Principles of Robot Autonomy I
Homework 1
Due Thursday, October 10 (5pm PT)

Several different software tools will be utilized throughout the course. To complete this assignment, make sure you have the following tools installed on your computer:

1. Git: a version control system for software development, an essential tool for software collaboration.
2. Python version 3.5+
3. Jupyter Notebook: A web application for interactive code development and prototyping.

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/PrinciplesofRobotAutonomy/AA274a-HW1-F24` 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 and relevant plots from code.

Your submission must be typeset in LaTeX.

Introduction

The goal of this homework 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.

Note that this homework represents the start of an incremental journey to build our robot autonomy stack. There is a mix of code that is to be implemented in Python and Jupyter Notebooks, and some ROS2 integrated code that will need to run with Gazebo.

Problem 1: A* Motion Planning & Path Smoothing

We will implement an A^* algorithm for motion planning, as outlined in pseudocode in Algorithm 1. In particular, we will apply this algorithm to plan discrete paths on a 2D grid (state $\mathbf{x} = (x, y)$).

Note: Execute in your Linux 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 `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.

Algorithm 1 A^* Motion Planning

Require: $\mathbf{x}_{\text{init}}$, $\mathbf{x}_{\text{goal}}$

$\mathcal{O}$.INIT($\mathbf{x}_{\text{init}}$) ▷ Open set initialized with $\mathbf{x}_{\text{init}}$
$\mathcal{C}$.INIT($\emptyset$) ▷ Closed set is initially empty
SET_COST_TO_ARRIVE_SCORE($\mathbf{x}_{\text{init}}, 0$)
SET_EST_COST_THROUGH($\mathbf{x}_{\text{init}}$, DISTANCE($\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}$))
while $\mathcal{O}$.SIZE > 0 **do**
    $\mathbf{x}_{\text{current}} \leftarrow$ LOWEST_EST_COST_THROUGH($\mathcal{O}$)
    if $\mathbf{x}_{\text{current}} = \mathbf{x}_{\text{goal}}$ **then**
        return RECONSTRUCT_PATH
    end if
    $\mathcal{O}$.REMOVE($\mathbf{x}_{\text{current}}$)
    $\mathcal{C}$.ADD($\mathbf{x}_{\text{current}}$)
    for $\mathbf{x}_{\text{neigh}}$ **in** NEIGHBORS($\mathbf{x}_{\text{current}}$) **do**
        if $\mathbf{x}_{\text{neigh}}$ **in** $\mathcal{C}$ **then**
            continue
        end if
        tentative_cost_to_arrive = GET_COST_TO_ARRIVE($\mathbf{x}_{\text{current}}$)+DISTANCE($\mathbf{x}_{\text{current}}, \mathbf{x}_{\text{neigh}}$)
        if $\mathbf{x}_{\text{neigh}}$ **not in** $\mathcal{O}$ **then**
            $\mathcal{O}$.ADD($\mathbf{x}_{\text{neigh}}$)
        else if tentative_cost_to_arrive $>$ GET_COST_TO_ARRIVE($\mathbf{x}_{\text{neigh}}$) **then**
            continue
        end if
        SET_CAME_FROM($\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{current}}$)
        SET_COST_TO_ARRIVE($\mathbf{x}_{\text{neigh}}$, tentative_cost_to_arrive)
        SET_EST_COST_THROUGH($\mathbf{x}_{\text{neigh}}$, tentative_cost_to_arrive + DISTANCE($\mathbf{x}_{\text{neigh}}, \mathbf{x}_{\text{goal}}$))
    end for
end while
return Failure

- `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 of 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).

Algorithm 2 RRT with goal biasing.

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

$\mathcal{T}$.INIT($\mathbf{x}_{\text{init}}$)
for $k = 1$ to K **do**
  Sample $z \sim$ Uniform([0, 1])
  if $z < p$ **then**
    $\mathbf{x}_{\text{rand}} \leftarrow \mathbf{x}_{\text{goal}}$
  else
    $\mathbf{x}_{\text{rand}} \leftarrow$ RANDOM_STATE()
  end if
  $\mathbf{x}_{\text{near}} \leftarrow$ NEAREST_NEIGHBOR($\mathbf{x}_{\text{rand}}, \mathcal{T}$)
  $\mathbf{x}_{\text{new}} \leftarrow$ STEER_TOWARDS($\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{rand}}, \varepsilon$)
  if COLLISION_FREE($\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}}$) **then**
    $\mathcal{T}$.ADD_VERTEX($\mathbf{x}_{\text{new}}$)
    $\mathcal{T}$.ADD_EDGE($\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}}$)
    if $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{goal}}$ **then return** $\mathcal{T}$.PATH($\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}$)
    end if
  end if
end for
return Failure

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.

