Basic Autonomous SLAM with Turtlebots

Daniel Heideman

1 Summary

This report covers the test code written to get a single turtlebot robot to autonomously map a room using the ROS "GMapping" simultaneous localization and mapping (SLAM) package. This project is intended to show that it is possible to read from a map as it is being made by the GMapping node to determine what the turtlebot knows about its surroundings. The GMapping node builds the map and localizes the turtlebot in that map. This autonomous mapping node reads from that map and figures out where its location corresponds to in the map, looks at its surroundings and moves accordingly. The test program is supposed to spin the turtlebot in place until it knows enough about what is to its sides, then proceed forward until the area to its sides are again unknown.

2 Big Picture

The ultimate goal of the use of turtlebots and ROS in Dr. Cortes's and Dr. Martinez's labs is to test multi-robot control systems with real robots. The current goal is to get multiple turtlebots to work together using SLAM algorithms to map out a room and create one master map from which the turtlebots can all navigate. The GMapping SLAM localization algorithm already exists for ROS, so our work will focus on making the turtlebots work autonomously, independently and together as a group.

A TurtleBot 2, with Microsoft Kinect 3d sensor and Kobuki base

3 My Contributions

My work on the turtlebots is involved in the motion of the turtlebot along with Randy Lewis. I have been looking specifically into autonomous motion, while Randy has been looking at image data from the Kinect sensor. Both will form the base of the lab's ROS system, upon which we will be able to write more complex nodes with test algorithms.

In order to get the turtlebots to autonomously map a room in a swarm, we first need to know how to read the maps they create. Using one turtlebot and running the GMapping node, it has been my job to get the turtlebots to react to their surroundings, if only very basically, to show that we can. I have written nodes that can read the map and determine the turtlebot's current position in the map, the next step being to react to the surroundings. Further work will build on this to test more complex algorithms.

4 Methodology

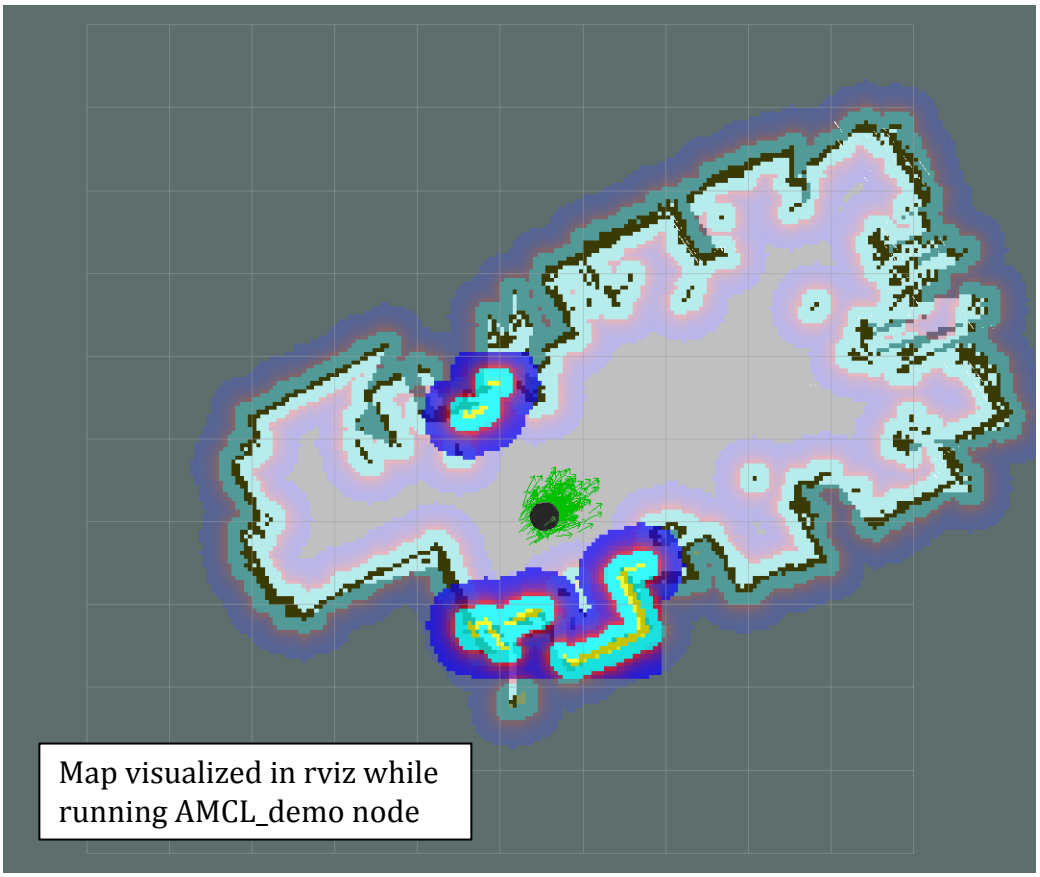
This section contains a list and high-level descriptions of test programs completed during Spring Quarter 2014. The autonomous SLAM mapping node was written with multiple proof-of-concept nodes in these steps:

