Principles of Robot Autonomy I
Problem Set 1
Due Tuesday, September 29 (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 2.7 (not Python 3.X!)
3. Jupyter Notebook: A web application for interactive code development and prototyping.

While not required, another useful tool is Conda, which is a package and environment management system that can be run on Windows, macOS, and Linux. With conda you can create local "environments" that utilize specific package versions without affecting other packages! For example, say you use Python 3.8 as default for all of your personal projects but need Python 2.7 for AA274A. Using conda, an environment can be created that uses Python 2.7 and within that environment you can download versions of any additional packages you need that depend on Python 2.7 without affecting your default setup.

Starter code for this problem set has been made available online through Github. To get started, download the code by running `git clone https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1.git` in a terminal window.

You will submit your homework to Gradescope. Your submission will consist of (1) a single pdf with your answers for written questions (denoted by the ✎ symbol) and (2) a zip folder containing your code (and any relevant files and plots) for the programming questions (denoted by the 🖥 symbol).

Your written part must be typeset (e.g., LaTeX or Word).

Also note this problem set represents the start of an incremental journey to build our robot autonomy stack. Thus, full completion of the coding modules in this set is key, as your implementation will be leveraged in the next problem set and in the final project.

Introduction

The goal of this problem set is to familiarize you with some of the techniques for controlling nonholonomic wheeled robots. The nonholonomic constraint here refers to the rolling without slipping condition for the robot wheels which leads to a non-integrable set of kinematic constraints. In this problem set we will consider 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).

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

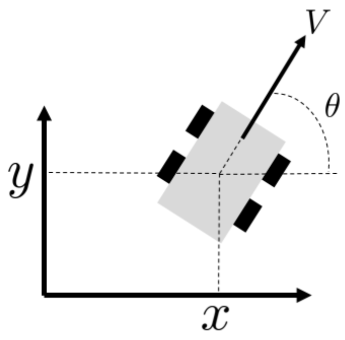

Figure 1: Unicycle robot model

In this model, the robot state is $\mathbf{x} = (x, y, \theta)$, where (x, y) is the Cartesian location of the robot center and θ is its heading with respect to the x-axis. The robot control inputs are $\mathbf{u} = (V, \omega)$, where V is the velocity along the main axis of the robot and ω is the angular velocity, subject to the control constraints:

$$|V(t)| \leq 0.5 \text{ m/s}, \qquad \text{and} \qquad |\omega(t)| \leq 1.0 \text{ rad/s}.$$

We'll start by exploring how we might generate dynamically feasible trajectories and their associated *open-loop* control sequences. However, the presence of un-modeled effects, external disturbances, and/or time sampling may lead to open-loop control being ineffective on real systems. In the second part of the problem set, we will focus on *closed-loop* control strategies for these systems that can reject these disturbances and provide robustness.

Problem 1: Trajectory Generation via Differential Flatness

Consider the dynamically extended form of the robot kinematic model:

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

where the two inputs are now $(a(t), \omega(t))$. Differentiating the velocities $(\dot{x}(t), \dot{y}(t))$ once more yields

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

Note that $\det(J) = V$. Thus for $V > 0$, the matrix J is invertible. Hence, the outputs (x, y) are the *flat outputs* for the unicycle robot, which means we may use the virtual control inputs $(u_1(t), u_2(t))$ to design the trajectory $(x(t), y(t))$ and invert the equation above to obtain the corresponding control histories $a(t)$ and $\omega(t)$.

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.

(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:

$$x(0) = 0, \quad y(0) = 0, \quad V(0) = 0.5, \quad \theta(0) = -\pi/2,$$
$$x(t_f) = 5, \quad y(t_f) = 5, \quad V(t_f) = 0.5, \quad \theta(t_f) = -\pi/2,$$

where $t_f = 15$.

(ii) Why can we not set $V(t_f) = 0$?

(iii) Implement the following functions in P1_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

```
$ python P1_differential_flatness.py
```

to see your trajectory optimization in action.

(iv) You may have noticed that in this process, we didn't specify that our robot is constrained to have a maximum speed $V_{\max} = 0.5$ and maximum angular velocity of $\omega_{\max} = 1$, and in fact, the resulting trajectory violates these constraints!

