Motion Planning RBE 550

Project 4: Out of Control Planning

DUE: Tuesday March 28 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
        car.cpp
        pendulum.cpp
        RG-RRT.cpp
        RG-RRT.h
        CollisionChecking.h
        CollisionChecking.cpp
        build
            benchmark.db
```

Present your work and your work only. You must *explain* all of your answers. Answers without explanation will be given no credit.

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. In this project, you will be planning for more complex systems – systems with dynamical constraints on their motion. For these complex systems, a collision-free path may not always be feasible. As an example, consider a car trying to parallel park. Although the a *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 implies that the car is a *non-holonomic* system; the system cannot control its position directly, and has constraints that include terms other than its position. This is a key point that differentiates planning between geometric and dynamic problems.

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 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.

Theoretical Questions (30 points)

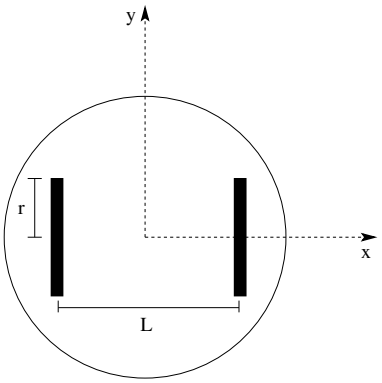

Figure 1: The Indoor Vacuum Robot

1. **(5 points)** What is the difference between asymptotic optimality and asymptotic near-optimality?
2. **(5 points)** If we have a system with non-holonomic constraints can we use RRT* to find an optimal path? Explain your answer.
3. **(20 points)** Figure 1 shows an indoor vacuum robot has a differential drive chassis with two wheels, each with a radius of r, that are separated by a distance L.

   Each wheel is controlled independently with its own motor. Presume that you can specify the torque for each motor, and the motors can directly change the velocities (u_l and u_r) of your Roomba. Then the dynamics of your Roomba can be specified with the following differential equations:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{r}{2}(u_l + u_r)\cos(\theta) \\ \frac{r}{2}(u_l + u_r)\sin(\theta) \\ \frac{r}{L}(u_r - u_l) \end{pmatrix}$$

   (a) **(5 points)** What is the configuration space, the control space, and the state-space of the robot?

   (b) **5 points** Is this a holonomic or non-holonomic robot? explain your answer

   (c) **(10 points)** Given the above equations of motion and using the Euler approximation, compute the next state of the robot if it starts from configuration $(x, y, \theta) = (0, 0, 0)$ and controls $(u_l, u_r) = (1, 0.5)$ are applied for 1 second. Show your work, don't just write a number.

Programming Component (70 points)

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 bar of the pendulum being horizontal. The objective is to move the pendulum from an initial state of hanging down at rest to a final state of pointing up at rest:

$$(\theta_s, \omega_s) = (-\tfrac{\pi}{2}, 0) \qquad \rightarrow \qquad (\theta_g, \omega_g) = (+\tfrac{\pi}{2}, 0)$$

A motor can apply a torque τ to the axis of rotation of the pendulum. The pendulum's dynamics are described by the following differential equation (g is gravity, ≈ 9.81):

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

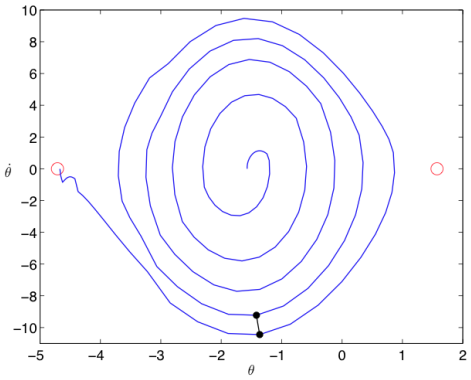

Figure 2: A possible solution trajectory for the pendulum in phase space. *From Shkolnik et al.; see below.*

When an arbitrarily large torque can be applied, the planning problem is not difficult as the torque can hold the system *quasi-statically* at any configuration. Many real manipulators use this strategy and utilize very large torques, allowing them to plan geometrically. However, is this always the case? In reality, the pendulum's motor can source only a finite torque.

In your implementation, use 3, 5, and 10 as the absolute value of the upper and lower bounds on the torque limits (i.e., $\tau \in [-3, 3]$ for a torque bound of 3). Use 10 as absolute value of the rotational velocity limit. Figure 2 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, an example of which is shown in Figure 3. The state of the vehicle is represented by its position (x, y), heading θ, and forward velocity v:

$$q = (x, y, \theta, v)$$

Note that forward velocity *can* be negative, in which case the car moves in reverse. The control inputs to this system u consist of the angular velocity ω and the forward acceleration of the vehicle $\dot{v}$,

$$u = (\omega, \dot{v})$$

Both terms of u are bounded in magnitude. The system dynamics are described by:

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

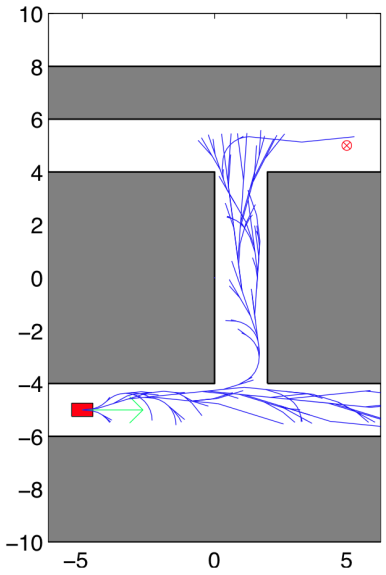

Figure 3: Vehicle navigating in street environment. *From Shkolnik et al.; see below.*

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.

