Algorithmic Robotics
COMP/ELEC/MECH 450/550

Project 6: 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 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 26th. In order to get approval you need 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.

An initial progress report is due at **1pm October 26th** 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 are and what progress you have made thus far. Only one report needs to be submitted for each group.

Final submissions are due **Tuesday November 21st 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. The **latest date for submission is 8am on 11/27th 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, ou are allowed to use any language (C++, Python, etc.), but your code must compile and run within a modern Linux environment. If your code 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 need to be done 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 on 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.

Please 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. Details of the presentation will be announced closer to the presentation dates. **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 describing in detail their own contribution to the project.

Project 1: Centralized Multi-robot Planning

Planning motions for multiple robots that operate in the same environment is a challenging problem. One method for solving this problem is to compute a plan for all of the robots simultaneously by treating all of the robots as a single composite system with many degrees of freedom. This is known as centralized multi-robot planning. A naive approach to solving this problem constructs a PRM for each robot individually, and then plans a path using a typical graph search in the composite PRM (the product of each PRM). Unfortunately, composite PRM becomes prohibitively expensive to store, let alone search. If there are k robots, each with a PRM of n nodes, the composite PRM has n^k vertices!

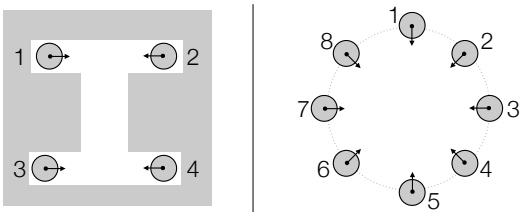

Figure 1: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1,\ldots,4$, needs to swap positions with robot $i+4$.

Searching the exponentially large composite roadmap for a valid multi-robot path is a significant computational challenge. Recent work to solve this problem suggests *implicitly* searching the composite roadmap using a discrete version of the RRT algorithm (dRRT). At its core, dRRT grows a tree over the (implicit) composite roadmap, rooted at the start state, with the objective of connecting the start state to the goal state.

The key steps of dRRT are as follows:

1. Sample a (composite) configuration q_{rand} uniformly at random.
2. Find the state q_{near} in the dRRT nearest to the random sample.
3. Using an *expansion oracle*, find the state q_{new} in the composite roadmap that is connected to q_{near} in the closest direction of q_{rand}. (Figure 2).
4. Add the path from q_{near} to q_{new} to the tree if it is collision free.
5. Repeat 1-4 until until the goal is successfully added to the tree.

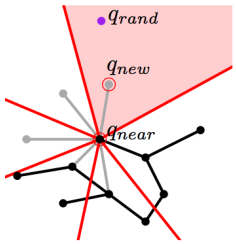

Figure 2: The dRRT expansion step. *From Solovey et al.; see below.*

Deliverables:
Implement the dRRT algorithm for multi-robot motion planning. For n robots, you will first need to generate n PRMs, one for each robot. If your robots are all the same, one PRM will suffice for every robot. Then use your dRRT algorithm to implicitly search the product of the n PRMs. The challenges here are the implementation of the expansion oracle (step 3) and the local planner to check for robot-robot collisions (step 4). The reference below gives details on one possible expansion oracle (section 3.1) and local planner (section 4.2). Evaluate your planner in scenarios with *at least* four robots, like those in Figure 1. For simplicity you may assume that the robots are planar rigid bodies. You may need to construct additional scenarios to evaluate different properties of the dRRT algorithm.

Address the following in your report: Is your dRRT implementation always able to find a solution? How does the algorithm scale as the number of robots increases? Elaborate on the relationship between the quality of the single robot PRMs and the time required to run dRRT. What do the final solution paths look like?

Reference:

K. Solovey, O. Salzman, and D. Halperin, Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning. In *Algorithmic Foundations of Robotics*, 2014. http://robot.cmpe.boun.edu.tr/wafr2014/papers/paper_20.pdf.

Project 2: Decentralized Multi-robot Coordination

Planning motions for multiple robots simultaneously is a difficult computational challenge. Algorithms that provide hard guarantees for this problem usually require a central representation of the problem, but the size and the dimension of the search space limits these approaches to no more than a handful of robots. Decentralization or distributed coordination of multiple robots, on the other hand, is an effective (heuristic) approach that performs well in practical instances by limiting the scope of information reasoned over at any given time. There exist many effective methods in the literature to practically coordinate the motions of hundreds or even thousands of moving agents.