-Go towards specified location on known map using AMCL demo node
-Read map messages into node
-Read tf transform between map and turtlebot
-Look in immediate surroundings, display information to screen
-Autonomously map room with simple algorithm

4.1 Move to specified location on known map

To figure out how locations are handled in a map generated by the GMapping node, we loaded a map and moved the turtlebot to different positions and watched it move. The demo AMCL node was run to navigate the map. Using rviz, it was possible to click on a location on the map, and the turtlebot would move to that location. The turtlebot's goal is set with a stamped pose message sent on the /move_base_simple/goal topic, so by echoing this topic we can figure out the coordinates of where we clicked. Based on where rviz tells the turtlebot to go, and where the turtlebot ends up, we can get an understanding of how the GMapping map coordinates work.

The GMapping nodes use the same Pose topic as used for Turtlebots. Coordinates in x and y are given in meters from an origin, the origin being where the turtlebot was when the GMapping node to make the map was started. Angle is given in radians, again relative to the origin. This origin does not appear to be in the center of the map image.



Map visualized in rviz while running AMCL_demo node

4.2 Read map messages into node

The GMapping node publishes the map on the topic /map, which the autonomous mapping node subscribes to, saving the newest instance of the map. The GMapping node does not appear to output new maps at a set rate, rather varying from one every 6 seconds to one every two minutes. It appears to send more updates when the map is updating quicker, such as when the turtlebot is mapping unexplored or uncertain areas, as opposed to known areas.

The map is saved as a .pgm image, with about 800x500 pixels for our lab. As a ROS message, it is saved as an integer vector in row-major order, which, given the width of the image, can be viewed as a matrix. The values start at the bottom left corner and work their way across right and up. Values of -1 represent unknown areas, values of 0 are areas known to be open, and 100 denotes a space known to be occupied. The header of the map topic contains:

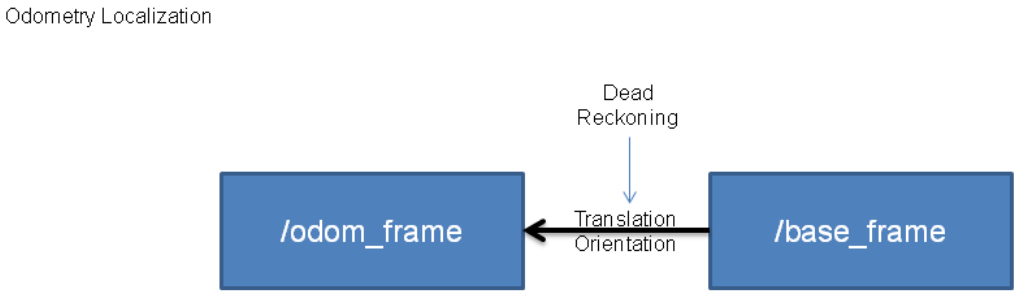
time map_load_time
float32 resolution
uint32 width
uint32 height
geometry_msgs/Pose origin

