Algorithmic Robotics
COMP/ELEC/MECH 450/550
Project G: Topics in Planning

In this assignment, you are free to select a project from the ones below. This project must be completed in groups of two or three and only by graduate students. **For PhD students only, you are allowed to propose your own project.** But you must receive approval from the instructor prior to October 31st by providing a one-page description of your project.

In contrast to previous projects, for this project, you are on your own. We want to see how far you can go with the project, and we will grade accordingly. If you do not complete the project, this is fine.

An initial progress report is due at **1pm October 28** on Canvas. This report should be short, no longer than one page in PDF format. At a minimum, the report should state who your partner is and what progress you have made thus far. Only one report needs to be submitted for each group.

Final submissions are due **November 18th at 1pm**. Submissions will be on Canvas and consist of three things. If your code or your report is missing by the deadline above, your project is late. Then by **November 25th at 1pm** you need to submit your presentation. The **latest date for submission is 8am on 12/1 for a grade.** No late policies apply after that and your project will not be graded.

First, to submit your project, clean your build space, zip up the project directory into a file named $Project5 < yourNetID >_< partner'sNetID >_< partner'sNetID > .zip$. Unless specified in the project, you are allowed to use any language (C++, Python, etc.), but your code must compile and run within a modern Linux environment. If your code is compiled on the environment we provided, then it will be fine. Also include a README with details on compiling and/or executing your code. Some projects require completion in OMPL.

Second, submit a written report that summarizes your experiences and findings from this project. The report should be no longer than 10 pages, in PDF format, and contain (at least) the following information:

- A problem statement and a short motivation for why solving this problem is useful.
- The details of your approach and an explanation of how/why this approach solves your problem.
- A description of the experiments you conducted. Be precise about any assumptions you make about the robot or its environment.
- A quantitative and qualitative analysis of your approach using the experiments you conduct.
- Rate the difficulty of each exercise on a scale of 1–10 (1 being trivial, 10 being impossible). Give an estimate of how many hours you spent on each exercise, and detail what was the hardest part of the assignment.

Submit the PDF write-up, separate from the source code archive.

Third, submit a slide presentation for a short ($\sim$15 minute) in-class presentation on your chosen topic. The slides are due at a later date than the first and second items above.

Note: Each team should provide one submission. When making the final submission, each student in the team must send a private email to kavraki@rice.edu stating the team composition and providing a detailed description of their own contribution to the project.

Note: Although it may not be stated explicitly, *visualization is essential* for projects, reports, and presentations to establish the correctness of the implementation.

Start this project early!

Project 1: Planning Under Uncertainty

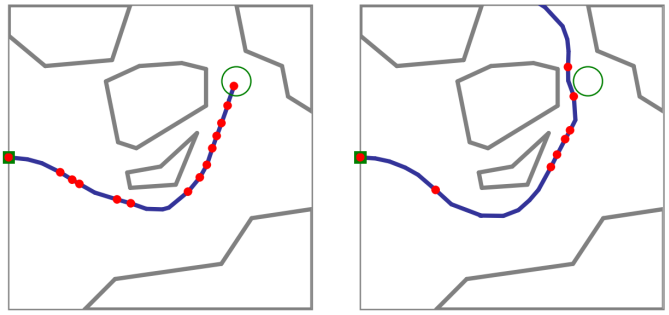

Figure 1: Even under an optimal policy, a system with actuation uncertainty can fail to reach the goal (green circle) consistently. *From Alterovitz et al.; see below.*

For physical robots, actuation is often imperfect: motors cannot instantaneously achieve a desired torque, the execution time to apply a set of inputs cannot be hit exactly, or wheels may unpredictably slip on loose terrain just to name a few examples. We can model the actuation error for many situations using a conditional probability distribution, where the probability of reaching a state q' depends on the initial state q and the input u, written succinctly as $P(q'|q,u)$. Because we cannot anticipate the exact state of the robot *a priori*, algorithms that reason over uncertainty compute a *policy* instead of a path or trajectory since the robot is unlikely to follow a single path exactly. A policy is a mapping of every state of the system to an action. Formally, a policy π is a function $\pi : Q \to U$, where Q is the configuration space and U is the control space. When we know the conditional probability distribution, an optimal policy to navigate the robot can be computed efficiently by solving a Markov decision process.

