PS01: Braitenberg Vehicles, Pose Estimation, Waypoint Navigation v1.1

Due: Feburary 3rd, 2015

1 Braitenberg Vehicles

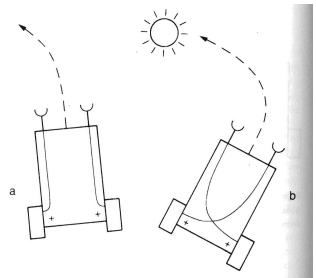


Figure 3
Vehicles 2a and 2b in the vicinity of a source (circle with rays emanating from it). Vehicle 2b orients toward the source, 2a away from it.

In this section, you will build several Braitenberg vehicles. We've given you a sample file to get started, PS01_Braitenberg.py. We don't want you to struggle with the Python programming language, so we've given you a lot of code to start with. Look for the lines: "`# student code start`", and the matching end line to know where to put your code.

1.1 Write motion helper functions

Finish the four motion helper functions:
`move_stop(time)`,
`move_forward(time)`,
`move_rotate_right(time)`, and
`move_rotate_left(time)`.
Each of these functions takes one argument, `time`. They move the wheels in the proper directions for the desired time to produce motion. Use the motor PWM function:

```
rone.motor_set_pwm('l', pwm_arg)
rone.motor_set_pwm('r', pwm_arg)
```

To directly control the left or right motor's PWM, where `pwm_arg` is the desired pwm. We have defined a global variable, `MOTOR_PWM` to use as the PWM value in all of these helper functions. This way, if you want to change the PWM, you can do so in one place. PWM values less than 60 won't produce motion, and PWM of more than 80 will make the robot move too fast. Use `sys.sleep(time)` to wait for a certain amount of time, units are milliseconds.

Hand-in: Write `move_stop(time)`, `move_forward(time)`, `move_rotate_right(time)`, `move_rotate_left(time)`**. (2 pts each)**

1.2 "Boredom" (Square motion)

Use your motion functions to write `square_motion()`, which will drive the robot in a 1 foot $\times$ 1 foot square. Your answer must use a for loop, and your robot must use `move_stop()` after the square is finished.

Hand-in: Use the motion helper functions and a for loop to write `square_motion()`. (5 pts)

1.3 "Love" (Move towards light)

We will use these motion functions and the light sensors to drive the robot towards light. Use these functions from the r-one library to read the light sensors:

```
light_fl = rone.light_sensor_get_value('fl')
light_fr = rone.light_sensor_get_value('fr')
```

1.3.1 Write `light_diff`

This function should read the two front light sensors, then return the difference between the left sensor and the right sensor: diff $=$ left $-$ right.

Hand-in: Write `light_diff()` (3 pts)

1.3.2 Complete `light_follow()`

Use your `light_diff()` function to complete `light_follow()`. We've given you a bit of starter code. Use an `if, elif, else` structure and your motion primitives from Section 1.1 to make the robot drive towards the light.

Hand-in: Use the motion helper functions, `light_diff()`, and a `if, elif, else` structure to write `light_follow()`. Explain why we are storing the initial diff value in `diff_start` (10 pts)

Check-off: Demonstrate your robot moving towards (loving?) a flashlight.

1.4 "Fear": Avoiding Obstacles with the Bump Sensors

Driving towards things is only half of what robots do. Now let's drive *away* from things. In this section, we will move away from obstacles we run into using the bump sensors.

We've given you three nifty functions: `bump_left_get_value()`,`bump_front_get_value()`, `bump_right_get_value()`. They each return a boolean variable indicating if the bump sensor is pressed from the indicated direction. Finish `bump_avoid()` to move the robot away from collisions. We've given you a bit of starter code. Use an `if, elif, elif, else` structure and your motion primitives from Section 1.1 to finish this. This answer will look similar to the answer from Section **??**.

Hand-in: Use the bump sensor helper functions and a `if, elif, elif, else` structure to write `bump_avoid()` (10 pts)

1.5 "Fear, Distantly": Avoiding Obstacles with the IR Sensors

Running into walls is sooo last problem. In the fourth half of this problem set, we will avoid walls altogether using the IR system to detect them from a distance.

We've given you a nifty function: `obstacle_detect()`. It returns a tuple of boolean variables indicating where the obstacle is relative to the front of the robot. Move your hands around the robot and watch the output. The range is pretty far, you will need to get your robot away from things around you to see the program work.