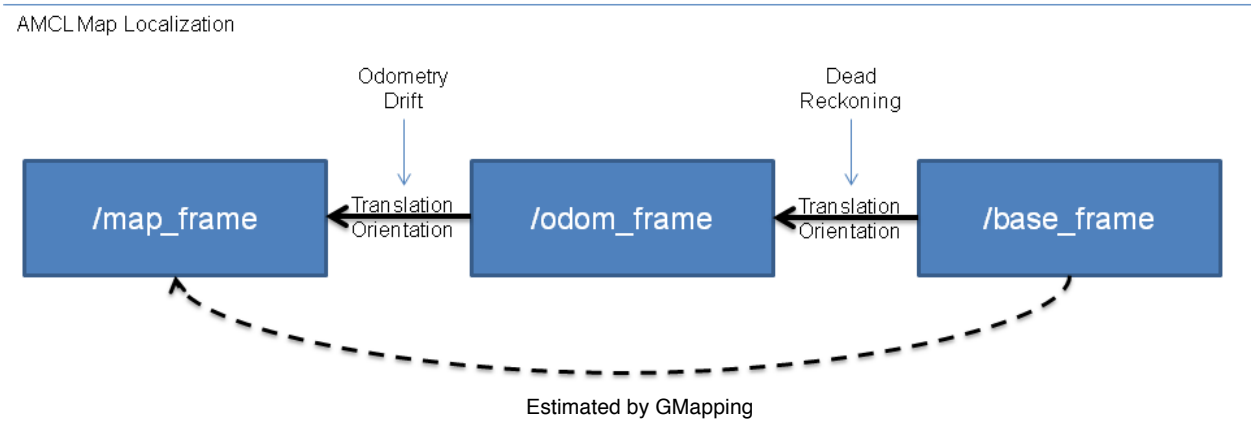
We can get coordinates, as we have the width and height of the image. We are given the pose of the bottom left corner of the image and the resolution/distance per pixel, so we can convert between pose and pixels in the map.

Full map image of lab

4.3 Read tf transform between turtlebot and map

To get the position of the turtlebot in the map it is making, we need its pose. All pose values are relative to the point at which the turtlebot started mapping. However, just looking at the /odom values will not be very accurate, since the GMapping node localizes as it maps. The GMapping node accounts for errors in the /odom values by comparing what it sees to what it has already mapped. These corrections are included in the tf relationship between /base_frame, or the turtlebot base, and /map_frame.





