EECS 467: Setpoint Challenge Final Report

Karl Koenig
University of Michigan
kamako@umich.edu

Brett Levenson
University of Michigan
brettlev@umich.edu

Mayukh Nath
University of Michigan
mayukhn@umich.edu

Luke Cohen
University of Michigan
lukecohe@umich.edu

1. Introduction

A core problem in robotics is that given a point, can a robot move to that point quickly and accurately. In this lab, we were tasked with solving this problem by using odometry data taken from the quadrature encoders on the wheels of the M-bot as well as a SLAM binary provided to us. In this report, we will discuss out methodology of implementing our odometric calculations and our motion control algorithm as well as our results based on just odometric readings and combined with the SLAM binary.

2. Methodology

2.1. Odometry Implementation

In our implementation of Odometry, we used the encoder readings from the quadrature decoders on the wheels as well as previous x, y and θ values to update an lcm message based on the following equations.

$$x_{new} += \cos\theta * (\Delta_{left} + \Delta_{right})/2.0 \tag{1}$$

$$y_{new} += \sin\theta * (\Delta_{left} + \Delta_{right})/2.0 \tag{2}$$

$$\theta_{new} += (\Delta_{right} - \Delta_{left})/Wheelbase \tag{3}$$

This gave us the pose of the robot based only on odometry, a pose that turned out to be riddled with accumulated error.

2.2. Motion Control Implementation

In our implementation of the motion controller, we used the suggested two state system. The robot continuously calculates the angular difference between the target heading and the current heading, and if this difference is greater than an adjustable threshold, the robot turns towards the target. Once it's within the threshold, the robot switches to the drive state and drives straight. This scheme wasn't ideal for performance (given that our goal was to drive across all four corners as fast and precisely as possible). A better scheme for performance–in terms of speed–would have been been driving in a circular shape. This would have allowed the robot to maintain more speed through the corners as opposed to stopping and turning on every corner. Even through the path length would have been longer, the robot would likely have finished the course faster due to its higher average speed. The trade-off for the performance gains this circular scheme offers is complexity and precision. The state machine for such a scheme would have been much larger, harder to implement, and, most importantly, harder to debug. Given our time constraints, we chose the simpler scheme, and focused our efforts optimizing it for speed while maintaining precision. After we finished our basic scheme, we attempted to run the code, without implementing any correction from SLAM. We quickly realized that we needed SLAM in order to drive in a path that, at the very least, resembled a square. It is clear in our figures that SLAM is crucial for this challenge. To utilize the data from SLAM, we calculated the error every time we received SLAM data, and used this error until we received the next set of data. This way we could use odometry for high frequency adjustments, and SLAM to account for the error that accumulates.

2.3. PID Tuning

Since we had two states, one for moving straight and one for turning, that meant we had two separate PID controllers to tune. We initially set both P values to 1 and the other constants to 0. We noticed that it would occasionally get stuck close to the setpoints when the speed value from our P was too low to actually cause the bot to move. To combat this we added a small I constant to the drive straight controller which fixed this problem. The next problem we noticed was that turning at slow speeds caused the robot to not turn smoothly around its center which caused issues with our map. We then transitioned to using a constant speed whenever we wanted to turn. The danger to this sort of control is having significant overshoot, but because of the dynamics of our bot we never noticed a significant issue. Finally the last issue we came across was that when moving at high speeds, our SLAM map would shift, so we reduced our P constant which made our map much stable even though it slowed our time down. We never found a need for a D term in our controller to combat overshoot. We believe this was because our tolerance was large enough to that our bot could easily brake in time. To summarize, our controller for driving straight used a P value of 1, an I value of 0.001 and a D value of 0. We changed our controller for turning from a PID controller to a bang bang controller that constantly turned at a speed of 1.

3. Results

In our best trial, we finished the course in 50 seconds, and our error for the final setpoint was, for all intents and purposes, zero. While our time taken to complete the course was about average, we had the best precision. Below we will discuss specific differences in our results based on only the odometry as well as the odometry corrected with SLAM.

