Algorithmic Robotics
COMP/ELEC/MECH 450/550
Project 3: Barking up a Random Tree

The documentation for OMPL can be found at http://ompl.kavrakilab.org.

A broad class of motion planning algorithms consider only geometric constraints (e.g., bug algorithms, visibility graph methods, PRM). These algorithms compute paths that check only whether the robot is in collision with itself or its environment. In the motion planning literature, this problem is also known as *rigid body* motion planning or the *piano mover's* problem. Physical constraints on the robot (e.g., velocity and acceleration limits, steering speed) are ignored in this formulation. While considering only geometric constraints may seem simplistic, many manipulators can safely ignore dynamical effects during planning due to their relatively low momentum or high torque motors. Formally, this is known as a *quasistatic* assumption.

The goal of this project is to implement a simple sampling-based motion planner in OMPL to plan for a rigid body and then systematically compare your planner to existing methods in the library.

Random Tree Planner

The algorithm that you will be implementing is the *Random Tree Planner*, or RTP. The RTP algorithm is a simple sampling-based method that grows a tree in the configuration space of the robot. RTP is loosely based on a random walk of the configuration space and operates as follows:

1. **Select** a random configuration q_a from the existing Random Tree.
2. **Sample** a random configuration q_b from the configuration space. With a small probability, select the goal configuration as q_b instead of a random one.
3. **Check** whether the straight-line path between q_a and q_b in the $\mathcal{C}$-space is valid (i.e., collision free). If the path is valid, add the path from q_a to q_b to the tree.
4. Repeat steps 1 – 3 until the goal state is added to the tree. Extract the final motion plan from the tree.

The tree is initialized with the starting configuration of the robot. You might notice that this procedure seems very similar to other sampling-based algorithms that have been presented in class and in the reading. Many sampling-based algorithms employ a similar core loop that utilizes the basic primitives of selection, sampling, and local planning (checking). RTP is one of the simplest possible approaches, with no additional intelligence in how configurations are sampled, selected, or checked. Improvements to this algorithm are **out of scope** for this project, simply implement RTP as described above.

Collision Checking

Sampling-based methods make use of a collision checking routine (also known as validity checker) that checks whether a given state is collision-free or not. For this project, we have provided two of these collision checking routines for environments made up of axis-aligned rectangular obstacles and either a point robot or a square robot. Function `isValidStatePoint(const ompl::base::State* state, const std::vector<Rectangle>& obstacles)` will return true if the given `state` is valid given the vector of rectangles `obstacles` for a point robot. Similarly, function `isValidStateSquare(const ompl::base::State* state, double sideLen, const std::vector<Rectangle>& obstacles)` will return true if the given `state` is valid given the vector of rectangles `obstacles` for a square robot of side `sideLen`. The collision checker for the square robot considers the robot's center as the reference point in the robot's body. A rectangular obstacle is defined with the coordinate (x, y) of its lower left corner, its width and its height. An environment will be a vector that contains several rectangles (including the boundaries) See `CollisionChecking.h` file for more details.

Benchmarking Motion Planners

As you have already experienced with Project 1, sampling-based motion planners (including RTP) require more than one run before you can draw statistically meaningful conclusions about the performance of your planner or any others. To help with evaluation, you will be using OMPL benchmarking functionalities that execute one or more planners many times while recording performance metrics in a database.

The `ompl::tools::Benchmark` class operates in two phases. First is the planning phase, where all of the planners are executed on the same problem for the given number of runs. Second is the analysis phase, where the log file emitted by the benchmark class is post-processed into a SQLite database, and statistics for many common metrics are plotted to a PDF using `ompl_benchmark_statistics.py`, or using the online analysis available through http://plannerarena.org/. The benchmark facilities are extensively documented at http://ompl.kavrakilab.org/benchmark.html#benchmark_code.

Note: `ompl_benchmark_statistics.py` requires `matplotlib` v1.2+ for Python, which can be installed through your favorite package manager or through Python's `pip` program. The Docker container should already have this installed. The script will produce box plots for continuously-valued performance metrics. If you are unfamiliar with these plots, Wikipedia provides a good reference. The script will merge with any existing SQLite database with the same name, so take care to remove any previously existing database files before running the script. You can also upload your SQLite database to http://plannerarena.org/ to interactively view your benchmark data.

Project exercises

1. Implement RTP for rigid body motion planning. At a minimum, your planner must derive from `ompl::base::Planner` and correctly implement the `solve()`, `clear()`, and `getPlannerData()` functions. `Solve` should emit an exact solution path when one is found. If time expires, `solve` should also emit an approximate path that ends at the closest state to the goal in the tree.
    - Your planner **does not** need to know the geometry of the robot or the environment, or the exact $\mathcal{C}$-space it is planning in. These concepts are abstracted away in OMPL so that planners can be implemented generically.
2. Compute valid motion plans for a *point robot* within the plane and a *square robot* with known side length that translates and rotates in the plane using OMPL. Note, instead of manually constructing the state space for the square robot, OMPL provides a default implementation of the configuration space $\mathbb{R}^2 \times \mathbb{S}^1$, called `ompl::base::SE2StateSpace`.
    - Develop at least two interesting environments for your robot to move in. Bounded environments with axis-aligned rectangular obstacles are sufficient.

