



Principles of Robotic Autonomy I

Problem Set 2

Due Friday, October 18 (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/Course1_Fall19_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 implicitly represent the free space by a graph, also implicitly defined, which is traversed by sampling and collision-checking states from a deterministic grid. Thus 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}_{init} , \mathbf{x}_{goal}

```

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

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:  $\mathcal{T}.\text{INIT}(\mathbf{x}_{init})$ 
2: for  $k = 1$  to  $K$  do
3:   Sample  $z \sim \text{Uniform}([0, 1])$ 
4:   if  $z < p$  then
5:      $\mathbf{x}_{rand} \leftarrow \mathbf{x}_{goal}$ 
6:   else
7:      $\mathbf{x}_{rand} \leftarrow \text{RANDOM\_STATE}()$ 
8:   end if
9:    $\mathbf{x}_{near} \leftarrow \text{NEAREST\_NEIGHBOR}(\mathbf{x}_{rand}, \mathcal{T})$ 
10:   $\mathbf{x}_{new} \leftarrow \text{STEER\_TOWARDS}(\mathbf{x}_{near}, \mathbf{x}_{rand}, \varepsilon)$ 
11:  if  $\text{COLLISION\_FREE}(\mathbf{x}_{near}, \mathbf{x}_{new})$  then
12:     $\mathcal{T}.\text{ADD\_VERTEX}(\mathbf{x}_{new})$ 
13:     $\mathcal{T}.\text{ADD\_EDGE}(\mathbf{x}_{near}, \mathbf{x}_{new})$ 
14:    if  $\mathbf{x}_{new} = \mathbf{x}_{goal}$  then return  $\mathcal{T}.\text{PATH}(\mathbf{x}_{init}, \mathbf{x}_{goal})$ 
15:    end if
16:  end if
17: end for
18: return Failure

```

the length of the resulting paths. This motivates a variety of post-processing methods which locally 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 velocity is fixed, i.e. the Dubins car. This will allow us to leverage as a steering function the

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

```

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

```

analytical solution that exists for Dubins car shortest paths.



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:

```
$ pip install dubins==0.9.2
```

See <https://github.com/AndrewWalker/pydubins/blob/0.9.2/dubins/dubins.pyx> for usage details.

- (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: Geometric Planning to Trajectories and 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 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: Bi-Directional Sampling-based Motion Planning

You may have noticed that it occasionally takes a large number of samples for RRT to reach the goal. Unfortunately, there are many seemingly simple planning scenarios which make this problem much worse. For example, consider the “bug trap” scenario in the following notebook:

```
$ jupyter notebook sim_bidirectional_rrt.ipynb
```

One way to address this issue and improve sample efficiency is through bi-directional sampling-based motion planning. To explore this, in this problem we will implement a very widely-used bi-directional variant of RRT called RRT-Connect [4], using the pseudocode outlined in Algorithm 4. You will notice that this is more detailed than the pseudocode in [4], as we are explicitly accounting for the difference in handling required for the forward-reaching tree \mathcal{T}_{fwd} and the backward-reaching tree \mathcal{T}_{bwd} , which is important for many non-geometric planning problems such as Dubins car planning.

- (i)  Implement RRT-Connect for 2D geometric planning problems by filling in `RRTConnect.solve` and the `GeometricRRTConnect` class functions `find_nearest_forward` and `steer_towards_forward` in `P4_bidirectional_rrt.py`. Notice that in the geometric planning case, the forward and backward versions of `find_nearest` and `steer_towards` are equivalent.
- (ii)  Implement RRT-Connect for Dubins car planning problems by filling the `DubinsRRTConnect` class functions `find_nearest_forward`, `find_nearest_backward`, `steer_towards_forward`, and `steer_towards_backward` in `P4_bidirectional_rrt.py`. You may also find the function `DubinsRRTConnect.reverse_heading` helpful.
- (iii)  Test the above in your notebook, generate the associated figures, and export the notebook as a pdf to be included in your written submission.