Markov decision process (MDP):
When the transitional probability distribution depends only on the current state, the robot is said to have the Markov property. For Markov systems, an optimal policy is obtained by solving a Markov decision process (MDP). An MDP is a tuple (Q,U,P,R), where:

- Q is a set of states
- U is a set of controls applied at each state
- $P : Q \times U \times Q \to [0,1]$ is the probability of reaching $q' \in Q$ via initial state $q \in Q$ and action $u \in U$.
- $R : Q \to \mathbb{R}$ is the reward for reaching a state $q \in Q$.

An optimal policy over an MDP maximizes the total *expected reward* the system receives when executing a policy. To compute an optimal policy, we utilize a classical method from dynamic programming to estimate the expected total reward for each state: Bellman's equation. Let $V_0(q)$ be an arbitrary initial estimate for the maximum expected value for q, say 0. We iteratively update our estimate for each state until we converge to a fixed point in our estimate for all states. This fixed-point algorithm is known as *value iteration* and can be succinctly written as

$$V_{t+1}(q) = R(q) + \max_{u \in U} \sum_{q'} P(q'|q,u) V_t(q'), \tag{1}$$

We stop value iteration when $V_{t+1}(q) = V_t(q)$ for all $q \in Q$. Upon convergence, the optimal policy for each state q is the action u that maximizes the sum in the equation above.

Sampling-based MDP:
Efficient methods to solve an MDP assume that the state and control spaces are discrete. Unfortunately, a robot's configuration and control spaces are usually continuous, preventing us from computing the optimal policy exactly. Nevertheless, we can leverage sampling-based methods to build an MDP that approximates the configuration and control space. The *stochastic motion roadmap* (SMR) is one method to approximate a continuous MDP. Building an SMR is similar to building a PRM; there is a learning phase and a query phase. During the learning phase, states are sampled uniformly at random and the probabilistic transition between

states are discovered. During the query phase, an optimal policy over the roadmap is constructed to bring the system to a goal state with maximum probability.

Learning phase: During the learning phase, an MDP approximation of the evolution of a robot with uncertain actuation is constructed. First, n valid states are sampled uniformly at random. Second, the transition probabilities between states are discovered. SMR uses a small set of predefined controls for the robot. The transition probabilities can be estimated by simulating the system m times for each state-action pair. Note that m has to be large enough to be an accurate approximation of the transition probabilities. For simplicity, we assume that the uncertainty is represented with a Gaussian distribution. Once a resulting state is generated from the state-action pair, you can match it with the n sampled states through finding its nearest neighbor, or if a sampled state can be found within a radius. It is also useful to add an *obstacle* state to the SMR to indicate transitions with a non-zero probability of colliding with an obstacle.

Query phase: The SMR that is constructed during the learning phase is used to extract an optimal policy for the robot. For a given goal state (or region), an optimal policy is computed over the SMR that maximizes the probability of success. To maximize the probability of success, simply use the value iteration algorithm from equation 1 with the following reward function:

$$R(q) = \begin{cases} 0 & \text{if } q \text{ is not a goal state,} \\ 1 & \text{if } q \text{ is a goal state.} \end{cases}$$

Note: it is assumed that the robot stops when it reaches a goal or obstacle state. In other words, there are no controls for the goal and obstacle states and $V_t(q) = R(q)$ is constant for these states.

Deliverables:

Part 1: Implement the SMR algorithm for motion planning under action uncertainty with a 2D steerable needle. Modeling a steerable needle is similar to a Dubins car; see section III of the SMR paper (cited below) for details on the exact configuration space and control set. Construct some interesting 2D environments and visualize the execution of the robot under your optimal policy. Simulated execution of the steerable needle is identical to the simulation performed when constructing the transition probabilities. Simulate your policy many times. Does the system always follow the same path? Does it always reach the goal? How sensitive is probability of success to the standard deviation of the Gaussian noise distribution?

Part 2: The value $V(q)$ computed by value iteration for each state q is the estimated probability of reaching the goal state under the optimal policy of the SMR starting at q. We can check how accurate this probability is. Keeping the environment, start, and goal fixed, generate 20 (or more) SMR structures with n sampled states and compute the probability of reaching the goal from the start state using value iteration. Then simulate the system many times (1000 or more) under the same optimal policies. How does the difference between the expected and actual probability of success change as n increases? Evaluate starting with a small n (around 1000), increasing until some sufficiently large n; a logarithmic scale for n may be necessary to show a statistical trend.

