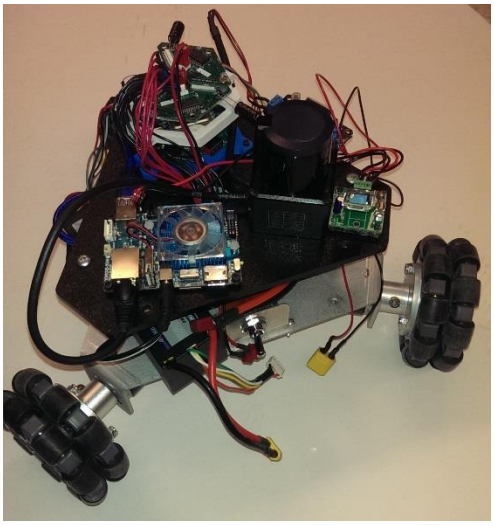
Programming Robots with ROS

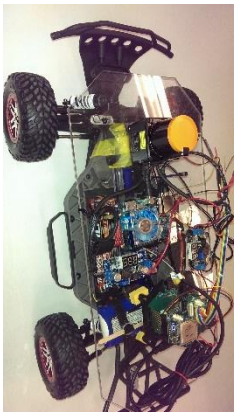
Lab 4 - Car Wall Following – Due Nov 12, 2017 – ver 1.7

Now, we have to pull everything together that we have been learning in class and prior labs: setting up ROS, copying files and setting permissions, creating new packages, understanding and using existing packages, writing python code, debugging modules and systems. You will no longer be given step-by-step instructions. Programming is a creative process and is more about debugging problems than it is about writing code. Use the ROS wiki for help with external packages. Read the textbook. Study the files provided and the tutorials we've done before.

In this lab assignment, you will explore simple PD control for wall following and path planning for a known map. We will be working with Hector_SLAM and AMCL in ROS and the Hokuyo LIDAR (urg_node). This assignment is intended for your groups of three, due to limitations on access to the Tri-Car and the Rally Cars, but feel free to work as you want. You should be using `ros_ws` as your ROS workspace.



The **Tri-Car**, shown in the picture, is an omnidirectional vehicle – meaning, it can go in any direction (in the plane) at any time. Technically, it is what we call "holonomic." An embedded controller interprets serial commands to control the car. The API to control the actuators consists of a simple command string starting with "H" for Holonomic steering, followed by three signed, 12-bit integers in ASCII: `H+0312-0448+0020`. The first value is the x velocity, the second value is the y velocity, and the third is the rotational velocity. All three values can range from -2048 to +2048. The linear velocities are interpreted in mm/sec and the angular velocity is interpreted in degrees/sec. The frame of reference of the car is right-handed and has +Y pointing straight ahead and +Z pointing straight up.



The **Rally Cars** are *Ackermann-steered* vehicles, which is just a fancy way of saying they are normal cars with the equivalent of a steering wheel and a gas pedal. An embedded controller interprets serial commands to control the cars. Both the rally car and the tri-car have the same Ackermann interface. The API to control the actuators consists of a simple command string starting with "A" for Ackermann steering, followed by two signed, 12-bit integers in ASCII: `A-0448+0312`. The first value is the "steering angle" and the second value is the "gas pedal." Both values can range from -2048 to +2048. The "gas pedal" is interpreted as -100% to +100% throttle (torque) and the steering angle is interpreted from -50 degrees to +50 degrees. The frame of reference of the cars is right-handed and has +Y pointing straight ahead and +Z pointing straight up.

