COMP/ELEC/MECH 450: Algorithmic Robotics

Project 4: Motion Planning for Dynamical Systems

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

In the previous project, you computed motion plans for a *rigid body* that was able to move in any direction and at any speed, so long as the path was collision free. When planning for complex systems, however, a collision-free path may not always be feasible. As an example, consider a car trying to parallel park. Although the *straight-line* path sideways to the parking spot may be collision free, the car cannot translate horizontally and must execute a longer path to reach the parking spot. The fact that the car cannot directly move sideways means that the car is *non-holonomic*; the system cannot control its position directly. This is a key point that differentiates planning for a rigid body (holonomic system) and a complex (AKA non-holonomic) system.

In this project you will plan motions for non-holonomic systems whose dynamics are described by an ordinary differential equation of the form $\dot{q} = f(q,u)$, where q is a vector describing the current state of the system and u is a vector of control inputs to the system. The systems you will plan for are a torque-controlled pendulum and a car-like vehicle moving in a street environment. Dynamically feasible and collision free motions for these systems will be computed using the RRT, EST, and KPIECE planners that are already implemented in OMPL. Additionally, you will also learn about and implement a new planner called Reachability-Guided RRT (RG-RRT), one of many variants of RRT.

A Torque-controlled Pendulum System

A pendulum is one of the simplest dynamic systems. The state $q = (\theta, \omega)$ of the system is described by the orientation of the pendulum (θ) and its rotational velocity (ω). Assume that $\theta = 0$ corresponds to the pendulum being horizontal. The objective is to move the pendulum from the initial state $(\theta, \omega) = (-\frac{\pi}{2}, 0)$ (i.e., the pendulum is hanging down at rest) to the final state $(\theta, \omega) = (+\frac{\pi}{2} \mod 2\pi, 0)$ (i.e., the pendulum is standing up). To accomplish this, a motor can apply a torque τ to the axis of rotation of the pendulum. The pendulum's dynamics can be described by the following differential equation:

$$\dot{q} = \begin{pmatrix} \dot{\theta} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} \omega \\ -9.81 \times \cos\theta + \tau \end{pmatrix}.$$

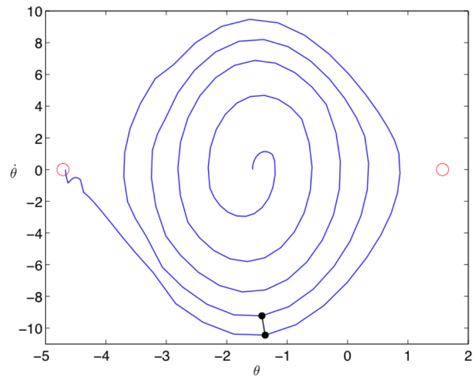

Figure 1: A possible solution trajectory for the pendulum in phase space. *From Shkolnik et al.; see below.*

When an arbitrarily large torque can be applied, this is not a very difficult planning problem. In reality, the pendulum's motor can source only a finite torque. In your implementation, you should set limits on the torque that requires the pendulum to swing back and forth a few times before it has enough momentum to swing up to the goal state. (By inspecting the ODE you might be able to predict the torque bounds that will prevent the pendulum from swing up without back-and-forth motions.) Figure 1 shows a possible solution on the (θ, ω) phase space of the pendulum. The start pose is at the center of the spiral and either one of the red circles are valid end poses.

A Car-like system

The second system involves a simple vehicle moving in a street environment. The state of the vehicle is represented by its position, heading, and forward velocity: $q = (x, y, \theta, v)$. The control inputs to this system consist of the angular velocity and the forward acceleration of the vehicle, $u = (u_0, u_1)$, both of which are bounded in magnitude. The system dynamics are described by:

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v\cos\theta \\ v\sin\theta \\ u_0 \\ u_1 \end{pmatrix}.$$

Similar to the pendulum system, the car cannot source an arbitrarily large acceleration or angular velocity. You need to set bounds on the control space to ensure a dynamically feasible trajectory.

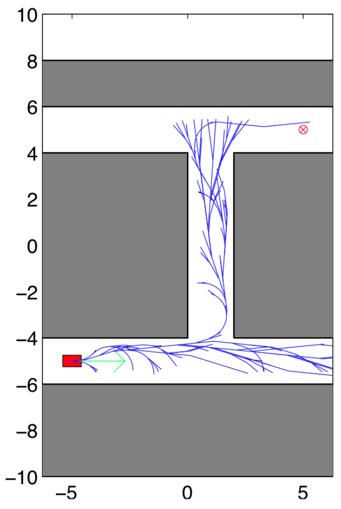

Figure 2: Vehicle navigating in street environment. *From Shkolnik et al.; see below.*

Planning for Dynamical Systems

The key difference when planning for dynamical systems is that the *input space* of controls must be sampled, along with a duration to apply the control. In the rigid body case, this input space was simply the configuration space since the motion between any two configurations is feasible. For dynamical systems, a control input is applied for a prescribed time, and the equations of motion are applied to compute the resulting configuration. The planners in OMPL for dynamic systems sample controls that are applied for some small period of time. It is thus necessary to integrate the equation $\dot{q} = f(q, u)$ to get the resulting state. Fortunately, you do not need to write your numerical integration code. OMPL includes support for numerical integration and you "only" have to implement $f(q, u)$. A detailed example is given at http://ompl.kavrakilab.org/odeint.html.

