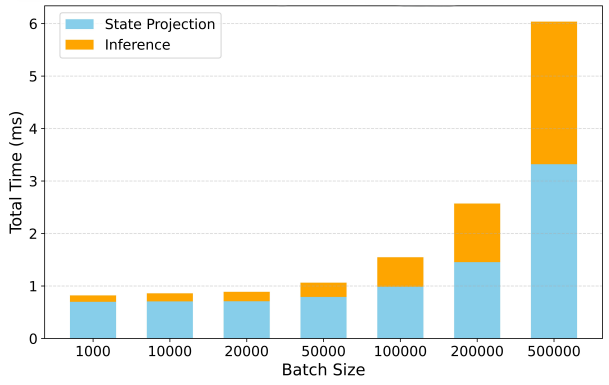


Fig. 7. This graph shows the time required to run state projection and node scoring (surrogate model inference) on a batch of states. Node scoring starts to dominate the runtime at higher batch sizes.

Fig. 7 shows that batch sizes greater than 500,000 are likely too large for real-time performance. Although the computational cost of both operations increases with batch size, inference is much faster at smaller batch sizes. As the batch size grows from 1,000 to 500,000, inference's contribution to total runtime triples from 15% to 45%. Fortunately, we perform state projection on a larger batch than we do inference: both nodes and edges are projected, while only nodes are scored. A roadmap of $N$ nodes has the following number of total states (TS):

$$\text{TS} = N + \frac{N \cdot K \cdot (R-2)}{2}$$

where $K$ is the number of neighbor connections and $R$ is the number of interpolated states used to represent each edge. For example, a roadmap with $N = 10^4$ and $K = R = 10$ would have $4.1 \times 10^5$ total states. Per Fig. 8, the time required to perform inference and projection is thus $\sim$3.5 ms, compared to $\sim$6 ms if inference and projection required the same batch size.

## IV. Discussion

Two key findings emerge from our analysis of our planning pipeline. First, we are well within our target for real-time performance. Second, while our **kNN** implementation is fast, its performance and memory usage scale poorly. Given our strong baseline performance, optimization is not immediately necessary, so we prioritize completing CUDA-Python integration and making the roadmap compatible with existing implementations of A-Star search. The remainder of this discussion is divided into two parts: (1) an analysis of our benchmarking results, and (2) a discussion of scaling issues and how to address them.

### A. Performance Analysis

Individually, both the CUDA and Python code perform exceptionally well. Even with several milliseconds of overhead (to account for slowdowns from Python bindings and the absence of graph search), we are on track to generate valid trajectories in $<$50 ms for even the most complex configurations tested. Our end-of-semester goal was to generate solutions in $<$100 ms. Although we have not achieved full integration (i.e., we have not tested our pipeline end-to-end), our current results demonstrate significant progress toward this goal.

All of our code exhibits excellent scalability except for the **kNN** algorithm; however, we do not anticipate requiring roadmaps with 10,000+ nodes and 15+ neighbor connections for our indoor 2D planning scenarios. Furthermore, while increasing the number of neighbor connections can improve the connectivity of the roadmap, it is well known that continually increasing the value of k yields diminishing returns. Performance degradation becomes noticeable primarily at high node counts and large k values; similarly, adding obstacles only meaningfully impacts performance in scenarios where both hyperparameters are set to large values.

### B. Inefficiencies and Moving Forward

The main bottleneck in our system is the **kNN** search. For simplicity during development, we currently use a brute-force exact search, which is feasible due to access to high-performance hardware and our low-dimensional configuration space – recall that the **kNN** search occurs in $(x, y, \theta) \in \mathcal{C} \subset \mathbb{R}^2 \times S^1$. Additionally, our implementation calculates duplicate neighbor relations (e.g., 2→3 and 3→2), a trade-off made to keep the system simple during development. These compromises are acceptable during our prototyping stage given the overall speed of the pipeline.

We identify two simple ways to improve the **kNN** module. First, we could eliminate the duplicate neighbor computations in our current exact search, though this would require higher thread complexity and could introduce bugs. Second, we could adopt an approximate nearest-neighbor method and sacrifice some accuracy for significantly improved performance. We will explore both approaches and start performance optimization once we finish prototyping our core pipeline.

This report summarizes our development of a GPU-accelerated motion planner that incorporates neural surrogate models to optimize motion plans based on learned metrics. Our current implementation achieves millisecond-level performance, positioning us well to meet our target of real-time operation. Looking ahead, we will focus on optimizing the **kNN** search step and integrating our CUDA and Python modules to ensure full compatibility with existing A-Star implementations. Ultimately, this research will culminate in a fully open-source, real-time planning library designed for human-centered robotic applications.

