Journal of UAB ECTC
Volume 17, 2018
Department of Mechanical Engineering
The University of Alabama, Birmingham
Birmingham, Alabama USA

LOWERING THE BARRIER-TO-ENTRY OF ROBOTICS DEVELOPMENT THROUGH THE USE OF ROBOT OPERATING SYSTEM

Matham Latif Al-Saaty, Kevin McFall PhD
Kennesaw State University
Marietta, Georgia, USA

ABSTRACT

In this paper Robot Operating System (ROS) will be used to integrate LiDAR (Light Detection And Ranging), wheel encoders and a rover with the desired result of creating a robot capable of autonomous navigation, without any specific tasks in mind. The goal is to provide an informal evaluation on the effectiveness of ROS as a means of enabling robot development and exposition on the process to aid and encourage other such endeavors. The process will include solving hardware communication, localization, mapping and navigation through the ROS environment. It will also test the limits of mobile processing power provided by the mounted single board computer.

KEY WORDS: *Robot Operating System, Localization, Mapping, Navigation*

INTRODUCTION

As robotic technologies have advanced, newer challenges continue to be proposed and solved, but the research required to build an autonomous or otherwise sophisticated robot leaves it unattainable for many. A core component of robot autonomy, localization, often relies on a Kalman filter for estimation of robot pose and for eliminating sensor noise. To implement a Kalman filter one would need extensive study into statistics, dynamics, linear algebra and control theory. This is without regard for the fact that the Kalman filter algorithm is only suitable for linear systems. Variations such as the Extended or Unscented Kalman Filter (EKF and UKF) are necessary for real world applications, which require a grasp and integration of even more disciplines such as calculus. Even then, experts consider the EKF to be "difficult to implement, difficult to tune and only reliable for systems that are almost linear on the time scale of the updates" [1]. This high barrier-to-entry limits research pertaining to autonomous robots to a post-graduate or equivalent level.

Many software development teams have sought to aid in enabling robotic research. ROS, an open-source platform, attempts to do this by providing "libraries and tools to help software developers create robot applications" [2]. Its principle is that collaboration and reusing code allow research and development to focus on specific solutions as opposed to the entire development of a new robot.

HARDWARE

The rover used is a Lynxmotion A4WD1 including the typical chassis with development platform, RC truck tires, and 12 V DC motors. It is outfitted with a Sabretooth 2×12 driver for variable motor control, a Lynxmotion BotBoarduino for motor input and a 4-cell Li-Po (Lithium Polymer) battery to deliver the necessary power. A Lynxmotion PS2 controller kit was used for wireless teleoperation during the testing phase. For sensing, Slamtec's RPLIDAR A2 provides 360-degree planar range scanning and two Lynxmotion motor encoders fed to an Arduino Uno provide wheel position data. A Raspberry Pi 3 loaded with Ubuntu Mate 16.04 and ROS were used as the processing unit communicating with all the hardware. A picture of all the hardware connected and secured to the chassis is shown on the next page. An Arduino Uno was also provided with the kit, programmed for odometry calculations via dead-reckoning. Due to the exploratory nature of this project a full graphics display was used with Virtual Network Computing to access the desktop remotely.

In initializing this setup, it became apparent that a solution for on-board power was necessary. Out of the box, the Raspberry Pi requires power via micro-USB, and the power needed to start up the LiDAR scanner can overdraw the Raspberry Pi (1.2-1.5A compared to 1.2A max). However, if a 5 V regulated power supply is available, USB devices can be powered by splicing into the voltage and ground wires. Thus, a 12 V to 5 V DC-DC converter was used to power the Raspberry Pi and LiDAR scanner from the Li-Po Battery.

Dealing with Li-Po batteries can be very dangerous, so research and adequate precautions are well-advised. In this case, long hours of programming make it easy to over-drain the cells and make the battery unusable. A simple protection circuit is advised, but in some cases a voltage meter, with a careful eye is sufficient.

