Homework 7: Follow-the-Leader

Due: November 6, 2014

Hand in your code on Owlspace before class, and bring a printout of the student code to class. Please only print the student functions, do not print the distribution code, the neighbor code, or the velocity code. All of your code should be contained in a single python file called PS07_*netid*.py. **Be sure to comment your code** so that the graders understand what you were trying to do. Questions? email `engi128-staff@rice.edu`.

Follow the Leader

For this assignment you will implement a Follow-the-Leader program that uses the infrared (IR) and radio communication systems on the robots. You will need to test your code in groups of three. There will be three robot roles:

- **Remote Robot:** Serves as a remote control to send radio commands to the leader robot.
- **Leader Robot:** Accepts radio commands and moves accordingly, while simultaneously broadcasting its ID over IR.
- **Follower Robot:** Receives IR messages, and moves towards a leader robot.

HONOR CODE: You must work with two other people on this assignment. Each student needs to do all of the problems.
HINTS: There is a lot of code in the **PS07.py** distribution, but you don't have to understand all of it. Don't get lost in the distribution code to try and understand the functions we are asking you to write. Read the question carefully, the longest function is about 10 lines, most are shorter.

1 Getting Started

1.1 Loading the supplied modules

We've abstracted your velocity controller from PS06, and put it into a separate file, `velocity.py`. We've also written a LED animation package, `leds.py` and a skeleton neighbor system (which you will finish in this lab) called `neighborsX.py`. You will need to program these files onto your robot. They come in a provided ZIP package file called `PS07Libs.zip`. The program and load-run commands have the ability to take this zip file and program all of the files within it onto the robot:

```
owlpy.connect()
... Robot Startup Stuff ...
owlpy>loadrun PS07_netid.py PS07Libs.zip
```

You have five functions to interact with the velocity controller:

```
velocity.init(kff, kff_offset, kp, ki)
velocity.update()
velocity.get(motor)
velocity.set(motor, velocity)
velocity.set_tvrv(tv, rv)
```

The `velocity.update()` function needs to be called at regular intervals to run the controller. We've already put the call to this function in the framework we're giving you. Please look at how we've structured this code to call this function repeatedly. Note that we call the update function more than it actually runs, it regulates its timing by itself and runs only at semi-regular intervals. There are three functions for LED animations:

```
leds.init()
leds.update()
leds.set_pattern(color, pattern, brightness)
```

The `leds.update()` function also needs to be called at regular intervals.

Run the `PS07_velocity_test.py` program and spend a few minutes testing these velocity and LED functions. You can easily command controlled velocities from other functions now. The units of the arguments to these functions are given in mm/sec and mrad/sec. The `velocity.set_tvrv(tv, rv)` function is the best way to control robot motion.

1.2 Communications with the r-one

Find a partner. First test the radio functions:

```
rone.radio_send_message(msg)
rone.radio_get_message_usr_newest()
```

from the interactive prompt to send and receive radio messages back and forth to your partner. Note the largest message that can be sent is 30 characters long. Likewise, test the:

```
rone.ir_comms_send_message()
rone.ir_comms_get_message()
```

functions to send and receive IR messages with your partner. Both of these functions return `None` if there is no message, but sure to check for this. The `rone.radio_get_message()` function returns a string is a message was received, and `None` if one was not. The `rone.ir_comms_get_message()` function returns a tuple of:

```
(nbr_id, receivers_list, transmitters_list, range)
```

This is the id of the neighboring robot, a list of the receivers which received this message, a list of the transmitters of the neighboring robot that the message was transmitted from, and the range estimate to the neighbor. We will use these lists to compute the bearing and orientation of the transmitting robot.

1.3 The `PS07.py` distribution code

We've given you a lot of code in this distribution. There is a neighbor system, and velocity package, and a LED animation package. The distribution code also includes the main loop. When the program starts, it checks the buttons to select a different role for the robot: red = remote control robot, green = leader robot, blue = follower robot. Once a mode is selected, the robot blinks the lights of the corresponding color. When the robot is in each mode, circling lights indicate that the robot is *idle*, and solid lights indicate that it is *active*. In the remote control mode, the robot is idle when there are no buttons being pressed, and active when a button is pressed. In the leader mode, the robot is idle when it is not receiving a radio message, and active when it is. In the follower mode, the robot is idle when it is not receiving an IR message, and active when it is. These lights will help you debug your program when the robot is disconnected from the computer.

