PS02: Multi-Robot Systems
v1.2

Due: February 26, 2015

Multi-Robot Systems

Multi-robot systems need to sense, communicate, compute, and move in a distributed fashion. For this assignment, you will build two basic multi-robot behaviors for coordinated motion: follow-the-leader, which is built from several primitive behaviors from many motion controllers, including navigation, and flocking, which has a deep theoretical underpinning in consensus.

1 Follow-the-Leader:

In this part, you will warm up to multi-robot communications by implementing 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:

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

We have gotten you started with **PS02.py** as a starting code base, and have provided a neighbor system library to ease communicating with nearby robots. You must work with two other people on this assignment. You will need to write code for each pat of the assignment. During class, we will check-ff each team with robots in random roles. We understand that you and your partners will have very similar code for each part of the assignment, but the code on your robot must be written by you.

1.1 Getting Started

Use the buttons to select a role for the robot: red = remote control robot, green = leader robot, blue = follower robot. When the robot is in each mode, use the circling lights indicate that the robot is *idle*, and the solid lights to 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.

```
owlpy.connect()
```
... Robot Startup Stuff ...
```
owlpy>loadrun PS02_netid.py PS02Libs.zip
```

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.

1.2.1 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.

1.3 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.

1.3.1 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()`**.**

1.3.2 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()`**.**

But wait, there's more...

1.4 Leader Robot

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

1.4.1 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()`**.**

But wait, there's more...

1.5 Follower Robot

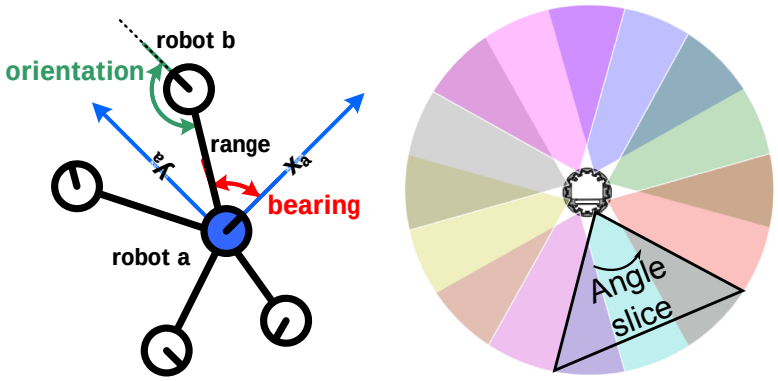
The follower robot will receive messages from a nearby leader robot over the InfraRed (IR) communications system. Once a message has been received, you will need to compute the bearing and orientation to the leader robot. IR signals are directional and will not work if occluded.

1.5.1 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 1(a) shows the local coordinate system around a robot.

[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.

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 1(b) 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.

Hand-in: Write `compute_bearing()` and `compute_orientation()`.

1.5.2 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 mea-

sured in the robot's local coordinate system, see Figure 1(a).

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()`**.**

2 Following with Distributed Algorithms

In this section, you will adapt the code from Section 1. You will make one function that programs the robots to follow in order of their robot ID. Each robot will run the same program, so you will need to design a distributed algorithm that let's them sort themselves. The lowest ID robot will become the group leader, and can simply move straight. You can have it avoid walls if you wish to overachieve.

Design a symmetry-breaking algorithm to ensure that the follower robot with the lowest ID becomes the leader, the next lowest ID follows the leader, the next lowest ID follows the follower, and so on. Using explicit IDs is not allowed, your code needs to be self-stablizing and robust to population changes. Done properly, any robot from any group shold work together. You can program the leader robot to drive straight, or in a large circle, or avoid walls, or anything you wish to test this section.

2.1 Hand-In / Check-Off

1. **Check-off:** Demonstrate your code with leader and two follower robots, sorted in order.

3 Distributed Agreement Theory

The remainder of this lab will deal with *distributed agreement*. Recall from lecture the difference between at agreement is the continuous version. In this section, we will construct some proofs about two different types of agreement.

