AA 274: Principles of Robotic Autonomy
Problem Set 1
Due January 25

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 roll without slip condition for the robot wheels which leads to a nonintegrable set of differential constraints. In this problem set we will consider the simplest nonholonomic wheeled robot model, the unicycle, shown below in Figure 1.

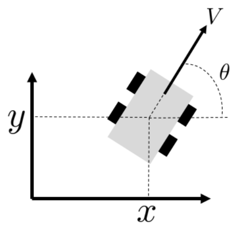


Figure 1: Unicycle robot model

The *kinematic* model we will use reflects the roll without slip constraint, and is given below in eq. (1).

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

where (x, y) is the Cartesian location of the robot center, θ is its alignment with respect to the x-axis, V is the velocity along the main axis of the robot, and ω is the angular velocity. We take (V, ω) to be our two control inputs, subject to the constraints: $|V(t)| \leq 0.5$ m/s and $|\omega(t)| \leq 1.0$ rad/s.

In the first part of the problem set you will explore different methods of generating feasible trajectories and associated *open-loop* control histories. The presence of un-modeled effects, external disturbances, and/or time sampling will undoubtedly lead to the need for *closed-loop* control and will be the focus for the second part of this problem set.

The nonholonomic constraint raises some challenging controllability issues. Chiefly, it can be shown that the unicycle (and more generally, the class of underactuated driftless regular systems) cannot be stabilized around any posture (position & orientation) using smooth (or even continuous) time-invariant feedback control laws [1] (Brockett's Theorem). This negative result has prompted extensive research in various control strategies such as smooth time-varying control laws, discontinuous control laws, and feedback linearization. In contrast, the trajectory tracking problem is significantly simpler (under some mild conditions). In this problem set you will experiment with a (discontinuous) kinematic and a (smooth) dynamic controller for both posture and trajectory feedback control, thereby imbuing your robot with some robustness.

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].

Problem 1

Consider the kinematic model of the unicycle given in (1). The objective is to drive from one waypoint to the next waypoint with minimum time & 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. Consider the following initial and final conditions:

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

Task: (i) Derive the Hamiltonian and conditions for optimality and formulate the problem as a 2P - BVP. (ii) Edit the script traj_opt.py to solve the 2P-BVP with the largest (Why?) possible value of λ such that the resulting optimal control history satisfies the constraints. That is, do not use the constrained control version of the optimality conditions, i.e., **use the conditions in Slide 9 in Lecture 3 and vary the value of λ yourself**. Experiment with different initial guesses for the solution. (iii) Extract the trajectory $(x(t), y(t))$ and the control history $(V(t), \omega(t))$ using a sample time of 5ms. (iv) Plot (1) trajectory $(x(t), y(t))$, (2) control history $(V(t), \omega(t))$. (v) Save the state and control histories as a .npy file with the format $(x, y, \theta, V, \omega)$.

Note: You will need the scikits.bvp_solver package. See https://pythonhosted.org/scikits.bvp_solver/. Try installing directly using `sudo pip install scikits.bvp_solve`. If this fails for some reason, update/reinstall gcc (make sure you include gfortran), download the source code for the package, and build 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.*

Validate: We will now simulate the car with the computed control history with and without the presence of disturbances (modeled as input noise). Run sim_traj.py using the filename containing your saved control history as: `python sim_traj.py <filename> <x_0> <y_0> <th_0> 1 open`, where (x_0, y_0, th_0) are the initial conditions from above. Confirm that the "disturbance-free" trajectory arrives at the goal while the "perturbed" trajectory deviates.

Problem 2

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 and 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 get the control history $a(t)$ and $\omega(t)$. 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 $\psi_i, i = 1, \dots, n$ are the basis functions, and x_i, y_i are the coefficients to be designed.

Task: (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 $x_i, y_i, i = 1, \dots, n$ to express the following initial and final conditions:

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

where $t_f = 15$ (Why can we not set $V(t_f) = 0$?). (ii) Edit the script `traj_df.py` to solve the system of equations above and compute the state-trajectory $(x(t), y(t), \omega(t))$, and control history $(V(t), \omega(t))$.

