Homework 8: Pose Estimation and Waypoint Motion

Due: November 20, 2014

Hand in your code on Owlspace before class, and bring a printout of the student code to class. Please only print your functions, do not print the distribution code. All of your code should be contained in a single python file called PS08_*netid*.py. This assignment will be due at the beginning of class, and you will need to demonstrate your pose estimation and waypoint navigation behavior during class.**Be sure to comment your code** so that the graders understand what you were trying to do.
Questions? email `engi128-staff@rice.edu`.

Pose Estimation and Waypoint Navigation

For this assignment, you will write software that lets the robot keep track of its current pose, and a waypoint navigator to command the robots to go to specified locations. You will test your software by programming the robot to move forward a fixed distance, turn left 90 degrees, then move forward the same fixed distance again and repeat four times to drive in a square.

Sound familiar? This is similar to the data collection assignment you had for PS04, but now you have a velocity controller, a pose estimator, and a waypoint motion controller. These systems will make your motions much more accurate. The goal of this assignment is to measure how much the robot's performance varies across trials.

Getting Started

Download the distribution code from the website. Program the following modules into flash memory on your robot:

```
owlpy.connect()
```
... Robot Startup Stuff ...
```
owlpy>loadrun PS08_netid.py PS08Libs.zip
```

1 Estimating Pose

The first step is to estimate the pose of the robot. Recall that the pose is a tuple of the position and angle of the robot on the 2-dimensional plane: pose $= (x, y, \theta)$. We can keep a running estimate of the robot's pose by using the wheel encoders to measure distance. We're going to write this software backwards, starting with the function that initializes the pose, them moving on to the functions that access the pose, print the pose, and finally update the pose. Why work in this way? Because deciding how you want to use a software module is one of the best ways to figure out its structure. In other words, defining the *interface* to a module constrains its design, and makes it easier to see how to build it. (Remember "interfaces"? From, like, 8,000 weeks ago?)

1.1 Review `pose_init()`

This function builds the state of your pose estimator. This state is stored in a dictionary. We've given this dictionary global scope, because there is only one pose state; *i.e.* the robot can only be in one place at a time. Using global variables is dangerous, and is almost always bad form, so don't use them in your code. In order to access a global variable within a function, you need to tell Python you are sure you want to do this. The **global** keyword tells python that the following variables are defined globally, and that this function should use these global variables instead of creating a new local variable.

1.2 Review the "Getters" and "Setters" of the pose state

Storing all of the pose estimator's state in a single variable is good, but we need a convenient way to access parts of it. That is what "getters" and "setters" are for: they access a particular piece of data from a larger, more complex data structure. We've written three getters:
`poseX.get_pose(),poseX.get_theta(),poseX.get_odometer()`

All of these return a single floating-point number, except for `pose.get_pose()`, which returns a tuple of floats: (x, y, θ). There is only one setter:
`pose.set_pose(x, y, theta)`

which takes as its argument the three components of the pose.

1.3 Write `pose_update()`

Recall the derivation from the lecture:

$$\Delta d = \frac{d_R + d_L}{2} \tag{1}$$
$$\Delta\theta = \frac{d_R - d_L}{b} \tag{2}$$
$$x' = x + \Delta d \cdot cos(\theta) \tag{3}$$
$$y' = y + \Delta d \cdot sin(\theta) \tag{4}$$
$$\theta' = \theta + \Delta\theta \tag{5}$$

Use these equations to write `pose_update(pose_state)`. The trigonometric functions sin and cos are part of the `math` package, call them with the syntax: `math.sin(angle)`. We've provided a `marh2.normalize_angle(theta)` function that takes an angle theta, and returns the same angle, but within the bounds of $-\pi < \theta \leq \pi$. Use it on θ after you update it in Equation 5, but before you store it in `pose_state`

You will also make an *odometer*, a counter of the total distance the robot has travelled since it was reset. It always increases, even when the robot is driving backwards. It should be a float. You will need to update it in this function, and store its current value in the pose state so that `poseX.get_odometer()` returns the correct value.
Hand-in: Write `pose_update()` (8 pts).

1.4 Run a Sanity Check

Run your pose estimator. The distribution code prints the pose every $250ms$. You may test your program by imagining a coordinate axis on the floor. Put your robot at a pose of $(0, 0, 0)$, which is the origin, facing the x-axis. Move your robot forwards, this should increase the x value. Rotate it $\frac{\pi}{2}$ degrees counter-clockwise (left turn). This should increase θ to $\frac{\pi}{2}$. Now push it forward again. This should increase the y value. If you don't get these kind of output, check your code for bugs, it was quite accurate on my desk. Maybe you need a better desk.
Hand-in: Sanity check passed?(1 pt).

2 Waypoint Navigation

Recall from lecture that you can program waypoint navigation by combining translational velocity and rotational velocity. In this section, you will refine this idea into a slick motion controller, then measure how well it works.

