Principles of Robot Autonomy I
Homework 2
Due Thursday, October 16 (5pm PT)

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-HW2-F25.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 and relevant plots from code.

Your submission must be typeset in LaTeX.

Introduction

The goal of this homework is to familiarize you with some Python fundamentals that will be used throughout the quarter, as well as techniques for controlling robots (a differential drive robot and a quadrotor robot) to track desired trajectories, e.g., as would be obtained from the planning and trajectory optimization methods from Homework 1.

Nonholonomic Wheeled Robot

Throughout this homework, we will consider a robot that operates with the simplest nonholonomic wheeled robot model, the unicycle, shown below in Figure 1.

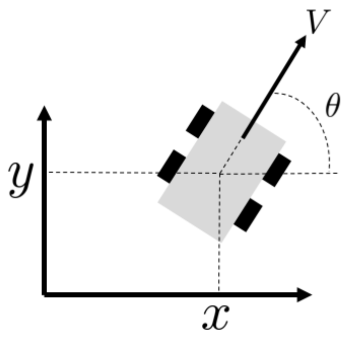

Figure 1: Unicycle robot model

The *kinematic* model we will use reflects the rolling without side-slip constraint, and is given below in Eq. (2).

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

In this model, the robot state is $\mathbf{x} = [x, y, \theta]^T$, where $[x, y]^T$ is the Cartesian location of the robot center and θ is its heading with respect to the x-axis. The robot control inputs are $\mathbf{u} = [v, \omega]^T$, where v is the velocity along the main axis of the robot and ω is the angular velocity.

Problem 1: Trajectory Optimization [30 Points]

Let's consider the problem of designing a dynamically feasible trajectory. In finding a trajectory that connects a starting state to the goal state in a dynamically feasible manner, 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 (2).

$$\begin{aligned}\dot{x}(t) &= v(t)\cos(\theta(t)),\\ \dot{y}(t) &= v(t)\sin(\theta(t)),\\ \dot{\theta}(t) &= \omega(t).\end{aligned} \tag{2}$$

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.

$$\begin{aligned} x(0) &= 0, \quad y(0) = 0, \quad \theta(0) = \pi/2,\\ x(t_f) &= 5, \quad y(t_f) = 5, \quad \theta(t_f) = \pi/2.\end{aligned}$$

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) **(15 Points)** 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) **(8 Points)** Complete the notebook in the Code Setup section and the `optimize_trajectory` function within `P1_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) **(7 Points)** Experiment with at least three different values of α used in the non-linear optimizer. Explain the differences that you see with the different choices of α.

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

Problem 2: Numpy and Class Inheritance [20 Points]

In this problem, we will demonstrate the use of class inheritance in Python classes and using Numpy for vectorized operations. The notebook associated with this homework problem is `P2_dynamics.ipynb`.

We will be using the `Dynamics` base class for two different dynamics models. The base class contains two unimplemented functions: `feed_forward` and `rollout`. The `feed_forward` function will propagate the dynamics a single time step with disturbances, and the `rollout` function will apply the `feed_forward` function multiple times to retrieve a trajectory of states over multiple time steps. Because the feed-forward dynamics are subject to disturbances, the same control sequence will result in different trajectories. We will observe this by executing multiple rollouts of the dynamics using the same control sequence from the same initial state.

The first model we consider is the kinematics model in Eq. (2). In the second, we will use a double integrator dynamics model. The equations for the double integrator model are as follows in Eq. (3):

$$\begin{aligned}\dot{x}(t) &= v_x(t),\\ \dot{y}(t) &= v_y(t),\\ \dot{v}_x(t) &= a_x(t),\\ \dot{v}_y(t) &= a_y(t).\end{aligned}\tag{3}$$

In this model, the robot state is $\mathbf{x} = [x, y, v_x, v_y]^T$ and the robot control inputs are $\mathbf{u} = [a_x, a_y]^T$.

(i) (**10 Points**) Fill in the `TurtleBotDynamics` class, in function `feed_forward` using discrete-time Euler integration, with the kinematic equations described in Eq. (2). Then in the same class, fill in function `rollout` with two `for`-loops, calling the `feed_forward` function. Run the cells that rollout the Turtlebot dynamics and plot the control and state trajectories (this code has been written for you). **Include the resulting plots in your write-up submission. Describe in a few sentences, what control inputs were used and how the plots of the state variables relate to the provided control inputs.**

(ii) (**5 Points**) Notice that in the previous problem, we used a `for`-loop to roll out several trajectories of the Turtlebot dynamics. In this problem, we will use the same base dynamics class for a `DoubleIntegratorDynamics` class, and use vectorization to reduce the number of `for`-loops needed to perform multiple rollouts. To do this, we will vectorize the feed-forward dynamics equations applied in the function `feed_forward`. **Write down the discrete time vectorized equations for a single dynamics step for multiple rollouts in your writeup** and fill in the function `feed_forward` in the `DoubleIntegratorDynamics` class. In your writeup, use notation $\mathbf{X}_t$ to denote the stacked state vectors from each rollout at timestep t, $\bar{\mathbf{A}}$ as the constructed matrix in the notebook code `A_stack`, and $\bar{\mathbf{B}}$ as the constructed matrix in the notebook code `B_stack`.

(iii) (**5 Points**) Fill in the code in function `rollout` in the `DoubleIntegratorDynamics` class using the `feed_forward` function you just wrote. Note that you should only need one `for`-loop! **Include the resulting plots in your writeup. Discuss the similarities and/or differences between the above two methods in terms of the states, control inputs and their plotted trajectories.**

Problem 3: LQR with Gain Scheduling [20 Points]

In this problem, you will implement LQR with gain scheduling for a planar quadcopter (drone) which wants to reach a goal position while avoiding an obstacle in the presence of a wind disturbance.

You should follow along the notebook `P3_gain_scheduled_LQR.ipynb`, which has 4 distinct parts. In parts 1 through 3 of the notebook, you will go through code that defines the dynamics, adds some visualization code, calculates an open-loop plan and sets up the wind disturbance. Note that you do not have to write any code in Parts 1 through 3. In Part 4 of the notebook, you will write a gain scheduled LQR algorithm that will allow the quadcopter to track the open loop trajectory as closely as possible.

When you are done with the notebook, return here and complete the following short answer questions.

(i) (**3 Points**) What is the dimensionality of the state space of the quadcopter? What do each of the values represent?

(ii) (**3 Points**) What is the dimensionality of the control space of the quadcopter? What do each of the values represent?

(iii) (**5 Points**) Briefly explain which method the notebook uses to calculate the open loop trajectory for the quadcopter.

(iv) Include the trajectory plot, from Part 4 in the notebook, here (**2 Points**). If implemented correctly, your drone should roughly follow the open-loop plan and come close to the goal position. Answer the following questions:

   (a) (**4 Points**) What are the dimensions of the gain matrices, K_i?

   (b) (**5 Points**) Which matrices can you modify to improve the gain correction, i.e, make the drone track the nominal trajectory more precisely?

Problem 4: ROS2 Navigation Node (Section Prep)

Note: This portion of the homework is **not graded**, but should be completed before Section on Week 5 (10/21 - 10/25) 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 HW1, as well as your differential flatness tracking controller from 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 HW2 Q2. 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 (HW1 Q1). 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 this homework in order to move things into the right places. The pseudo code for this function is detailed in Algorithm 1. 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.
3. 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?
4. See `compute_smooth_plan` function from `sim_astar.ipynb`.
5. See the block below `compute_smooth_plan` on how to construct a `TrajectoryPlan`.

Algorithm 1 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.