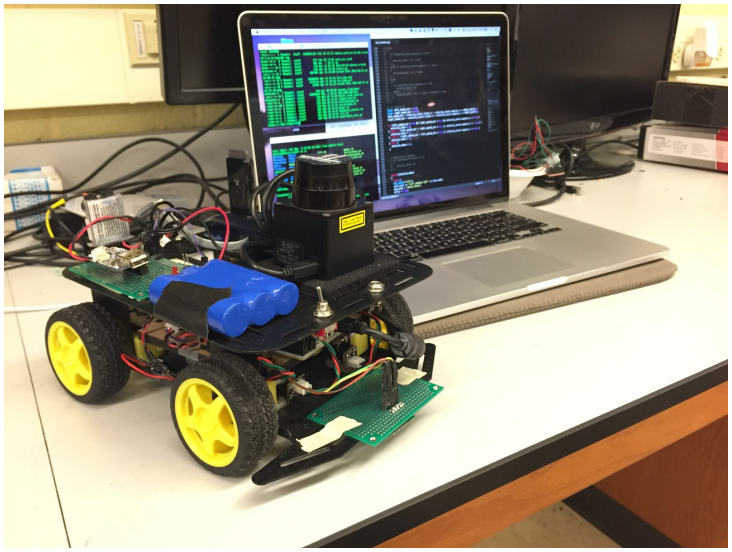
Documentation for Autonomous Vehicle Testbed

Zhi Li
2015 Spring

Roadmap

1. Setting up hector_slam package and ROS on VM and writing launch/config files for mapping on a trolley. (**21 Jan - 1 Feb**)
2. Researching on how individual packages of hector_slam works, finish integrating encoder data into hector_mapping. (**2 Feb - 15 Feb**)
3. Setting up ROS and mapping packages on BeagleBone Black, write packages for remote keyboard control over wifi. (**16 Feb - 22 Feb**)
4. Enable wifi connection(~ 200 ms level), automatically reconnected on detecting failure, setup local network to enable cooperation from multiple sources; enable setting up distributed ROS for running visualization on VM; (**23 Feb - 28 Feb**)
5. Figuring out to use special power kit for booting up from battery, write code on mbed to control different H-bridges and find a proper one that could provide stable current output. I haven't work on anything related to zigbee, though it turns out to be faulty. (**15 March - 20 March**)
6. Encounter power insufficient and back-emf issue, poking around to find a solution, figure out to use Y-cable for Hokuyo (suggested by Yash from Grasp), with back-emf problem solved by Matt; configure to provide robust and reliable wifi connection(~10 ms level); configuring USB power supply to reconnect on detecting failure. (**21 March - 7 April**)
7. Simplify the procedure of presenting a test/demo by combining packages and writing bash scripts; enhance control to car over mbed to enable real time speed change (PWM), write code for basic straight line following; write ROS package to integrate encoder into hector_mapping (**8 April - 15 April**)
8. Getting realtime encoder data with acceptable tick loss; time synchronization problem solved across ROS platforms, and will apply to all sensor inputs; trying to feed encoder data into mapping process, but performance is not promising due to precision of encoder and skidding wheels. (**16 April - 22 April**)
9. Applying PI control and straight driving using encoder ticks per sec as speed input; enabling line following around center table in a circular track, with several scenarios handling added to the code. (**23 April - 1 May**)
10. Achieving waypoint navigation with existing map and real-time exploration; obstacle avoidance can also be achieved by tuning parameters in yaml files of costmap, as well as the inflation of obstacles. (**7 May - 8 May**)

Table of Content



Introduction

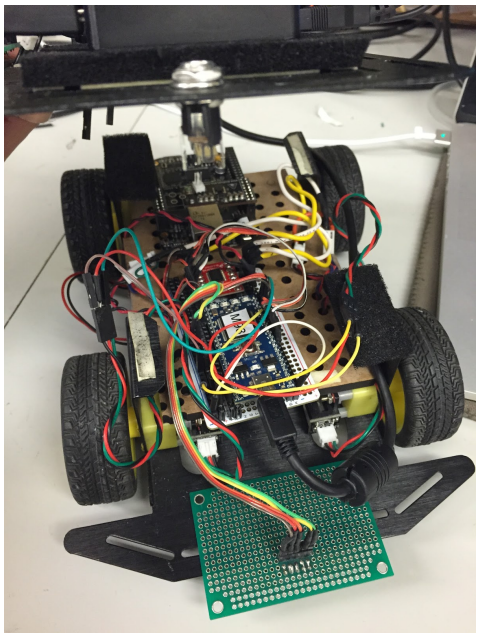
This autonomous vehicle testbed is designed to be a common platform for multiple research topics in autonomous driving, including scenario handling for safety assurance, localization and autonomous navigation. The car could now perform real time exploration and waypoint navigation using the LIDAR on top, as well as the traditional approach of building a map first and then achieve navigation capability based on the map generated. A PI controller is used to keep the car moving at a set speed calculated out of encoder ticks per sec, and could also enable straight line driving in the meantime. Line following capability enables the car to follow a preset track and cope with different turning scenarios, the detection of lanechange could also be achieved leveraging those IR sensors.