2.1 Review the Motion Controller API

The motion controller API is int he zip file we've provided. You don't need to unzip it to program the robots. You can read files in the zip file without unzipping them: `motionX.init()`, `motionX.update()`,`motionX.is_done()`,`motionX.set_goal(goal_pos, tv_max)`, `motionX.get_goal()`

2.2 Write helper functions

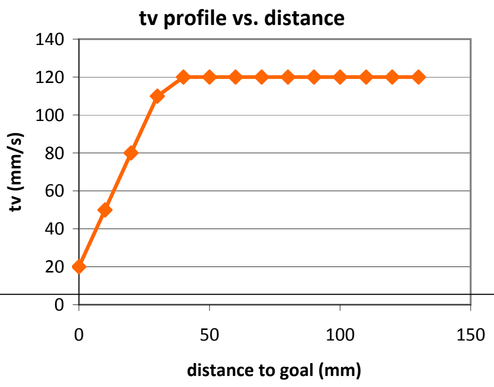
You'll need three functions to compute the distance and the direction to the goal position. Write a function called `topolar(x, y)` to convert cartesian coordinates to polar coordinates and return a tuple of the form: `(r, theta)`. We use this to compute the distance to the goal. You can compute x^2 in two ways: `x**2.0` or `x*x`. Use the second way, it's faster. The square root function is in the math package, access it with the statement `a = math.sqrt(b)`.
Hand-in: Write `topolar(x, y)` **(2 pts).**

Second, write `compute_goal_distance_and_heading()`. This returns a tuple of the form `(goal_distance, goal_heading, robot_heading)`. The goal heading is the angle between the current (x, y) position and the goal (x, y) position. In other words, this is the heading along which the robot must travel. It **is not** the angle the robot needs to rotate to point itself at the goal position. That is the next function.
Hand-in: Write `compute_goal_distance_and_heading()` **(6 pts).**

Finally, write a third function called `smallest_angle_diff(current_angle, goal_angle)` that computes the smallest angle difference from the `current_angle` to the `goal_angle`. This is the angle that the robot needs to rotate from its current heading to the goal heading, `heading_error`. Normalize this error to lie between $-\pi < \theta \leq \pi$, in other words, compute the most direct rotation to point the robot towards the goal position. The robot should never robot more than π or $-\pi$. **Computing this angle is tricky.** Test this function carefully. Be sure to test with start and goal positions in multiple quadrants. Pay careful attention as all the different angles wrap around from 0 to 2π and $-\pi$ to π. Review the lecture notes on global coordinates before you start this section.
Hand-in: Write `smallest_angle_diff()` **(3 pts).**

2.3 Build a Controller for tv

We want the robot to slow down as it approaches the goal position. In order to do this, we want to command a velocity profile of the form:



$$tv_{temp} = k_{tv} \cdot d + tv_{min} \tag{6}$$

$$tv = \begin{cases} tv_{temp} & \text{if } tv_{temp} \leq tv_{max} \\ tv_{max} & \text{otherwise} \end{cases} \tag{7}$$

You need to write `motion_controller_tv(d, tv_max)` We've provided these parameters for you:
tv_{max} = `tv_max`, an argument to the function
tv_{min} = `MOTION_TV_MIN`
k_{tv} = `MOTION_TV_GAIN`
Hand-in: Write `motion_controller_tv()` **(3 pts).**

2.4 Build a Controller for rv

The controller for rv is simpler:

$$rv = k_{rv} \cdot \theta_{\text{rotate}} \tag{8}$$

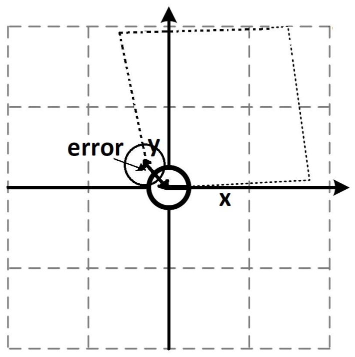
Set k_{rv} = `MOTION_RV_GAIN` and use the `math2.bound(rv, MOTION_RV_MAX)` function to limit the values of rv to `MOTION_RV_MAX`.
Hand-in: Write `motion_controller_rv()` **(3 pts).**

2.5 Test with the waypoint list

Build a list to store a series of *waypoints*, which are (x, y) tuples that are the points that the robot is supposed to visit. We've given you the example list I used to test with the 1 ft-square tiles in my office. Units are in millimeters. Make your list appropriate for the tiles you have to work with. The list is on line 165 of the code.
Hand-in: Waypoint list working?(1 pt).

3 Data Collection

Find a floor with a regular grid tile pattern. We'll use these to make collecting data easier. Note how many tiles can fit into a meter, probably three, if your tiles are one foot squares. This distance will be our reference distance, d. Place the robot at a tile intersection. This will define our coordinate system:



The program we've given you waits for the user to press the red button to load the waypoint list. Modify this waypoint list to move the robot in a square 1 m on a side. Measure the actual final position, (x, y), of the robot. Measure from the center of the robot. This is one experimental trial. Move the robot back to the starting position and run the program for a total of 10 trials. Each robot will perform differently in these tests and the measurements will vary. Make a scatter plot of the (x, y) positions of the robot.
Hand-in: A plot of the final positions of the robot after moving in a 1 m square.(6 pts).