Principles of Robot Autonomy I
Homework 2
Due Thursday, October 17 (5:00pm)

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-F24.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.

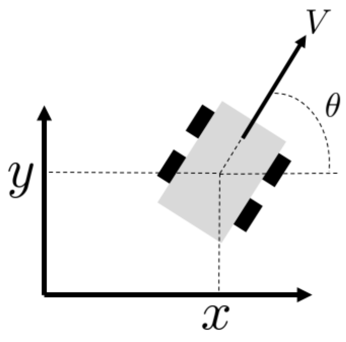

Figure 1: Unicycle robot model

The *kinematic* model we will use reflects the rolling without side-slip constraint, and is given below in Eq. (1).

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

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, subject to the control constraints:

$$|v(t)| \leq 0.75 \text{ m/s}, \qquad \text{and} \qquad |\omega(t)| \leq 1.0 \text{ rad/s}.$$

Problem 1: Numpy and Class Inheritance

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 `P1_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. (1). In the second, we will use a double integrator dynamics model. The equations for the double integrator model are as follows in Eq. (2):

$$\begin{aligned} \dot{x}(t) &= v_x(t), \\ \dot{y}(t) &= v_y(t), \\ \dot{v}_x(t) &= a_x(t), \\ \dot{v}_y(t) &= a_y(t). \end{aligned} \tag{2}$$

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) Fill in the `TurtleBotDynamics` class, in function `feed_forward` using discrete-time Euler integration, with the kinematic equations described in Eq. (1). 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) Notice that in the previous problem, we used a `for`-loop to rollout 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) 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 2: Trajectory Generation via Differential Flatness

Consider the dynamically extended form of the robot kinematic model:

$$\begin{aligned}\dot{x}(t) &= v(t)\cos(\theta(t)),\\ \dot{y}(t) &= v(t)\sin(\theta(t)),\\ \dot{v}(t) &= a(t),\\ \dot{\theta}(t) &= \omega(t),\end{aligned} \tag{3}$$

where the two inputs are now $(a(t), \omega(t))$.

Differentiating the velocities $(\dot{x}(t), \dot{y}(t))$ once more yields

$$\begin{bmatrix}\ddot{x}(t)\\ \ddot{y}(t)\end{bmatrix} = \underbrace{\begin{bmatrix}\cos(\theta) & -v(t)\sin(\theta)\\ \sin(\theta) & v(t)\cos(\theta)\end{bmatrix}}_{:=J}\begin{bmatrix}a\\ \omega\end{bmatrix} := \begin{bmatrix}u_1\\ u_2\end{bmatrix}. \tag{4}$$

Note that $\det(J) = v$. Thus for $v > 0$, the matrix J is invertible.

Roughly speaking, we say a system is *differentially flat* if we can find a set of outputs (*flat outputs*), equal in number to the number of inputs, such that all states and inputs can be determined from the flat outputs without integration [1].

For the unicycle robot, the *flat outputs* are given by (x, y). Thus, if we design a trajectory $(x(t), y(t))$, we can determine all the states and inputs that correspond to it. Specifically, we can get $(\ddot{x}(t), \ddot{y}(t))$ and invert equation (4) above to obtain the corresponding control input histories $a(t)$ and $\omega(t)$. In this case, we will call $(\ddot{x}(t), \ddot{y}(t))$ the *virtual control inputs*, (u_1, u_2).

In this problem, we will design the trajectory $(x(t), y(t))$ using a polynomial basis expansion of the form

$$x(t) = \sum_{i=1}^{n} x_i\psi_i(t), \quad y(t) = \sum_{i=1}^{n} y_i\psi_i(t),$$

where ψ_i for $i = 1, \dots, n$ are the basis functions, and x_i, y_i are the coefficients to be designed. (Note that basis expansion is just one possible way to design trajectories.)

(i) Take the basis functions $\psi_1(t) = 1$, $\psi_2(t) = t$, $\psi_3(t) = t^2$, and $\psi_4(t) = t^3$. Write a set of linear equations in the coefficients x_i, y_i for $i = 1, \dots, 4$ to express the following initial and final conditions:

$$\begin{aligned}x(0) = 0, \quad y(0) = 0, \quad v(0) = 0.5, \quad \theta(0) = -\pi/2,\\ x(t_f) = 5, \quad y(t_f) = 5, \quad v(t_f) = 0.5, \quad \theta(t_f) = -\pi/2.\end{aligned}$$

Solve for the coefficients x_i, y_i, $i = 1, \dots, 4$ when $t_f = 25$.

(ii) When finding $a(t)$ and $\omega(t)$, why can we not set $v(t_f) = 0$?