Hardware

Development Environment

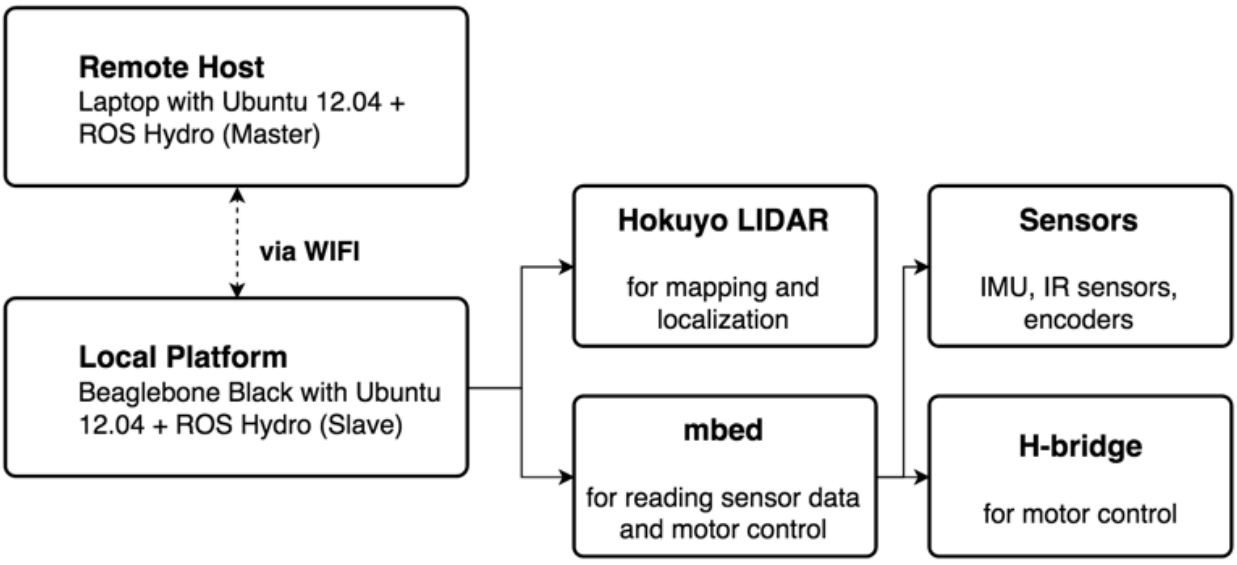
OS: Ubuntu 12.04
ROS ver: Hydro
LIDAR: Hokuyo URG-04LX-UG01
Robot car base: DFRobot with encoder
Platform: Beaglebone Black (ubuntu 12.04 + ROS Hydro)
Hardware: mbed LPC1768, H-bridge, Wifi Dongle (WNA1100), Adafruit Powerboost 500
Battery: Adafruit Lithium Ion 3.7V 6600mAh
Param sheet for parts mentioned above:
https://www.dropbox.com/sh/ujqpz4cp4h5fzak/AAAGaUGlNwOzu-JbHHehNdq4a?dl=0

System Overview



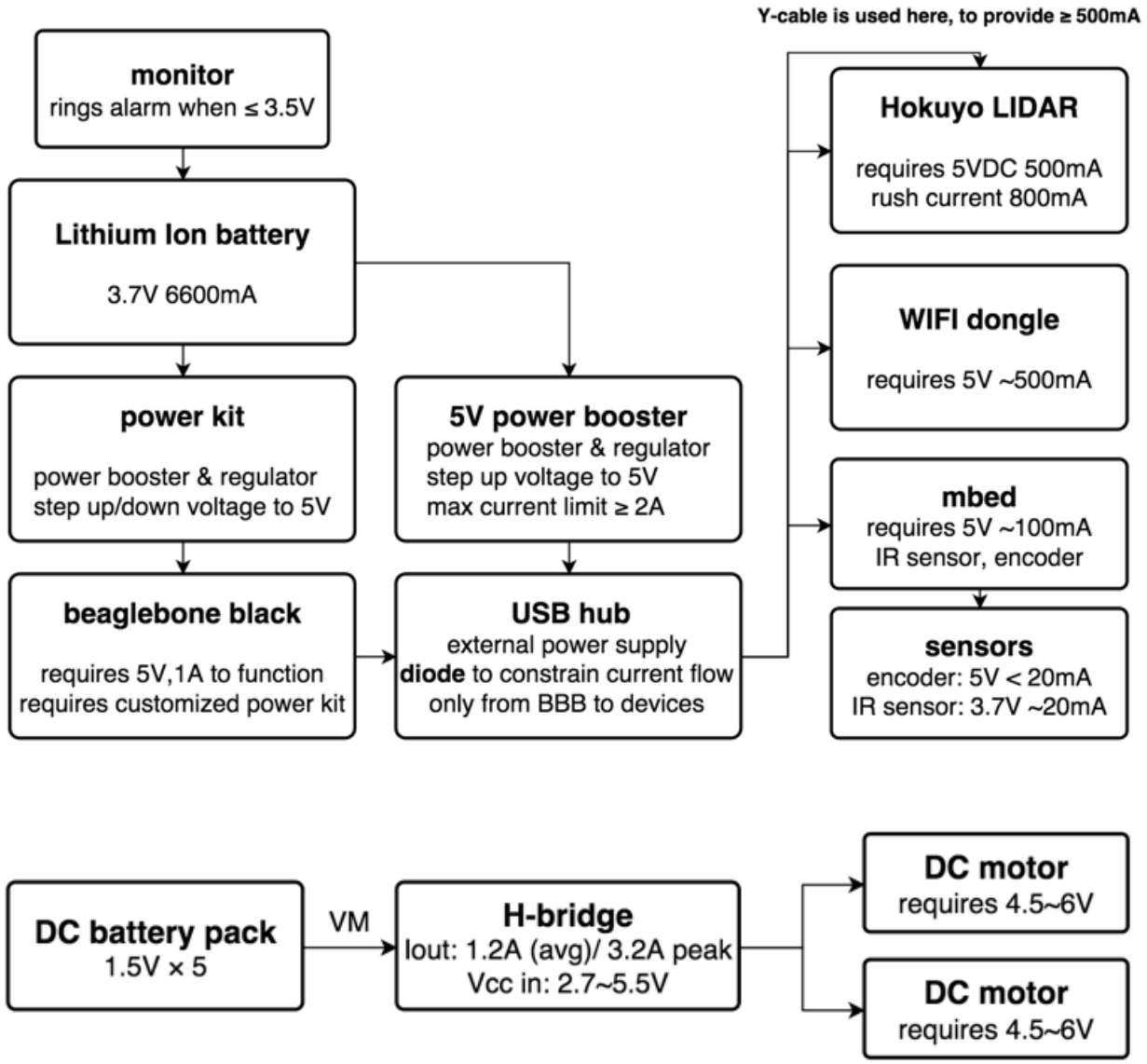
A system diagram can be found below. The core system is running on beaglebone black, which communicates with remote host via WIFI network, talks to mbed via USB serial port, and collects laser point-cloud data from hokuyo module. The mbed is in charge of collecting data from the encoder, IR sensors and IMU, and will then forward them to beaglebone via serial port. The mbed also reads commands from the beaglebone via serial port, and convert them to set of control inputs to the h-bridge, which manipulates the current to motor and achieve actions including forward, backward, left, right, brake etc.

Since the beaglebone black platform has only one USB port that limits a maximum output of 500mA, the car uses an external powered usb hub to provide multiple ports to connect the LIDAR, mbed and wifi dongle at the same time. Notice that the beaglebone black platform has little tolerant over its USB port, i.e. it doesn't allow the device connected to consume more than 500mA, and also doesn't allow current to flow backwards to the USB port. The two requirements are found in practice, and will cause convoluted troubles during implementation of this system. The solution can be found in the power system design part below.