(iii) Since there are only 4 basis polynomials and 4 constraints (for x and y at each endpoint), we get a unique solution. However, the trajectory 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) re-scale the velocity trajectory while keeping the *geometric* aspects of the trajectory the same. We will adopt the 3rd strategy. To do this, define a path parameter s (i.e., arc-length) for the trajectory. Then, we can write the control inputs as:

$$V(t) = V_s(s(t))\dot{s}(t), \quad \omega(t) = \omega_s(s(t))\dot{s}(t).$$

Here $V_s(s)$ and ω_s are the "pseudo-velocities" defined with respect to the arc-length parameter s, and $\dot{s}(t)$ can be thought of as a "timing-law". Adjust $V(t)$ so that $V(t) \leq 0.5$ m/s and $|\omega(t)| \leq 1$ rad/s. Note that you will have to re-generate the trajectories since the final time t_f may be longer than that obtained by solving the optimal control problem above. (iv) Compute the scaled trajectory and save the data as a `.npy` file with the format: $(x, y, \theta, V, \omega, \dot{x}, \dot{y}, \ddot{x}, \ddot{y})$ using a timestep of 5 ms (we will use the desired velocity and acceleration in Problem 4). Plot (1) trajectory $(x(t), y(t))$, (2) arc-length $s(t)$ for the original and scaled trajectories, (3) control history for the original and scaled trajectories.

Validate: Run `sim_traj.py` using the filename containing your saved control history as: `python sim_traj.py <filename> <x_0> <y_0> <th_0> 1 open`. Confirm that the "disturbance-free" trajectory arrives at the goal while the "perturbed" trajectory deviates.

Problem 3

We will now study *closed-loop* control for posture stabilization, i.e., stabilize the robot around some desired position and pose. 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 invariant state-feedback control laws, is to use a change of variables. Consider Figure 2.

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

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

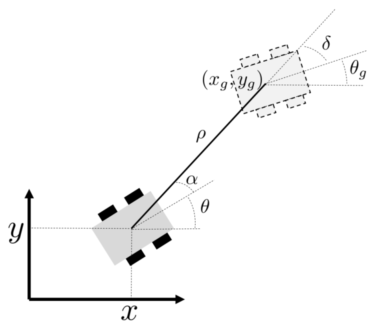


Figure 2: New coordinate system

The posture stabilization problem is now equivalent to driving the state (ρ, α, δ) to the unique equilibrium $(0, 0, 0)$. Note that in these coordinates, Brockett's condition no-longer applies so we will 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 (1) 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. (2) The control law is discontinuous in the original coordinates, as necessary due to Brockett's result.

Task: Edit the control function `ctrl_pose` in the file `car_dyn_control.py` and program the control law above.

HINTS: (1) You should use the function `wrapToPi` also present in that file 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) 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!

Validate: Run the script `sim_parking.py` with a variety of start locations and poses using the syntax: `python sim_parking.py x_0 y_0 th_0 t_f` where (x_0, y_0, th_0) is the initial position and pose of the robot, and t_f is the length of time to simulate. Plot the trajectories for forward, reverse, and parallel parking scenarios. Plot also the control histories $(V(t), \omega(t))$.

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

Problem 4

We will now address *closed-loop* control for (i) trajectory tracking using the differential flatness approach, and (ii) parking using the non-linear kinematic controller from Problem 3. Consider the trajectory designed in Problem 2. 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.

Task: (i) Write down the form of the dynamic compensator as a set of differential-algebraic equations with input (u_1, u_2) and output (V, ω). Clearly define any internal states that you may need. (ii) Edit the function `ctrl_traj` in the file `car_dyn_control.py` and program in the virtual control law above *and* the dynamic compensator to convert (u_1, u_2) into the actual control inputs (V, ω). (We have already filled in the integration step for the compensator internal state).

WARNING: You must be very careful when defining the actual control inputs due to a potential singularity regarding the internal state of the compensator. For instance, you may wish to use a "reset" strategy if this value gets too small.

(iii) Modify this control function to *switch* to the pose stabilization controller from Problem 3 when you are "sufficiently close" to the goal position $(5, 5)$.

Validate: Run `sim_traj.py` using the filename containing your saved trajectory history as: `python sim_traj.py <filename> <x_0> <y_0> <th_0> 1 closed`. Confirm that both the "disturbance-free" and "perturbed" trajectories arrive at the goal. Experiment with different initial conditions.

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

Problem 5

As discussed during lecture, we will be using ROS throughout the semester. ROS stands for Robot Operating System. Effectively it allows us to break down a robot software stack in different modules called nodes (each of them a process) that seamlessly communicate with each other over topics (using TCP/UDP-like connections). Every week we will implement a piece of the software stack (most likely a node) that we will use to accomplish a complex task with a real robot at the end of the semester.

The simple publisher and subscriber

In order to facilitate your experience with ROS, we require that you use a clean install of Ubuntu 16.04.1. You can either install Ubuntu in dual-boot on your machine or simply have a virtual machine. Stanford provides access to a good virtualization solution (VMware) for free.