http://wiki.ros.org/amcl

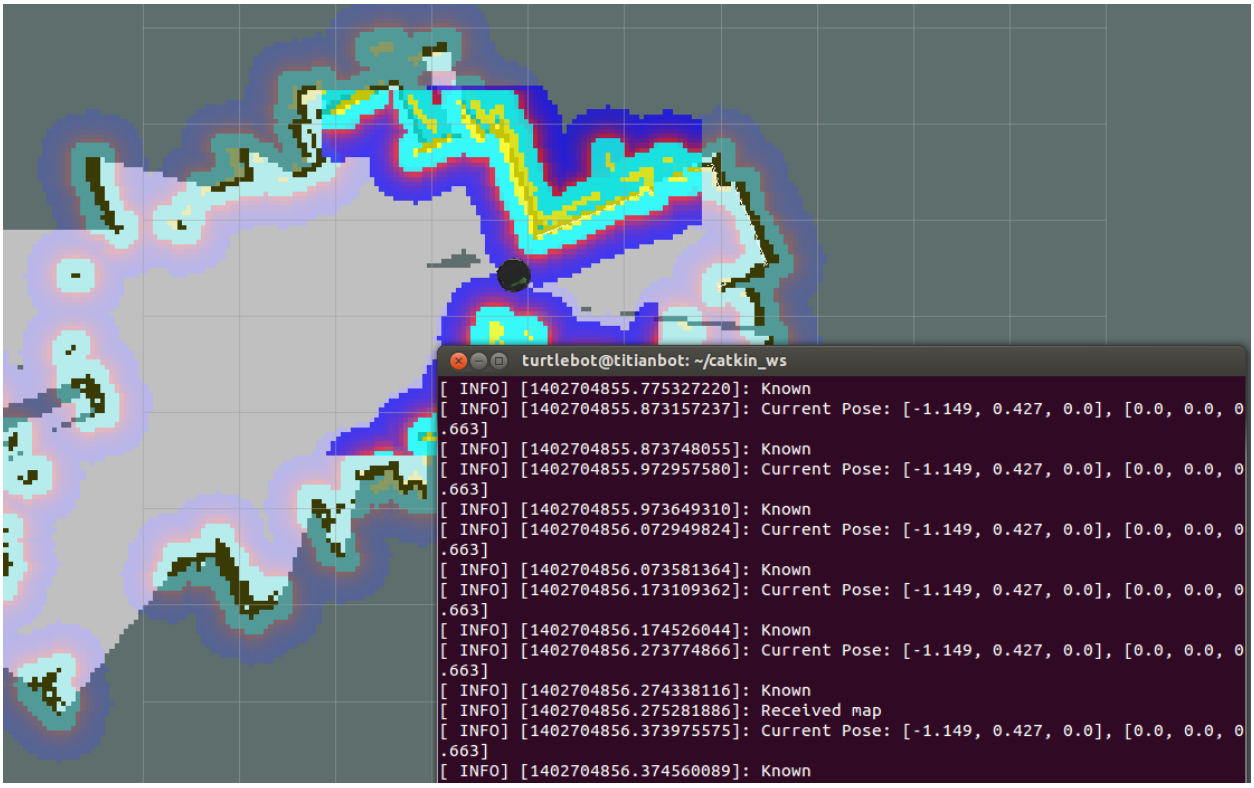
In spinordrive.cpp, a tf listener is set up as a global variable, as there were some problems with it being defined in the main function. However, tf listeners cannot be initialized before the node handle is created, within main. The solution to this is defining a pointer that points to the listener, initializing it to NULL, and constructing the listener object in the main loop, after the node handle has been created.

The transform between the turtlebot and the map is read by the listener and saved into a StampedTransform message. The transform is similar to but not directly compatible with the Pose messages used by the turtlebot. The main difference is that the orientation/rotation values are given from -1 to 1in tf and in radians, $-\pi$ to π, in Pose. The tf rotation in spinordrive is multiplied by π when being saved to the robot's Pose. To check the tf listener, the robot's pose is written to the terminal screen.

4.4 Look in immediate surroundings and display information to screen

To make sure the turtlebot is localizing itself within the map properly, spinordrive.cpp looks directly to its left to see if it knows what is there. It looks through each pixel until it reaches a space that is not empty. If the not-empty space is occupied, the turtlebot knows everything to its left and displays "Known" to the screen. However, if that space is unknown, there is more to explore to the turtlebot's left, and it displays "Unknown".

Before proceeding further, it was important to verify that the node was properly relating its position in the room with its position in the map. Tests with spinordrive indicate that the turtlebot knows to within less than a meter of where it is on its map.



rviz representation of map while running GMapping node and output of spinordrive.
In this image, the front of the turtlebot is pointed up and right, roughly parallel to its nearest wall.

4.5 Autonomously map room with simple algorithm

This step of the project was not completed by the end of the quarter. The turtlebot is supposed to gauge how well it knows the area to its sides, and map it if it is not known enough. Once it knows enough about the area, it should move on and repeat. Once the node ends, the node should have a complete map of the room. The spinordrive.cpp program was to do this, and contains most of the code to read its surroundings as planned, lacking only the decision-making algorithm and movement commands.