Power System Design

Since the autonomous car runs on the ground, there's no need to concern too much about the weight it carries. Thus, considering the fact that better performance is achieved doing so, we choose to use separate power supply for motors and electronic devices. Two diagrams of power system are shown below.



The first graph shows the overall power system for the electronic devices. A power circuit is designed to supply power to the beaglebone and the USB hub at the same time. One of the outputs of Lithium Ion battery goes to the power kit of BBB to power the core system, the other is first stepped up to 5V through a power booster, and then power the USB hub connecting the LIDAR, mbed and WIFI dongle. A monitor is also connected directly to the battery, thus when the voltage drops too low, it will ring the alarm. Notice that there's a general requirement to tie the grounds of all electronic devices together.

The second graph shows the connection for motor control, a battery pack of five DC batteries powers the H-bridge through VM port, and the H-bridge will apply different set of current supply to DC motors according to control signal inputs from the mbed. Notice that current implementation ties the two motors on the left side together, along with the same approach applied to the right side.

Notice:

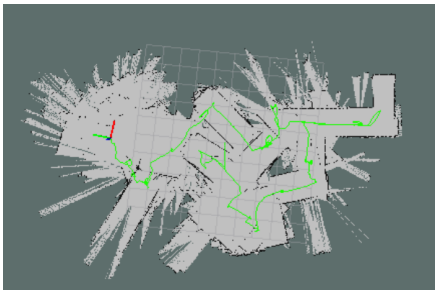
- **Quick fix with LIDAR**
  A Y-cable is used to enable external power supply to the sensor itself via two USB ports. The reason of this approach is that the Hokuyo LIDAR needs more than 500mA to function but USB standard has a limit of 500mA top. An alternative is to separate the power and signal wire in a USB cable and provide external power supply.
- **Diode in USB hub**
  When a motor abruptly runs (start/stop), it will generate a back-emf that will pass the USB hub to the BBB, which will in turn immediately shut down the USB power supply. The LIDAR also has a step motor built inside. Thus, the solution here is to use a USB hub with built-in diode to prevent back current to the platform, or simply solder a diode.
- **External power supply for USB hub**
  It's essential that USB hub has external power supply, since the LIDAR and the wifi dongle could both be power consuming. If not given external power supply, the wifi dongle will not providing robust network connection, sometimes just turn itself off arbitrarily.
- **Power supply for encoders**
  One could supply either 3.7V or 5V to the encoders, both will work fine.
- **Charger for Lithium Ion batteries**
  The adafruit power booster 500 serials can also be used as a charger for LiPo and Lithium Ion batteries, simply attach the battery pack to it, and then use micro USB cable to connect the circuit to a PC. Normally a full recharge will take 6~8 hours, which depend on the remained volume. Also, it's suggested that the voltage should not drop below 3.4V for maintenance purposes.
- **Set microSD card as default bootup**
  sudo dd if=/dev/zero of=/dev/mmcblk1 bs=1024 count=1024
  execute the command above on Linux at BBB
  ref: http://www.twam.info/hardware/beaglebone-black/u-boot-on-beaglebone-black
- **Static IP can't resolve nameserver**
  /sbin/route add -net 0.0.0.0 gw 192.168.1.1 wlan0
  # then able to ping ip addresses but not host names
  add one line dns-nameserver 8.8.8.8 under inet ~ static ~ in /etc/network/interfaces
  then restart network/interface: sudo /etc/init.d/networking restart
  ref: http://askubuntu.com/questions/465729/ping-unknown-host-google-com-in-ubuntu-server

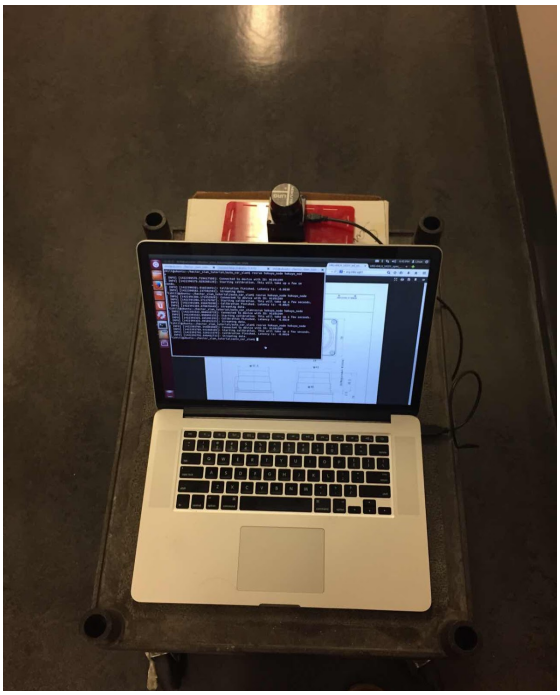
Software

Simulation on VM

- **Reason for simulation**
  It's always good practice to first run simulation on virtual machine with sensors (e.g. LIDAR) connected to it. Reasons are as follows. First, you could rule out any problem caused by power insufficiency, since laptop is a robust testing environment with sufficient power supply. Then you could make sure that your implementation of ROS packages are bug-free before deployed to embedded platform. Also, you could tune the parameters for tasks such as mapping and PI control using just the VM, which will save a lot of time assembling the car up.
- **Types of simulation**
  - mapping using LIDAR (ROS on VM + LIDAR)
  - motor control/ IR sensor tuning (terminal + mbed)
  - incorporate encoders/IMU data into mapping/publish odometry (ROS on VM + mbed)

Mapping using LIDAR:
ref: http://wiki.ros.org/hector_slam/Tutorials/SettingUpForYourRobot
Experiment setup: LIDAR + trolley + laptop
ROS packages: hector_slam(hector_mapping, hector_map_server, hector_trajectory_server)
Virtual Machine: VMware over virtual box
(the former one has more video memory)





right: experiment setup;
map above: LIDAR not fixed to the trolley;
map below: mapping with less roll/pitch/yaw;

Conclusion:
Good quality of mapping can only be achieved when LIDAR is stable. There are two solutions for this, one physical way is to stuff foam under the edges of the LIDAR and stick the sensor on top of it. The other approach is to leverage IMU data and hector_imu_attitude_tf to improve mapping by discarding laser data during roll/pitch/yaw.

System Overview
ROS is installed on both platforms, PC is configured as host machine and embedded platform is configured as slave. As long as time between the two platforms are tightly synchronized, host could subscribe topic collecting sensor data published by slave, while slave could subscribe topic sending control command published by host. Open-source packages and customized packages works as talker and listener components, and are integrated to achieve different goals and purposes. In this project, host machine is in charge of visualization and major computation, while slave (along with mbed) is in charge of adapting low-level control and sensor data forwarding.

