MBot Software and Odometry Report

(ROB 330)

Akash Sarada (asarada) and Alexandru Otlacan (aotlacan)

MBot Software

This section highlights how ROS nodes work and how we used ROS nodes to communicate data to different parts of our program.

ROS Nodes Overview

ROS nodes are individually executable programs that allow a robot to perform specific actions when required. These nodes communicate asynchronously, allowing programs to both publish to and subscribe to send or receive messages. These nodes communicate over channels called ROS topics, which are the commands that activate the specific nodes required. The individual data packets transmitted over the topics are called messages, which contain specific information about the timestamp at which they were published. Usually, a program publishes to or subscribes to a node, and sends/receives the data over their respective topic repeatedly, sending/reading packages and performing actions.

Odometry Estimation using ROS nodes

ROS nodes are used for odometry estimation by allowing the encoders to publish a ROS topic called `/encoders`, which the `external_odom.cpp` file subscribes to, allowing it to receive the latest odometry values from the encoders through the latest message. A standard 3D geometry vector is used to communicate the data inside each message, which is then decoded by the `encoder_calback()` function. After decoding the message, it adds to the current robot state and calculates the dt value. This allows us to know the period at which the messages are coming, and allows us to calculate the current wheel speed, as well as the difference between the current wheel speed and current body velocity, allowing us to integrate the velocity over time to estimate the position of the robot. It then publishes the odometry data it calculated to the ROS topic called `/odom`, allowing other programs to use the values.

Motor Controller using ROS Nodes

Motion_controller_diff uses many different ROS nodes to input and output data values for its calculations. It subscribes to data topics like `/odom` and `/goal_pose_array`, which give

the `motion_controller_diff.cpp` the data it needs to calculate the pose of the robot and the movement required. It then publishes its calculations to `/cmd_vel`, `/debug_cmd_vel`, `/odom_path`, and `/goal_path`. This allows other programs to use the data (especially in `/odom_path` and `/cmd_vel`) to perform more calculations that send voltage values to the motor controller.

Odometry Estimation

This section explores our final odometry model and the sources of error.

Resulting Odometry

Our resulting odometry was (0.97, 1.03, 1.56).

Odometry Difference

The difference between the ideal odometry and our resulting odometry was (0.03, 0.03, 0.01), which is quite small. This shows that the odometry was only slightly off in more ideal conditions. However, this was done with a large amount of force being applied to the robot while it moved very slowly, which helped reduce the amount of error.

Causes for Error

However, there was still some error in the odometry, which could have been caused by a multitude of things. The wheel sensors used are quite cheap, which means that the resolution isn't very high, allowing for a bit of play in the wheels before the sensors register a tick. Furthermore, any bit of slip in the wheels or unevenness in the floor or tape would have caused the odometry to be slightly off, as it would not have traveled a perfect meter.

MBot PID Controller

This section highlights our PID values and how we arrived at them.

PID Gains

```
// PID gains used for linear (translational) control
double Kp_lin = 0.45, Ki_lin = 0.0, Kd_lin = 0.1;

// PID gains used for angular (rotational) control
double Kp_ang = 2.9, Ki_ang = 0.0, Kd_ang = 0.5;
```

PID Tuning

We spent many hours tuning the PID, attempting to autonomously move the robot to the exact real-world points we wanted to hit. When we ran the default values, the robot jittered and overshot a lot, which told us to decrease the values until the robot was stable. However, after a day of tuning, the robot was only stable at slow speeds, which introduced lots of drift. This also introduced another issue, where we discovered that our robot had difficulty driving in a straight line, heavily relying on the angular PID to maintain its straight position. After many conversations with the IA and GSIs, we decided to restart our PID tuning journey, this time attempting to keep the driving PIDs around the default, and increasing our angular PID until the robot is able to drive in a straight line. We tuned these values by increasing P till the robot overshot, then increasing D till it was accurate. Finally, we added a tiny amount of I to help the robot whenever any small errors occurred, giving us our current PID values.

Visualizing Robot Odometry

This section highlights how we used visualization software like RVIZ to see our odometry and PID in action.

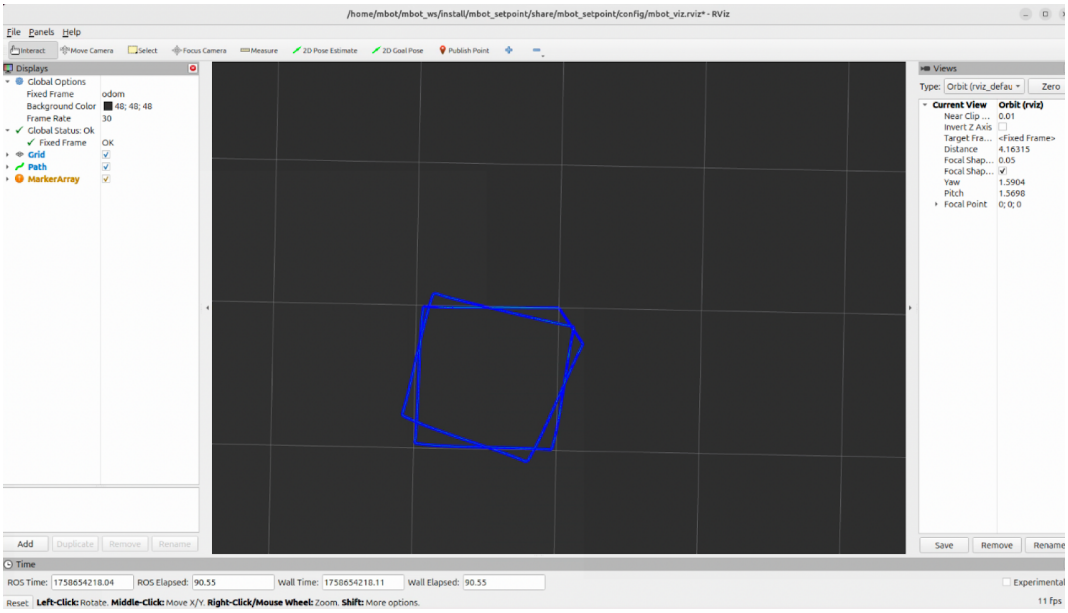

Figure 1: RVIZ Robot Odometry Pathing

Robot Performance

Overall, the quality of the robot traversal was pretty good, especially in the real world. It was able to hit all the marks around the square pretty well and achieve a pretty good result in the challenge. Furthermore, the data visually represents a square, which is great for our square challenge. It shows the sharp 90-degree angles and the almost perfect distance traversals, which made the robot very accurate. The slight offset in the second square may have been due to the slight change in the poses to make the real-world robot more accurate. Quantitatively, the robot reached the points flawlessly (within our threshold of 5cm/15 degrees), and responded accordingly to any changes while it was driving.

Overall Task Reflection

Overall, we think that we performed well, especially with the robot we were given. We had a major challenge with the inconsistencies in the drive motors, but we overcame it by tuning our angular PID to near-perfection. However, this meant that our drive PID was not the best, which caused us to miss the 5cm circle by a hair. Furthermore, we adapted our setpoints to account for the drift coming from the motors, which made the robot much more accurate when driving. We noticed that after doing so, we were able to estimate the drift of the robot and account for that, making the robot nearly perfect. Nevertheless, one piece of advice that we would give others is to ensure that the robot can drive straight and tune the angular PID till it is perfect. This approach is much better than assuming that the robot can, saving both time and energy when tuning the robot's PID.