In Python (or C if preferred):

1. First, on a Linux workstation, create and debug a wall following algorithm in ROS. First, create a new package called "wall_follower". (LAB4interfacing.pdf has some tips.) Set this up like the

listener in Lab 1, but use "`scanListener()`" and "`scanCallback()`" instead of "`listener()`" and "`callback()`". The `urg_node` publishes a "`scan`" topic using the "`LaserScan`" format from `sensor_msgs`. You must figure out the format of a `LaserScan` from the ROS wiki and how to use the nodes. (You already know how to use `rosnode`, `rostopic`, and `rosrun` from prior labs, so you can test things out, such as the `urg_node`.) Check out a Hokuyo LIDAR to test your assumptions and to detect the wall. In the hallway, we will position the car 0.5 meters from the wall, following the UPenn tutorial 6 for this initial problem. (UPenn sets it up 1.0 m from the wall. The distance "AC" is a function of the speed of the vehicle. 10% of the Tri-Car in Ackermann mode is about 10 cm/s.) Do not use mapping for this problem, just subscribe to the Hokuyo sensor and use the raw data. We will treat the gas pedal and steering angle as decoupled values. Use a PD controller on the steering angle with `Kp = 4.5` and `Kd = 100` and just run as fast as you can using the callback tied to the "`scan`" topic. Set the `gas pedal` to about +10% (or what you're comfortable with).

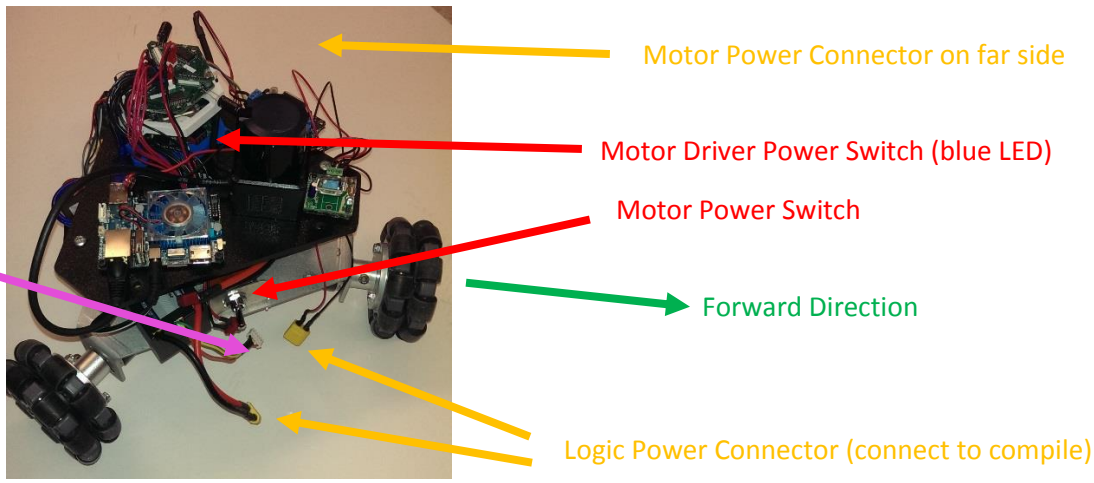
You can test your code in the lab by holding the sensor in your hand and moving it with respect to the wall. NOTE: You have a limited amount of time to unplug the Ethernet cable from the back of the workstation tower and plug in the Hokuyo Ethernet cable. (About 3 seconds!) Power up the Hokuyo, unplug the workstation Ethernet cable, plug in the Hokuyo Ethernet cable and you should be good to test. If not, re-plug the Ethernet, test for network connectivity, and re-try.

To make sure you can communicate between functions in python, make your "main function" of your `scanListener()` compute and print out the data indexes of the $+90^{o}$ and -90^{o} scan values based upon the data in the LaserScan data record. You can use the "`global`" variable declaration to do so. Google "python global variable" to research how these work. You need a mechanism like globals to communicate between functions because you cannot print anything from `scanListener()` until `scanCallback()` has executed once.