## V. Acknowledgment

I would like to recognize Qingxi Meng, a member of the Kavraki Lab at Rice University, whose research acts as the foundation for my work on parallelized motion planning. I also thank my advisors, Lydia Kavraki and Jose Moreto, for their continued support and feedback. I extend my gratitude to the creators of Pytorch, Torch2TRT, and Pytorch FK, as well as to the engineers at NVIDIA for their work on CUDA and GPU hardware.

## References

[1] L. Bartolomei, L. Teixeira, and M. Chli, "Perception-aware Path Planning for UAVs using Semantic Segmentation," in 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Oct. 2020, pp. 5808–5815. doi: 10.1109/IROS45743.2020.9341347.

[2] J. Blankenburg, R. Kelley, D. Feil-Seifer, R. Wu, L. Barford, and F. C. Harris, "Towards GPU-Accelerated PRM for Autonomous Navigation," in 17th International Conference on Information Technology–New Generations (ITNG 2020), S. Latifi, Ed., Cham: Springer International Publishing, 2020, pp. 563–569. doi: 10.1007/978-3-030-43020-774.

[3] M. Campana, F. Lamiraux, and J.-P. Laumond, "A gradient-based path optimization method for motion planning," Advanced Robotics, vol. 30, no. 17–18, pp. 1126–1144, Sep. 2016, doi: 10.1080/01691864.2016.1168317.

[4] J. Cortés and T. Siméon, "Sampling-Based Tree Planners (RRT, EST, and Variations)," in Encyclopedia of Robotics, M. H. Ang, O. Khatib, and B. Siciliano, Eds., Berlin, Heidelberg: Springer, 2020, pp. 1–9. doi: 10.1007/978-3-642-41610-1170 − 1.

[5] G. Costante, C. Forster, J. Delmerico, P. Valigi, and D. Scaramuzza, "Perception-aware Path Planning," Feb. 10, 2017, arXiv: arXiv:1605.04151. doi: 10.48550/arXiv.1605.04151.
[6] C. H. Huang, P. Jadhav, B. Plancher, and Z. Kingston, "pRRTC: GPU-Parallel RRT-Connect for Fast, Consistent, and Low-Cost Motion Planning," Mar. 09, 2025, arXiv: arXiv:2503.06757. doi: 10.48550/arXiv.2503.06757.
[7] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," IEEE Transactions on Robotics and Automation, vol. 12, no. 4, pp. 566–580, Aug. 1996, doi: 10.1109/70.508439.
[8] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time Motion Planning and Global Navigation using GPUs," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 24, no. 1, Art. no. 1, Jul. 2010, doi: 10.1609/aaai.v24i1.7732.
[9] G. Marsaglia, "Xorshift RNGs," Journal of Statistical Software, vol. 8, pp. 1–6, Jul. 2003, doi: 10.18637/jss.v008.i14.
[10] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," IEEE Transactions on Robotics, vol. 21, no. 4, pp. 597–608, Aug. 2005, doi: 10.1109/TRO.2005.847599.
[11] D. Shah, N. Yang, and T. M. Aamodt, "Energy-Efficient Realtime Motion Planning," in Proceedings of the 50th Annual International Symposium on Computer Architecture, in ISCA '23. New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 1–17. doi: 10.1145/3579371.3589092.
[12] B. Sundaralingam et al., "cuRobo: Parallelized Collision-Free Minimum-Jerk Robot Motion Generation," Nov. 03, 2023, arXiv: arXiv:2310.17274. doi: 10.48550/arXiv.2310.17274.
[13] S. Teng et al., "Motion Planning for Autonomous Driving: The State of the Art and Future Perspectives," IEEE Transactions on Intelligent Vehicles, vol. 8, no. 6, pp. 3692–3711, Jun. 2023, doi: 10.1109/TIV.2023.3274536.
[14] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in Microseconds via Vectorized Sampling-Based Planning," in 2024 IEEE International Conference on Robotics and Automation (ICRA), May 2024, pp. 8749–8756. doi: 10.1109/ICRA57147.2024.10611190.
[15] Y. Zhao, Z. Xiong, S. Zhou, J. Wang, L. Zhang, and P. Campoy, "Perception-Aware Planning for Active SLAM in Dynamic Environments," Remote Sensing, vol. 14, no. 11, Art. no. 11, Jan. 2022, doi: 10.3390/rs14112584.
[16] Zhong, Sheng and Power, Thomas and Gupta, Ashwin and Mitrano, Peter, "Pytorch Kinematics", Version 0.7.1, Feb. 2024, doi: 10.5281/zenodo.7700587
[17] Hello Robot. (2025). Accessed: April 28, 2025. [Online Photo]. Available: https://hello-robot.com/