Use `obstacle_detect()` to move the robots away from walls. Use an `if, elif, else` structure and your motion primitives from Section 1.1 to finish this. This answer will look similar to the answer from Section 1.4, copy and paste your code and modify it, don't start from scratch. We have given you the code to call `obstacle_detect()` and unpack the tuple into three variables: `obs_front, obs_left, obs_right`. After they have been unpacked, you can forget about the tuple, and use these booleans just like any other variable.

Hand-in: Use the motion helper functions, `obstacle_detect()`, and a `if, elif, else` structure to write `obstacle_avoid()` (10 pts)

2 Pose estimation

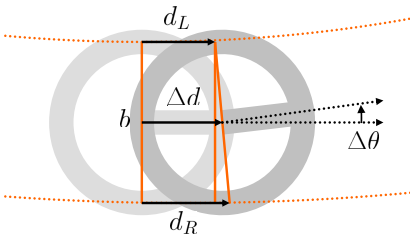
2.1 Getting Started

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

```
owlpy.connect()
... Robot Startup Stuff ...
owlpy>loadrun PS01_netid.py PS01Libs.zip
```

2.2 Derive the pose update equation

Recall the derivation we started in lecture for the incremental pose update equation:



Complete this derivation. Solve for (x', y', θ') in terms of (x, y, θ), d_L, d_R, and b. Assume the change in heading, $\Delta\theta$, is very small, so you can make use of the trigonometric approximation $tan(\theta) \approx \theta$.

Hand-in: Your derivation.

2.3 Implement the pose update function

Complete the pose estimator in Python on your robot. 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 `math2.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 **??**, 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()` **(20 pts).**

2.4 Run a Sanity Check

Run your pose estimator program. 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 $\pi/2$ degrees counter-clockwise (left turn). This should increase θ to $\pi/2$. 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.

Check-off: Working pose estimator(2 pts).

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.

2.5 Review the Motion Controller API

Review the motion controller API we've provided: `motionX.init()`,`motionX.update()`, `motionX.is_done()`,`motionX.set_goal(goal_pos, tv_max)`,`motionX.get_goal()`

2.6 Write helper functions

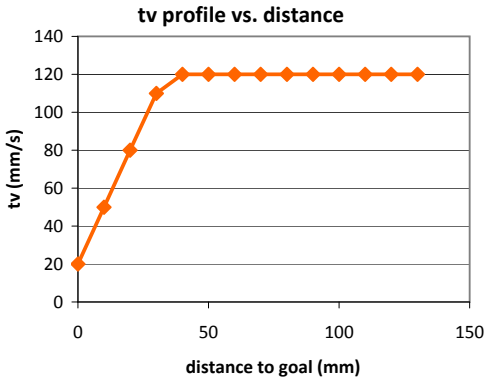
You'll need three functions to compute the distance and the direction to the goal position. Write a function called `topolar(x, y)` to convert cartesian coordinates to polar coordinates and return a tuple of the form: `(r, theta)`. We use this to compute the distance to the goal. You can compute x^2 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)` **(4 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()` **(10 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()` **(6 pts).**

2.7 Build a Controller for tv

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



$$tv_{temp} = k_{tv} \cdot d + tv_{min} \tag{1}$$

$$tv = \begin{cases} tv_{temp} & \text{if } tv_{temp} \leq tv_{max} \\ tv_{max} & \text{otherwise} \end{cases} \tag{2}$$

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()` **(6 pts).**

2.8 Build a Controller for rv

The controller for rv is simpler:

$$rv = k_{rv} \cdot \theta_{\text{rotate}} \tag{3}$$

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()` **(4 pts).**

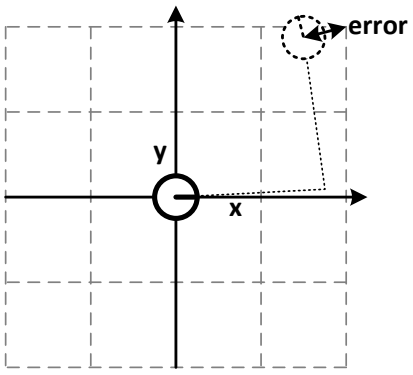
2015-01-28

2.9 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.(8 pts).

4 Write-up

Write a **one-page** report about your experimental results. Longer reports **will not** be graded. Be sure to include your data plots from Section 3. Answer the following questions in your report:

- Why do you think your robot does not move straight, even with your velocity controller and pose estimator? (answers may vary by robot and location)?
- What kind of sensors could improve your robot's ability to get to waypoints with higher precision?