It is important to construct the correct state space for each of the systems using the `CompoundStateSpace` class, as well as a correct `StateValidityChecker`. For the pendulum system, you can assume an environment without obstacles. The state validity checker, however, must ensure that the angular velocity of the pendulum is within the bounds that you specify. Similarly you must verify that the translational velocity of the vehicle is within the bounds you set. For simplicity, you can assume that the vehicle is a point system for collision checking purposes, but collision checking methods from the previous project can also be used to add geometry to the vehicle if you wish.

See the OMPL demo programs for some examples of how to put all the different pieces together. Look in particular at `${OMPL_DIR}/share/ompl/demos/RigidBodyPlanningWithODESolverAndControls.cpp` (also available online on the "demos" page).

Reachability-guided RRT

An article by Shkolnik et al. (2009) proposes a new motion planning algorithm for dynamic systems. The authors propose that an RRT should only be grown towards a random state that is likely to be *reachable* from its nearest neighbor in the tree. For dynamic systems, states that are near a given state are not always easily reachable; the vehicle system, for instance, cannot easily move sideways. You will use the information in their article (summarized below) to understand the problem and the performance characteristics of the systems they assessed. Next, you will implement the motion planner they describe, the *reachability-guided* RRT (RG-RRT) and test its performance.

The main change of RG-RRT over RRT is that for each state q in the tree, an approximation of the *reachable set* of states, $R(q)$, is maintained. The reachable set $R(q)$ is the set of states the system can arrive at, starting at state q, after applying any valid control(s) for a small period of time. You can approximate $R(q)$ by choosing a small number of controls and computing the states that were reached after applying them for a short duration. For the pendulum you could pick, say, 10 values for τ, evenly spaced between the torque limits. For the car you can do the same for u_0 (you can ignore u_1 for the reachable set). Next, apply each of the controls to the start state q for a small time step to obtain $R(q)$ (use the `SpaceInformation::propagate` method for this).

The next step is to systematically modify the nearest neighbor queries. Given a random state q_{rand}, you need to find the state q_{near} that is closest to q_{rand} **and** has a state in $R(q_{\text{near}})$ that is closer to q_{rand} than q_{near}. An easy strategy is to use the NearestNeighborLinear data structure over Motion objects with a special distance function that you need to write. This distance function returns the distance between two states q_0 and q_1 when q_1 is reachable from q_0 (i.e., there exists a state in $q^r \in R(q_0)$ that is closer to q_1 than q_0 is to q_1) and returns ∞ otherwise. Alternatively, you can also leave the initial nearest neighbor selection the same as in RRT and simply reject random samples until the nearest neighbor can reach it. (Note that these two approaches are not *exactly* equivalent, but you can ignore the difference.)

The recommended way to implement RG-RRT is to start with the RRT implementation from the OMPL.app source distribution. These files are found in `omplapp/ompl/src/ompl/control/planners/rrt` if you compiled from source (or the virtual machine). The files are also found at https://bitbucket.org/ompl/ompl. Copy these to your project directory, rename them to `RG-RRT.{h,cpp}`, and modify them as necessary. You can store the reachable set approximation by extending RRT'ss `Motion` class to also store the reachable set.

Project exercises

1. Implement the state validity checker and differential equations for the pendulum and the vehicle systems described above. Solve the motion planning problems described for these systems using the RRT planner. Visualize the solution paths to make sure they are correct.
2. Extend the program from #1 to solve the pendulum and the vehicle problems using the KPIECE planner. KPIECE (among other planners) requires the definition of a projection for the state space being planned in. The projection is used to estimate how well different parts of the state space have been sampled. For the existing state spaces in OMPL a projection is already defined. You will need to define a projection for the state spaces you create for the pendulum and the car. See http://ompl.kavrakilab.org/projections.html for details on how to define a projection and associate it with a state space.
3. Implement RG-RRT (see Scholnik et al., 2009) and solve the pendulum and vehicle problems as in #1 and #2. Make sure to visualize the solution paths.
4. Compare your RG-RRT against RRT and KPIECE 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 35% of your total project grade. **Submissions are due Tuesday Oct. 28 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 RRT, KPIECE and RG-RRT planners for both systems. This data must be presented in a quantitative manner (i.e., plots or tables) and any conclusions must be supported from this data. Consider the following metrics: computation time, path length, number of states sampled, and success rate.
- A head-to-head comparison of RRT and RG-RRT. Discuss the performance trade-offs of RG-RRT for computing reachable states in terms of the *length* of the time period and the *number* of controls. Support your claims with images and/or benchmark data.
- 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 groups 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

- Solution paths from the *control-based* planners in OMPL contain more information than their simpler, geometric counterpart. You can "geometrize" a controlled path using the `asGeometric` method. This will return a `PathGeometric` object that you can use with your existing visualization tools.
- Make sure to perform state validity checking for the reachable set in your RG-RRT implementation. If a state is not valid, it certainly is not *reachable*.

Reference

A. Shkolnik, M. Walter, and R. Tedrake, Reachability-guided sampling for planning under differential constraints, in *IEEE Intl. Conf. on Robotics and Automation*, pp. 2859–2865, 2009. http://dx.doi.org/10.1109/ROBOT.2009.5152874, http://dspace.mit.edu/openaccess-disseminate/1721.1/61653