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 for rigid-body planning and then tune its parameters, along with those of other planners, to identify the best path according to a chosen metric.

This project must be completed in pairs, with each pair submitting only one solution. We will create a piazza post to help you find a partner. Please read the provided code's README carefully, as it contains important instructions for running the code.

Deadlines

This project consists of two parts. In the first part, you will implement a sampling-based motion planner and test it in your own customized environment. The deadline for this part (Exercise 1 and 2) is **October 9th at 1:00 pm**. You are expected to submit your implementation of the random tree planner along with a short write-up as specified in the deliverables. The second part (Exercise 3) is due on **October 16th at 1:00 pm**. For this submission, you will turn in all components of the project, including your full codebase and a complete report.

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.

Parameter Tuning of Planners for Optimal Path

Note: Please complete the following required and optional steps before starting this portion of the project:

1. **[Required] In your Docker container,** run `pip install xmltodict`.
2. **[Optional]** You may encounter a long list of build warnings due to a package called `eigen`. To resolve these warnings, please take the following steps:
   (a) **On your host machine,** `git clone https://gitlab.com/libeigen/eigen.git -b 3.4` in a convenient location.
   (b) Copy the directory that was just created (`eigen`) into your Docker container using the following command: `docker cp eigen comp450-workspace:/usr/local/include/eigen`
   (c) On line 6 of the Makefile in the Project 3 directory, change `-I/usr/include/eigen3` to `-I/usr/local/include/eigen`.

For sampling-based motion planning, each planner exposes a set of parameters that can significantly affect its performance. For example, the Rapidly-exploring Random Tree (RRT) has parameters such as *goal bias* and *step size* that influence how quickly and effectively it finds a path. As a result, depending on the evaluation metric of interest, it is often necessary to tune these parameters to achieve the best possible outcome.

In this part of the project, you will tune the parameters of your implemented RTP planner, along with three existing planners: **RRT**, **RRT-Connect**, and **PRM**. For each planner, you may adjust parameters such as:

- Planning time
- Goal bias
- Range
- Maximum number of neighbors
- Connection radius

We will evaluate your planned paths using the following four metrics:

1. **Planning time** – how quickly the planner returns a solution.
2. **Path smoothness** – how continuous and natural the trajectory is.
3. **Path length** – the total distance traveled.
4. **Minimum clearance** – how safely the path avoids obstacles.

Your task is to carefully tune each planner's parameters to optimize performance for each metric. In other words, for each of the four metrics, you should identify a set of parameter values that leads to the best performance you can achieve. In the provided code, we include five different sets of start–goal pairs. Your planner will be run 5 times on each pair, and the results will be used to compute statistical averages across the four metrics.

Finally, you are required to document your full analysis process. This should include your reasoning for the parameter choices you made, the trade-offs you observed, and any patterns or general insights about how different parameters affect planner behavior across the different metrics.

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`.
    - We provide a starting code in `Project3Exercise2.cpp`.
    - 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. You are required to experiment with different parameter settings for each of the four planners: your implemented RTP planner, **RRT**, **RRT-Connect**, and **PRM**. We have already provided template code in `Project3Exercise3.cpp`. You will need to set up planner configurations as shown in the examples in this file to run your experiments. We also provide a visualization script, `visualize_trajectory.py` so that you can check the trajectory for your tuning experiment. For each run, it saves one solution path that can be visualized with the script. Please refer to the script for more details. While running experiments is straightforward, the key part of this exercise is to perform a *thorough analysis* of the results. In particular, you should:

   - Ensure your RTP planner has a function via which the goal bias parameter (”small probability” mentioned earlier in this document) may be modified.
   - Complete the `ManipulatorPlanner::setRTPParameters` function in `provided/ManipulatorPlanner.cpp` to use your RTP planner’s function to set its goal bias.
   - Modify the `main` function in `Project3Exercise3` to add your desired planner configurations.
   - Explore how different parameter values (e.g., planning time, goal bias, range, maximum number of neighbors, connection radius) affect the performance of each planner.
   - Compare the results across planners for the four metrics we defined earlier: planning time, path smoothness, path length, and minimum clearance.
   - Identify trade-offs between different metrics (e.g., shorter path length may reduce clearance, or more planning time may increase smoothness).
   - Reason about why certain parameter values lead to better or worse outcomes, drawing connections to the underlying algorithmic behavior of each planner.
   - Look for patterns or trends across multiple experiments, rather than reporting results from a single trial.
   - Visualize the paths you generate using the `visualize_trajectory.py` script.

   Your final submission should not only include the numerical results but also provide a clear explanation of how you arrived at the best parameter settings for each metric, along with your observations, insights, and justifications. The quality of your analysis and reasoning will be an important part of the evaluation. In addition, we will have a leaderboard competition. Your place in the leaderboard will not count toward your grade, but the winner will receive a prize. More details will be shared soon.

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.

Deliverables

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?
- **(40 points)** Compare and contrast the performance of your RTP planner with the PRM, EST, and RRT planners from Exercise 3. Your analysis should go beyond a simple description: you are expected to critically evaluate how your planner performs relative to the others.
  - Present your conclusions *quantitatively*, supported directly by the experiments you run (figures and tables would be very helpful).
  - Use the following metrics in your comparison: **planning time**, **path length**, **path smoothness**, **minimum clearance**.
  - Discuss trade-offs among planners. For example, does your RTP achieve faster solutions at the cost of longer paths, or vice versa?
  - Provide reasoning for the observed differences, linking them to the algorithmic design of each planner.

  Your write-up should not only report the data but also demonstrate clear reasoning and insight into why certain planners perform better under specific metrics or conditions.
- **(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 results you should include the referenced data as figures or tables 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. Please also include comments in your code to explain your implmentation. The breakdown of the grading of the implementation is as follows:

- **(35 points)** A correct, well-organized implementation of RTP that generates valid motion plans.