3.1 Pairwise Agreement

In this section, we will use agreement to compute the average value of a quantity on each robot. Assume each robot r_i has a quantity x_i. We wish to compute the mean of these quantities, μ. We will do so by selecting two neighboring robots at each round, and having them compute their pairwise average. The number of robots, n is not known, and the number of neighbors of each robot is also unknown and time-varying.

Existence: Show that a global average, μ, does indeed exist, and provide an expression for it. (Yes, this is a gimmie)

Conservation: Show that local pairwise averages between two robots, r_i and r_j, preserve the global average, *i.e.* the global average is invariant to pairwise averages.

Convergence: Show that each local pairwise average between two non-equal values reduces the global variance, σ^2, of the distribution of averages. Explain how this provides convergence in a well-mixed network of robots, *i.e.* one in which the likelihood of robot r_i being a neighbor of robot r_j is uniform and bounded away from 0 for all robots i, j. Use the formula for sample variance to get started:

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2$$

3.2 Tolerating Errors:

In a real-world application, robots will have multiple neighbors. There will need to be some kind of communications protocol (a simple algorithm) to select pairs of neighbors who will then compute their pairwise average.

Protocol Design: Design a communications handshaking protocol to compute a correct pairwise average. Your answer does not need to be very detailed, just describe which robot says what to which other robot. Explain how they eventually agree to average with each other and then compute the correct average.

Correctness: Sometimes messages are lost during communication. Explain if your protocol will work if messages are lost between the two robots during communication. If the answer is no, what can you do to fix it? Spend no more than 20 minutes on this question.

3.3 General Agreement

Let's remove our pairwise average constraint from above and have each robot compute a local average of all its neighbors simultaneously.

Conservation: Show that local averages between all the robots that are neighbors of r_i, *does not* preserve the global average. It is sufficient to provide a counterexample to demonstrate this, and it it possible to do so with a network of three robots. Do not construct a full proof of the eventual average to answer this part.

Convergence: Show that each local average between neighbors with non-equal values reduces the global variance of the distribution of averages. Again, start with the sample variance formula from above to show this.

Calculation: Describe what information you would need to know to analytically compute the final agreement value, but do not attempt to derive a analytical formulation for this value. (Hint: You can illustrate the information required with a network of three robots)

3.4 Hand-In / Check-Off

1. **Hand-in:** Answers to the above questions.

4 Distributed Agreement in Practice

Now we'll implement both of these agreement algorithms on the robots.

4.1 Orientation Math

Refer to the local coordinate system for the robots shown in Figure 1(a). Derive an expression for the angle the reference robot would need to rotate to face in the same direction (the same global heading) as its neighbor. Assume that the reference robot can measure the bearing, range, and orientation of the neighboring robot.

4.2 Pairwise Orientation Matching

Using only two robots, write a `match_orientation()` function that makes the robot with the higher ID face the same direction as the robot with the lower ID. You should reuse your heading controller from Section 1.5, but with the equation you derived above.

4.3 General Orientation Agreement

Write a `average_orientation()` function that has each robot face the average heading of all of its neighbors. Be careful how you compute the average of angles, there are many clever-sounding ways, but only one correct way to do this. Test this with all the robots you can find. Is this cool or what? But wait, it gets better...

4.4 Flocking

We will combine the remote control from follow-the-leader with this application with the process of *superposition*. Write a `flock()` function that combines the `(tv, rv)` from the leader robot with the `(tv, rv)` from the `average_orientation()` function. (Hint: This is easier than it sounds.) Test your massive multi-robot flocking with-distributed-agreement machine! w00t!

4.5 Hand-In / Check-Off

1. **Hand-in:** Your bearing equation from Section 4.1.
2. **Check-off:** Your `match_orientation()` function in operation.
3. **Hand-in:** Your bearing equation from Section 4.3.
4. **Check-off:** Your `average_orientation()` function in operation.
5. **Check-off:** Your `flock()` function in operation.