Go to `https://stanford.onthehub.com/WebStore/Welcome.aspx`, login (top right), search "VMware" in the top box, select "VMware Fusion 8.0" if you are using Mac, select "VMware Workstation 12" otherwise.

We are providing you with a virtual machine (located at `https://stanford.box.com/v/aa274vm`) that already has most of the required software installed. For reference, it contains the following software (for manual installations, `sudo apt-get install <item>` on the command line will install the turtlebot dependencies listed below):

- Ubuntu 16.04.1
- ROS Kinetic
- ros-kinetic-turtlebot
- ros-kinetic-turtlebot-apps
- ros-kinetic-turtlebot-interactions
- ros-kinetic-turtlebot-simulator
- git

The virtual machine also contains a catkin workspace which was created using the `catkin_init_workspace` command. You can think of catkin as a combination of a compiler (built on top of cmake) and a way for you to manage your environment.

If you look at the end of `~/.bashrc` you will see that we call a script called `setup.bash` contained in your workspace every time we launch a bash shell. This then allows bash to find the different packages contained in your catkin workspace.

Lets start by creating a package in your workspace.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg turtlebot_control std_msgs roscpp rospy
$ cd ~/catkin_ws
$ catkin_make
```

Now create a script directory in your new `turtlebot_control` package. Then create a file called `publisher.py` in the scripts directoy and add the following code to the file.

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def publisher():
    pub = rospy.Publisher('random_strings', String, queue_size=10)
    rospy.init_node('publisher', anonymous=True)
    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        # your code to pulish a string goes here
        rate.sleep()

if __name__ == '__main__':
    try:
        publisher()
    except rospy.ROSInterruptException:
        pass
```

fill in the section we commented out. The code should take one or two lines and use the `pub.publish()` command.

Make the script executable using the following command (you need to do this for every new python node you write).

```
$ chmod a+x publisher.py
```

Once your script is written, build your package using `catkin_make` in the `catkin_workspace` directory (required even for python scripts because of the occasional messages you will add so a good habbit to have). Then run your script by first launching master in one terminal (`roscore`) and then in another terminal starting your node (`rosrun turtlebot_control publisher.py`).

You should be able to see the strings you are publishing using the `rostopic` tool. Make sure you do, otherwise you will not be able to move forward in the homework.

```
$ rostopic echo random_strings
data: HELLO WORLD!
---
data: HELLO WORLD!
---
data: HELLO WORLD!
---
...
```

Now let's try to write a node that will subscribe to our topic. Copy and complete the following code into a file called `subscriber.py`. Make sure to make the script executable again (the command is written above).

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo('received: %s', data.data)

def subscriber():
    rospy.init_node('subscriber', anonymous=True)
    # fill in the construction of the subscriber
    rospy.spin()

if __name__ == '__main__':
    subscriber()
```

Your code should use the Subscriber constructor `rospy.Subscriber()` as well as the topic name you used in the publisher (i.e. `random_strings`).

Keep your publisher node running, and run the subscriber node (so you should now have three things running). It should receive the messages you are publishing on the topic.

```
$ rosrun turtlebot_control subscriber.py
[INFO] [1483896494.733081]: received: HELLO WORLD!
[INFO] [1483896495.733632]: received: HELLO WORLD!
[INFO] [1483896496.733577]: received: HELLO WORLD!
...
```

Task: Make your publisher node publish your first and last name (as one single string). Then use the following command to record a rosbag of this and keep that file for submission with your homework.