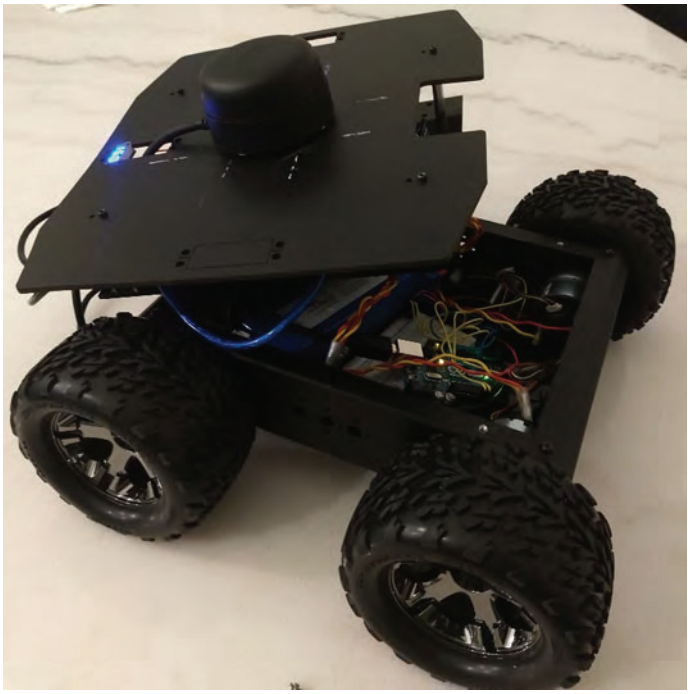

Figure 1 Final hardware configuration

INTRODUCTION TO ROS

While ROS is designed to help robotics development, it is not an easy system to learn; a strong background in programming and adequate time to familiarize with the environment are strongly encouraged. Understanding the ROS eco-system starts with nodes and topics. Nodes are computation processes and topics are the communication bus which nodes typically communicate through. Nodes are designed to be modular and as such, each device should be handled by a separate, dedicated node. Topics are public names in which messages can be published to or subscribed from. They offer a decoupled transport system that allows any process access if the required message type is matched. Messages are data structures that can consist of basic variables or arrays of basic variables. Multiple nodes can be combined with the necessary configuration files and supporting data into packages which can then be shared and installed on any ROS system. This is where the architecture begins to shine, as packages have been developed for most generic robot needs and can be installed in a modular fashion. Device drivers such as for the RPLIDAR have been wrapped into downloadable packages that are built ready to publish to the appropriate topics. Other useful packages include simulation, visualization, and analysis suites, as well solutions to problems such as implementing an EKF for robot pose estimation, arm manipulation, navigation, data conversions, and even simultaneous localization and mapping (SLAM).

IMPLEMENTATION OF ROS

In this workspace there are 3 device nodes, one to publish the LiDAR scans, one for sending an 8-bit integer command to the motors, and one for interpreting the encoder data and publishing the wheel position. The topics they employ are a motor command and encoder value per side, and the laser scan. These are the active streams of real-world data upon which everything else builds. Note that forcing the system to communicate through these topics safeguards the hardware from improper use, also known as abstraction. The next step is to find the appropriate packages that can provide the desired functionality of the robot and the package requirements. In this case the navigation stack of packages is highly-documented and offers the desired goal of path-planning by taking in a desired goal point and publishing local velocity commands for the robot i.e. driving forward and rotating [3]. To do this, it also requires odometry, a laser scan, and optionally localization and a built map. To satisfy these requirements, the system needs to translate velocity commands into motor values, and wheel movement needs to be translated into odometry with regards to the global reference frame.

To have an accurate response to global velocity commands the differential-drive package was used [4]. It employs a PID (Proportional-Integral-Differential) controller per side, with the control variable configured to match the 8-bit integer taken by the motor controlling Arduino and process variable measured by changes in encoder values converted to meters per second. PID gains can be extremely difficult to tune manually and a standard method or tuning software should be applied. For this study, the Zeigler-Nichols method was used [5]. It is performed by increasing the proportional gain until a step response produces steady oscillation. The recommended proportional, integral, and derivative gains can be calculated using two measurements in this state: the ultimate gain, K_U, which is found experimentally as the proportional gain needed to achieve the steady oscillation, and the period of the oscillation, T_U. Different control types, shown in table 1, can be used depending on the desired characteristics of the response.