ROS
installation: http://wiki.ros.org/hydro/Installation/UbuntuARM

1. **Distributed ROS setup**
   ref: http://wiki.ros.org/ROS/Tutorials/MultipleMachines;
   http://wiki.ros.org/ROS/NetworkSetup;
   ROS is designed with distributed computing in mind. A well-written node makes no assumptions about where in the network it runs, allowing computation to be relocated at run-time to match the available resources. The approach is to add the following lines in both host's and slave's bash file, set ROS_IP & ROS_HOSTNAME to the machine's IP address, and set ROS_MASTER_URI to host machine's IP:11311. (where roscore runs) Always remember to check if roscore could run with no warning/error messages on host end, and it's suggested that for the first time running ros across distributed platforms, run simple talker/listener and see if the network is correctly configured.

```
export ROS_MASTER_URI=http://172.31.100.171:11311
export ROS_IP=172.31.100.171
export ROS_HOSTNAME=172.31.100.171
```

2. **Synchronization across platforms**
   A crucial part to operate on distributed packages is to ensure synchronization across platforms. If timestamp is not correctly synchronized, you'll probably run into the following problems. program on slave machine will not be able to compile; sensor data published is not correctly subscribed; receiver end complains about "lookup

exploration into the future"; mapping and navigation could not proceed etc.
The solution is to use chrony as suggested on the ros forum. First, installing chrony on both host and slave machine. Then, modify parameters according to the link attached. The key point here is to keep server time and client time closely tightened, regardless the relation between server time and that of outside ntp servers. After fully setup, every time when the slave boot up, just execute **init.sh**, which contains the following two lines to synchronize with server's timestamp:
sudo ntpdate $SERVER_IP *# clear discrepancy between server and client*
sudo invoke-rc.d chrony restart *# restart chrony service to keep synchronized*
ref: https://code.google.com/p/rosbee/wiki/chrony

3. **ROS control**
   custom packages: car_slam (keyboard_control), control_to_mbed
   ros topic: /motro_control
   Instead of applying direct control to mbed and providing more flexibility, the current implementation use two custom ROS nodes to achieve remote control over the car. One node is called **keyboard_control**, located in package car_slam on host machine, which could read keyboard inputs(space, ↑, ↓, ←, →, A, D: stop, forward, backward, turn left, turn right, accelerate, decelerate), and publish control messages to **/motor_control** topic. A sample forward message is in the form of "BMAIN A EMAIN". The other node, **control_to_mbed**, will subscribe this topic, and trigger a callback function whenever there's a control message available, then will write corresponding control command to mbed through serial port. The implementation can be further expanded to publish/subscribe pair that can handle control messages containing velocity requirement, i.e. in the form of "BMAIN LEFT 0.25 RIGHT 0.45 EMAIN". On receiver side, the velocity information will be parsed down in control_to_mbed, and sent to mbed as inputs for PID controllers' goals.

   Since **control_to_mbed** doesn't care who's publishing /motor_control topic, it could also receive a sequence/set of commands sent by the navigation stack(move_base) or custom package to finish specific task. This project has a customize node naming **nav_control** under car_slam on host machine, which will subscribe /cmd_vel topic published by the navigation stack(move_base) and convert control message to the type that current controller could take in, and publish the converted message to /motor_control topic. Detail implementation could be found in the part of **ROS navigation**.

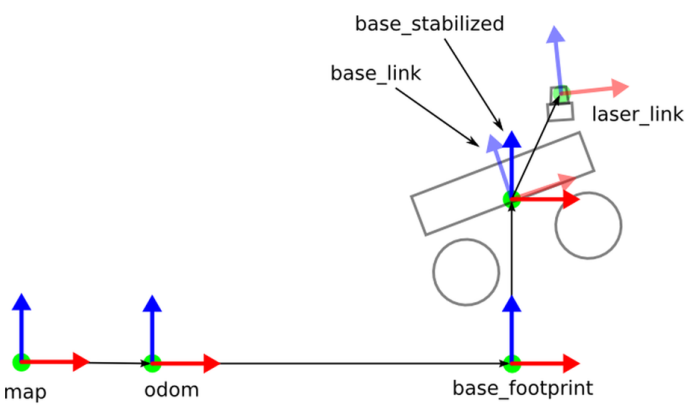
4. **Line following**
   custom packages: serial_generic, ircmd_to_motor
   The serial_generic node read whatever mbed writes to BBB via serial port and publish to /serial_generic topic. For line following functionality, mbed will write message in the format of “BIR 0 EIR”, the digit in the middle corresponds to different command (0: stop; 1: forward; 2: back; 3: left; 4: right). The **ircmd_to_motor** node subscribes /serial_generate topic and trigger a callback function to parse “BIR X EIR” type message when available, then write to mbed for low-level control. The current structure could also expand to achieve closed loop functionality, i.e. leveraging /slam_out_pose of hector_mapping package, whenever the car comes back to its original startpoint considering error, it will stop sending out commands to the mbed and thus achieving closed loop functionality. The current implementation is to detect a right-angled bend and stop (which is the starting point of the track).

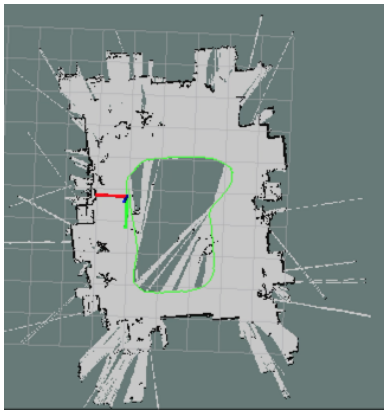
   demo video:
   https://drive.google.com/file/d/0B2jEHYaQEznjSEJTbzNIRnVKZFU/view?usp=sharing

5. **Mapping**
   ros packages(essential): hector_slam, hokuyo_node
   custom packages(optional): serial_generic, encoder_from_mbed, imu_generic
   Mapping functionality can be achieved leveraging several open-source ros packages including **gmapping** and **hector_slam**. This project uses the hector_slam package to generate a static/dynamic map. The reason is that hector_slam could take full advantage of laser data and “fake” decent odometry in a non-clear environment, i.e. detectable difference in every 4 meters required by the LIDAR. Thus, this SLAM implementation of mapping won’t need wheel odometry, and could provide decent performance when the LIDAR is stable(not experiencing much roll/pitch/yaw). Current implementation has also provided a custom ros package to process IMU data and another package to process encoder data collected from slave(mbed) for better mapping. Detail implementation could be found in node **imu_generic** and **encoder_from_mbed** on slave machine. In general, sensor data is collected from mbed, printed to serial port, parsed on slave side, processed and published to specific topic/tf_broadcaster, then subscribed by packages on host machine (e.g. imu_attitude_to_tf, odom frame etc.).