Since there are only 4 basis polynomials and 4 constraints each for x and y, we get a unique solution that satisfies the constraints in part (i), but this solution may not necessarily satisfy the control constraints. Some ways to circumvent this are to (1) increase the number of basis polynomials and solve a constrained trajectory optimization problem, (2) increase the value of t_f until the control history becomes feasible (this can be very suboptimal), or (3) "edit" the infeasible trajectory, by re-scaling the velocity trajectory while keeping the *geometric* aspects of the trajectory the same. We will adopt option (3).

First, we need to reparameterize the trajectory $(x(t), y(t), \theta(t))$ from being a function of time t to a function of arc-length s, which is the distance traveled along the path from the starting point. In this way, $s(t) = \int_0^t V(\tau) d\tau$ represents how far along the trajectory we are at time t, and $(x(s), y(s), \theta(s))$ represent the state at that arc-length.

Now, in order to satisfy control constraints, we can follow this geometric trajectory at a different pace, applying a different velocity as a function of s, $\widetilde{V}(s)$. However, note that choosing $\widetilde{V}$ also defines a new angular velocity $\widetilde{\omega}$. This occurs since the kinematic equation $\dot{\theta}(t) = \omega(t)$ can be written as:

$$\frac{d\theta(s)}{ds} \dot{s} = \omega(s),$$

by using the chain rule. Thus, noting that from the original trajectory $\dot{s} = V > 0$ the geometric path $\theta(s)$ satisfies:

$$\frac{d\theta(s)}{ds} = \frac{\omega(s)}{V(s)}.$$

Therefore, since the new trajectory will also have to satisfy the same kinematic equation and we also want to keep the same geometric path it can be seen that:

$$\frac{d\theta(s)}{ds} \tilde{V}(s) = \tilde{\omega}(s),$$

and substituting in the geometric path information:

$$\frac{\omega(s)}{V(s)} = \frac{\tilde{\omega}(s)}{\tilde{V}(s)}.$$

Therefore once the new velocity profile $\tilde{V}(s)$ is chosen the new angular velocity profile is defined by $\tilde{\omega}(s) = \tilde{V}(s)\frac{\omega(s)}{V(s)}$.

Once $\tilde{V}(s)$ and $\tilde{\omega}(s)$ are defined, they need to be mapped back into functions of time. To accomplish this we can define a function $\tau(s)$ that maps each s to a new time t by leveraging the fact that $\widetilde{V}(s) = \frac{ds}{dt}$ and integrating (choosing $\tau(0) = 0$):

$$\tau(s) = \int_0^t dt' = \int_0^s \frac{ds'}{\widetilde{V}(s')}.$$

Implement the remaining functions in `P1_differential_flatness.py`:

- `compute_arc_length` which computes s at each point of the trajectory computed in part (iii).
- `rescale_V` which chooses $\widetilde{V}$ at each point along the original trajectory, ensuring that the resulting $\widetilde{V}$ and $\widetilde{\omega}$ satisfy the control constraints.
- `compute_tau` which uses $\widetilde{V}$ and s to compute the new time profile.
- `rescale_om` which computes the adjusted angular velocity $\widetilde{\omega}$ from $\widetilde{V}$ and the original (V, ω).

Run the script again to check your results! The new trajectory should take the same geometric path, but take a longer time to satisfy the control constraints. The resulting plot will be saved to `plots/differential_flatness.png`.

(v) ✎ Please include `differential_flatness.png` in your write up.

(vi) ✎ 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. However, the nonholonomic nature of the system raises some challenging controllability issues. Chiefly, it can be shown that the unicycle cannot be stabilized around any pose (position & orientation) using smooth (or even continuous) time-invariant feedback control laws [1] (Brockett's Theorem). This result has prompted extensive research in various control strategies such as smooth time-varying control laws, discontinuous control laws, and feedback linearization. In contrast to the pose stabilization problem, the trajectory tracking problem is significantly simpler. In this problem set you will experiment with a discontinuous kinematic pose controller and a smooth dynamic trajectory tracking controller for the unicycle model[1], thereby imbuing your robot with some robustness.