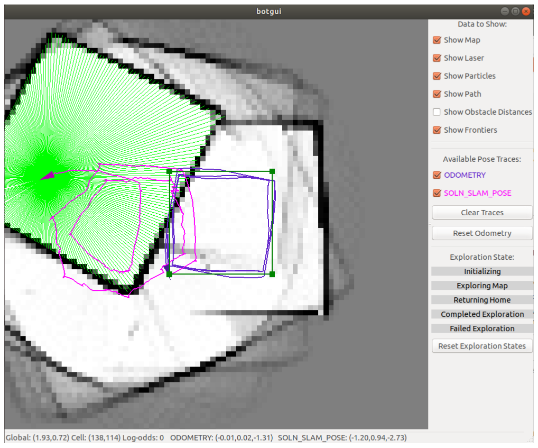
3.1. Odometry

In our testing, we found that using just the odometry data to drive to the setpoint worked poorly due to the uncertainty of the environment and motors. Wheels slippage led to the true pose of the robot and the odometric pose of the robot conflicting. This error between the two poses made it impossible to drive a true square based on purely odometry.

One of the major issues we observed came up when we investigated the cause of the wheel slippage. The root cause of the error was the wheels having insufficient traction with the floor because of a buildup of dust on the floor. This problem permeated up throughout the system because the encoders would record the wheels having traveled further than reality, resulting in an incorrect location being computed. When we did not actively clean both the floor and the wheels, we had a very noticeable degradation in tracking quality, as shown by the below figure where it is unable to even build a map. This phenomenon can be seen in Figure 1.

Once we accounted for this slack by wiping down both the field and the wheels, we were able to obtain a more realistic trace shown in Figure 2. While this is clearly far from ideal, due to the error introduced when the robot turns and drifts, we can see that it is much closer.

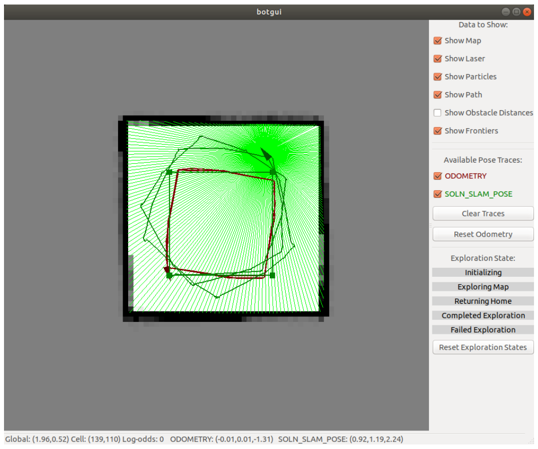
When driving a square with just odometry, we can see in Figure 3, the error continues to grow as the amount of drift continues to grow in the system. There are noticeable dips in the chart where the robot accidentally drives back towards the square. Because we are controlling the robot based only on odometry, we can see dips in the error when the robot moves towards the correct square from it's outer orbit. This explains the sinusoidal pattern as the robot goes towards the corner at an angle, and then shoots past it, resulting in a dip in the error, followed by it quickly increasing again.



✓ Show Map
✓ Show Laser
✓ Show Particles
✓ Show Path
☐ Show Obstacle Distances
✓ Show Frontiers
✓ ODOMETRY
✓ SOLN_SLAM_POSE

```
Global: (1.93,0.72) Cell: (138,114) Log-odds: 0   ODOMETRY: (-0.01,0.02,-1.31)   SOLN_SLAM_POSE: (-1.20,0.94,-2.73)
```

Figure 1. M-bot attempts to drive square based only on odometry, wheel slippage



✓ Show Map
✓ Show Laser
✓ Show Particles
✓ Show Path
☐ Show Obstacle Distances
✓ Show Frontiers
✓ ODOMETRY
✓ SOLN_SLAM_POSE

```
Global: (1.96,0.52) Cell: (139,110) Log-odds: 0   ODOMETRY: (-0.01,0.01,-1.31)   SOLN_SLAM_POSE: (0.92,1.19,2.24)
```

Figure 2. M-bot attempts to drive square based only on odometry, less wheel slippage