   hector_slam uses the hector_mapping node for learning a map of the environment and simultaneously estimating the platform's 2D pose at laser scanner frame rate. The frame names and options for hector_mapping have to be set correctly. Parameters for

mapping has been tuned to adapt current platform, and could be found in **slam_demo.launch** under **car_slam** package on **host machine**. The image below shows all potential frames of interest in a simplified 2D view of a robot travelling through rough terrain, leading to roll and pitch motion of the platform:



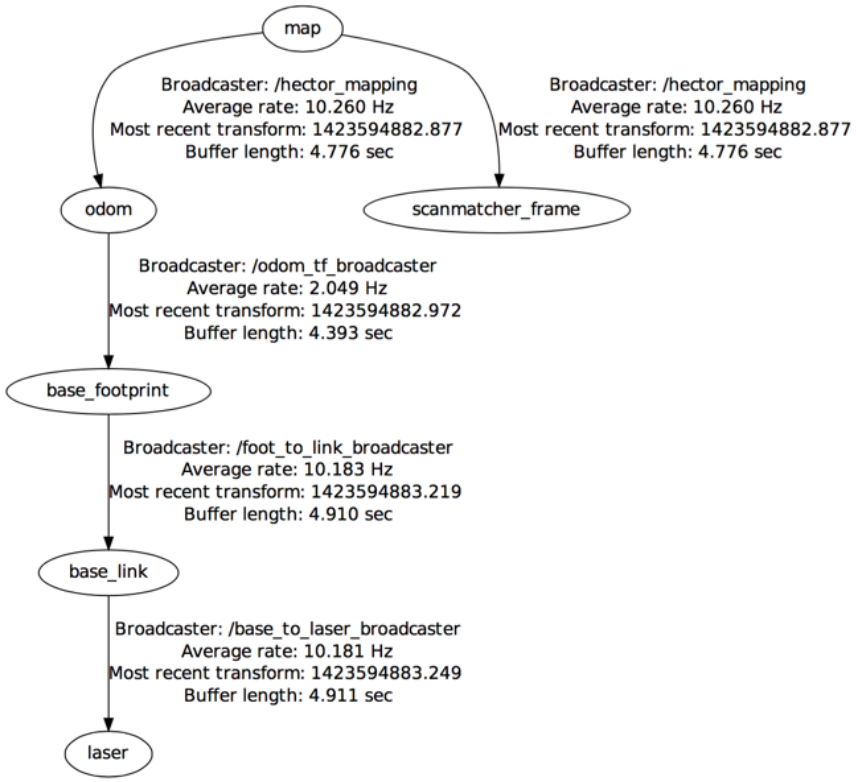


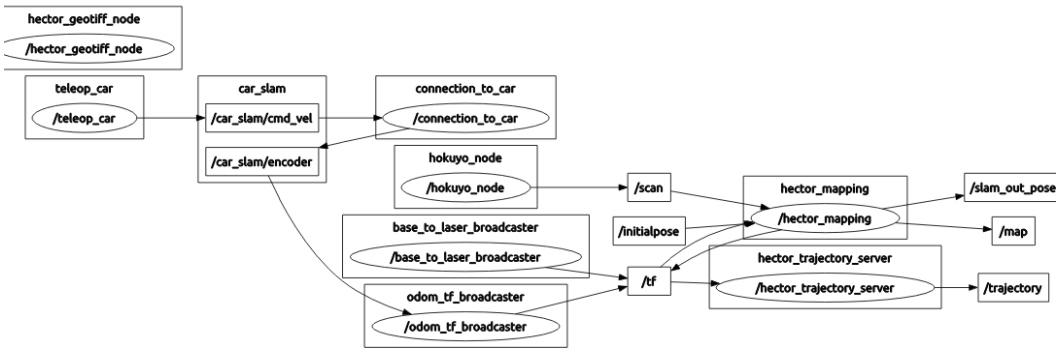
https://drive.google.com/file/d/0B2jEHYaQEznjOGVacDRyTERuckE/view?usp=sharing

A further peek at the launch file could give detail information of relationship between these frames. Our implementation has used the following frames. The map frame and scanmatcher_frame are generated by hector_mapping, the latter of which is the frame that records laser odometry with corresponding topic /slam_out_pose. hector_mapping node is also in charge of publishing tf between map frame and odom frame if **pub_map_to_odom** parameter is set true in the launch file. Here in order to add in an odom frame to leverage wheel odometry data, **imu_generic(optional)** and **encoder_from_mbed** node come into use, the two nodes will publish tf between odom frame and base_footprint frame, then hector_mapping will take in these tf and publish a map to odom tf in general.

Notice that after numerous experiments, I figured out that wheel odometry can **barely** help with perfecting the map generated since not only error accumulates along the way, the wheel encoders suffer a lot from skid-steering driving, i.e. the skidding behaviour of wheels during turning will ruin the odometry drastically. Thus, encoder data is only used for straight driving and PI control.

base_footprint frame is the lower frame at the same level of the car's wheels, and base_link is the frame where laser is set (a little higher than base_footprint), the tf (transform) between this two frames is published by a **static_tf_publisher**. Also, in order to let the hector_mapping package know where it can get the laser data, we also need a **static_tf_publisher** between base_link frame and laser frame. Notice that the laser frame and /scan topic is provided by an external ros package called **hokuyo_node**. A tf graph is shown below, along with a system diagram with tf and active topics under it.