Table 1. Zeigler-Nichols Gains

Rule	Proportional Gain	Integral Gain	Derivative Gain
Classic	0.6 K_U	0.5 T_U	0.125 T_U
Pessen Integral	0.7 K_U	0.4 T_U	0.15 T_U
Some overshoot	0.33 K_U	0.5 T_U	0.33 T_U
No overshoot	0.2 K_U	0.5 T_U	0.33 T_U

The package then rearranges the simple set of dynamic equations in Equations 1 and 2 to translate the robot frame's x and Θ velocities into wheel speeds. Note that x and θ in the robot frame correspond to forward and positive rotation movement as shown in figure 2 and that velocity, the rate of change in position can be represented as just the change (Δ) in position per period, in this case per refresh. L_{wheel} and R_{wheel} correspond to the distance that the left and right wheels travel, and w is the width of the mobile robot. It should also be noted that movement in the local Y direction will always be 0 because the robot cannot drive sideways.

$$\Delta x = \frac{\Delta Lwheel + \Delta Rwheel}{2} \quad (1)$$

$$\Delta \Theta = \frac{\Delta Lwheel + \Delta Rwheel}{w} \quad (2)$$

Figure 1. Robot frame convention

ROS can now control the robot as needed with generic commands, without regard for the hardware or setup specifics.

Visualizing the robot within the environment is the next requirement to set navigation goals, but a few more ROS concepts need to be discussed first. For a representation of the robot model and to display its movements, ROS uses Universal Robot Description Format (URDF) files written in XML (eXtensible Markup Language) [6]. It describes the robot as multiple shapes (box, cylinder, sphere or even a mesh loaded from file) known as links connected via joints. Links are fully dimensioned with visual options for materials and can store data of mass and rotational inertias for force calculations. Joints can be fixed attachments or movable and are defined by coordinate frame transformations, performed by a rotation via roll pitch yaw and translation in the x, y and z axes. Motion of joints such as wheel rotation or arm translation is defined through a joint state publishing node. In this case, joint state publishing only shows the wheel rotation but in other cases such as arm manipulation it is crucial that dimensions are represented precisely as it will affect performance and calculations. The resulting robot model created by the URDF is shown in figure 3.

Coordinate reference frames are also used to define the robot pose relative to the world frame. In a standard ROS setup however, there is no single 'world' frame; instead there is the map frame, and an odom frame [7]. These both start as the world origin, with the robot pose in map frame defined by external sensors and the robot pose in odom frame defined by wheel odometry. This design allows for accurate short-term reference to odom but due to growing error from slipping, wheel compression and other complications in the dynamic model, sometimes called drift, the map frame is necessary for long-term reference.

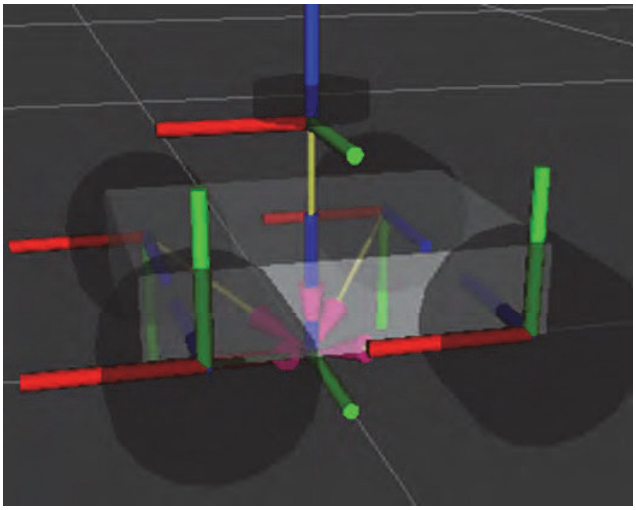

Figure 2. Robot model

Sensors also produce discrete jumps in pose estimation each reading, making odom necessary for continuous smooth movement in local maneuvers. Localization methods can take advantage of this structure and compensate for drift by defining the transformation between map and odom, thus creating a single fused pose prediction. All frame transformations need to be defined during initialization to utilize ROS's visualizing suite, rviz and are often handled by the tf node/package. The tf package also has a filtering function to help synchronize sequentially generated frames via message timestamps. If produced by the same source, messages can be synched to be published at the same time, or in the case of comparing multiple sensors, time thresholds can be set to ensure frames are being published at the desired rate and that the most recent is compared/used for calculation. This will also generate log errors whenever a process times out while waiting for transforms (or other messages) with acceptable timestamps. The final frame tree structure can be seen in figure 4.

