Principles of Robot Autonomy I
Problem Set 2
Due Monday, October 20 (11:59pm)

Starter code for this homework has been made available online through GitHub. To get started, download the code by running `git clone https://github.com/StanfordASL/AA174a-HW2.git` in a terminal window.

You will submit your homework to Gradescope. Your submission will consist of a single pdf with your answers for written questions (denoted by the ✎ symbol), which in this assignment will include a 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: Tracking LQR

This problem needs to be completed in a colab notebook, linked here. **You will need to login through your Stanford email to access the notebook.** Please email the CA team or post a question on Ed if you have trouble accessing it. Go to File → "Save a copy in Drive" to create your own editable copy of the notebook.

You will need to follow along Parts 1 through 3 in the notebook and write code in Part 4. Once you are done with the colab notebook, return to this overleaf to complete the following written questions:

(i) 🖥 Include a screenshot of the final state of your quadrotor from the visualization output at the end of Problem 4.

(ii) ✎ Why does there exist some "steady state error" at the end of the trajectory?

(iii) ✎ If we found ourselves running up against control limits (i.e., activating the np.clip in the cell above; this shouldn't be the case with the numbers given in this problem as written), what could we change in (a) the tracking LQR formulation, or (b) the computation of the nominal trajectory, to make this less likely to happen?

(iv) ✎ Even with closed-loop control, we see that the red "safety bubble" surrounding the quad intersects the obstacle over a short time interval. What could we do to avoid this?

Problem 2: A* Motion Planning & Path Smoothing

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 $\mathbf{x} = (x, y)$).

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: SET_COST_TO_ARRIVE_SCORE($\mathbf{x}_{\text{init}}, 0$)
4: 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** 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** NEIGHBORS($\mathbf{x}_{\text{current}}$) **do**
13:        **if** $\mathbf{x}_{\text{neigh}}$ **in** $\mathcal{C}$ **then**
14:            **continue**
15:        **end if**
16:        tentative_cost_to_arrive = GET_COST_TO_ARRIVE($\mathbf{x}_{\text{current}}$) + 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** tentative_cost_to_arrive > GET_COST_TO_ARRIVE($\mathbf{x}_{\text{neigh}}$) **then**
20:            **continue**
21:        **end if**
22:        SET_CAME_FROM($\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{current}}$)
23:        SET_COST_TO_ARRIVE($\mathbf{x}_{\text{neigh}}$, tentative_cost_to_arrive)
24:        SET_EST_COST_THROUGH($\mathbf{x}_{\text{neigh}}$, tentative_cost_to_arrive + DISTANCE($\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{goal}}$))
25:    **end for**
26: **end while**
27: **return** Failure

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).

Note: Execute in your VM environment using the system python, as we'll leverage functions from `asl_tb3_lib`. Ensure Jupyter is installed (if not, run `sudo apt install jupyter`).