6. **Navigation**

   Navigation stack(move base) will publish speed commands to /cmd_vel according to the planned track route based on the pose of the car and its 2D goal. After reading through topics on ros forum and documentation of related packages, I've organized the following approaches to achieve waypoint navigation and obstacle avoidance. I choose to implement the first solution, and have achieved exploration, waypoint navigation along with static obstacle avoidance using solely LIDAR as sensor input.

   real-time exploration (hector_mapping + move_base):

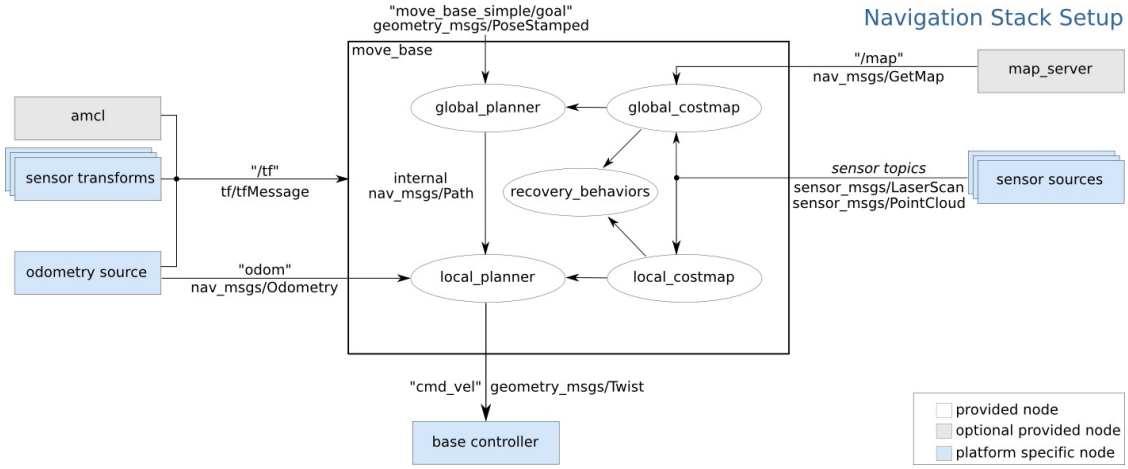
   1. http://answers.ros.org/question/63360/extrapolation-error-using-hector_mapping-move_base/
   2. http://answers.ros.org/question/73261/autonomous-navigation-using-navigation-stack-and-hector-slam/

AMCL+ Hector_SLAM (might be redundant)
http://answers.ros.org/question/59153/how-to-extract-information-from-hector-slam-for-autonomous-navigation/

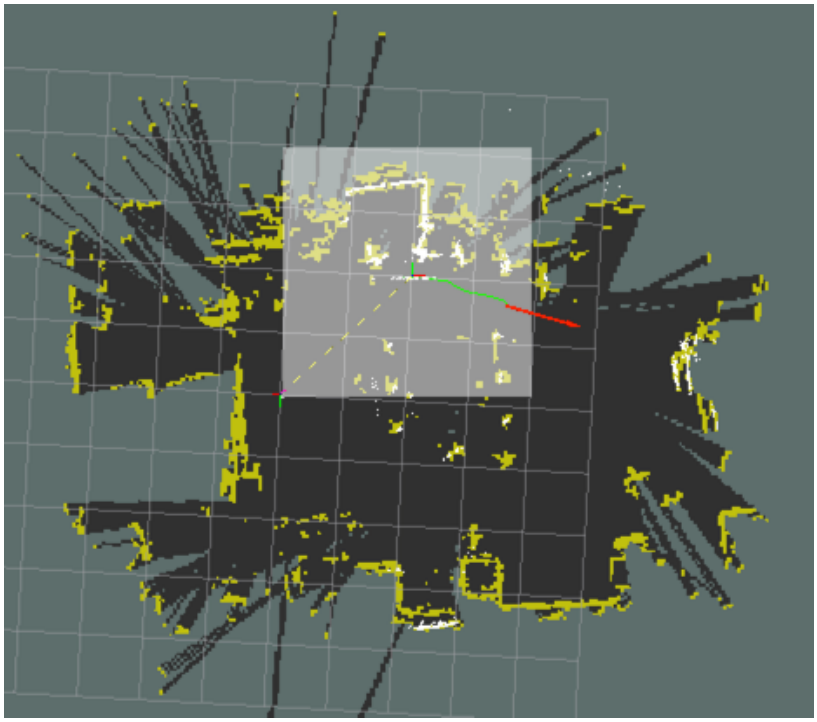
hector_navigation using hector_mapping
http://answers.ros.org/question/112576/how-can-i-run-hector-navigation-for-exploration/
http://answers.ros.org/question/38575/navigate-in-hector-map/

Navigation Summary

ref: http://wiki.ros.org/navigation/Tutorials/RobotSetup



move_base is the core plugin in the navigation stack to take in static/dynamic map, sensor data, result of localization and generate controller commands. amcl is the localization package to locate the mobile platform on a static map given, while odometry source could be nav_msgs/Odometry messages come from customized packages that process IMU data/wheel odometry. Since hector_mapping could fake a laser odometry with decent precision, our implementation does not use amcl or external odometry sources.



real time exploration + waypoint navigation in rviz: red arrow is the goal, tf on the left is the fixed map frame, while the tf in the middle (the red axis shows the front side of the car) of local costmap shows current pose of the car.

Navigation General Setup

Navigation stack requires a **robot configuration launch file** (here as **hector_hokuyo.launch**) that runs sensor_node_pkg, odom_node_pkg, and transform_configuration_pkg. Here in my implementation of exploration, **sensor_node_pkg** corresponds to **hokuyo_node** running on slave machine, **odom_node_pkg** corresponds to **hector_mapping** with preset mapping parameters (stored in **default_mapping.launch**) as well as generating a **scanmatcher_frame** as laser odometry frame, along with other transform_configuration_pkg.

Also, the stack requires a **main launch file** for navigation (here as **move_base.launch**), which requires a map service, a localization package, and move_base package with costmap configurations. In my implementation, **map service** is provided by **hector_mapping** since the goal is to achieve real time exploration, and the car is generating/filling a map while it's navigating to a spot within the range of global costmap. **hector_mapping** also provides **localization functionality** through /slam_out_pose topic and the scanmatcher_frame. Thus, my main launch file only contains **move_base** package with costmap configuration, and includes in **hector_mapping.launch** to run hector_mapping package in the meantime.
usage(on host machine): roslaunch navigation2d_example move_base.launch

Costmap Configuration
It's better to look through parameter setting in ***.yaml files** under **ros-navigation2d-example/param** on slave machine. Generally, for real time exploration, global/local costmap needs to set to rolling window and update periodically. Configuration of obstacle avoidance goes to costmap_common_params.yaml, move_base node configuration goes to move_base_params.yaml, and global/local planner configuration goes to base_local_planner_params.yaml.