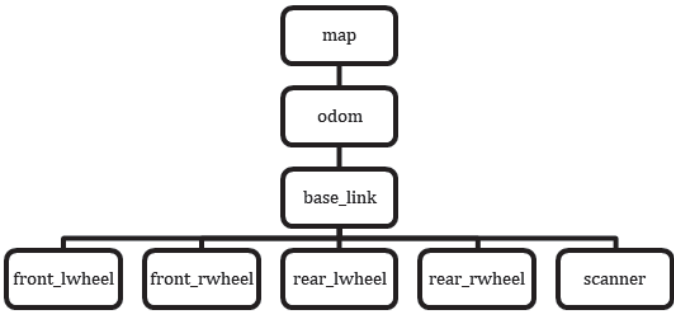

Figure 3. Full frame tree

While the navigation stack considers providing localization and mapping to be optional, all introductory tutorials assume this is completed. As such, utilizing a 2-D SLAM package would make achieving navigation much easier. The hector_slam package was used because of its clear documentation, set up simplicity and minimal CPU usage

compared to other SLAM implementations [8]. Even though this is the most computationally intense component so far, no additional requirements need to be fulfilled; using hector_slam is as simple as installation and feeding input parameters such as the laser scan topic name. It relies solely on laser scan matching, making the process less intensive but does not provide any process for comparing the laser scan to the odometry. To use both sources it would require manual switching of map and odom frame references or periodically setting the odom transform equal to the map transform effectively resetting the drift. Even with hector_slam being a minimal solution for SLAM it is effective and taxing on the hardware.

The robustness of mapping can be experimentally evaluated by revisiting the same feature and measuring the drift between subsequent scans. In occupancy grid representation this can be shown by the thickness of a wall in grid cells multiplied by the resolution, provided that the resolution is small enough to see this variation. In full exploration of a 10×6 meter room and a resolution of 5 cm/cell, the largest variation seen was 3 cells, or 15 cm; however, most walls were between 1 and 2 cells thick. The map of a smaller room is used to illustrate this in figure 5.

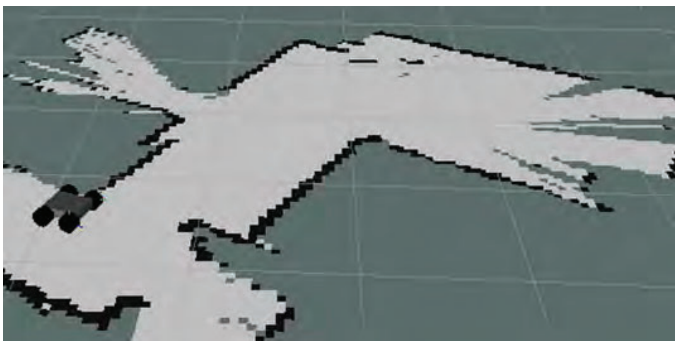

Figure 4. Occupancy grid cells produced by hector_slam

Despite the success of hector_slam it was clear that the limitations of the hardware had been reached. The system frequently overheated and subsequent hardware deterioration was evident. Performance continued to drop and without rebuilding a more optimized software system, no further progress would be possible. The final architecture of the ROS system can be seen via the node and topic message flow in figure 6. The ovals are nodes, the boxes topics and the two larger boxes, arduino_uno 1 and 2 are namespaces, another way to reuse code or duplicate nodes while avoiding name conflicts.

FUTURE OPTIMIZATION

First, the transform from wheel base to global movement was very imprecise at times and should be improved to ensure accurate path following. A large reason for this is deformation in the wheels and variance in friction. The robot required significantly more power to overcome static friction versus kinetic friction and was less consistent at lower speeds. Varying friction between surfaces also rendered tuned PID gains useless. In order to fix this, different wheels such as solid rubber should be used. A more complex motor control package designed specifically for skid-steer configuration should also be used to interpret global commands.

Next, using the Raspberry Pi's processing power to display and share the desktop, including 3D graphics of the robot and occupancy grid, while helpful in the startup phase, is unnecessary and extremely taxing. Running the operating system headless (no display) would help but remove all visualization capabilities. However, ROS offers distributed computing across multiple devices via internet protocol. This allows for most computation, not just display, to be processed by a wirelessly connected, higher performance machine. The only limitations are that drivers must be run at the connection of the device and that the network speed can set the data communicated in real time. As such, to further develop this robot, it would make sense to recreate the architecture across a laptop or other capable computer while only relying on on-board computing for hardware communication. This would allow for more complex SLAM algorithms that effectively fuse sensors as well as further developments.

