



# AA 274: Principles of Robotic Autonomy

## Problem Set 1

Due January 23 11:59pm

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/StanfordASL/AA274_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). To zip up your code, run `./make_submission.sh` inside the folder.

Your written part must be typeset (e.g., L<sup>A</sup>T<sub>E</sub>X or Word).

## 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 non-integrable set of differential constraints. In this problem set we will consider the simplest nonholonomic wheeled robot model, the unicycle, shown below in Figure 1.

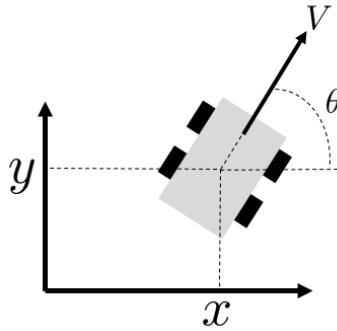


Figure 1: Unicycle robot model

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

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

The robot state is  $\mathbf{x} = (x, y, \theta)$ , where  $(x, y)$  is the Cartesian location of the robot center and  $\theta$  is its alignment 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  $\omega$  is the angular velocity, subject to the control constraints:

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

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 nonholonomic 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: Optimal Control


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:

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

where  $\lambda \in \mathbb{R}_{\geq 0}$  is a weighting factor and  $t_f$  is free. Consider the following initial and final conditions:

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

- (i)  Derive the Hamiltonian and conditions for optimality and formulate the problem as a 2P-BVP. Make sure you explain your steps, and that you include boundary conditions.
- (ii)  Complete `P1_optimal_control.py` to solve the 2P-BVP with the largest possible value of  $\lambda$  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 7 in Lecture 3 and vary the value of  $\lambda$  yourself**. Experiment with different initial guesses for the solution. Run `python P1_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. Further, the solution trajectory should be smooth.
- (iii)  Please include `optimal_control.png` in your write up.
- (iv)  Explain the significance of using the largest feasible  $\lambda$ .

- (v)  Validate your work. We will now simulate the car with the computed control history with and without the presence of disturbances (modeled as input noise).

Run `python sim_traj.py --data optimal_control --dist 1 --ctrl open`. This will add disturbance to the system, and apply the control sequence computed from (ii). Verify that the “disturbance-free” trajectory arrives at the goal while the “perturbed” trajectory deviates.

The plot will be saved as `plots/sim_traj_optimal_control_1_open.png`. Please include it in your write up.

**Note:** You will need the `scikits.bvp_solver` package (which is already installed in the virtual machine we provide - see question 5 below). See [https://pythonhosted.org/scikits.bvp\\_solver/](https://pythonhosted.org/scikits.bvp_solver/). Try installing directly using `sudo pip install scikits.bvp_solver`. 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.

## Problem 2: 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 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.

- (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:

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

where  $t_f = 15$ .

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

- (iii)  Complete the function `differential_flatness_trajectory` in `P2_differential_flatness.py` to compute the state-trajectory  $(x(t), y(t), \theta(t))$ , and control history  $(V(t), \omega(t))$ .
- (iv)  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 option (3).

Define a path parameter  $s$  (i.e., arc-length) for the trajectory by  $\dot{s}(t) = V(t)$ ,  $s(0) = 0$ . That is,  $s(t)$  denotes the distance traveled along the path up to time  $t$ . Thus, we can refer to the path as  $(x(s), y(s), \theta(s)) = (x(s(t)), y(s(t)), \theta(s(t)))$  which gives us the state  $(x, y, \theta)$  now parameterized by the arc-length  $s$ . Let's consider choosing an alternative velocity control as a function of  $s$ , call this new velocity profile  $\tilde{V}(s)$ . This corresponds to a new time history  $\tau(s)$  specified by  $\frac{ds}{d\tau} = \tilde{V}(s)$ ,  $\tau(0) = 0$ , which we may rearrange and integrate to yield

$$\tau(s) = \int_0^s d\tau' = \int_0^s \frac{ds'}{\tilde{V}(s')}.$$

The alternative angular velocity control input is given by



$$\tilde{\omega}(s) = \frac{d\theta}{d\tau} = \frac{d\theta}{dt} \frac{dt}{ds} \frac{ds}{d\tau} = \omega(s) \cdot \frac{1}{V(s)} \cdot \tilde{V}(s)$$

where  $\omega$  denotes the original angular velocity control input produced in part (ii).