Problem 2: Pose Stabilization

We will now study *closed-loop* control for pose stabilization, i.e., stabilizing the robot around some desired position and orientation. For consistency, we will set the desired goal position to be $(x_g, y_g) = (5, 5)$ and desired pose as $\theta_g = -\pi/2$. One way to get around Brockett's controllability result, which prevents the use of smooth time-invariant state-feedback control laws, is to use a change of variables.

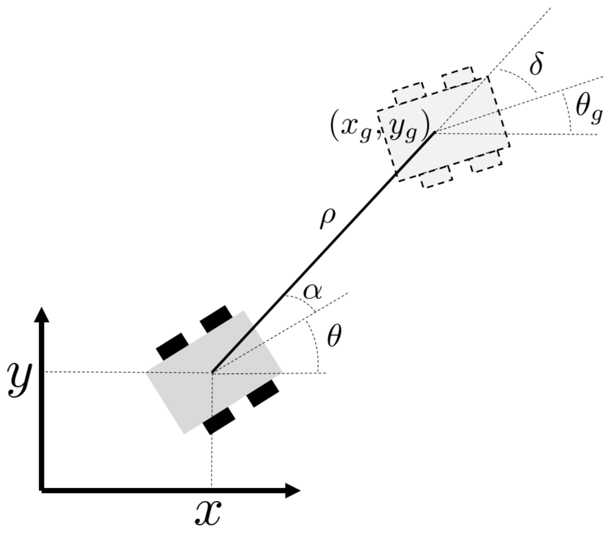

Figure 2: New coordinate system

In Figure 2, α is the bearing error and (ρ, δ) represent the robot's polar coordinate position with respect to the goal. In these new coordinates our kinematic equations of motion become:

$$\begin{aligned} \dot{\rho}(t) &= -V(t)\cos(\alpha(t)), \\ \dot{\alpha}(t) &= V(t)\frac{\sin(\alpha(t))}{\rho(t)} - \omega(t), \\ \dot{\delta}(t) &= V(t)\frac{\sin(\alpha(t))}{\rho(t)}. \end{aligned} \tag{3}$$

The pose stabilization problem is now equivalent to driving the state (ρ, α, δ) to the unique equilibrium $(0, 0, 0)$. In these coordinates, Brockett's condition no longer applies, so we can use the simple (smooth) control law

$$\begin{aligned} V &= k_1 \rho \cos(\alpha), \\ \omega &= k_2 \alpha + k_1 \frac{\sin(\alpha)\cos(\alpha)}{\alpha}(\alpha + k_3 \delta), \end{aligned} \tag{4}$$

where $k_1, k_2, k_3 > 0$. Note that the kinematic equation (3) is undefined at $\rho = 0$. However, the control law above drives the robot to the desired goal *asymptotically* (i.e., as time goes to infinity) so we never have to worry about the equations becoming undefined. Furthermore, note that this control law is still discontinuous in the original coordinates, as necessary due to Brockett's result.

[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].

(i) 🖥 Complete the function `compute_control` in the `PoseController` class in `P2_pose_stabilization.py` and program the control law above. The goal position and controller gains k_1, k_2, k_3 are member variables of the class.

(ii) 🖥 Validate your work. Open the visualization Jupyter notebook by running the following command:

```
$ jupyter notebook sim_parking.ipynb
```

Test your controller with different initial positions and goal poses to simulate different parking scenarios. For each section of the notebook, choose initial and final poses to demonstrate forward, reverse, and parallel parking scenarios.

(iii) ✎ The plots will be saved as `plots/sim_parking_<parking-type>.png`. Please include all three plots (forward, reverse, and parallel) in your write up.

Hints: (1) You should use the function `wrapToPi` also present in `utils.py` to ensure that α and δ remain within the range $[-\pi, \pi]$. This is needed since the control law has terms linear in α and δ. (2) Use the function `np.sinc` to handle values $\alpha \approx 0$ in the control law for ω above. (3) You are free to choose the gains k_1, k_2, and k_3 (try staying between $(0, 1.5]$), however, do not be too aggressive since we have saturation limits!