```
$ rosbag record random_strings
```

Simluation in Gazebo

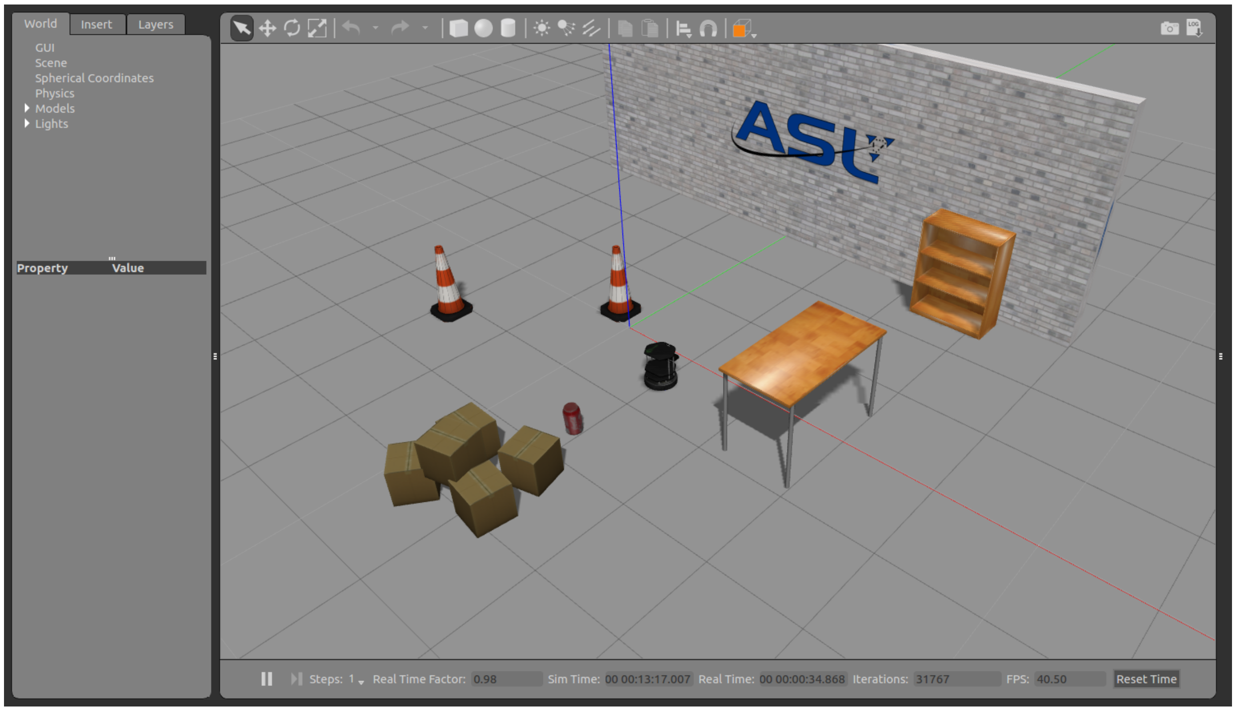
One of the most important (if not the most important) tool when developing robotic software is simulation. A good simulator is one that accurately represents the physics of your system as well as lets you seamlessly execute the same code you would run on the real system.

There is already a package in your workspace called asl_turtlebot. This package is a clone of the repository that is available online here: https://github.com/StanfordASL/asl_turtlebot. We encourage you to explore this package.

Lets try running a simulation of our turtlebot using roslaunch.

```
$ roslaunch asl_turtlebot turtlebot_sim.launch
```

You should see Gazebo start running a simlation that looks like this:



Steps: 1 Real Time Factor: 0.98 Sim Time: 00 00:13:17.007 Real Time: 00 00:00:34.868 Iterations: 31767 FPS: 40.50

Now let's try to run the parking controller you designed in problem 3 on the simulated turtlebot.

Add the following node to your `turtlebot_control` package and modify it so that it implements the controller you designed in problem 3. You should more or less be able to just copy and paste your code from problem 3 into the `get_ctrl_output` method. We also provide a copy of the controller template in the code release, but make sure you get familiar with what the rest of the code does. This is representative of what we expect you to be able to implement later this semester.

```python
#!/usr/bin/env python

import rospy
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import Twist
import tf

class Controller:

    def __init__(self):
        rospy.init_node('turtlebot_controller', anonymous=True)
        rospy.Subscriber('/gazebo/model_states', ModelStates, self.callback)
        self.pub = rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
        self.x = 0.0
        self.y = 0.0
        self.theta = 0.0

    def callback(self, data):
        pose = data.pose[data.name.index("mobile_base")]
        twist = data.twist[data.name.index("mobile_base")]
        self.x = pose.position.x
        self.y = pose.position.y
        quaternion = (
            pose.orientation.x,
            pose.orientation.y,
            pose.orientation.z,
            pose.orientation.w)
        euler = tf.transformations.euler_from_quaternion(quaternion)
        self.theta = euler[2]

    def get_ctrl_output(self):
        # use self.x self.y and self.theta to compute the right control input here
        cmd_x_dot = 0.0 # forward velocity
        cmd_theta_dot = 0.0
        # end of what you need to modify
        cmd = Twist()
        cmd.linear.x = cmd_x_dot
        cmd.angular.z = cmd_theta_dot
        return cmd

    def run(self):
        rate = rospy.Rate(10) # 10 Hz
        while not rospy.is_shutdown():
            ctrl_output = self.get_ctrl_output()
            self.pub.publish(ctrl_output)
            rate.sleep()

if __name__ == '__main__':
    ctrl = Controller()
    ctrl.run()
```

Task: Run a simulation with the target state $x = 1.0$, $y = 1.0$ and $\theta = 0.0$. Take a rosbag recording of both the state and control topics. You will submit this rosbag as part of your homework.

```
$ rosbag record /gazebo/model_states cmd_vel_mux/input/navi
```

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.