Reference:

R. Alterovitz, T. Siméon, and K. Goldberg, The Stochastic Motion Roadmap: A Sampling Framework for Planning with Markov Motion Uncertainty. In *Robotics: Science and Systems*, pp. 233–241, 2007. http://goldberg.berkeley.edu/pubs/rss-Alterovitz2007_RSS.pdf.

Project 2: Elastic Kinematic Chains

NOTE: This project was designed for mechanical engineering students.

Let an elastic kinematic chain be defined as a planar kinematic chain with n revolute joints, where each joint has an angular spring. The spring energy of the chain is $\sum_i u_i^2$, where u_i corresponds to joint angle i. Suppose the base link is fixed at $(0,0)$ with orientation 0. Now imagine that the end effector is slowly moved around to different positions and orientations so that the arm is always at a stable equilibrium configuration. How many parameters would you need to describe *all* stable configurations for n joints? The surprising answer is 3 for any n! There is a relatively straightforward algorithm to determine which configurations are stable (although understanding its correctness is far from trivial).

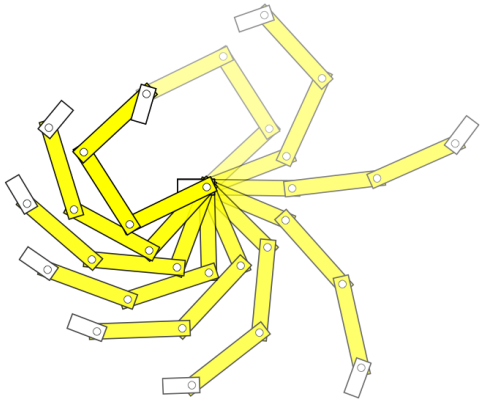

Figure 2: A sequence of stable configurations for a chain with 4 joints. *From McCarthy & Bretl; see below.*

There is one small typo in the algorithm in the reference below. The line that reads

$$P_{n-4} = (A^\dagger B)^T (I - NK)^T M (I - NK)(A^\dagger B)$$

should be changed to

$$P_{n-4} = Q_{n-4} + (A^\dagger B)^T (I - NK)^T M (I - NK)(A^\dagger B).$$

As you can tell from this one line, you will want to use a linear algebra package such as Eigen (a C++ header-only library) or Numpy (a Python library with many Matlab-like functions) for your implementation.

Reference:

Z. McCarthy and T. Bretl, Mechanics and Manipulation of Planar Elastic Kinematic Chains, in *IEEE Intl. Conf. on Robotics and Automation,* 2012. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6224693

Deliverables: Implement the `isStable` function from the reference above and use it as a StateValidityChecker to plan paths between stable configurations of an elastic kinematic chain. Add an option to check whether the joint angles (denoted by u_i in the paper) are within $-\pi$ and π. Then use any OMPL planner to find a path of stable configurations between any pair of stable configurations. Next, write an algorithm that using this function finds a pair of configurations that requires a long path (in terms of number of states on the path). It is assumed that you will use path simplification on each path produced by the planner so that a long path is not simply the result of an unlucky run of the planner. How do the $[-\pi, \pi]$ joint limits change the results?

Picking reasonable bounds for the 3-parameter space of stable configurations is not entirely obvious. For a 10-joint arm you could use the following bounds on the configuration space: $[-15, 15] \times [-15, 15] \times [-\pi, \pi]$. They may have to be adjusted for a different number of joints. Implementing the bonus below can help you visualize when the free space becomes almost fractal-like (in which case, you would need to specify a very small value for setStateValidityCheckingResolution). Consider the symmetries that should exist in your c-space obstacles. If your visualization does not match the expected symmetries, there may be an error in your computations.

Write code that visualizes the free space of the three-parameter configuration space. This can be done by creating plots of 2D slices of the configuration space, as was done in the reference above. You don't actually have to compute the boundaries of the configuration space obstacles. It is sufficient to create a dense 3D grid of the configuration space, where for each grid point you compute the state validity. You can then plot each slice of the grid by drawing invalid configurations as black pixels and free configurations as white pixels.