(i) 🖥 Implement the remaining functions in `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. You can 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
```

Please include the plot from the "Simple Environment" section of the notebook in your write-up. In the "Random Cluttered Environment" section, feel free to play with the number of obstacles and other parameters of the randomly generated environment.

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.

(ii) 🖥 In the final segment of Problem 1, we transition from the geometric paths obtained from the A* algorithm to generating feasible trajectories for our differential drive robot.

Smooth the paths from A* by fitting a cubic spline to the path nodes. Implement this within the `compute_smooth_plan` function of `sim_astar.ipynb`. You may need to use the `splrep` function from `scipy.interpolate` (read through the documentation to understand its usage and parameters).

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. Compute the cumulative time along the path waypoints and use it for spline fitting.

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.

Please include the plot generated in the "Smooth Trajectory" section of the notebook in your write-up.

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.

[Section Prep]: ROS2 Navigation Node

Note: This portion of the homework is **not graded**, but should be completed before Section on Week 5 to test in hardware.

Objective: Implement a Path Planning and Trajectory Tracking Node in ROS2 using A* Algorithm and Spline Interpolation

Import note: all the URLs are highlighted in blue. Make sure you click into them as they are important references and documentation!

In this assignment, you are tasked with developing a ROS2 node in Python that utilizes the A* algorithm for path planning and spline interpolation for trajectory generation and tracking for a TurtleBot3 robot. The node will be implemented using the `rclpy` library and will interact with custom messages and utility functions provided in the `asl_tb3_lib` and `asl_tb3_msgs` packages. You will be leveraging your implementations of A* and path smoothing from Problem 1, as well as your differential flatness tracking controller from HW1 (Problem 2).

First, take a brief look at the `navigation.py` from `asl_tb3_lib`. Specifically, you will be implementing the functions `compute_heading_control`, `compute_trajectory_tracking_control`, and `compute_trajectory_plan`. In this file, you can also find the definition of the `TrajectoryPlan` class.

Unlike HW1, you will build your navigation node from scratch for this homework. However, feel free to use the given code for HW1 as a reference.

Implement the Navigation Node

Step 1 – Create a new node. You can use the same autonomy workspace from HW1. In it, make a new script at `~/autonomy_ws/src/autonomy_repo/scripts/navigator.py`. Write the necessary code to create your own navigator node class by inheriting from `BaseNavigator`.

Hints:

1. Some examples for importing from `asl_tb3_lib`,

   > **from** asl_tb3_lib.navigation **import** BaseNavigator
   > **from** asl_tb3_lib.math_utils **import** wrap_angle
   > **from** asl_tb3_lib.tf_utils **import** quaternion_to_yaw

2. Use HW1, section, or this minimal node example as references on how to write the basic structure of a Python ROS2 node.

3. Make sure this script is a proper executable file (i.e. shebang + executable permission).

4. Register your new node in `CMakeLists.txt` at the root of your ROS2 package. See for example here.

Step 2 – Implement / Override `compute_heading_control`. This should be identical to the function `compute_control_with_goal` from `heading_controller.py` in HW1. You may also want to add gain initialization to the `__init__` constructor.

Step 3 – Implement / Override `compute_trajectory_tracking_control`. Migrate and re-structure the `compute_control` function in `P2_trajectory_tracking.py` from HW1. This is not as straightforward as step 2. Use the following hints as a guide:

1. Make sure to understand the data structures `TurtleBotControl` and `TrajectoryPlan`.

2. The desired states `x_d, xd_d, xdd_d, y_d, yd_d, ydd_d` need to be computed differently. Use `scipy.interpolate.splev` to sample from the spline parameters given by the `TrajectoryPlan` argument.

3. The variable initialization in the constructor (`__init__`) function also needs to be migrated. Constants like `V_PREV_THRESH` also needs to be moved into the constructor.

4. The control limit can be removed since the base navigator class has its built-in clipping logic to prevent generating unreasonably large control targets.

Step 4 – Implement / Override `compute_trajectory_plan`. You will borrow / migrate code from the A* problem (HW2). You don't need to implement additional logic in this question, but you will need solid understanding on all the code from Problem 2 in order to move things into the right places. The pseudo code for this function is detailed in Algorithm 2. Here are some hints for implementing each step of the algorithm:

1. Make sure you understand everything about the `AStar` class. The easiest way to implement this step is to copy the entire class into your navigator node, and directly use it in the `compute_trajectory_plan` method. See the notebook `sim_astar.ipynb` for examples on how to

   (a) construct an `AStar` problem
   (b) solve the problem
   (c) access the solution path

2. See `sim_astar.ipynb` for examples on how to check if a solution exists.

5. The `compute_trajectory_tracking_control` method uses some class properties to keep track of the ODE integration states. What are those variables? How should we reset them when a new plan is generated?

6. See `compute_smooth_plan` function from `sim_astar.ipynb`.

7. See `compute_smooth_plan` function from `sim_astar.ipynb`.

8. See the block below `compute_smooth_plan` on how to construct a `TrajectoryPlan`.

Algorithm 2 Compute Trajectory Plan

Require: `state`, `goal`, `occupancy`, `resolution`, `horizon`
 1: Initialize A* problem using `horizon`, `state`, `goal`, `occupancy`, and `resolution`   ▷ A* Path Planning
 2: **if** A* problem is not solvable **or** length of path < 4 **then**
 3:     **return** None
 4: **end if**
 5: Reset class variables for previous velocity and time   ▷ Reset Tracking Controller History
 6: Compute planned time stamps using constant velocity heuristics   ▷ Path Time Computation
 7: Generate cubic spline paramteres   ▷ Trajectory Smoothing
 8: **return** a new `TrajectoryPlan` including the path, spline parameters, and total duration of the path

---

Create the Launch File

Create a launch file at `~/autonomy_ws/src/autonomy_repo/launch/navigator.launch.py`. The launch file needs to

1. Declare a launch argument `use_sim_time` and make it defaults to `"true"`.

2. Launch the following nodes

    (a) Node `rviz_goal_relay.py` from package `asl_tb3_lib`. Set parameter `output_channel` to `/cmd_nav`.

    (b) Node `state_publisher.py` from package `asl_tb3_lib`.

    (c) Node `navigator.py` from package `autonomy_repo` (This is your navigator node!). Set parameter `use_sim_time` to the launch argument defined above.

3. Launch an existing launch file `rviz.launch.py` package `asl_tb3_sim` with the following launch arguments

    (a) Set `config` to the path of your `default.rviz`.

    (b) Set `use_sim_time` to the launch argument defined above.

Hint: take a look at `heading_control.launch.py` provided from HW1. You may copy the entire file over and make some really small changes to satisfy the requirements above. These requirements are mostly just descriptions of what the previously provided launch file is doing.