To control the Tri-Car or Rally Car, here is some example code:

```
/*****************************************************************/
/* To send commands to the Tri-Car or Rally Car using the serial port,
you can use the "serial" package. */
/*****************************************************************/
# import the serial package to use the hardware serial ports

import serial

# initialize serial port to communicate with Tri-Car

#We are using a USB to RS232 converter on ttyUSB0 for Tri-Car
console_ser = serial.Serial(port = /dev/ttyUSB0",baudrate=115200)

# the port can be changed if there are multiple USB devices


# initialize serial port to communicate with Rally Car

#We are using direct USB on ttyACM0 for Rally Car
console_ser = serial.Serial("/dev/ttyACM0",baudrate=115200)
```

```
#commands to open and close the serial port

console_ser.open()
console_ser.close()

/* construct and send the Ackermann steering commands to Car */
console_ser.write("A-0500+0319")

#Of course, you will have to construct the string dynamically.

# Examples of converting an integer to a string

str(45)

"%+05s" 45
```



Turn in a testable `wall_follower` package online, well-commented, submitting to "Lab4 Prob1". First, fully test your code on the Tri-Car or Rally Car (check it out from MGL 1310 or MGL 1219), then demonstrate your code to Prof. Voyles in the hallway outside the lab by next lab session. **Time your algorithm to see how fast it is running (how many times per second `scanCallback()` is running). Include a screen shot of your `scanListener()` node printing out the +/- 90 indexes and the frequency of your `scanCallback()` as well as a brief write-up of how you timed it.**

2. Modify your `wall_follower` package to run the PD controller at 150 Hz. Wait! The sensor does not run that fast (and, therefore, `scanCallback()` does not run that fast)! So, modify your `scanListener()`, using talker() as a model, to run the PD controller in `scanListener()` at 150 Hz. (How are you going to do that?? – hmmm, how did you find the +/- 90° indices??)

   With PD controller running faster than the sensor, you can initially just use the most recent sensor value to compute a new control value (which will be the same value as before). Test your

code to make sure it still works, but measure the new PD controller to make sure it is running at 150 Hz.

A better thing to do is interpolate the movement of the car based on last position, last speed and last steering angle. Use this information to update the estimated distance of the car from the wall (and angle) and re-compute a new control command. You may do this any way you choose, including: coarse estimate by look-up table, piecewise linear estimate, or compute a circular arc. Whenever you get a new update from the LIDAR, just use that for the PD controller. **HOWEVER, you must compare your estimates** of distance to the measured distance and log this error.

Create a graph of your error versus time for your test run and turn it in to "Lab4, Prob 2." No code to turn in.

3. If we had a map, we could use it in place of the sensor, as long as we know where we (the car) is. To do this, create a **new package** called "map_follower". Download the initial map of the hallway outside MGL 1219 (`hallmap`) from the class website. You need both the `pgm` and `yaml` files. Load this map using `map_server`. Here is an example launch file to load a map (Remember reading about launch files in Chapter 2 of the book?):

```
<launch>

    <!-- Load a Map  -->
    <arg name="map_file" default="$(find wall_follower)/resource/map/hallmap.yaml"/>   # load map
file
    <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

</launch>
```

You can use the AMCL topics of `map` and `amcl_pose` to read the map loaded by the `map_server` and to read the current pose of the robot, published by AMCL. Note that the map is a `width`-by-`height` occupancy grid, stored as a linear array of 8-bit values from 1 to 100 (with -1 indicating "unknown"). To read location `(r,c)`, you must access `data[r * width + c]`. You must query the `MapMetaData` to find the size and resolution of the occupancy grid and you must search for the first occupied cell along a ray.

