



Principles of Robot Autonomy I

Problem Set 2

Due Tuesday, October 19 (11:59pm)

Starter code for this problem set has been made available online through Github. To get started, download the code by running `git clone https://github.com/PrinciplesofRobotAutonomy/AA274A_HW2.git` in a terminal window.

You will submit your homework to Gradescope. Your submission will consist of (1) a zip folder containing your code for the programming questions (denoted by the  symbol), and (2) a single pdf with your answers for written questions (denoted by the  symbol), which in this assignment will be a merged pdf printout of your Jupyter notebook with all figures included.


Introduction

The goal of this problem set is to familiarize you with algorithms for path planning in constrained environments (e.g. in the presence of obstacles) and techniques to integrate planning with trajectory generation and control.

Problem 1: A* Motion Planning

To begin, we will implement an A^* algorithm for motion planning, as outlined in pseudocode in Algorithm 1. In particular, we will apply this algorithm to 2D geometric planning problems (state $\mathbf{x} = (x, y)$).

In this implementation, we will represent the free space by a graph, which is traversed by sampling and collision-checking states from a deterministic grid. This implementation can be categorized as informed, deterministic sampling-based planning (informed due to the A^* heuristic).

- (i)  Implement the remaining functions in `P1_astar.py` within the `Astar` class. These functions represent many of the key functional blocks at play in motion planning algorithms:
- `is_free` which checks whether a state is collision-free and valid.
 - `distance` which computes the travel distance between two points.
 - `get_neighbors` which finds the free neighbor states of a given state.
 - `solve` which runs the A^* motion planning algorithm.

Be sure to read the documentation for every function for a more detailed description.

- (ii)  Now let's test this implementation in a couple planning environments. To do so, open the associated Jupyter notebook by running the following command:

```
$ jupyter notebook sim_astar.ipynb
```

Feel free to play with the number of obstacles and other parameters of the randomly generated environment. When you are satisfied with your figures, print the notebook as a pdf to be merged into your written pdf submission.

Algorithm 1 A^* Motion Planning**Require:** $\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}$

```

1:  $\mathcal{O}.\text{INIT}(\mathbf{x}_{\text{init}})$  ▷ Open set initialized with  $\mathbf{x}_{\text{init}}$ 
2:  $\mathcal{C}.\text{INIT}(\emptyset)$  ▷ Closed set is initially empty
3:  $\text{SET\_COST\_TO\_ARRIVE\_SCORE}(\mathbf{x}_{\text{init}}, 0)$ 
4:  $\text{SET\_EST\_COST\_THROUGH}(\mathbf{x}_{\text{init}}, \text{DISTANCE}(\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}))$ 
5: while  $\mathcal{O}.\text{SIZE} > 0$  do
6:    $\mathbf{x}_{\text{current}} \leftarrow \text{LOWEST\_EST\_COST\_THROUGH}(\mathcal{O})$ 
7:   if  $\mathbf{x}_{\text{current}} = \mathbf{x}_{\text{goal}}$  then
8:     return  $\text{RECONSTRUCT\_PATH}$ 
9:   end if
10:   $\mathcal{O}.\text{REMOVE}(\mathbf{x}_{\text{current}})$ 
11:   $\mathcal{C}.\text{ADD}(\mathbf{x}_{\text{current}})$ 
12:  for  $\mathbf{x}_{\text{neigh}}$  in  $\text{NEIGHBORS}(\mathbf{x}_{\text{current}})$  do
13:    if  $\mathbf{x}_{\text{neigh}}$  in  $\mathcal{C}$  then
14:      continue
15:    end if
16:     $\text{tentative\_cost\_to\_arrive} = \text{GET\_COST\_TO\_ARRIVE}(\mathbf{x}_{\text{current}}) + \text{DISTANCE}(\mathbf{x}_{\text{current}}, \mathbf{x}_{\text{neigh}})$ 
17:    if  $\mathbf{x}_{\text{neigh}}$  not in  $\mathcal{O}$  then
18:       $\mathcal{O}.\text{ADD}(\mathbf{x}_{\text{neigh}})$ 
19:    else if  $\text{tentative\_cost\_to\_arrive} > \text{GET\_COST\_TO\_ARRIVE}(\mathbf{x}_{\text{neigh}})$  then
20:      continue
21:    end if
22:     $\text{SET\_CAME\_FROM}(\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{current}})$ 
23:     $\text{SET\_COST\_TO\_ARRIVE}(\mathbf{x}_{\text{neigh}}, \text{tentative\_cost\_to\_arrive})$ 
24:     $\text{SET\_EST\_COST\_THROUGH}(\mathbf{x}_{\text{neigh}}, \text{tentative\_cost\_to\_arrive} + \text{DISTANCE}(\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{goal}}))$ 
25:  end for
26: end while
27: return Failure


```

Note: Notice that we collision-check states but do not collision-check edges. This saves us some computation (collision-checking is often one of the most expensive operations in motion planning). Also, in this case the obstacles are aligned with the grid, so paths will remain collision-free. However, outside such special circumstances one should add edge collision-checking and/or inflate obstacles to guarantee collision-avoidance.

