Motion Planning RBE 550

Project 2: Barking Up A Random Tree

DUE: Tuesday February 18 at 11:59 pm.

All written components should be typeset using LaTeX. A template is provided on Overleaf. However, you are free to use your own format as long as it is in LaTeX.

Submit two files **a)** your code as a zip file **b)** the written report as a pdf file. If you work in pairs, only **one** of you should submit the assignment, and write as a comment the name of your partner in Canvas. Please follow this formatting structure and naming:

```
project_YOUR_NAME.pdf
project_YOUR_NAME.zip
    code
        CMakeLists.txt
        CollisionChecking.cpp
        CollisionChecking.h
        RTP.cpp
        RTP.h
        PlanningRTP.cpp
        visualise.py
        build
            obstacles1.txt
            obstacles2.txt
            path1.txt
            path2.txt
            path3.txt
            path4.txt
            benchmarkChain20.db
            benchmarkChain10.db
```

Present your work and your work only. You must *explain* all of your answers. Answers without explanation will be given no credit. Points will be deducted for failing to follow the submission naming and structure.

Goal of This Project

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.

Theoretical Questions (40 points)

1. **(5 points)** Write two key-differences between the Bug 1 and Bug 2 algorithms.

2. **(10 points)** A fundamental part of A* search is the use of a heuristic function to avoid exploring unnecessary edges. To guarantee that A* will find the optimal solution, the heuristic function must be admissible. A heuristic h is admissible if

$$h(n) \leq T(n)$$

where T is the true cost from n to the goal, and n is a node in the graph. In other words an admissible heuristic never overestimates the true cost from the current state to the goal.

A stronger property is consistency. A heuristic is consistent if for all consecutive states n, n'

$$h(n) \leq T(n, n') + h(n')$$

where T is the true cost from node n to its adjacent node n' .

Answer the following:

   (a) **(5 points)** Imagine that you are in the grid world and your agent can move up, down, left, right, and diagonally, and you are trying to reach the goal cell $f = (g_x, g_y)$. Define an admissible heuristic function h_a and a non-admissible heuristic h_b in terms of x, y, g_x, g_y. Explain why each heuristic is admissible/non-admissible.

   (b) **(5 points)** Let h_1 and h_2 be consistent heuristics. Define a new heuristic $h(n) = max(h_1(n), h_2(n))$. Prove that h is consistent.

3. **(10 points)** Consider workspace obstacles A and B. If A∩B $\neq \emptyset$, do the configuration space obstacles QA and QB always overlap? If A ∩ B $= \emptyset$, is it possible for the configuration space obstacles QA and QB to overlap? Justify your claims for each question.

4. **(15 points)** Suppose you are planning for a point robot in a 2D workspace with polygonal obstacles. The start and goal locations of the robot are given. The visibility graph is defined as follows:

   - The start, goal, and all vertices of the polygonal obstacles compose the vertices of the graph.
   - An edge exists between two vertices of the graph if the straight line segment connecting the vertices does not intersect any obstacle. The boundaries of the obstacles count as edges.

   Answer the following:

   (a) **(10 points)** Provide an upper-bound of the time it takes to construct the visibility graph in big-O notation. Give your answer in terms of n, the total number of vertices of the obstacles. Provide a short algorithm in pseudocode to explain your answer. Assume that computing the intersection of two line segments can be done in constant time.

   (b) **(5 points)** Can you use the visibility graph to plan a path from the start to the goal? If so, explain how and provide or name an algorithm that could be used. Provide an upper-bound of the run-time of this algorithm in big-O notation in terms of n (the number of vertices in the visibility graph) and m (the number of edges of the visibility graph). If not, explain why.

Programming Component (60 points)

Project Description

A broad class of motion planning algorithms consider only geometric constraints (e.g., bug algorithms, visibility graph methods, Randomly Exploring Random Trees). 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.

New Planner: Random Tree Planner (RTP)

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 (Specially RRT). 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.

Benchmarking Motion Planners

Since many of the state-of-the-art algorithms for motion planning are built upon the concept of random sampling (including RTP), one run is not sufficient to draw statistically meaningful conclusions about the performance of your planner or any others. To help with evaluation, OMPL provides 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 the online analysis available through PlannerArena or using `ompl_benchmark_statistics.py`.

The benchmark facilities are extensively documented here.

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 virtual machine 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 PlannerArena to interactively view your benchmark data.

Project exercises

1. **(15 points)** Fillout the missing functions in `CollisionChecking.cpp` by implement collision checking for a *point robot* within the plane, and a *square robot* with known side length that translates and rotates in the plane in. There are many ways to do collision checking with a robot that translates and rotates. Here are some things to consider
   - The obstacles will only be axis aligned rectangles.
   - You can re-purpose the point inside the squre code from Project1.
   - Using a line intersection algorithm might be helpful.
   - Make sure that your collision checker accounts for all corner cases
2. **(15 points)** 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, it should also emit an approximate path that ends at the closest state to the goal in the tree. It may be helpful to start from an existing planner, and modify it, such as RRT. You can check the source files of RRT.h, RRT.cpp, and the online documentation available here.
   Note that:
   - You need to fill the implementations in `RTP.h` and `RTP.cpp`
   - 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.
3. **(20 points)** Fill-out the missing functions at `PlanningRTP.cpp` You can check the OMPL demos on how to setup different planning problems here. The RigidBodyPlanning might be the most relevant.
   - Develop at least two interesting environments for your robot to move in. Bounded environments with axis-aligned rectangular obstacles are sufficient.
   - Perform motion planning in your developed environments for the *point robot* and the *square robot*. Collision checking must be exact, and the robot should not leave the bounds of your environment. 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`.
   - Include in your report the images of your environments, corresponding solution paths, and start and goal queries. In your report you will need to provide 4 images. Two for the point robot, and defined environments and two for the square robot and defined environments.
   - You can use the visualise.py script provided to you for visualizing the environments and the paths. You can run the script with:

     ```
     python3 visualise.py --obstacles <obs file> --path <path file>
     ```

     We have provided you with `example_path.txt` and `example_obstacles.txt` as reference file for the visualizer. You should make your own files and provide them under the build directory.

4. **(10 points)** Download and modify appropriately the KinematicChain environment to benchmark your planner. If all of the planners fail to find a solution, you will need to increase the computation time allowed. Benchmark with a kinematic chain of 20 and 10 links. Compare and contrast the solutions of your RTP with EST, and RRT planners. Elaborate on the performance of your planner 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). In your submitted files include the 2 generated databased named `benchmarkChain20.db` and `benchmarkChain10.db`

Your code must compile, run, and solve the problem correctly. You are allowed to add new functions but you are not allowed to use external libraries, or modify the names of the provided functions. Correctness of the implementation is paramount, but succinct, well-organized, well-written, and well-documented code is also taken into consideration.

Protips

- The `getPlannerData()` function is implemented for all planners in OMPL. This method returns a `PlannerData` object that contains the entire data structure (tree or graph) generated by the planner. `getPlannerData()` is very useful for visualizing planner samples and edges, and also 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.
- Solution paths can be easily dumped to a txt file using the `printAsMatrix()` function from the `PathGeometric` class.