Complete the rest of `P2_differential_flatness.py` to select  $\tilde{V}(s)$  so that it and  $\tilde{\omega}(s)$  satisfy the control saturation constraints  $|\tilde{V}(s)| \leq 0.5$  m/s and  $|\tilde{\omega}(s)| \leq 1$  rad/s. In particular, do not simply choose  $\tilde{V}(s)$  to be a uniform scaling of  $V(s)$  — you should only slow the robot down (i.e., select  $\tilde{V}(s) < V(s)$ ) when one of the above constraints would otherwise be violated. In order to get all of the state and control histories in terms of uniform timesteps in  $\tau$  we'll have to do a little interpolation (this is already implemented in the code).

Run `python P2_differential_flatness.py` to test your code. In particular, it will generate a plot of original and scaled path, arc-length and control history.

The plot will be saved to `plots/differential_flatness.png`. Use this to verify that the control constraints and boundary conditions are met.

- (v)  Please include `differential_flatness.png` in your write up.
- (vi)  Validate your work. Run `python sim_traj.py --data differential_flatness --dist 1 --ctrl open`. Using the generated plot, you can verify that the “disturbance-free” trajectory arrives at the goal while the “perturbed” trajectory deviates.

The plot will be saved as `plots/sim_traj_differential_flatness_1_open.png`. Please include it in your write up.

### Problem 3: Closed-loop Control I

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.

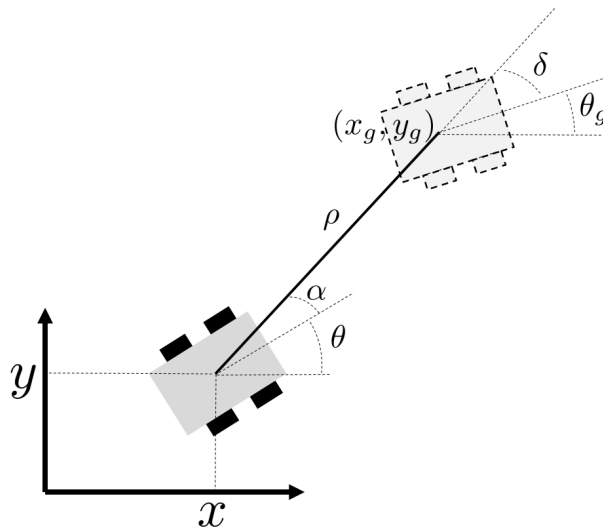


Figure 2: New coordinate system




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

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

The posture stabilization problem is now equivalent to driving the state  $(\rho, \alpha, \delta)$  to the unique equilibrium  $(0, 0, 0)$ . 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.

- (i)  Complete the control function `ctrl_pose` in the file `P3_pose_stabilization.py` and program the control law above.
- (ii)  Validate your work. Test your controller by implementing a forward, reverse, and parallel parking scenario. Edit `sim_parking.py` to specify the initial pose of each parking scenario. Then run `python sim_parking.py --parking <parking-type>` to simulate it. For example, run `python sim_parking.py --parking forward` to simulate the forward parking scenario. This will generate a plot of the position, angle, and control history. Use this to verify your code.
- (iii)  The plots will be saved in `plots/<parking-type>_parking.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  $\alpha$  and  $\delta$  remain within the range  $[-\pi, \pi]$ . This is needed since the control law has terms linear in  $\alpha$  and  $\delta$ . (2) Use the function `np.sinc` to handle values  $\alpha \approx 0$  in the control law for  $\omega$  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 to this problem.


## Problem 4: Closed-loop Control II

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 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:

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

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  $(V, \omega)$  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 function `ctrl_traj` in the file `P4_trajectory_tracking.py`.

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  $(V, \omega)$ .

WARNING: You must be careful when computing the actual control inputs due to the potential singularity in velocity discussed in Problem 2 — 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. In this case you should use a “reset” strategy to the nominal velocity if the true velocity gets too small.

HINT: At each timestep you may consider the current velocity to be that commanded in the previous timestep.

- (iii)  Modify your controller to *switch* to the pose stabilization controller from Problem 3 when you are “sufficiently close” to the goal position  $(5, 5)$ .