Problem 2: Rapidly-exploring Random Trees (RRT)

While our A^* planning relies on a predefined set of viable samples on the edges of a graph, in some scenarios it is useful to draw samples incrementally and in a less structured fashion. This motivates sampling-based algorithms such as Rapidly-exploring Random Trees (RRT) [1], which we will implement in this problem.

Since vanilla RRT builds its tree by extending from the nodes nearest to random samples, we cannot add the same heuristic as A^* to bias search in the direction of the goal. Instead, we will use a goal-biasing approach, included in the pseudocode in Algorithm 2.

- (i)  Implement RRT for 2D geometric planning problems (state $\mathbf{x} = (x, y)$) by filling in `RRT.solve`, `GeometricRRT.find_nearest`, and `GeometricRRT.steer_towards` in [P2_rrt.py](#).

You can validate your implementations of the parts of this problem in the associated notebook:

```
$ jupyter notebook sim_rrt.ipynb
```


- (ii) You may have noticed that due to the random sampling in RRT, there is plenty of room to optimize the length of the resulting paths. This motivates a variety of post-processing methods which locally

Algorithm 2 RRT [1] with goal biasing.

Require: \mathbf{x}_{init} , \mathbf{x}_{goal} , maximum steering distance $\varepsilon > 0$, iteration limit K , goal bias probability $p \in [0, 1]$

- 1: $\mathcal{T}.\text{INIT}(\mathbf{x}_{\text{init}})$
- 2: **for** $k = 1$ to K **do**
- 3: Sample $z \sim \text{Uniform}([0, 1])$
- 4: **if** $z < p$ **then**
- 5: $\mathbf{x}_{\text{rand}} \leftarrow \mathbf{x}_{\text{goal}}$
- 6: **else**
- 7: $\mathbf{x}_{\text{rand}} \leftarrow \text{RANDOM_STATE}()$
- 8: **end if**
- 9: $\mathbf{x}_{\text{near}} \leftarrow \text{NEAREST_NEIGHBOR}(\mathbf{x}_{\text{rand}}, \mathcal{T})$
- 10: $\mathbf{x}_{\text{new}} \leftarrow \text{STEER_TOWARDS}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{rand}}, \varepsilon)$
- 11: **if** $\text{COLLISION_FREE}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$ **then**
- 12: $\mathcal{T}.\text{ADD_VERTEX}(\mathbf{x}_{\text{new}})$
- 13: $\mathcal{T}.\text{ADD_EDGE}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$
- 14: **if** $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{goal}}$ **then return** $\mathcal{T}.\text{PATH}(\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}})$
- 15: **end if**
- 16: **end if**
- 17: **end for**
- 18: **return** Failure

optimize motion planning paths. As it turns out, even very simple methods can perform quite well on this task. We will implement one of the simplest of these algorithms, which we simply call Shortcut [2].

 Implement the shortcutting algorithm outlined in the pseudocode in Algorithm 3 by filling in [RRT.shortcut_path](#). You can test your implementation in the notebook and should notice that in nearly all cases, Shortcut will be able to refine to a shorter path.

Note: Post-processing algorithms such as this are performing a *local* optimization, which means the result may be far from a globally optimal path. For example in this case, shortcutting is not likely to move the path to the other side of an obstacle (i.e. to a different solution homotopy class), even if this would result in lower path length. This motivates the use of asymptotically optimal varieties of sampling-based planners such as RRT*, which perform a *global search* and are thus guaranteed to approach the globally optimal solution.

- (iii) While geometric RRT does rapidly generate collision-free paths, these paths are not ideal candidates to track with our wheeled robot as the paths' sharp corners would require stopping and turning at most nodes.

One way to resolve this issue is through kinodynamic motion planning, where kinematically- and dynamically-feasible trajectories are built directly in the planner by concatenating subtrajectories. There are two ways to do this, which each have potential drawbacks. The first is control sampling, which (a) requires defining a sensible control subtrajectory sampling strategy, (b) can require a large number of samples when the control-sampling scheme does not provide effective state sampling, and (c) can struggle to reach a precise goal state or small goal region. The second is state sampling, which requires a steering function, i.e. connecting to sampled states by solving a 2P-BVP. Unfortunately, these 2P-BVP problems must be solved many times — for each candidate sub-trajectory and possibly to measure distance while finding nearest neighbors. In our case, a candidate steering method could be solving the optimal control problem in the Extra Problem of HW 1. However, this runs far too slowly to be practical. A more reasonable steering candidate would be the differential flatness approach of Problem 1 in HW 1. However, this may still require careful implementation to run efficiently, since with kinodynamic planning we must add orientation and velocities to the state, tripling the problem dimension.