2 Remote Control Robot

The remote control robot will process button pushes from a user holding it. When a person presses buttons on the Remote Robot, it sends a radio signal to control the Leader Robot. Keep in mind all radios are on the same channel so be careful when testing your program with other robots nearby.

2.1 Write `check_buttons()`

Complete the `check_buttons` function to read the buttons and return a string of characters consisting of 'r', 'g', 'b'. For example, if the user presses only the red button, the message would be 'r'. If multiple buttons are pressed, the string might be 'rg', 'rb', or even 'rgb'.

When you have this function implemented correctly, you should be able to run the distribution code, press a button and place the robot in one of three modes: remote(red), leader(green), follower(blue). **Hand-in: Write** `check_buttons()` **(3 pts).**

2.2 Write `test_radio_receive()` and `test_ir_receive()`

Complete the `test_radio_receive()` and `test_ir_receive()` functions. This test functions run forever, and print IR or radio messages if they receive them. When you get a message, print it and turn on the green LEDs. If not, turn on the red LEDs. When you have this function implemented properly, you should be able to receive radio and IR messages from your partner's robot running in remote mode. **Hand-in: Write** `test_radio_receive()` **and** `test_ir_receive()` **(4 pts each).**

But wait, there's more...

3 Leader Robot

The leader robot will receive radio commands from the remote control and move according to the button presses.

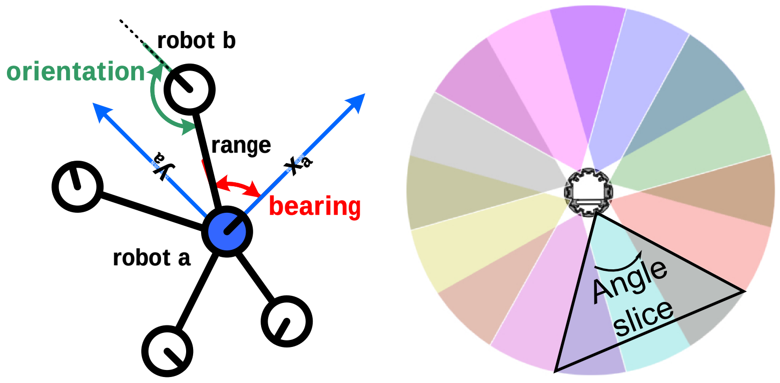
3.1 Write `leader_motion_controller()`

This function takes the received radio message and controls the motors. It returns a tuple of `(tv, rv)`. In the distribution code, this tuple is sent to the `velocity.set_tvrv(tv, rv)` function to control the motors. Use the `FTL_TV` and `FTL_RV` global variables as the velocities you are commanding. You can decide how you want to process the buttons. One option uses red for left wheel forward, blue for right wheel forward[1]. I prefer green for forward, red to rotate left, and blue to rotate right. You can do anything you want[2]. When you have this function implemented properly, the motors will turn, and you will have a remote controlled robot! **Hand-in: Write** `leader_motion_controller()` **(6 pts).**

But wait, there's more...

[1]This is like the classic 1980's video game *"Battlezone"*. A game worthy of Googling and playing.

[2]If you want to be hardcore, you can use the accelerometer x and y axes to compute a linear tv and rv, but transmitting and processing will be tricky. (and no, there is no extra credit)



(a) Local coordinate system. (b) Sectors from the IR receivers.

Figure 1: a: The local coordinate system of the blue robot, 'a'. The bearing to robot 'b' is measured in robot 'a's coordinate system. The orientation of robot 'b' is measured from the line between 'a' and 'b'. The range. is the distance between the two robots, but this measurement isn't very accurate. **b:** Each IR receiver can receive from a wide angle. The sectors are designed to overlap so that a message from a transmitting robot will be received on either one or two receivers. Depending on which receiver(s) get the message, the bearing of the transmitting robot can be measured. measuring orientation works in a similar way, but by noting which transmitters on the neighboring robot sent the message.