For navigation using static map, first you'll need to record a rosbag that contains laser data in future navigation setting (must have same frame structure with same tf relationship). Link on how to build a map using rosbag using hector_slam: http://wiki.ros.org/hector_slam/Tutorials/MappingUsingLoggedData.
Then add this line in **move_base.launch**:
<node name="map_server" pkg="map_server" type="map_server" args="$(find my_map_package)/my_map.pgm my_map_resolution"/>
and set param **publish_map_service** in hector_mapping to **false**.
Next set **global_costmap_params.yaml** to use static map. Now launch move_base.launch, you should be able to use static map for navigation.

Base Controller
The navigation stack assumes that it can send velocity commands using a **geometry_msgs/Twist** message assumed to be in the base coordinate frame of the robot on the **"cmd_vel"** topic. This means there must be a node subscribing to the "cmd_vel" topic that is capable of taking (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) velocities and converting them into motor commands to send to a mobile base. My implementation of this node goes to **nav_control.cpp** under **car_slam** package on the host machine. The algorithm to generate a compatible control message based on the velocity command sent from move_base is simplified as follows.

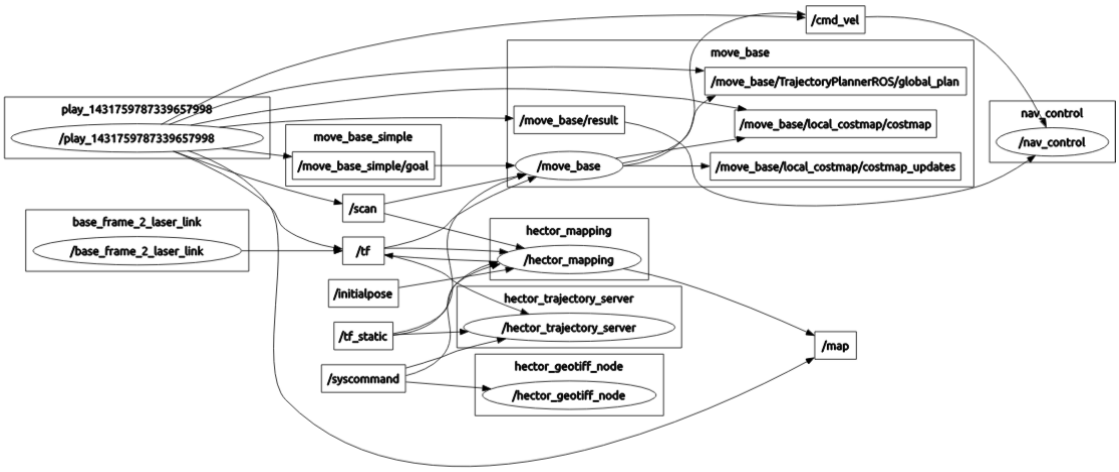
```
left_speed_out = velocity_twist.linear.x - velocity_twist.angular.z * ROBOT_WIDTH/2;
right_speed_out = velocity_twist.linear.x + velocity_twist.angular.z * ROBOT_WIDTH/2;

if (left_speed_out < 0.1 && right_speed_out < 0.1 && velocity_twist.angular.z < 0.1) {
  velocity.data = 0; // stop
} else if (left_speed_out > right_speed_out * 1.2 || velocity_twist.angular.z < - 0.3) {
  velocity.data = 4; //right
} else if (right_speed_out > left_speed_out * 1.2 || velocity_twist.angular.z > 0.3) {
  velocity.data = 3; //left
} else {
  velocity.data = 1; // forward
}
```

Notice that if the thresholds above are not properly set, the car may not go along the pre-planned route, or even keeps deviating from the track defined and might not reach its destination. Another issue is that if the car could not meet the time limit set for control loop(here 5Hz for control frequency), it will easily lead to poor performance by

not achieving the ideal pose on reaching destination, or simply sending out STOP command after receiving a new navigation goal. Thus, in my implementation, I have to tune down the integral part of the PI controller, or even set it to zero to achieve quicker response.

System rqt_graph

A general navigation system graph should assemble the graph below. It's generated using rosbag to replay tf and topics. (will use runtime system graph when I get back)



mbed

Summary

The mbed software is using multithreading for reading encoder data, imu data and ir data, as well as handling motor control commands received on serial port. Detail implementation could be found under **mbed** repository.

1. **PID driver control**

   General control interface:

   driver thread listens on pc.serial, react and apply corresponding set of control signals to H-bridge, use PWM output to control motor speed. ros node **control_to_mbed** on the slave end will map commands parsed from **/motor_control** topic to int8 type values, i.e. 0, 1, 2, 3, 4, 5, 6: stop, forward, backward, turn left, turn right, accelerate, decelerate

   PI control:

   mbed has an existing PID library for control, the essential part is to understand the api designed and adapt to current implementation. For current implementation, the PI controller takes in encoder ticks per sec as inputs, and generate PWM cycle as output. The bottleneck of current project is the precision of the encoder as well as the speed of car, i.e. when the car runs at full speed (i.e. duty cycle = 100%), it could only reach 20 ticks/second, which hugely limits the performance of PID control since the speed can fluctuate a lot from the baseline due to small denominator. Therefore, the PI

parameters are tuned according to experiments instead of mathematical computation, and the goal of tuning is to make the transition in between each move smoother.

Interface of PID library:

1. input/output: I: counts/sec; O: pwm duty cycle
2. setSetPoint: set goal velocity(unit: counts/sec)
3. setProcessValue set realtime feed of velocity(unit: counts/sec)
4. compute calculate pwm duty cycle for H-bridge control (current compute rate = 10ms)

Experiments to calculate Kp:

duty cycle	**speed (counts/sec)**
1	19.41
0.8	14.56
0.6	12.10
0.4	6.72

tuning ref: https://developer.mbed.org/cookbook/PID

2. **encoder data reading**

After switching between using ISR and analogIn for reading encoder data, I finally choose the latter one since it's more robust and could provide precise ticks. The former one used to experience a lot of tick losses due to interleaving of ISR processes. The encoder data is smoothed on every 3 inputs and printed to serial port every 100ms. Notice that a funny thing about mbed is that when I use two adjacent pins as analogIn pins, I'll get weird readings from the encoder output. The solution is to use separate pins.

3. **ir data reading**

After several experiments, it turns out that solely IR receivers could achieve ideal performance. The car has three IR receivers set under the head of the front frame, each will detect the black tape with readings above 88. Since compared to the width of the car, the distance between IR sensors is rather small, naive implementation of turning will not work, and will basically stop during turning. By recording the previous 3 moves, my implementation could detect and react upon the following scenarios.

a) recover from turning out of track during straight line driving
b) keep turning when sensor could not see the track but hasn't fully turned
c) [cheat protocol] drive back a little when encounter a sharp turn
d) stop at a right-angled bend after driving straight

Notice that during the implementation of line following, the mbed receives commands to drive the motor at a constant pace(1 command per 200ms). The current PI controller could not make the turn fast enough, and better performance is achieved by setting the duty cycle of pwm control to a constant value during turning.

bash scripts

The reason of providing these bash scripts are as follows.