- (iv)  Validate your work. Run `python sim_traj.py --data differential_flatness --dist 1 --ctrl closed`.

Using the generated plot, you can verify that both the “disturbance-free” and “perturbed” trajectories arrive at the goal. Feel free to experiment with slightly different initial conditions and see if you can still track the path and reach your desired goal pose. You can change your initial pose by adding `--x0 <x0>, --y0 <y0>, --th0 <th0>` flags.

The plot will be saved as `plots/sim_traj_differential_flatness_1_closed.png`. Please include it in your write up.

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

## Problem 5: Robot Operating System

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.

For this homework, you will familiarize yourself to the ROS environment. Namely, the goals of this ROS question are:

- Setting up the VM/ROS.
- Creating a package in your catkin workspace and building it.
- Writing publisher/subscriber nodes and executing them.
- Use the rostopic tool.
- Familiarize yourself with the Gazebo environment.

There is a lot to read, please read the entire text first before starting on the problem!

### Setting up

Setting up ROS and getting it to work with the turtlebots requires installing several packages on a linux system. For this class, we recommend using a virtual machine (VM) that we provide which has all the required software preinstalled.

Alternatively, you can natively install Linux alongside your normal OS as a dual boot and install the required packages yourself. This may give better performance, especially for graphics rendering. However, as there can be driver issues and other edge cases that can come up when installing linux on hardware, *we can only officially support the VM* in terms of providing help with your setup.

To set up the VM on your computer, follow the instructions here: <https://docs.google.com/document/d/1Ge94AbHGz5uqgsjbWyG7QK-7AHh-XMkchFFKmbbinNw/edit?usp=sharing>

Let's now create a package in your workspace!

### The Basic Publisher and Subscriber

Let's go to the `src` folder in your catkin workspace, and then make a new package.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg hw1 std_msgs roscpp rospy
$ cd ~/catkin_ws
$ catkin_make
```

The `catkin_create_pkg` command creates a new package called `hw1`, and tells catkin that `std_msgs`, `roscpp`, and `rospy` are the packages that our package depends on. You will need to `catkin_make` in the your catkin workspace in order to build your newly created package.

Now create a `scripts` folder in your new `hw1` package; this is where catkin expects all python source code for the project to be.

```
$ cd ~/catkin_ws/src/hw1
$ mkdir scripts
```

Then create a file called `publisher.py` in the scripts directory 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 HERE ###
        # your code to publish a string
        # goes here
        ### END OF YOUR CODE ###
        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 it is a good habit to have). Then run your script by first launching master in one terminal

```
$ roscore
```

Then in another terminal starting your publisher node

```
$ rosrn hw1 publisher.py
```

To check if your messages are being published, you need to use the `rostopic` tool <http://wiki.ros.org/rostopic>. In particular, you will need to `echo` the topic you want to see. In a separate terminal, type:



```
$ rostopic echo random_strings
```

Make sure you are able to see the published messages, otherwise you will not be able to move forward in the homework. You should see something like this (if the published message was "HELLO WORLD!"):

```
data: HELLO WORLD!
---
data: HELLO WORLD!
---
data: HELLO WORLD!
---
...
```

Note that if your script produces errors, they will not necessarily display if you run your node using `roslaunch`. This is because ROS will capture the error messages. You can get them by running your script directly instead.

```
$ cd ~/catkin_ws/src/hw1/scripts
$ python publisher.py
```

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)
    ### YOUR CODE HERE ###
    # fill in the construction
    # of the subscriber
    ### END OF YOUR CODE ###
    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 in another terminal (so you should now have three things running). It should receive the messages you are publishing on the topic.