(iii) Implement the following functions in `P2_differential_flatness.py` to compute the state-trajectory $(x(t), y(t), \theta(t))$, and control history $(v(t), \omega(t))$:

- `compute_traj_coeffs` to compute the coefficients x_i, y_i, for $i = 1, \dots, 4$.
- `compute_traj` to use the coefficients to compute the state trajectory $[x, y, \theta, \dot{x}, \dot{y}, \ddot{x}, \ddot{y}]$ from $t = 0$ to $t = t_f$.
- `compute_controls` to compute the actual robot controls (v, ω) from the state trajectory.

Run the script in the terminal by typing

```
$ python3 P2_differential_flatness.py
```

to see your trajectory optimization in action. Please include `differential_flatness.png` in your write up and answer the following questionss:

(a) Briefly describe how you obtained the θ state from the coefficients (x_i, y_i) in `compute_traj`.

(b) Briefly explain how the plotted trajectory relates to the provided initial and final conditions.

(iv) Now, let's see what happens when we execute these actions on a system with disturbances. To do so, open the visualization Jupyter notebook by running the following command:

```
$ jupyter notebook sim_trajectory.ipynb
```

This notebook will use the code you wrote for the this problem to plan trajectories for the initial and final states specified in the notebook. Run the cells up until the closed loop simulation to validate your work up to this point. You should see that the planned trajectory indeed reaches the goal, while in the presence of noise, executing actions open-loop causes the trajectory to deviate. Experiment with different initial and final states to see how the system behaves! Each time you run the plotting code it will save the plot to `plots/sim_traj_openloop.pdf`. **Include a plot with the initial conditions given in part (i) in your write up.**

Closed-Loop Control

Clearly, in order to deal with unmodeled effects and noise, we need to use feedback control. Now you will experiment with a smooth dynamic trajectory tracking controller for the unicycle model[1], thereby imbuing your robot with some robustness.

We will now address *closed-loop* control for trajectory tracking using the differential flatness approach. Consider the trajectory designed in parts (i)-(iv) (we will refer to the center coordinates of this desired trajectory as (x_d, y_d)). We will implement the following virtual control law for trajectory tracking:

$$\begin{aligned} u_1 &= \ddot{x}_d + k_{px}(x_d - x) + k_{dx}(\dot{x}_d - \dot{x}) \\ u_2 &= \ddot{y}_d + k_{py}(y_d - y) + k_{dy}(\dot{y}_d - \dot{y}) \end{aligned} \tag{5}$$

where $k_{px}, k_{py}, k_{dx}, k_{dy} > 0$ are the control gains.

(v) Write down a system of equations for computing the true control inputs (v, ω) in terms of the virtual controls $(u_1, u_2) = (\ddot{x}, \ddot{y})$ and the vehicle state. HINT: it includes an ODE (you are allowed to have $\dot{v}$ in your answer).

(vi) Complete the `compute_control` function in the `TrajectoryTracker` class which is defined in the file `P2_trajectory_tracking.py`. Specifically, program in the virtual control law Eq. (5) to compute u_1 and u_2, *and* integrate your answer from part (v) to convert (u_1, u_2) into the actual control inputs (v, ω).

Hint: At each timestep you may consider the current velocity to be that commanded in the previous timestep. The controller class is designed to save this as the member variable `self.v_prev`

Warning: You must be careful when computing the actual control inputs due to the potential singularity in velocity discussed in Problem 1 — although we took care to ensure that the nominal trajectory did not have $v \approx 0$ it is possible that noise might cause this singularity to occur. If the actual velocity drops below the threshold `V_PREV_THRES`, you should "reset" it to `V_PREV_THRES`.

Validate your work. Open the trajectory tracking visualization notebook:

[1]Note that many of the control laws discussed in this problem set can also be extended to related nonoholonomic robot models, e.g., a rear/front wheel drive robot with forward steering and a trailer/car combination. Interested students are referred to [**ADL-GO-CS:98**].

```
$ jupyter notebook sim_trajectory.ipynb
```

Run the closed loop simulation section and see if using the controller, the robot can now track the planned trajectory even in the presence of noise. Feel free to experiment with different initial and final states, as well as different amounts of noise or different controller gains.

Each run will save a plot in `plots/sim_traj_closedloop.png`. Run the script with the initial and final positions as given in part (i) and a nonzero noise amount, and include the resulting plot in your write up and answer the following briefly:

(a) Discuss how you would use the matrix J from equation (4) to simplify the code to obtain the control inputs (If you didn't use J in your code, think about how it can be used to simplify the code and discuss here).

(b) Discuss how the closed-loop trajectory differs from the open-loop trajectory in the plot. Does the closed-loop trajectory perfectly track the nominal trajectory?

Problem 3: LQR with gain scheduling

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) What is the dimensionality of the state space of the quadcopter? What do each of the values represent?

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

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

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

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

  (b) 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 (10/23 - 10/27) 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 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.

References

[1] Richard M Murray, Muruhan Rathinam, and Willem Sluis. “Differential flatness of mechanical control systems: A catalog of prototype systems”. In: *ASME international mechanical engineering congress and exposition*. Citeseer. 1995.