- this project requires numerous testing, and every time after the embedded platform boots up, a lot of initialization needs to be done.
- To grant r/w access to the devices every time the device is reconnected.
- To source devel/setup.bash in order to run ros packages, as well as every time after one modifies a package and builds the workspace with catkin_make.
- To reconnect/re-configure wifi when it loses connection/re-powered.
- To re-power the USB port on BBB after it's shut down.

All scripts can be found under **beagleboneSide/scripts** directory. **init.sh** is used every time after the system on embedded platform boots up, and will grant r/w access to LIDAR and mbed, source .bashrc to configure distributed ROS settings, source devel/setup.bash to run custom ros packages and syncronize client with server using chrony. The other three scripts are running in background periodically enabled by crontab. A reference for the usage of crontab is provided here:
http://alexsleat.co.uk/2011/01/09/a-more-elegant-solution-to-ubuntu-wi-fi-reconnecting-issue/

Conclusion

Future Development

a. Navigation control by speed

Current implementation: (simplified version)

1. calculate desired left/right motor speed based on velocity messages (linear.x, angular.z) published on /cmd_vel topic provided by move_base.
2. output action command(left, right, forward, backward in an int8 type message) to mbed based on the relationship between left & right motor speed.

Potential improvement: (complete version)

The implementation can be further expanded to publish/subscribe pair that can handle control messages containing velocity requirement, i.e. in the form of "BMAIN LEFT 0.25 RIGHT 0.45 EMAIN". On receiver side, the velocity information will be parsed down in control_to_mbed, and sent to mbed as inputs for PID controllers' goals.

A complete base_controller should be able to take in the figures in **geometry_msgs/Twist** message directly without mapping to discrete actions. However, the reason that the current implementation works is that during planning, the navigation stack implements dead-reckoning mechanism, i.e. the slave will receive commands periodically due to its pose and relative location at the time. Thus, it breaks the continuous plan/trajectory into discrete steps, and should be able turn specific angle with different speed goal set for left/right motors. However, current implementation just arbitrarily set the speed for left/right motors, which won't

reflect this implementation. This change with lead to better navigation as long as the control loop is responsive, i.e. the PI controller can react fast enough, also leading to higher requirement for wheel odometry provided by encoders.

b. Advanced obstacle avoidance handling

Currently obstacle avoidance is only achieved by setting inflation in local/global costmaps based on laser scan data. It should also be able to detect moving object of proper size and reflect the obstacle detected to local planner. The functionality should be achievable through further tuning on the parameters of the costmap, or building a vexel map with additional sensor sources to enrich local costmap.
ref: http://wiki.ros.org/costmap_2d
http://www.aerialroboticscompetition.org/2011SymposiumPapers/BITS-2_2011.pdf

c. Ability to abort navigation on detecting unreachable goal

Now the cancel of unreachable goal is achieved through manually execute this command:
rostopic pub /move_base/cancel actionlib_msgs/GoalID -- {}
However, it should be incorporated into current implementation of **nav_control** by adding in a publisher to publish to /move_base/cancel topic when detecting an unreachable goal. Otherwise the current planner will keep giving a velocity command of turning around in circles.
ref: http://answers.ros.org/question/57772/how-can-i-cancel-a-navigation-command/

d. Further research goals

Robot city is really a great vision to chase, but I'd suggest that a more practical approach is to further researching into more advanced and improved navigation capability. Since the mobile platform is built with decent control/mapping/navigation capability, it could finally be used as autonomous vehicle testbed. Avoiding moving object in runtime can definitely be the next reachable goal. Another topic could be adapting APEX DSL to real-time implementation, i.e. write a custom node to execute a set/sequence of velocity commands for specific functionality, such as lane change, traffic light handling, pedestrian scenario handling etc. Meanwhile, more sensor sources can be added to the existing platform for new functionality, such as a digital camera (http://wiki.ros.org/Sensors/Cameras) for video recording and image/video processing to handle traffic light scenarios, or hall effect sensors for GPS functionality.

Bookmarks:

1. DFRobot hardware parameter sheet
   http://www.dfrobot.com/index.php?route=product/product&product_id=98
   http://www.robotshop.com/en/dfrobot-4wd-arduino-platform-encoders.html?gclid=CjwKEAjwx9KpBRCAiZ_tgYKWvhQSJABQjGW--Ypt_Migjfc426Nasrbxb6b9wLQvnEr3Rd70H85p_xoCtPTw_wcB
2. Establish ros on mbed
   https://developer.mbed.org/users/nucho/code/rosserial_mbed/
   http://developer.mbed.org/users/jjzak/code/rosserial_mbed_lib/
   http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber
3. Get reliable wifi on BBB:
   https://groups.google.com/forum/#!topic/beagleboard/9KCIs7yqsa8
   http://embeddedprogrammer.blogspot.com/2013/01/beaglebone-using-usb-wifi-dongle-to.html
4. Setup Ubuntu & ROS on BBB
   a. download image file (ubuntu 12.04)
      http://rayhightower.com/blog/2014/01/02/beaglebone-black-ubuntu-part-1/
      http://elinux.org/Beagleboard:Ubuntu_On_BeagleBone_Black
   b. partition and expand the disk
      http://elinux.org/Beagleboard:Expanding_File_System_Partition_On_A_microSD
5. BBB Battery bootup
   http://elinux.org/AndiceLabs:PowerBar
6. MicroSD bootup instead of eMMC
   http://www.twam.info/hardware/beaglebone-black/u-boot-on-beaglebone-black
7. Sparkfun H-bridge tutorial
   http://bildr.org/2012/04/tb6612fng-arduino/
   https://www.sparkfun.com/datasheets/Robotics/TB6612FNG.pdf
8. Improve mapping through adding in encoder/IMU data
   http://answers.ros.org/question/44639/how-to-broadcast-a-transform-between-map-and-odom/
   http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom
   http://answers.ros.org/question/39132/improve-gmapping-results/
9. Real-time exploration with hector_mapping
   https://github.com/DaikiMaekawa/ros-navigation2d-example
10. Mailcar project
    http://mailmanforpenn.blogspot.com/2013/12/ese519-final-project-lidar-mailcar-team.html