    - Visualize the world and the path that the robot takes to ensure that your planner is implemented correctly. You must include visualizations of your worlds as well as some example paths your planner generated in your report.

3. Benchmark your implementation of RTP in the 3D *Apartment* and *Home* scenarios from OMPL.app. Make sure to compare your planner against the PRM, EST, and RRT planners over at least 50 independent runs. If all of the planners fail to find a solution, you will need to increase the computation time allowed. **Note**: see the appropriate OMPL.app `.cfg` files for the mesh names, start and goal configurations, etc. They can be found at `/usr/local/share/ompl/resources/3D`. Do not refer to the `.cfg` files used in project 1, as values were tuned to be different. Use the results from your benchmarking to back up claims in your report about RTP's performance quantitatively.
    - The `SE3RigidBodyPlanning.cpp` demo provided in project 1 shows a code example of how to setup a problem using triangle meshes and benchmark planners. Feel free to use this as a reference for the benchmark exercise. The `ompl::app::SE3RigidBodyPlanning` class derives from `ompl::geometric::SimpleSetup`.

Protips

- It may be helpful to start from an existing planner, like RRT, rather than implementing RTP from scratch. Check the files at https://github.com/ompl/ompl/tree/main/src/ompl, and on the online documentation available at http://ompl.kavrakilab.org/.
  Make sure you understand what each line of the RRT algorithm is doing. RRT is a much smarter algorithm than RTP, and while a good starting point, requires you to remove what is not required. You will be penalized if code that is not a part of the RTP algorithm remains in your implementation.
  Additionally, although RRT has a variant that saves intermediate states, you **should not** implement a variant of RTP that saves intermediate states.
- The `getPlannerData` function is implemented for all planners in OMPL. This method returns a `PlannerData` object that contains the entire data structure (tree/graph) generated by the planner. `getPlannerData` is very useful for visualizing and debugging your tree.
- If your `clear()` function implemented in RTP is incorrect, you might run into problems when benchmarking as your planner's internal data structures are not being refreshed.
- You **do not need** a nearest neighbor data structure for RTP. RTP has no need for neighbor proximity queries, as it selects states to extend from at random.
- RTP is not an intelligent planner, and is a very simple algorithm compared to other sampling-based planners. Keep this in mind when observing its performance.
- For benchmarking, make sure you give the planners a reasonable amount of time to solve the problem. If the time limit is too low, then it is highly unlikely your planner will solve any instances, and the resulting benchmark will not be informative. Make sure you understand when the planner returns an exact solution (it has solved the problem) and when it has returned an approximate solution (it has not solved the problem, and has returned a guess).
- Solution paths can be easily visualized using the `printAsMatrix` function from the `PathGeometric` class. Use any software you want to visualize this path, but provide your script with your submission. See http://ompl.kavrakilab.org/pathVisualization.html for more details.
- In exercise 3, make sure to link your program against `libompl`, `libompl_app`, and `libompl_app_base` against the problem instances. The provided Makefile given in the OMPL installation handout and the Docker container does this.

Deliverables

This project must be completed in pairs. **Submissions are due Thursday September 28th at 1pm**.

To submit your project, clean your build space with `make clean`, zip up the project directory into a file named `Project3_<your NetID>_<partner's NetID>.zip`, and submit to Canvas. Your code must compile within a modern Linux environment. If your code compiled in the Docker container, then it will be fine. Include a README with details on compiling and executing your code. In addition to the archive, submit a short report that summarizes your experience from this project. The report should be anywhere from 1 to 5 pages in length in PDF format, and contain the following information:

- **(2.5 points)** A succinct statement of the problem that you solved.
- **(2.5 points)** A short description of the robots (their geometry) and configuration spaces you tested in exercise 2.
- **(5 points)** Images of your environments and a description of the start-goal queries you tested in exercise 2.
- **(5 points)** Images of paths generated by RTP in your environments you tested in exercise 2.
- **(5 points)** Summarize your experience in implementing the planner and testing in exercise 2. How would you characterize the performance of your planner in these instances? What do the solution paths look like?
- **(15 points)** Compare and contrast the solutions of your RTP with the PRM, EST, and RRT planners from exercise 3. Elaborate on the performance of your RTP. Conclusions *must* be presented quantitatively from the benchmark data. Consider the following metrics: computation time, path length, and the number of states sampled (graph states).
- **(5 points)** 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. **Additionally**, describe your individual contribution to the project.

Take time to complete your write-up. It is important to proofread and iterate over your thoughts. Reports will be evaluated for not only the raw content, but also the quality and clarity of the presentation. When referencing benchmarking results you must include the referenced data as figures within your report.

Additionally, you will be graded upon your implementation. Your code must compile, run, and solve the problem correctly. Correctness of the implementation is paramount, but succinct, well-organized, well-written, and well-documented code is also taken into consideration. Visualization is an important component of providing evidence that your code is functioning properly. The breakdown of the grading of the implementation is as follows:

- **(30 points)** A correct implementation of RTP that generates valid motion plans.
- **(30 points)** Benchmarking of RTP against other planners in all prescribed environments.

Each pair need to provide only one submission.