Rather than exploring this further here, we will instead experiment with a simpler dynamics model where speed is fixed, i.e. the Dubins car. This will allow us to leverage as a steering function the analytical solution that exists for Dubins car shortest paths.

Algorithm 3 Shortcut (deterministic)**Require:** $\Pi_{\text{path}} = (\mathbf{x}_{\text{init}}, \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{\text{goal}})$

```

1: SUCCESS = False
2: while not SUCCESS do
3:   SUCCESS = True
4:   for  $\mathbf{x}$  in  $\Pi_{\text{path}}$  where  $\mathbf{x} \neq \mathbf{x}_{\text{init}}$  and  $\mathbf{x} \neq \mathbf{x}_{\text{goal}}$  do
5:     if COLLISION_FREE(PARENT( $\mathbf{x}$ ), CHILD( $\mathbf{x}$ )) then
6:        $\Pi_{\text{path}}$ .REMOVE_NODE( $\mathbf{x}$ )
7:       SUCCESS = False
8:     end if
9:   end for
10: end while

```

 Implement RRT for Dubins car planning problems (state $\mathbf{x} = (x, y, \theta)$, see [3] for steering connection details) by filling in `DubinsRRT.find_nearest` and `DubinsRRT.steer_towards` in `P2_rrt.py`. Computing steering solutions in this code relies on installing the following python package:


```
$ pip3 install dubins
```

See <https://github.com/AndrewWalker/pydubins/blob/master/dubins/dubins.pyx> for usage details.

Installation notes for dubins:


If you are prompted to install Microsoft Visual C++ Build Tools, you can find them here <https://visualstudio.microsoft.com/visual-cpp-build-tools/>. However, we **highly** recommend using MacOS or Linux to complete this part of the problem since installation has proven difficult on Windows. Farmshare and genbu are good resources to access a Linux environment. Another option is to use [Google Colab](#) to run your jupyter notebook. In Colab, you can install dubins using

```
!pip3 install dubins
```

- (iv)  Test all of the above in your notebook, generate the associated figures, and export the notebook as a pdf to be included in your written submission.

Problem 3: Motion Planning & Control


In this problem, we will finally bridge the gap between our planned paths and trackable trajectories for our differential drive robot. Currently our geometric paths have no time derivatives associated with them to constitute trajectories, and our Dubins paths are based on the assumption of a fixed velocity, which is an undue constraint on our robot. As mentioned previously, we can use kinodynamic motion planning to directly plan trajectories which meet our robot's kinematic and dynamics constraints, but this can be tricky to do efficiently. Instead, we will use another simple post-processing technique for our planned paths and leverage our controllers from Problem Set 1 to generate and follow corresponding trajectories.

- (i)  Smooth the paths from A* by fitting a cubic spline to the path nodes. This will be implemented within the `compute_smoothed_traj` function of `P3_traj_planning.py`, and you may need to use the `splrep` and `splev` functions from `scipy.interpolate` (read through their documentation). The output of this function will include a trajectory $[x, y, \theta, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}]$ from $t = 0$ to $t = t_f$. Since all we have is a geometric path, you should estimate the time for each of the points assuming that we travel at a fixed speed V_{des} along each segment. Adjust the smoothing parameter α (denoted s in `splrep`) to strike a balance between following the original collision-free trajectory and risking collision for additional smoothness.

You can validate your implementations of the parts of this problem in the associated notebook:


```
$ jupyter notebook sim_traj_planning.ipynb
```

Note: There are many ways to ensure smoothed solutions are collision-free (e.g. collision-checking smoothed paths and running a dichotomic search on α to find a tight fit against obstacles, or inflating obstacles in the original planning to give additional room for smoothing). This strategy can be used on geometric sampling-based planning methods as well.


- (ii)  Now we will begin leveraging our modules from Problem Set 1. Replace the scripts in the [HW1](#) folder with your own implementations so that they can be imported by [P3_traj_planning.py](#).

Next, fill in the function `modify_traj_with_limits` to generate control-feasible trajectories using the time-scaling strategy and differential flatness, just as in in HW 1, Problem 1. You can then step through the notebook to track these trajectories using the trajectory-tracking controller from HW 1, Problem 3.