If you do not want to use OMPL, you can implement your own planner. Keep it as simple as possible.

Project 3: Protein Folding

NOTE: This assignment is for students with a background or interest in computational structural biology.

Understanding how proteins fold into their native structure is one of the most fundamental problems in biology. In this assignment you will explore how motion planning algorithms can be used to find possible folding pathways. You can use a Python package called the Molecular Modeling Toolkit (MMTK). It is recommended that you only choose this assignment if you already know Python and have a basic understanding of proteins.

The protein folding can be formulated as follows: given a small unfolded protein (such as a beta hairpin[1]), find a series of bond rotations that will bring the protein from an arbitrary initial state to a folded state. Conformations with energy above some absolute energy threshold should be considered as invalid states.

In this project, you will use a *probabilistic* acceptance criterion for transitions between two conformations c_A and c_B. For this, we interpolate the conformations at some fixed resolution to obtain conformations $c_A = c_0, c_1, \ldots, c_{n-1}, c_n = c_B$. The transition between c_i and c_{i+1} is accepted using the following probability distribution function:

$$P(c_i \to c_{i+1}) = \begin{cases} e^{\frac{E(c_i)-E(c_{i+1})}{k_B T}} & \text{if } E(c_i) < E(c_{i+1}), \\ 1 & \text{if } E(c_i) \geq E(c_{i+1}), \end{cases}$$

where $E(c_i)$ is the energy of conformation i, k_B is the Boltzmann constant ($1.3806488 \times 10^{-23}$) and T is the temperature (typically $300K$). In other words, transitions from high energy to low energy are always accepted, while transitions in the opposite direction become unlikely at an exponential rate as the energy difference increases.

Rather than planning one path from one random start state to the goal state, it is more informative to get a more global view of which parts of the conformational space are reachable. To do this, we can make the folded state the start state and create a Goal-derived class whose isSatisfied() method always returns false. This forces the planning algorithm to keep exploring the space until time runs out.

Deliverables:
You need to define a new state space for proteins. The degrees of freedom are formed by the bond angles, which you can model as a CompoundStateSpace with n SO2StateSpaces where n is the number of bond angles. You will need to implement a new MotionValidator-derived class that implements the probabilistic acceptance criterion described above and make sure the planner is using it. You can use any standard molecular energy function. Save the states generated by a planning algorithm (using PlannerDataStorage) and write a separate program or script to analyze the results. Describe which planning algorithms you tried and why. You should visualize the energy landscape by projecting the conformations onto 2 dimensions and plotting each conformation as a point colored by energy. You can also try to visualize the pathway between two distinct low-energy conformations (for this you could use a molecular viewing program such as VMD, Chimera, PyMol).

Tip:
Install MMTK in your machine and use it.

[1] The beta hairpin occurs in many proteins. You can use the substructure formed by residues 23–42 from PDB entry 1NXB as your reference folded structure, i.e., the goal state.

Project 4: Chance Constrained RRT (CC-RRT)

NOTE: This is a challenging assignment.

Understand and implement the Chance Constrained RRT algorithm[2] [1] for motion planning under action uncertainty using OMPL. A useful reference to understand the algorithm is [2]. In your implementation, do not take into account the uncertainty due to dynamic obstacles described in Sec IV. Implement the offline version of the algorithm and disregard the branch-and-bound heuristic shown in line 19 of Algorithm 1. Design 2 different environments (basic and cluttered) with polyhedral obstacles to test your implementation and use a small square car robot. Test the performance of your implementation by using different values of the probability of safety and assess the quality of your results.

The state of the system is represented by its position (x,y), orientation θ and linear forward velocity v, i.e., $x = (x,y,\theta,v)$. The control inputs to this system consist of the angular velocity ω and the forward linear acceleration a, i.e., $u = (\omega,a)$. The system dynamics are:

$$\begin{aligned} x_{t+1} &= f(x_t,u_t,w_t) \\ &= \begin{pmatrix} x_t + \tau v_t \cos\theta_t \\ y_t + \tau v_t \sin\theta_t \\ \theta_t + \tau\omega \\ v_t + \tau a \end{pmatrix} + w_t \end{aligned}$$

where τ is the duration of the time step.

