AA 174A: Principles of Robot Autonomy I
Problem Set 1
Due Friday, October 13 (11:59pm 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.

   - An easy way to install this: use VSCode and get the Jupyter notebook extension.

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/AA274a-HW1.git` in a terminal window. *You can ignore* `P4_trajectory_optimization.ipynb` *as it won't be needed in this problem set.*

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.

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 nonholonomic wheeled robots.

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.

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.

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

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

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:

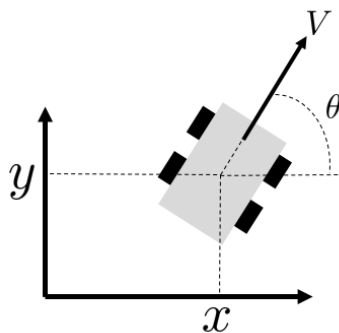$$|v(t)| \leq 0.75 \text{ m/s}, \quad \text{and} \quad |\omega(t)| \leq 1.0 \text{ rad/s}.$$

Figure 1: Unicycle robot model

[5 pts] 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.

(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.

Controlling Nonholonomic Wheeled Robots

In Problem 2, we start by exploring how we might generate dynamically feasible trajectories and their associated *open-loop* control sequences.

[15 pts] 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 $i = 1, \ldots, n$ 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 $i = 1, \ldots, 4$ to express the following initial and final conditions:

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

Solve for the coefficients x_i, y_i, $i = 1, \ldots, 4$ when $t_f = 15$.

(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 $i = 1, \ldots, 4$.
  - `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.

(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.

(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

---

[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 [2].

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 to 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:

```
$ 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.

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.

[10 pts] Problem 3: 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).

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

Instructions:

1. Workspace Setup:

   - Open your Ubuntu installation on your Virtual Machine. 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
  $ 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
# on the VM no GUI should appear.

# In Terminal 2 run
$ ros2 launch autonomy_repo heading_controller.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.

References

[1] R. M. Murray, M. Rathinam, and W. Sluis, "Differential flatness of mechanical control systems: A catalog of prototype systems," in *ASME international mechanical engineering congress and exposition*. Citeseer, 1995.

[2] A. De Luca, G. Oriolo, and C. Samson, "Feedback control of a nonholonomic car-like robot," in *Robot motion planning and control*. Springer, 1998, pp. 171–253.