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), 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
```

Please include the generated plot from the "Geometric Planning" section of the notebook in your write-up.

Algorithm 3 Shortcut (deterministic)

Require: $\Pi_{\text{path}} = (\mathbf{x}_{\text{init}}, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{\text{goal}})$
SUCCESS = False
while **not** SUCCESS **do**
    SUCCESS = True
    for $\mathbf{x}$ **in** Π_{path} where $\mathbf{x} \neq \mathbf{x}_{\text{init}}$ and $\mathbf{x} \neq \mathbf{x}_{\text{goal}}$ **do**
        if COLLISION_FREE(PARENT($\mathbf{x}$), CHILD($\mathbf{x}$)) **then**
            Π_{path}.REMOVE_NODE($\mathbf{x}$)
            SUCCESS = False
        end if
    end for
end while

(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 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

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.

Please include the generated plot from the “Adding shortcutting" section of the notebook in your write-up.

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.

These techniques will be core components of the trajectory generation and control modules of our autonomy stack. However, you may notice that so far our trajectory generation neglects key constraints such as the presence of obstacles. Thus, in later problems, we will integrate these components with motion planning algorithms to find and track feasible trajectories while avoiding obstacles.

Problem 3: Trajectory Optimization

Let’s revisit the problem of designing a dynamically feasible trajectory. In Problem 2, we were only interested in finding a trajectory that connected the starting state to the goal state in a dynamically feasible manner. However, there are often many such dynamically feasible trajectories, and we might prefer some to others. In this problem, we will utilize tools from optimal control to design a trajectory that explicitly optimizes a given objective.

Consider the kinematic model of the unicycle given in (1).

$$\begin{aligned} \dot{x}(t) &= v(t)\cos(\theta(t)), \\ \dot{y}(t) &= v(t)\sin(\theta(t)), \\ \dot{\theta}(t) &= \omega(t). \end{aligned} \tag{1}$$

where v is the linear velocity and ω is the angular velocity. Suppose the objective is to drive from one waypoint to the next waypoint with minimum time and energy, i.e., we want to minimize the functional

$$J = \int_0^{t_f} \left[\alpha + v(t)^2 + \omega(t)^2\right] dt,$$

where $\alpha \in \mathbb{R}_{\geq 0}$ is a weighting factor and t_f is unbounded. We'll use the following initial and final conditions.

$$x(0) = 0, \quad y(0) = 0, \quad \theta(0) = \pi/2,$$
$$x(t_f) = 5, \quad y(t_f) = 5, \quad \theta(t_f) = \pi/2.$$

In addition, we consider an object in the environment that the robot must avoid. We formulate this collision avoidance as follows:

$$\sqrt{(x(t) - x_{obstacle})^2 + (y(t) - y_{obstacle}(t))^2} - (r_{ego} + r_{obstacle}) \geq 0,$$

where r_{ego} is the radius of the robot we control, and $r_{obstacle}$ is the radius of the obstacle to avoid. In our optimized trajectory, we want to make sure that each state in the trajectory at time t does not violate this constraint.

In this problem, we use:

$$x_{obstacle} = 2.5, \quad y_{obstacle} = 2.5, \quad r_{obstacle} = 0.3, \quad r_{ego} = 0.1.$$

(i) Transcribe this optimal control problem into a finite dimensional constrained optimization problem. Be sure to include the function to be minimized, the initial and final conditions, the collision avoidance constraint, and dynamics constraint.

(ii) Complete the notebook in the Code Setup section and the `optimize_trajectory` function within `P3_trajectory_optimization.ipynb`, implementing a direct method for optimal control using `scipy.optimize.minimize`.[1] Portions in the notebook are marked where you need to write your code. If implemented correctly, your optimizer should produce a trajectory that reaches the goal position without colliding with the obstacle! Include the generated trajectory plot of the open-loop plan generated by the non-linear optimizer in `trajectory_optimization.ipynb`.

(iii) Experiment with a different value of α used in the non-linear optimizer. Explain the differences that you see with the different choices of α.

Problem 4: Heading Controller (Section Prep)

Objective: Develop a ROS2 node for heading control using a proportional controller. This controller aims to minimize the error between the current heading of the TurtleBot3 robot and a desired goal heading. You'll be using ROS2 (Robot Operating System) with the rclpy library, utilizing the given messages and utility functions.