4 Follower Robot

The follower robot will receive messages from a nearby leader robot over the InfraRed (IR) communications system. Once a message has been received, you will need to compute the bearing and orientation to the leader robot. IR signals are directional and will not work if occluded.

4.1 Write `compute_bearing()` and `compute_orientation()`

We've done most of the hard work of receiving and processing the IR message for you. You will need to do the last step and compute the bearing and orientation of the neighboring robot. Figure **??** shows the local coordinate system around a robot.

In lecture, we talked about the measurement of bearing and orientation using the IR communications system. Now let's put that knowledge to use. Recall that the robot can transmit and receive messages on many different transmitters and receivers. Figure **??** shows the sectors of the 8 receivers, and how they overlap. When a message is received, it will be received on one or more receivers. We need a way to convert the list of receivers to a bearing and the list of transmitters to an orientation.

The `rone.ir_comms_get_message()` function returns a tuple of `(message, receivers_list, transmitters_list)`. The transmitters and receivers are labeled on the top of your robot as {T1 - T8} and {IR1 - IR8} respectively. These indices start from 0 in the software, so the transmitter and receiver lists will contain numbers from {0 - 7}. In order to compute the bearing, or the orientation you will need to average the angles from the sectors that a message was received on.
The IR receivers are located at angles of: $\{\frac{1\pi}{8}, \frac{3\pi}{8}, \frac{5\pi}{8}, \frac{7\pi}{8}, \frac{9\pi}{8}, \frac{11\pi}{8}, \frac{13\pi}{8}, \frac{15\pi}{8}\}$
The IR transmitters are located at angles of: $\{\frac{0\pi}{4}, \frac{1\pi}{4}, \frac{2\pi}{4}, \frac{3\pi}{4}, \frac{4\pi}{4}, \frac{5\pi}{4}, \frac{6\pi}{4}, \frac{7\pi}{4}\}$.
Note that we usually use angles within $[\pi, -\pi)$, but using larger angles will be ok for this part of

the assignment, they will be normalized to $[\pi, -\pi)$ by the trigonometric functions.

It is easy to compute the angle of a single transmitter or receiver with a simple mathematical function – just multiply the transmitter/receiver number by the right amount to get the angle from above (you might need an offset, too). However, averaging angles is tricky. One way to do it is to break each angle into component unit vectors, compute the vector summation, then compute the angle of the resultant vector. You want to implement the following equations:

$$y = \sum_i sin(\text{angle}_i) \tag{1}$$

$$x = \sum_i cos(\text{angle}_i) \tag{2}$$

$$\text{avg} = atan2(y, x) \tag{3}$$

You can use the syntax:

```
for r in receivers_list:
```

to make a for loop that iterates through all elements in the list, adding up th ecomponents of each one as they proceed. **Hand-in: Write** `compute_bearing()` **and** `compute_orientation()` **(5 pts each).**

4.2 Write `follow_motion_controller()`

Finally, the good stuff! The last function you need to implement is the heading controller in the `follow_motion_controller()` function. This is responsible for taking a neighbor, finding the bearing, and steering the follower robot in the proper direction. Remember, the bearing is measured in the robot's local coordinate system, see Figure **??**.

You want to design a controller to set the translational velocity, , `tv`, and the rotational velocity, `rv`, to steer the follower robot towards the leader robot. To accomplish this, we'll compute the *angular error*: θ_{error}, between the our current heading and the bearing of the leader robot. If the leader robot is to the left, the error should be positive. If the leader is to the right, the error should be negative, *i.e* $\theta_{error} \leq \pi$ then you rotate left, otherwise rotate right.

Once you have determined θ_{error}, you need to compute the rotational velocity (rv) as a function of this error, $rv = K_{rv}\theta_{error}$. Positive values of rv will make the robot turn to the left, negative values will make it turn to the right. Your function will require you to tune K_{rv} to get the best performance. This will behave similar to your velocity controller; making this constant too small will make your follower slugish, but making it too large will make the system unstable. **Hand-in: Write** `follow_motion_controller()` **(10 pts).**

Congratulations! You've built a remote-follower-leader program! Have some fun with it!

But wait, there's more...