```
$ roslaunch hw1 subscriber.py
[INFO] [1483896494.733081]: received: HELLO WORLD!
[INFO] [1483896495.733632]: received: HELLO WORLD!
[INFO] [1483896496.733577]: received: HELLO WORLD!
...
```

## Simulation 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.

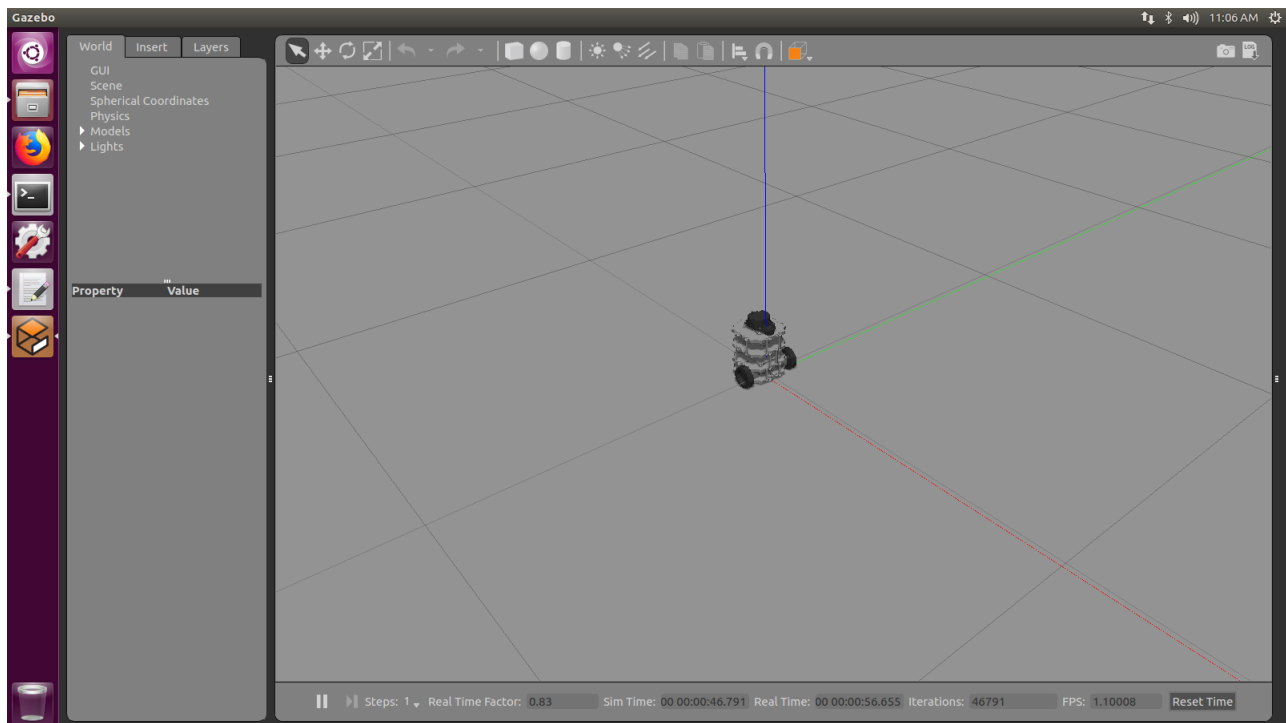
There is already a package in your workspace called `asl_turtlebot`. This package is a clone of the repository that is available online here: [https://github.com/StanfordASL/asl\\_turtlebot](https://github.com/StanfordASL/asl_turtlebot). We encourage you to explore this package. Start by pulling any new changes we may have published there.


```
$ cd ~/catkin_ws/src/asl_turtlebot
$ git pull
```

Let's try running a simulation of our turtlebot using `roslaunch` (make sure you include the 3 or you will be running last year's version!).

```
$ roslaunch asl_turtlebot turtlebot3_sim.launch
```

Feel free to take a look at the contents of `asl_turtlebot/launch/turtlebot3_sim.launch`. You should see Gazebo start running a simulation that looks like this:







- (i)  Make your publisher node publish your first and last name as one single string (e.g., `'marco`

`pavone`'. Then use the following command to record a rosbag of it and keep that file for submission with your homework.

```
$ rosbag record -O ~/catkin_ws/name.bag random_strings
```

Note: the `-O name.bag` flag specifies the name of the bag. Without it, it will default to the current timestamp. See <http://wiki.ros.org/rosbag/Commandline> for more details.

- (ii)  You can verify the contents of your rosbag. What is the command to play back a rosbag with name `bagname.bag`?
- (iii)  What command must you run after writing a new python node, and before running `catkin_make`? What does it do?
- (iv)  What is the command to show information about active topics?
- (v)  Now let's try to run the parking controller you designed in Problem 3 on the simulated turtlebot.

In the `asl_turtlebot` package, you should have a node called `controller.py`. Modify it so that it implements the controller you designed in Problem 3 and has target state  $x = 1.0$ ,  $y = 1.0$  and  $\theta = 0.0$ .

You should more or less be able to just copy and paste your code from Problem 3 into the `get_ctrl_output` method. Make sure you have a look at the rest of the code too. This is representative of what we expect you to be able to implement later this quarter.

After launching Gazebo, in a separate terminal, run your `controller.py` node in the same way you ran your publisher and subscriber node before.

Take a rosbag recording of both the state and control topics. You will submit this rosbag as part of your homework.