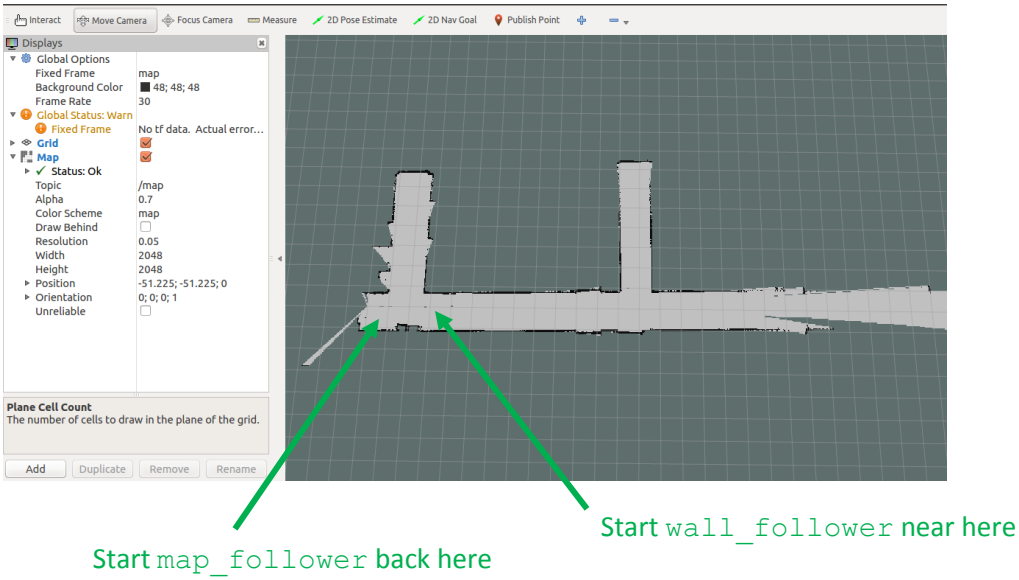
Assume you are starting in the middle of the hallway outside the door to 1219 and are facing the outer door. See below, but exact placement is not too critical.

4. If we had a map to know what is in front of us, we could control the speed of the vehicle, as well. Create a separate PD controller for the speed, but clip all negative values to zero. (Backing up in an Ackermann-steered vehicle is more complex and requires more planning – and race cars rarely go backwards, anyway!) Set `Kp = 10` and `Kd = 80`. Include the speed PD controller in the same 150 Hz process with steering. To interface with the onboard car controller, scale the speed in mm/sec and multiply by 10 to transmit over the serial port. (An error of 20 mm from the PD controller, when multiplied by a Kp of 1 and then multiplied by 10 will produce a request of +0200 to the controller.) Download the initial map of the hallway outside MGL 1219 from the class website. You need both the `pgm` and `yaml` files. Load this map using `map_server`. Here is an example launch file to load a map:

```
<launch>    t

    <!-- Load a Map  -->
    <arg name="map_file" default="$(find nav)/resource/map/map.yaml"/>   # load map file
    <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

</launch>
```

(See the "Lab4interfacing.pdf" and "AMCLhints.pdf" files for more details.)

First, we will do this by hand, setting a series of way points along a route using the map reference frame. A dense series of way points can be used to control both the speed and the lateral relationship to the wall. Plan these way points for an update rate of 15 Hz and a speed of 200 mm/s. (The PD controller, running at 150Hz, will simply use the most recent values of the way points.) Display the way points on the map. You can do this, as in the figure below, with the following:



Turn in a testable `map_follower` package online to "Lab 4 Prob 4". Demonstrate your code in the hallway outside the lab to Prof. Voyles by third lab session.

5. With the map data, plan circular arcs around a corner, offline, using the `cornermap` found on the website for the hallway outside MGL 1219. Again, do this first by hand, setting a series of way points along a route using the map reference frame. Plan these way points for an update rate of 10 Hz and a speed of 200 mm/s. You can click on the map to get a list of waypoints and hard code these into the program. Use `rviz` and the `waypoint.py` example to query a set of waypoints, exploring the hints in Lab4interfacing.pdf.

No code to turn in, but demonstrate your code in the hallway outside the lab.

6. Now, let's plan the corners semi-automatically. Normally, a path planner would select where to turn and which way, but that's Lab6. Instead, you will act as the path planner, so you get to manually choose an apex point in each curve (allow safe distance from the corner) and manually specify an input point and angle and an output point and angle. (You will choose the angle to be parallel to the walls.) From these, find the largest radius curve that moves through it, intersecting straight-line paths along the angles specified, parallel to the walls on either side of the apex a maximum (safe) distance from the wall. (The curve will not necessarily pass through the two points on either side of the apex of the turn.)

Turn in a testable `autoWayFollower` package online. Demonstrate your code in the hallway outside the lab.