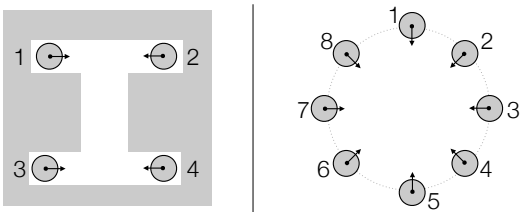

Figure 3: Two challenging multi-robot problems. *Left:* robots 1 and 2 need to swap positions with robot 4 and 3, respectively. *Right:* robot $i, i = 1,\ldots,4$, needs to swap positions with robot $i+4$.

A more recent approach reasons over the relative velocities between robots, employing the concept of a *reciprocal velocity obstacle* to simultaneously select velocities for each robot that both avoid collision with other robots and allow each robot to progress toward its goal. This technique is used not only in robotics but also in animation and even video games (e.g., Warhammer 40k: Space Marine, Crysis 2). Briefly, a reciprocal velocity obstacle RVO^B_A defines the space of velocities of robot A that will cause A to collide with another robot B at some point in the future, given the current position and velocity of A and B. Using this concept, a real-time coordination algorithm can be developed that simulates the robots for some short time period, then recalculates the velocities for each robot by finding a velocity that lies outside the union of each robot's RVO.

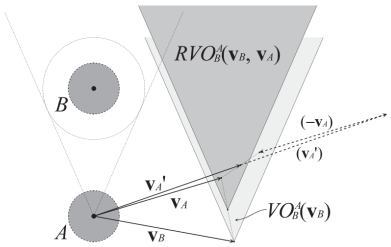

Figure 4: The reciprocal velocity obstacle formed by robot B's current position and velocity relative to robot A. *From Berg et al. 2008; see below.*

Deliverables:

Implement the reciprocal velocity obstacles algorithm for decentralized multi-robot coordination and evaluate the method in multiple scenarios with varying numbers of robots. You may assume that the robots are disks that translate in the plane.

Address the following: Is your method always successful in finding valid paths? Expound upon the kinds of environments and/or robot configurations that may be easy or difficult for your approach? How well does your method scale as you increase the number of robots? Qualitatively, how do the solution paths look? Depending on the method you chose, answer the following questions.

Discuss the challenges faced by the RVO approach as the number of robots increases. Is there always a

velocity that lies outside of the *RVO*? How practical do you think this algorithm is?

Protips:
The implementation of the *RVO* method requires reasoning over the velocities of your robots. You can greatly reduce the complexity of the implementation by planning geometrically and bounding the velocity by limiting the distance a robot can travel during any one simulation step. Moreover, the *RVO* implementation requires reasoning over the robot geometry; choose simple geometries (circles) and ensure your implementation is correct before considering more complex geometries.

Visualization, particularly animation, is key in debugging this method. Matlab, Matplotlib, and related packages support generating videos from individual frames.

References:

J. van den Berg, M. Lin, and D. Manocha, Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In *IEEE Int'l Conference on Robotics and Automation*, pp. 1928-1935, 2008. https://www.cs.unc.edu/~geom/RVO/icra2008.pdf.

Project 3: Dynamic Manipulation

NOTE: This project is popular with **mechanical engineers**, but is very complicated if you do not understand the math. Please only choose this project if you are confident with your ability to understand the referenced work.

Robotic manipulators consist of *links* connected by *joints*. They are often arranged in a serial chain to form an arm. Attached to the arm is a hand or a specialized tool. Clearly, manipulators are essential if we want robots to do useful work. However, planning for manipulators can be very challenging. The state of a manipulator can be described by a vector θ 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. The state vector then becomes $(\theta, \dot{\theta})$ (i.e., a vector that contains both joint positions and velocities). In this assignment you will develop an OMPL model for general planar manipulators with n revolute joints. The equations of motion are given in the first reference below. Although it is important that you understand this paper, *you do not need to derive any equations yourself.* Specifically, the kinematics and dynamics of a planar manipulator are given. The dynamics equations are of the form:

$$\tau = M(\theta)\ddot{\theta} + C(\theta, \dot{\theta}) + V(\theta),$$

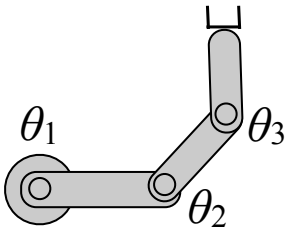

Figure 5: Three-link planar manipulator.