Algorithm 4 RRT-Connect [4]**Require:** \mathbf{x}_{init} , \mathbf{x}_{goal} , maximum steering distance $\varepsilon > 0$, iteration limit K

```

1:  $\mathcal{T}_{fwd}.\text{INIT}(\mathbf{x}_{init})$ 
2:  $\mathcal{T}_{bwd}.\text{INIT}(\mathbf{x}_{goal})$ 
3: for  $k = 1$  to  $K$  do
4:    $\mathbf{x}_{rand} \leftarrow \text{RANDOM\_STATE}()$ 
5:    $\mathbf{x}_{near} \leftarrow \text{NEAREST\_FORWARD\_NEIGHBOR}(\mathbf{x}_{rand}, \mathcal{T}_{fwd})$ 
6:    $\mathbf{x}_{new} \leftarrow \text{STEER\_FORWARD\_TOWARDS}(\mathbf{x}_{near}, \mathbf{x}_{rand}, \varepsilon)$ 
7:   if  $\text{COLLISION\_FREE}(\mathbf{x}_{near}, \mathbf{x}_{new})$  then
8:      $\mathcal{T}_{fwd}.\text{ADD\_VERTEX}(\mathbf{x}_{new})$ 
9:      $\mathcal{T}_{fwd}.\text{ADD\_EDGE}(\mathbf{x}_{near}, \mathbf{x}_{new})$ 
10:     $\mathbf{x}_{connect} \leftarrow \text{NEAREST\_BACKWARD\_NEIGHBOR}(\mathbf{x}_{new}, \mathcal{T}_{bwd})$ 
11:    while True do
12:       $\mathbf{x}_{newconnect} \leftarrow \text{STEER\_BACKWARD\_TOWARDS}(\mathbf{x}_{new}, \mathbf{x}_{connect}, \varepsilon)$ 
13:      if  $\text{COLLISION\_FREE}(\mathbf{x}_{newconnect}, \mathbf{x}_{connect})$  then
14:         $\mathcal{T}_{bwd}.\text{ADD\_VERTEX}(\mathbf{x}_{newconnect})$ 
15:         $\mathcal{T}_{bwd}.\text{ADD\_EDGE}(\mathbf{x}_{newconnect}, \mathbf{x}_{connect})$ 
16:        if  $\mathbf{x}_{newconnect} = \mathbf{x}_{new}$  then
17:          return  $\text{RECONSTRUCT\_PATH}$ 
18:        end if
19:         $\mathbf{x}_{connect} \leftarrow \mathbf{x}_{newconnect}$ 
20:      else
21:        break
22:      end if
23:    end while
24:  end if
25:   $\mathbf{x}_{rand} \leftarrow \text{RANDOM\_STATE}()$ 
26:   $\mathbf{x}_{near} \leftarrow \text{NEAREST\_BACKWARD\_NEIGHBOR}(\mathbf{x}_{rand}, \mathcal{T}_{bwd})$ 
27:   $\mathbf{x}_{new} \leftarrow \text{STEER\_BACKWARD\_TOWARDS}(\mathbf{x}_{near}, \mathbf{x}_{rand}, \varepsilon)$ 
28:  if  $\text{COLLISION\_FREE}(\mathbf{x}_{new}, \mathbf{x}_{near})$  then
29:     $\mathcal{T}_{bwd}.\text{ADD\_VERTEX}(\mathbf{x}_{new})$ 
30:     $\mathcal{T}_{bwd}.\text{ADD\_EDGE}(\mathbf{x}_{new}, \mathbf{x}_{near})$ 
31:     $\mathbf{x}_{connect} \leftarrow \text{NEAREST\_FORWARD\_NEIGHBOR}(\mathbf{x}_{new}, \mathcal{T}_{fwd})$ 
32:    while True do
33:       $\mathbf{x}_{newconnect} \leftarrow \text{STEER\_FORWARD\_TOWARDS}(\mathbf{x}_{connect}, \mathbf{x}_{new}, \varepsilon)$ 
34:      if  $\text{COLLISION\_FREE}(\mathbf{x}_{connect}, \mathbf{x}_{newconnect})$  then
35:         $\mathcal{T}_{fwd}.\text{ADD\_VERTEX}(\mathbf{x}_{newconnect})$ 
36:         $\mathcal{T}_{fwd}.\text{ADD\_EDGE}(\mathbf{x}_{connect}, \mathbf{x}_{newconnect})$ 
37:        if  $\mathbf{x}_{newconnect} = \mathbf{x}_{new}$  then
38:          return  $\text{RECONSTRUCT\_PATH}$ 
39:        end if
40:         $\mathbf{x}_{connect} \leftarrow \mathbf{x}_{newconnect}$ 
41:      else
42:        break
43:      end if
44:    end while
45:  end if
46: end for
47: return Failure

```

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*.
- [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. J. Kuffner and S. M. LaValle, “RRT-Connect: an efficient approach to single-query path planning,” in *Proc. IEEE Conf. on Robotics and Automation*, 2000.