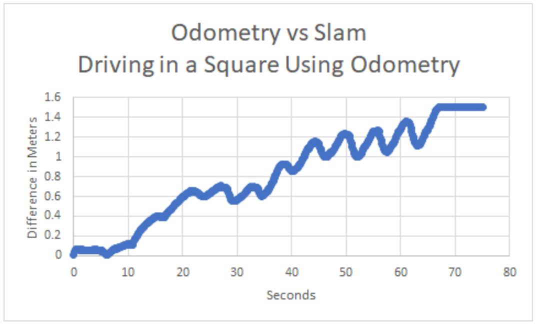


Figure 3. Error between Odometry and SLAM

In Figure 4 we can see how the error in odometry relates to the real world location through the SLAM pose. The odometry believes it is efficiently controlling the robot to the desired set point, but as we can see in orange, the true pose very quickly diverges from where the robot thinks it is. For the initial driving straight the odometry does a good job tracking the true pose with some error, but once it begins to turn, it quickly begins to accumulate error. By the time it has driven the second leg and turned, it begins to drastically differ. Eventually around 50 seconds, the error flips, and as it attempts to get closer to the set point, it ends up going further away.

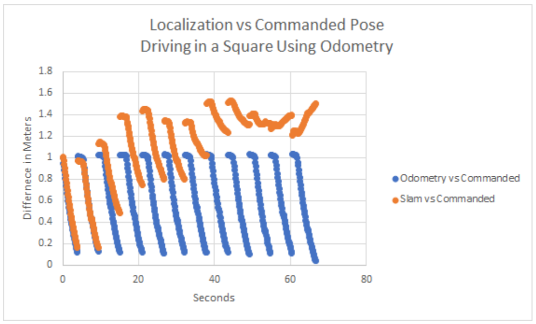
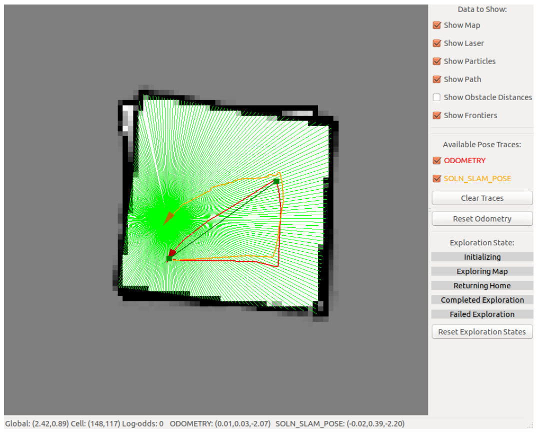


Figure 4. Error between Odometry and SLAM vs Commanded Location

When running just on odometry, any work that does not bring us closer to the goal, will make our ending state further away. This can be seen in Figure 5 where we take a longer path, and because of this, the robot thinks it has traveled an adequate distance to reach its target. The small error in the heading after the turn, increases the total error at the end because we drive a far distance at a wrong heading, which adds up to being off by a lot.



Global: (2.42,0.89) Cell: (148,117) Log-odds: 0   ODOMETRY: (0.01,0.03,-2.07)   SOLN_SLAM_POSE: (-0.02,0.39,-2.20)

Figure 5. Trace of Odometry path and the SLAM path

Additionally, when we look at Figure 6 we can see how the error increases when turning. As the M-Bot moves based just on odometry, it starts off moving in a relatively straight line with some error and during its first turn, it accidentally corrects this error. These behaviors can be seen in the graph, Figure 6 from time 0-16 and 17-35 seconds respectively. It is after this point that the turning causes a large amount of error in the pose of the robot which causes it to veer off course to the right. In Figure 5, this is represented by the SLAM pose being largely off from the odometric pose and in Figure 6, this is displayed by the large spike in error near time = 38 seconds.

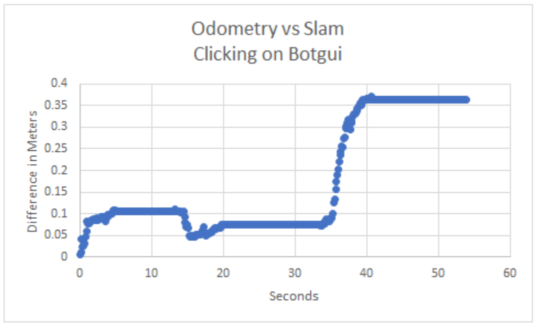
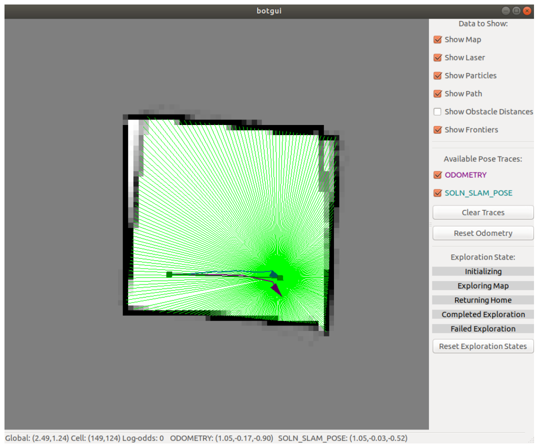