5 Tips

I have found that while trying out a new part of ROS, it is best to try a test with parts you know are reliable and build from there. Hence the multiple small steps in this project. ROS is not very intuitive, so it is usually a good idea to test the basics first. For instance, a subscriber node might be tested by subscribing to something that you know is being published to consistently, and it might only write the data to the terminal. After this test, you at least know the subscriber works and that the problem does not lie elsewhere in a more complicated program.

With nodes that rely on readings from messages, it is important to check whether you have received that message before you try to use it, or that it is initialized. Otherwise, it will generate a segfault. If the structs are not initialized and your ROS::spinOnce() command is at the end of your loop, you will never have that message the first loop, since you have not looked for those messages yet. As most of these messages are large data structures, initializing would be complicated, so I tend to use Boolean values to check whether a message has been received.

6 Pitfalls

The biggest problem in this project was a particular quirk of tf. As mentioned in section 4.3, tf listeners do not appear to work when defined locally in main, yet must be initialized after the node handle is created. This required that the listener variable used be a pointer that was initialized as a global variable as NULL, to be initialized later in main(). It works, but I do not understand why the program failed to compile when the object was a local variable in main, as it is only used in main.

Tf is different enough from ordinary ROS topics and messages to take me a few hours to get a simple tf listener working. While tf transforms can be accessed on the /tf topic, they usually are read with a tf listener. Within either there is a lot of information about relationships between various tf frames. Anything that collects data or moves has its own frame, and the transform between any two of them can be accessed. With a listener, this is done with the listener's lookupTransform method and saved to a tf::StampedTransform data structure.

The tf transform data structures are very similar to geometry_msgs/Pose messages, but cannot be used interchangeably. There are however tf services that will convert between the two, or the values can be individually copied over. The one significant difference is that while the orientation/rotation value for straight forward is 0 in both, transform gives -1 to 1 for all other angles, whereas Pose uses $-\pi$ to π. Both scales appear to be linear, with a simple factor of π being all that is needed to convert between the two, but this has not been dealt with yet in this project.

It appears that the tf data can be accessed at any time with a listener, as opposed to ordinary messages, which must be caught in a callback function when the data is published. This means that data can be local to any method instead of being copied from a callback function through a global variable. While I am not sure whether this makes much of a difference in the operation of the nodes, it does make it easier to read and write the code. That perk is negated by the fact that a global listener pointer appears to be necessary for tf listeners to work.

7 Conclusions & Future Directions

This project shows that we can read data from a map as it is being generated by the GMapping node and analyze the robot's surroundings. From here, algorithms determining where to go to better map the room can be run to make the turtlebots autonomously map a room. While this would initially be limited to individual robots, it should be possible to expand this to multiple robots working in concert to map the same room.

The next step in this project is to complete the simple autonomous mapping node that was to end the project this quarter. This node would only move down a straight path and only analyze the area immediately around it, so more advanced algorithms will be required to make the mapping more efficient, effective, and work in larger and more complex spaces. These algorithms should look for patterns, anticipate features of the room and search accordingly. This project has been just to make the platform work and be able to test such algorithms, so the simple mapping algorithm would be sufficient.

To make multi-robot systems work, basic multi-robot functionality is more important than advanced single-robot mapping, so following the completion of the simple autonomous mapping node, the next step would probably be in relating the individual robots' maps through tf and determining where the other robots are in the map. A test of this could be passing the tf between the two robot bases into the autonomous mapping node and displaying its Pose. However, finding the tf transform between the two in actual operation would be much harder. Relationships between origin points, relative positions and orientations of the robots, or compared and overlaid maps all could be used to find the transforms, but I have no idea how to incorporate the latter two. Given specific origin points, this can easily be estimated, but this is the least robust of the three and requires precise placement. Given that the other two are more related to custom SLAM or AMCL algorithms, it might be best to keep this particular project to storing the relations and little more.