where M is the inertia matrix, C is the vector of Coriolis and centrifugal forces, and V is the vector of gravity forces. You need to rewrite them to obtain the following systems of ordinary differential equations:

$$\frac{d}{dt}\begin{pmatrix}\theta\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}\dot{\theta}\\ M^{-1}(\theta)\big(\tau - C(\theta,\dot{\theta}) - V(\theta)\big)\end{pmatrix}$$

and implement this as a "propagate" function [1]. For simplicity, you do not need to deal with friction or collisions. Given the "propagate" function, a planner needs to find a series of controls $\tau_1, \ldots, \tau_n$ that, when

[1] You should not write your own numerical matrix inversion routine. Instead, use a linear solver (a function that finds x s.t. $Ax = b$) from a standard library, such as LAPACK, the C++ Eigen library, or Python's numpy module.

applied for $t_1, \ldots, t_n$ seconds, respectively, will bring a robot from an initial pose and velocity to a final pose and velocity.

When you test your code, it is helpful to check the total energy $E(\theta, \dot{\theta})$ when you integrate the equations above with $\tau = 0$ and the initial values for $(\theta, \dot{\theta})$ chosen arbitrarily (for a straight horizontal configuration you might have some intuition what motion would "look right"). In that case, the system is closed and the energy should remain constant (up to numerical precision). The energy is equal to $E(\theta, \dot{\theta}) = \frac{1}{2}\dot{\theta}^T M(\theta)\dot{\theta} + g\sum_{i=1}^{n} m_i h_i$, where $g = 9.81$ is the gravitational constant, m_i is the mass of link i, and h_i is the height of the center of mass of link i.

In the references below, the variables (x_i, y_i) do *not* refer to the position of joint i in the workspace, but the position of the end effector relative to the position of joint i. This seemingly odd parametrization is useful, because it greatly simplifies the dynamics equations. Using the notation of the references, the endpoint of link i is simply $(\sum_{i=1}^{n} l_i \cos\phi_i, \sum_{i=1}^{n} l_i \sin\phi_i)$. For visualizing the output of your program, you can print these positions and plot them with any plotting program you are familiar with.

References:

[1] L. Žlajpah, Dynamic simulation of n-R planar manipulators. In *EUROSIM '95 Simulation Congress*, Vienna, pp. 699–704, 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.1622&rep=rep1&type=pdf. *This is a concise description of the dynamics.*

[2] L. Žlajpah, Simulation of n-R planar manipulators, *Simulation Practice and Theory*, 6(3):305–321, 1998. http://dx.doi.org/10.1016/S0928-4869(97)00040-2. *This paper has a few typos, but it has more details on how to compute* $C(\theta, \dot{\theta})$.

Deliverables:

Part 1: You will implement a general state space for dynamic planar manipulators with n revolute joints. You need to produce solutions for manipulators with 3 joints between two states with arbitrary joint positions and velocities. It should be possible to set the number of joints, the link lengths, and limits on joint velocities and torques. If you model each joint position with a SO2StateSpace (which is recommended), you do not have to worry about joint *position* limits (the velocity and torque limits are still important). If the limits are too small, it may not be possible to solve a given motion planning problem, while large limits might force a planner to search parts of the configuration space that are not all that useful.

The mass matrix M (called H in the papers above) is defined in equations 8–11 of ref. 1 above. The Coriolis and centrifugal term C (called h in the papers above) is defined by the fourth equation on p. 310 of ref. 2, equation 11 and the two equations right above it (all in ref. 2). Finally, the gravity term V (called g in the papers above) is defined in equation 15 of ref. 1. There is a typo in ref. 2 when computing the Coriolis and centrifugal term C:

$$\frac{\partial^2 y_{ci}}{\partial q_j \partial q_k} = \begin{cases} -y_r + y_i - l_{ci}\sin(\varphi_i), & j \leq i \text{ and } k \leq i \\ 0, & j > i \text{ or } k > i \end{cases}$$

This is correct in reference 1.

There is another typo when computing the gravity term. Equation 15 in ref. 1 and Equation 13 in ref. 2 differ on the initial subscript in the sum. The correct subscript is $k = i + 1$.

Part 2: Implement collision checking for planar manipulators. You can model a manipulator as a collection of connected line segments.

Part 3: I Compare the solutions you get by using a sampling-based planner with those obtained with a pseudo-inverse type controller as described in the references for a variety of queries.

Project 4: Implementing Chance Constrained RRT (CC-RRT)

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:

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

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

For the disturbance covariance and initial state covariance use:

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

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

- Briefly describe 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.

[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.