```
$ rosbag record -O ~/catkin_ws/gazebo.bag /gazebo/model_states /cmd_vel
```

NOTE: Depending on your computer, it might have a hard time rendering Gazebo's environment from inside a virtual machine. You can opt to run the problem set without the Gazebo graphics by passing an argument to the launch file.

```
$ roslaunch asl_turtlebot turtlebot3_sim.launch gui:=false
```

We also implemented for you a simple visualizer so that you can still see your turtlebot moving. It should leave a 'trail' as it moves around. See Figure 3.

```
$ rosrun asl_turtlebot gazebo_plot.py
```

## Compiling your submission

Compile your written responses into a single typeset pdf document.

The code, generated figures, and data files should be compiled into a single zip file. Once you are finished with your assignment, you can run `./make_submission.sh` to automatically compile the required files.

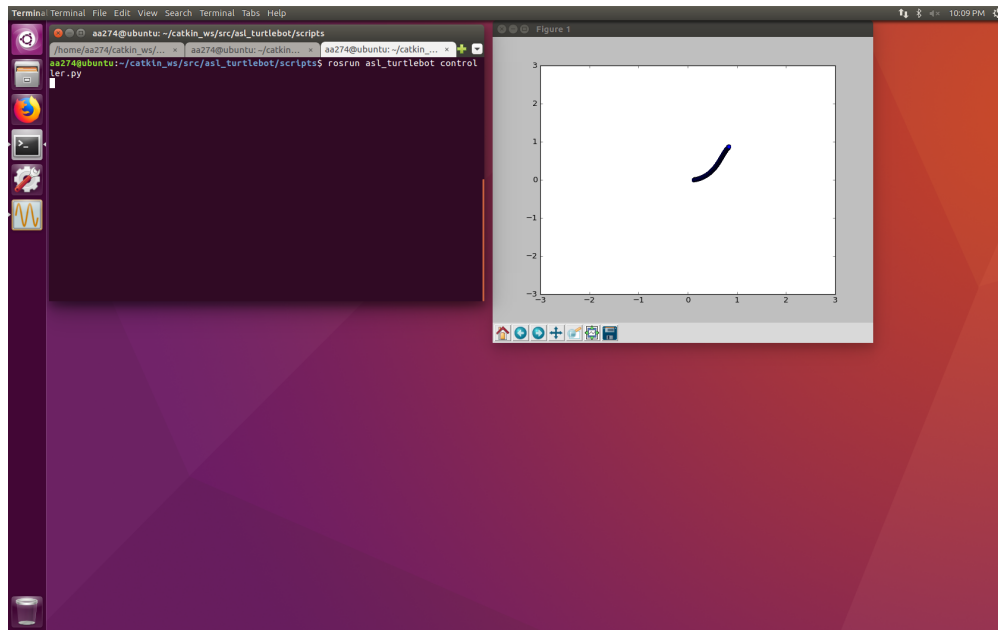


Figure 3: Plotting without Gazebo gui

If you named your folders and file as suggested in this document, and are working from the provided VM, this script should work without issues. Whether you use the script or zip the files yourself, please check that the following files are stored in the zip file before submitting:

1. `P1_optimal_control.py`
2. `P2_differential_flatness.py`
3. `P3_pose_stabilization.py`
4. `P4_trajectory_tracking.py`
5. `sim_parking.py`
6. `sim_traj.py`
7. `utils.py`
8. `~/catkin_ws/src/hw1/scripts/publisher.py`
9. `~/catkin_ws/src/hw1/scripts/subscriber.py`
10. `~/catkin_ws/src/asl_turtlebot/scripts/controller.py`
11. `~/catkin_ws/gazebo.bag`
12. `~/catkin_ws/name.bag`

## 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.