- (iii) It may be tempting to stop here. However, run the notebook further to zoom in on the endpoint of the trajectory. You will notice that the endpoint of the actual trajectory does not match the final desired pose, due to a combination of both controller tracking error and potentially smoothing of the original path. Even if we did get lucky and the robot landed close to the desired endpoint, the tracked trajectory may have a nonzero velocity at its endpoint, quickly moving it off the goal. And in any case, ongoing disturbances would certainly cause the robot to drift off the goal.

 The solution to this is to switch from trajectory tracking to our pose stabilization controller from HW 1, Problem 2 as the robot approaches the end of the trajectory. To do this, fill in the function `SwitchingController.compute_control` to perform this controller switch some number of seconds before reaching the goal.

In the notebook, we can continue the simulation for some extra time past the nominal final time to make sure the robot reaches and stays near the final pose. Adjust the control gains on the pose controller until you are satisfied with the final pose stabilization.

- (iv)  Finish testing all of the above in your notebook, generate the associated figures, and export the notebook as a pdf to be included in your written submission.

Extra Problem: Parallel Parking with RRT

Extra problem is required for students taking AA 274A/CS 237A/EE 260A for 4 credits


In problem 2 we considered the extension of Rapidly-exploring Random Trees (RRTs) to the case of kinodynamic constraints, specifically modeling the robot as a Dubins car. In this problem we will consider another practical extension – to the case of nontrivial robot geometry.

As discussed in lecture, one way to plan safe trajectories for a mobile robot with a nontrivial footprint is to consider a sphere of radius r that encloses the robot’s body and to solve the planning problem with all obstacles inflated by r , still treating the robot as a point. This strategy does ensure safety, but it also engenders a potentially undesirable degree of conservatism which may make truly feasible problems appear infeasible (consider, e.g., a particularly oblong robot for which the enclosing “safety bubble” would contain mostly free space, or the “alpha” puzzle¹ for which even the initial configuration would seem unsafe). Fortunately, sampling-based motion planning algorithms (e.g., RRT) are designed to accommodate nontrivial collision geometry through the black-box abstraction afforded by methods like `COLLISION_FREE`.

In this question we will consider the problem of planning a parallel parking maneuver for a simple car. Since parallel parking typically requires reversing in addition to driving forward, we will consider a Reeds-Shepp

¹See <https://parasollab.web.illinois.edu/resources/mpbenchmarks/data/mp/Alpha/> for details on this classic motion planning benchmark problem.

car model. This model is similar to the Dubins car in that the magnitude of its speed is fixed but in this case it may be applied forward or backward. Also like the Dubins car, an analytical shortest-path steering solution exists for the Reeds-Shepp car (see [4] for details).

- (i)  Implement the necessary methods of ParkingRRT in `P4_parallel_parking.py` to account for both
- (i) Reeds-Shepp steering (as opposed to geometric, i.e., straight-line, steering or Dubins steering) and
 - (ii) collision checking for the car as a rigid body (as opposed to a point). Computing steering solutions in this code relies on installing the following python package:

```
$ pip3 install reeds-shepp
```

See <https://github.com/ghliu/pyReedsShepp> for usage details.²

Installation notes for `reeds-shepp`:

If you are prompted to install Microsoft Visual C++ Build Tools, you can find them here <https://visualstudio.microsoft.com/visual-cpp-build-tools/>. However, we **highly** recommend using MacOS or Linux to complete this part of the problem since installation has proven difficult on Windows. Farmshare and genbu are good resources to access a Linux environment. Another option is to use [Google Colab](#) to run your jupyter notebook. In Colab, you can install `reeds-shepp` using

```
!pip3 install reeds-shepp
```

- (ii)  Test your implementation in the parking scenario provided in the following notebook:

```
$ jupyter notebook sim_parallel_parking.ipynb
```

Generate the associated figure and export the notebook as a pdf to be included in your written submission.

²Note that `reeds_shepp.path.sample` actually returns tuples of 5 numbers not 3 as written; the last two numbers encode controls (corresponding to whether the car is turning left/straight/right or moving forward/backward). Dealing with somewhat half-baked code/impartially documented code is one of the realities of being a roboticist, and arguably one of the learning objectives of this class. Contributing back to the community, e.g., making a pull request to update the `reeds_shepp` API to be in line with the new `dubins` API (<https://github.com/AndrewWalker/pydubins/pull/8>) would be a cool mini-project!

References

- [1] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” Illinois State University, Tech. Rep., 1998.
- [2] R. Geraerts and M. Overmars, “Creating high-quality paths for motion planning,” *Int. Journal of Robotics Research*, vol. 26, no. 8, pp. 845–863, 2007.
- [3] L. E. Dubins, “On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents,” *American Journal of Mathematics*, vol. 79, pp. 497–516, 1957.
- [4] J. Reeds and L. Shepp, “Optimal paths for a car that goes both forwards and backwards,” *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990.