Since CC-RRT assumes a linear model, a linearization must be done at every state:

$$\begin{aligned} x_{t+1} &\approx \tilde{A}x_t + \tilde{B}u_t \\ \tilde{A} &= \frac{\partial f}{\partial x}(x_t,u_t) \\ \tilde{B} &= \frac{\partial f}{\partial u}(x_t,u_t) \end{aligned}$$

The state propagation can be done as follows:

$$\begin{aligned} \hat{x}_{t+1} &= f(\hat{x}_t,u_t,0) \\ P_{x_{t+1}} &= \tilde{A}P_{x_t}\tilde{A}^\top + P_w \end{aligned}$$

For the disturbance covariance and initial state covariance use:

$$P_{x_0} = \begin{pmatrix} 0.01 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0.001 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, P_w = \begin{pmatrix} 0.02 & 0.01 & 0 & 0 \\ 0.01 & 0.02 & 0 & 0 \\ 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0.01 \end{pmatrix}$$

[2]Paper: https://drive.google.com/file/d/1QZQZu5cCZTDMt-fOYxIHJR2vVU3U-sVT/view?usp=sharing

but feel free to experiment with different values of P_w after you have designed your environments.

- Briefly describe why the CC-RRT algorithm (what problems is it good for) and what were the the biggest challenges of implementing the CC-RRT algorithm and specify what assumptions you made.
- Show the linearized state and input matrices $\tilde{A}, \tilde{B}$
- Create figures showing your designed planning problems (start, goal, environment) together with the trees created by the CC-RRT algorithm for different values of p_{safe} and draw conclusions from them.
- Compare qualitatively the performance of CC-RRT with that of RRT using your figures to support your conclusions.

References

[1] B. Luders, M. Khotari and J. How, "Chance Constrained RRT for Probabilistic Robustness to Environmental Uncertainty", Proceedings of the AIAA Guidance, Navigation, and Control Conference, Toronto, Ontario, Canada, 2-5 August 2010. https://dspace.mit.edu/handle/1721.1/67648

[2] L. Blackmore, Hui Li and B. Williams, "A probabilistic approach to optimal robust path planning with obstacles," 2006 American Control Conference, 2006, pp. 7 pp.-, doi: 10.1109/ACC.2006.1656653. https://groups.csail.mit.edu/mers/papers/BlackmoreLiWilliams_ACC06.pdf

Project 5: Trajectory Optimization with Multiple Initial Motion Plans

NOTE: This is a research-oriented project, suitable for students from engineering and computer science. This project may involve investigating and prototyping various solutions to trajectory optimization. This project is proposed by Thai Duong.

The state of a manipulator can be described by a vector $\mathbf{q}$ of joint angles (assuming there are only revolute joints). Usually, the joint angles cannot be controlled directly. Instead, the motors apply torques to the joints to change their acceleration. While it is possible to plan a geometric path, i.e., a sequence of waypoints $\{\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_n\}$, it is often desirable to plan a trajectory with smooth motions, e.g., continuous velocity $\dot{\mathbf{q}}$ and acceleration $\ddot{\mathbf{q}}$. In this assignment, you will develop trajectory optimization [1] for an n-joint manipulators that find an optimal trajectory, given collision avoidance constraints, and velocity and acceleration limits. However, trajectory optimization is often slow, can get stuck in a local minima, and is sensitive to initialization. You will address this issue by generating multiple initial solutions and picking the best final trajectory.

Deliverables:

Part 1: Your assignment is to adapting the TrajOpt code: https://github.com/tesseract-robotics/trajopt to work with a 7-DOF Panda robot with 7 controllable joint angles.

TrajOpt examples with a KUKA robot can be found in: https://github.com/tesseract-robotics/tesseract_python in Python, and https://github.com/tesseract-robotics/tesseract_ros2/tree/master/tesseract_ros_examples in ROS. TrajOpt requires an SRDF and a URDF file, which describe the robot and the scene (including obstacles). The SRDF and URDF files for Panda can be found here: https://github.com/KavrakiLab/vamp, *without* the obstacle descriptions. Test your TrajOpt with a Panda in an empty environment and make sure the motion looks reasonable.