These developments would ideally include successfully utilizing the navigation stack to produce accurate goal paths and a method of autonomously generating goals. A specific case of this would be using an exploration package to direct the robot to map undefined areas.

CONCLUSION

Although the initial goal of creating an autonomous robot was not achieved, ROS definitively aided in the process of implementing SLAM. In a similar time frame, the same would not be possible from scratch. With adequate coaching and lessons, robotics frameworks such as ROS could accelerate undergraduate studies and allow for more specialized work.

ACKNOWLEDGEMENTS

All project supplies were graciously provided by the Kennesaw State University Mechatronics Department for this study. This project is predicated on the work by the open source robotics foundation and the countless ROS community members that continue to share their work, free to use through the ROS framework.

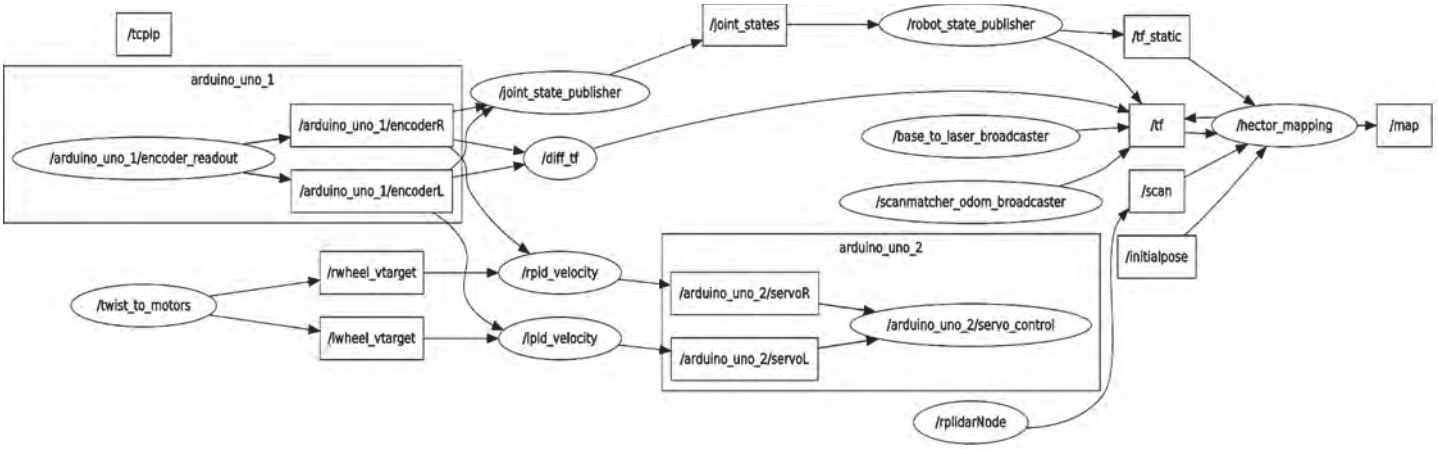

Figure 6 Full node graph

REFFERENCES

[1] S. Julier and J. Uhlmann, 2004, "Corrections to 'Unscented Filtering and Nonlinear Estimation,'" *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422.

[2] n.a., "About ROS," n.d. ROS.org, from http://www.ros.org/about-ros/.

[3] E. Marder-Eppstein, "navigation," n.d. ROS.org, from http://wiki.ros.org/navigation/.

[4] J. Stephan, "differential_drive," 2012, ROS.org, from http://wiki.ros.org/differential_drive.

[5] L. Trammell, "Ziegler-Nichols Tuning Rules for PID," n.d. Microstar Laboratories. from http://www.mstarlabs.com/control/znrule.html.

[6] I. Sucan and J. Kay, "URDF," n.d. ROS.org, from http://wiki.ros.org/urdf.

[7] W. Meeussen, "Coordinate Frames for Mobile platforms," 2010, ROS.org, http://www.ros.org/reps/rep-0105.html.

[8] B. M. F. D. Silva, R. S. Xavier, T. P. D. Nascimento, and L. M. G. Gonsalves, 2017, "Experimental evaluation of ROS compatible SLAM algorithms for RGB-D sensors," 2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)