Planning for Dynamical Systems

The key difference when planning for dynamical systems is that the *input space* of controls must be sampled (a *control space*), 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. 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.

You must construct the correct state space for each of the systems, as well as a correct `StateValidityChecker`. For the pendulum system, you can assume an environment without obstacles. However, the state validity checker must ensure that the angular velocity of the pendulum is within the bounds that you specify. Similarly you must verify that the 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.

Reachability-guided RRT

An article by Shkolnik et al. (2009) proposes a 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. For instance, vehicle systems 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 (a fixed value you set). 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 should pick 10 evenly spaced values for τ between the torque limits (i.e., {-10, -8, ..., 8, 10}). 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). You can store the reachable set approximation by extending `Motion` objects (such as those in RRT) to also store the reachable set (as a vector of states). You can assume for this project that the control space is a `RealVectorControlSpace` with bounds. Please make sure you write your assumptions in your report.

The next step is to modify 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 that contains `Motion` objects (such as those in RRT) with a special distance function (as described above) 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.

Projections

KPIECE (among other planners) requires the definition of a projection for the state space being planned in. KPIECE uses the projection 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; however, a good projection often requires some information about the specific robot and task. Although projections are generally a reduction in dimension, projections in OMPL do not have a limit on how many dimensions you wish to define nor do they require that you chose a subset of the dimensions in your state space.

For example, one of the dimensions in your projection could even be the Cartesian position of some point on the robot. Often times, a desirable projection might be one which has a strong correlation with progress toward the goal or indication of greater coverage in the workspace. With control spaces, such a projection usually includes some information from the dynamic dimensions in the state.

Project exercises

Fill a the necessary functions in `car.cpp`, `pendulum.cpp`, `CollisionChecking.cpp`, `CollisionChecking.h` `RG-RRT.cpp` `RG-RRT.h` to:

1. **(20 points)** Implement the state validity checker and differential equations for the pendulum and the vehicle systems described above by Solve the motion planning problems described for these systems using the RRT planner.
   - **(15 points** Visualize the solution paths for car and pendulum (by creating your own script) and attach them in your report to make sure they are correct. Also include a description of the obstacles you used (for the car) and start and goal queries.
   - **(5 points)** Compare the solution paths found using torque values of 3, 5, and 10 for the pendulum problem. Explain the differences in solution paths when torque is limited to 3, 5, and 10 for the pendulum problem.
2. **(10 points)** Extend the program from above to solve the pendulum and the vehicle problems using the KPIECE planner. You will need to define a projection for the state spaces you create for the pendulum and the car. See projections for details on how to define a projection and associate it with a state space.
3. **(30 points)** Implement RG-RRT (see Scholnik et al., 2009) and solve the pendulum and vehicle problems as in. Make sure to visualize the solution paths, and attache them in your report. To implement RG-RRT, we recommended you start with the control RRT implementation from the ompl_control_rrt.
4. **(10 points)** Compare your RG-RRT against RRT and KPIECE using the OMPL `Benchmark` class for both the car and pendulum environments. Use a torque value of 3 for the pendulum. Any conclusions must come from at least 20 independent runs of each planner. Consider the following metrics: computation time, path length, number of tree nodes, and success rate. When referencing benchmarking results you must include the referenced data as figures. 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.

Protips

- Be sure to use the `ompl::control` namespace instead of `ompl::geometric` in this project. This includes the `SimpleSetup` class and all of the planners.

- Solution paths from the *control-based* planners in OMPL contain more information than their 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*.
- It is *highly* unlikely that a control-based planner will be able to find a goal configuration exactly. Think about why this is the case. The call to `SimpleSetup.setStartAndGoalStates()` has a third parameter that defines an acceptable radius around the goal state. Any state that is within this radius will be considered an exact goal state. You will need to play with this parameter to successfully plan.
- Generally, geometric planners (such as RRT and your Random Tree Planner from the last assigment) use `NearestNeighborsGNAT`, which requires a distance function that is a *metric*. However, for RG-RRT, the distance defined by the reachability set (∞ if it cannot be reached) is assymetric which breaks the necessary metric properties to use `NearestNeighborsGNAT`. For this reason, you must use `NearestNeighborLinear` to build a nearest-neighbor structure for RG-RRT.
- In this exercise you will have to generate new states for the reachability set of a state. To do this, you want to apply some control to an existing state. You can allocate controls by allocating new controls from `ompl::control::SpaceInformation` using `allocControl`. Just like with states, you can cast controls to their control type defined in the control space. For example, `RealVectorControlSpace` has `RealVectorControlSpace::ControlType`. Use `ompl::control::SpaceInformation` to generate new states through `propogate`, which applies a control to get a new state.

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. Visualization is an important component of providing evidence that your code is functioning properly. The breakdown of the grading of the implementation is as follows:

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