Background:

- `rclpy`: Python library to write ROS2 nodes.
- `TurtleBotState`: A message type that contains state information of the TurtleBot3, including its heading (theta).
- `TurtleBotControl`: A message type that contains control commands for the TurtleBot3, including angular velocity (omega).

[1] See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

- `wrap_angle`: A utility function that wraps angles between $-\pi$ and π.

Instructions:

1. Workspace Setup:
   - Open your Ubuntu installation on your Linux environment. From the home directory, create directory `~/autonomy_ws/src`:

   ```
   $ mkdir -p ~/autonomy_ws/src
   ```

   - From the `HW1` folder in the homework repository, move the `autonomy_repo` folder in to `autonomy_ws/src`.
   - Navigate to the `~/autonomy_ws` directory, build the workspace, and source:

   ```
   $ cd ~/autonomy_ws
   $ colcon build --symlink-install
   $ source install/local_setup.bash
   ```

   - Navigate to `~/autonomy_ws/src/autonomy_repo/scripts/heading_controller.py`. This is the file you will be editing in the next steps, and should be completely blank when you start.
2. Initial Imports:
   - Set the shebang at the top of your Python file: `#!/usr/bin/env python3`.
   - Import necessary libraries: numpy and rclpy.
   - From the `asl_tb3_lib.control` package, import the `BaseHeadingController` module.
   - From the `asl_tb3_lib.math_utils` package, import `wrap_angle`.
   - From the `asl_tb3_msgs.msg` package, import `TurtleBotControl` and `TurtleBotState` messages.
3. Define the `HeadingController` Class:
   - Create a class `HeadingController` that inherits from `BaseHeadingController`.
   - In the `__init__` method
     - Call the parent's `__init__` method.
     - Define a class variable for the proportional control gain `kp` and set it to 2.0.
4. Proportional Control:
   - Inside your `HeadingController` class, you should override the `compute_control_with_goal()` method from the `BaseHeadingController` class. This method is left unimplemented in the base class and serves as a placeholder for you to define your heading control logic.
     - The method takes in the current state and the desired state of the TurtleBot, both of which are of type `TurtleBotState`.
     - It should return a control message of type TurtleBotControl.
     - Use type annotations in your method signature to enforce these types.
   - Inside this method calculate the heading error ($\in [-\pi, \pi]$) as the wrapped difference between the goal's theta and the state's theta.
   - Use the proportional control formula, $\omega = k_p \cdot \text{err}$, to compute the angular velocity required for the TurtleBot to correct its heading error.
   - Create a new `TurtleBotControl` message, set its `omega` attribute to the computed angular velocity, and return it.
5. Node Execution:
   - In the main block (`if __name__ == "__main__":`) initialize the ROS2 system using `rclpy.init()`.
   - Create an instance of the `HeadingController` class.
   - Spin the node using `rclpy.spin()` to keep it running and listening for messages.
   - Ensure to shut down the ROS2 system with `rclpy.shutdown()` after spinning.

6. Running with your Simulator:

```
        # Open three terminals, and run the following in each of them.
        $ cd ~/autonomy_ws
        $ source install/local_setup.bash

        # In Terminal 1 run
        $ ros2 launch asl_tb3_sim root.launch.py
        # This will start your Turtlebot simulator
        # no GUI should appear.

        # In Terminal 2 run
        $ ros2 launch autonomy_repo heading_control.launch.py
        # This will start up the heading controller that you
        # just created, and open RVIZ. DO NOT SET A GOAL POSE.
        # Allow thirty seconds for the code in Terminal 2 to fully start
           up.

        # In Terminal 3 run.
        $ ros2 run autonomy_repo p3_plot.py
        # This will command a fixed goal position to test your controller,
        # and produce a plot that you can submit for grading.
        # You should see your Turtlebot moving in RVIZ.
        # This function will take at least ten seconds to produce a plot.

        # After you've successfully produced a plot,
        # try setting a new goal post in RVIZ by selecting the
        # "Goal Pose" button on the top toolbar and then
        # clicking and dragging in the environment to set
        # a goal heading.
```

7. Please include the resulting plot `p3_output.png` in your write-up.