COMP/ELEC/MECH 450: Algorithmic Robotics

Project 3: Randomized Rigid Body Planning

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

The first motion planning algorithms (e.g., bug algorithms, visibility graph, PRM) considered just geometric constraints, meaning that the 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 and the *piano mover's* problem. Other constraints on the robot, like limits on velocity or acceleration, are ignored in this formulation. While considering only geometric constraints may seem simplistic, many small manipulators and wheeled robots safely ignore dynamical effects during planning due to their relatively low momentum. Formally, this is known as a *quasistatic* assumption.

In this project, you will implement a simple sampling-based motion planner in OMPL to plan for a free moving rigid body. Additionally, you will create two interesting 2D environments to test your planner. You will also exercise your knowledge of rigid transformations in 2D as well by implementing methods that check for collision in your environments for robots with three different geometries. Once your implementation is complete, you will compare the performance of your planner against existing state-of-the-art algorithms.

Random Tree Planner

The Random Tree algorithm is a simple sampling-based method that grows a tree in the configuration space of the robot. The algorithm is loosely based on a random walk of the configuration space and operates using the following strategy:

1. Select a random configuration a from the existing Random Tree.
2. Sample a random configuration b in the configuration space. With a small probability select the goal configuration instead of a random one.
3. Check whether the straight-line path between a and b is valid (i.e., collision free). If the path is valid, add the path to the Random 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.

Collision Checking

For this project, you will implement methods for collision checking for three different kinds of robots in the 2D environments that you create: a point robot, a circle robot, and a square robot. For simplicity, you may assume all the obstacles in the environment are **axis-aligned rectangles**.

1. For the point robot, the collision checker must decide whether the robot is inside an obstacle. The condition below becomes true when the robot is in collision with an axis-aligned rectangle:

$$(x_{min} \leq x \leq x_{max}) \; and \; (y_{min} \leq y \leq y_{max}) \tag{1}$$

where x_{min}, x_{max}, y_{min} and y_{max} are the range of the axis-aligned obstacle's coordinates, and x, y are the coordinates of the point robot.

2. When the robot has geometry, as in the circle robot, collision checking becomes a bit more complex. Borrowing from the Minkowski sum, we can *inflate* the obstacles by the radius of the circle robot, effectively shrinking the circle to a point. When the obstacles are axis-aligned rectangles, the Minkowski sum with the circle robot results in axis-aligned rectangular obstacles, except the corners are now rounded. The condition below becomes true when the circle robot with radius r intersects with an axis-aligned rectangle:

$$\begin{aligned} &((x_{min} - r \leq x \leq x_{max} + r) \text{ and } (y_{min} \leq y \leq y_{max})) \text{ or} \\ &((x_{min} \leq x \leq x_{max}) \text{ and } (y_{min} - r \leq y \leq y_{max} + r)) \text{ or} \\ &\qquad \exists e, \text{ a vertex of the rectangle, } \|\overrightarrow{ce}\| \leq r \end{aligned} \tag{2}$$

   where x_{min}, x_{max}, y_{min} and y_{max} are the range of the axis-aligned rectangle's coordinates, $c = (x, y)$ is the center of the circle robot, and $\|\overrightarrow{ce}\|$ is the Euclidean norm of the vector from point c to point e.

3. For a square robot that translates and rotates, we apply a 2D rigid body transformation $\begin{pmatrix} R(\theta) & t \\ 0 & 1 \end{pmatrix}$ to the four vertices of the square. By doing this, we know the new location of each vertex in the global frame after the transformation. Then, for each side of the square robot we check for intersection with each side of every polygonal obstacle. This is a line segment intersection in 2D. Feel free to reuse the algorithm you developed in Homework 1 to determine whether two given edges intersect or not.

Project exercises

1. Implement the Random Tree Planner 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 emit an approximate path that ends at the closest state to the goal in the tree.
2. Create at least two interesting (2D) environments for your robots to navigate in. Bounded environments with axis-aligned obstacles are sufficient.
3. For the environments you create, implement exact collision checking methods for the following 2D robot geometries:
   - A point robot that can translate
   - A circular robot (with a known radius) that can translate
   - A square robot (with a known side length) that can translate and rotate
4. Compute motion plans for the different robot geometries using your Random Tree. Visualize the world and the path that the robot takes to ensure that your planner and collision checker are working properly.
5. Compare your Random Tree against the existing implementations of PRM, EST, and RRT using the OMPL `Benchmark` class. Any conclusions must come from at least 20 independent runs of each planner.

Deliverables

This project may be completed in pairs and comprises 20% of your total project grade. **Submissions are due Thursday Oct. 2 at 1pm**.

Submissions are on OwlSpace. Submit a single archive (.tar, .zip, etc.) containing only your source code, Makefile, and visualization code. If you deem it necessary, also include a README with details on how to

compile and run your code. Your code must compile on a modern Ubuntu/Mac operating system running OMPL v0.14.2. In addition to the source, a short write-up must be submitted that summarizes your experiences and findings from this project. The report should be no longer than 5 pages, in PDF format, and contain (at least) the following information:

- A succinct statement of the problem that you solved.
- Images of your environments and a description of the start-goal queries you are evaluating.
- A short description of the robots (their geometry) and configuration spaces.
- A summary of benchmarking data for the Random Tree, PRM, EST, and RRT in both worlds for all robot geometries. Data must be presented quantitatively and conclusions drawn must be supported from the data. Consider the following metrics: computation time, path length, and number of states sampled.
- Concluding remarks about what you liked and disliked about this project. Include a difficulty rating for each exercise on a scale of 1–10 (1=trivial, 10=impossible), and an estimate of the time spent on each exercise.

Please submit the PDF write-up separate from the source code archive. Those completing the project in pairs need to provide just one submission for code and report.

Grading

Your code should 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. 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.

Protips

- It may be helpful to start from an existing planner, like RRT, rather than implementing the random tree from scratch. Check `omplapp/ompl/src/ompl/geometric/planners/rrt` if you compiled from source (or the virtual machine). The files are also found at https://bitbucket.org/ompl/ompl/src.
- You can implement and debug your planner without the special collision checking methods. For a point robot in an empty world, the state is always valid.
- 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`.
- The `getPlannerData` function is implemented for all planners in OMPL. This method returns a `PlannerData` object that contains the entire data structure generated by the planner. Once you implement `getPlannerData` for your RandomTree, use this method to visualize and debug your tree.
- Your planner **should not and does not** need to know the geometry of the robot or the environment. These concepts are abstracted away in OMPL so that planners can be implemented generically. Your planner should not make any assumptions about the robot's geometry or what the world looks like.
- Solution paths can be easily visualized using the `printAsMatrix` function from the `PathGeometric` class. You should write visualization tools for worlds and paths so that you can reuse this for future projects. Use any software you want: MATLAB, Excel, gnuplot, Python's matplotlib, etc. See http://ompl.kavrakilab.org/pathVisualization.html for more details.