Note: It is possible to extend this method to allow path tracking [3] however we will adopt a simpler and more elegant approach for tracking trajectories.

Problem 3: Trajectory Tracking

We will now address *closed-loop* control for both trajectory tracking using the differential flatness approach, as well as parking using the non-linear kinematic controller from Problem 3. Consider the trajectory designed in Problem 1 (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.

(i) ✎ 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.

(ii) 🖥 Complete the `compute_control` function in the `TrajectoryTracker` class which is defined in the file `P3_trajectory_tracking.py`. Specifically, program in the virtual control law Eq. (5) to compute u_1 and u_2, *and* integrate your answer from (i) 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 the nominal velocity In this case you should use a "reset" strategy to the nominal velocity.

(iii) 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 Problem 1 and a nonzero noise amount, and include the resulting plot in your write up.

Note: It is possible to modify the flatness-based trajectory tracking controller to also allow pose stabilization. See [4] for details.

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 Problem Set 2 we will integrate these components with motion planning algorithms to find and track feasible trajectories through cluttered environments.

Extra Problem: Optimal Control and Trajectory Optimization

Let's revisit the problem of designing a dynamically feasible trajectory. In Problem 1, we only were 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). 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

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

where $\lambda \in \mathbb{R}_{\geq 0}$ is a weighting factor and t_f is free. We'll use the following initial and final conditions, which are the same as in Problem 1.

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

(i) Derive the Hamiltonian and conditions for optimality and formulate the problem as a 2P–BVP. Make sure you explain your steps, and include the boundary conditions.

(ii) Complete P4_optimal_control.py to solve the 2P–BVP with the largest possible value of λ such that the resulting optimal control history satisfies the control constraints. Note, for this problem, do not use the constrained control version of the optimality conditions, and rather vary the value of λ youself until the unconstrained problem yields a feasible solution. Experiment with different initial guesses for the solution. Run python P4_optimal_control.py to test your code. In particular, it will generate a plot of the trajectory $(x(t), y(t))$ and the control history $(V(t), \omega(t))$. The plot will be saved to plots/optimal_control.png. Use this to verify that the control constraints and boundary conditions are met, and that the solution trajectory is smooth.

(iii) Please include optimal_control.png in your write up. Compare the result with the trajectory obtained through the differential flatness approach in Problem 1.

(iv) ✎ Explain the significance of using the largest feasible λ.

(v) ✎ Validate your work. We will now simulate the car with the computed control history in the presence of disturbances (modeled as input noise).

Open and run the visualization notebook for this problem:

```
$ jupyter notebook sim_optimal_control.ipynb
```

Try experimenting with different amounts of noise. The plot will be saved as `plots/sim_traj_optimal_control.png`. Please include it in your write up.

Note: You will need the `scikits.bvp_solver` package. See `https://pythonhosted.org/scikits.bvp_solver/`, and try installing directly using `pip install scikits.bvp_solver`. If this fails for some reason, try `pip install --user scikits.bvp_solver` (sometimes the default folder that pip installs to is owned by the "root" account in your computer rather than your own user account. `--user` tells pip to install the package to a directory that your user account certainly has access to). If even that fails for some reason, update/reinstall `gcc` (make sure you include `gfortran`), download the source code for the package, and build it yourself.

Hint: You will need to re-formulate the problem into "standard" form - see *Reformulation of boundary value problems into standard form," SIAM review, 23(2):238-254, 1981.*

References

[1] R. W. Brockett, "Asymptotic stability and feedback stabilization," *Differential geometric control theory*, vol. 27, no. 1, pp. 181–191, 1983.

[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.

[3] M. Aicardi, G. Casalino, A. Bicchi, and A. Balestrino, "Closed loop steering of unicycle like vehicles via lyapunov techniques," *IEEE Robotics & Automation Magazine*, vol. 2, no. 1, pp. 27–35, 1995.

[4] G. Oriolo, A. De Luca, and M. Vendittelli, "Wmr control via dynamic feedback linearization: design, implementation, and experimental validation," *IEEE Transactions on control systems technology*, vol. 10, no. 6, pp. 835–852, 2002.