Next, you will add spheres as obstacles to the environment, either programmatically (which is preferred) or by modifying the URDF file. Add the following spheres with radius 0.2 and centers:

$$\begin{bmatrix} 0.55 & 0 & 0.25 \\ 0.35 & 0.35 & 0.25 \\ 0 & 0.55 & 0.25 \\ -0.55 & 0 & 0.25 \\ -0.35 & -0.35 & 0.25 \\ 0 & -0.55 & 0.25 \\ 0.35 & -0.35 & 0.25 \\ 0.35 & 0.35 & 0.8 \\ 0 & 0.55 & 0.8 \\ -0.35 & 0.35 & 0.8 \\ -0.55 & 0 & 0.8 \\ -0.35 & -0.35 & 0.8 \\ 0 & -0.55 & 0.8 \\ 0.35 & -0.35 & 0.8 \end{bmatrix} \tag{2}$$

Initialize TrajOpt with a random solution, and verify that the planned trajectory from TrajOpt is reasonable and collision-free. Report the trajectory length and the planning time with and without obstacles in the environment.

Part 2: Run the RRT-Connect planner from VAMP [2] (https://github.com/KavrakiLab/vamp) with different random seeds on a Panda manipulator, and interpolate the resulting geometric paths to have denser initial solutions. Run TrajOpt with these initial motion plans, and choose the best final trajectory, e.g., in terms of length and duration, to visualize. Compare the trajectory length and the planning time with those from **Part 1** (with random initializations).

Bonus: Write code to run TrajOpt with the initial solutions in different threads in parallel, to further improve the planning time.

References:

[1] . Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," The International Journal of Robotics Research, vol. 33, no. 9, pp. 1251–1270, 2014. https://rll.berkeley.edu/~sachin/papers/Schulman-IJRR2014.pdf
[2] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in IEEE International Conference on Robotics and Automation (ICRA), 2024, pp. 8749–8756. https://ieeexplore.ieee.org/document/10611190

Project 6: Making Sampling-based Planners Ultra Fast

NOTE: This is a project for students who are proficient in C++ or Rust. It is an amazing experience if you have a good handle of the programming language. The project is proposed by Clayton Ramsey.

The purpose of this project is to provide students with experience writing planning code at the cutting edge of research. Students will reimplement a key part of a recent publication, then benchmark and present their findings.

Specifically, you will implement the core parts of the paper "Motions in Microseconds via Vectorized Sampling-Based Planning" [1] by Thomason et al. This paper presents VAMP, a motion planning framework that uses single-instruction, multiple-data (SIMD) parallelism to speed up sampling-based motion planning performance. Nearly all modern CPUs have access to SIMD instructions, which allow user code to use a single instruction to compute the output of a computation across a large block of data (typically, 8 or 16 floating-point numbers). The key benefit of SIMD is that it comes at essentially no overhead (unlike GPU parallelism, which suffers from high communication overhead). For the project, you will write a high-performance state validator, which is an essential component of any sampling-based planner. This project is recommended for students with an interest in systems programming and high-performance software engineering. Since it requires access to low-level primitives, you must use a low-level language (such as C, C++, or Rust) for your implementation.

Deliverables:

1. Read the original vamp paper closely, especially the Method section (Sec. III). Make sure you fully understand the approach.
2. Write a simple collision checker using a struct-of-arrays model for environment representation. To keep things simple, you need only support axis-aligned bounding boxes and spheres in the scene. You may want to refer to existing literature on collision checking [2] for assistance.
3. Write a state validator that, given a robot configuration q and environment representation E, determines whether a robot at configuration q collides with E. Make your state validator support both sequential queries (i.e., not using SIMD parallelism) and SIMD-parallel batched queries. Your state validator must also support at least two robots of your design.
4. Use your state validator to build a motion planning system (using e.g., RRT-connect or PRM). You can use the planner implementations in OMPL or a custom planner implementation.
5. Benchmark your state validator. What is the collision-checking throughput of your implementation? How does parallelism affect performance? What is the experimental performance of the planners? Perform experiments and produce graphs to support your arguments. Discuss.

References:

1. W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in IEEE International Conference on Robotics and Automation (ICRA), 2024, pp. 8749–8756. https://ieeexplore.ieee.org/document/10611190
2. C. Ericson, Real-Time Collision Detection. USA: CRC Press, Inc., 2004