Figure 6. Error between Odometry and the true position

3.2. Motion Control

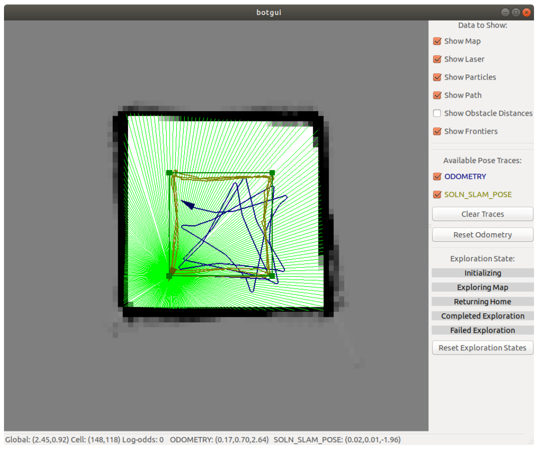
Using the data from SLAM to augment odometry, we were able to drive in a pretty decent square. In the following figures, it is evident that the robot did much better with slam.



Global: (2.49,1.24) Cell: (149,124) Log-odds: 0   ODOMETRY: (1.05,-0.17,-0.90)   SOLN_SLAM_POSE: (1.05,-0.03,-0.52)

Figure 7. M-bot attempts to drive in a line based on odometry corrected with SLAM

In Figure 8 we can see that despite the odometry being totally off (that triangular looking path) our robots path was almost a perfect square. The odometry only drifts on longer intervals, so by resetting it to a known pose from SLAM, it helps to fill in the gaps between SLAM updates. Figure 9 shows that how far our odometry is from slam without any compensation. However, since we are constantly computing the offset to reset the odometry, this higher error does not



- [✓] Show Map
- [✓] Show Laser
- [✓] Show Particles
- [✓] Show Path
- [ ] Show Obstacle Distances
- [✓] Show Frontiers
- [✓] ODOMETRY
- [✓] SOLN_SLAM_POSE

```
Global: (2.45,0.92) Cell: (148,118) Log-odds: 0   ODOMETRY: (0.17,0.70,2.64)   SOLN_SLAM_POSE: (0.02,0.01,-1.96)
```

Figure 8. M-bot attempts to drive in a square based on odometry corrected with SLAM

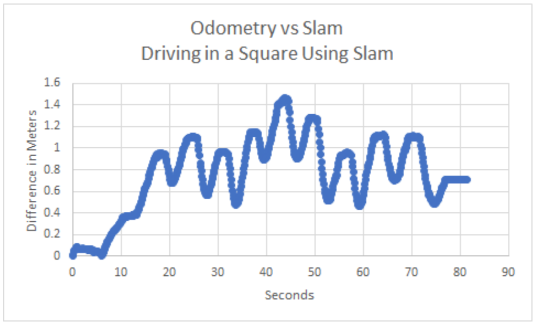


Figure 9. Error between Odometry and SLAM vs Commanded Location

affect our accuracy. Finally in figure 10 we see that our motion controller does a very good job at reducing the error from the slam pose to the set point even though our raw odometry would have too much error to complete this by itself.

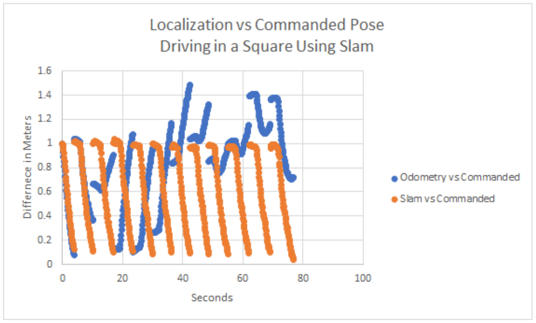
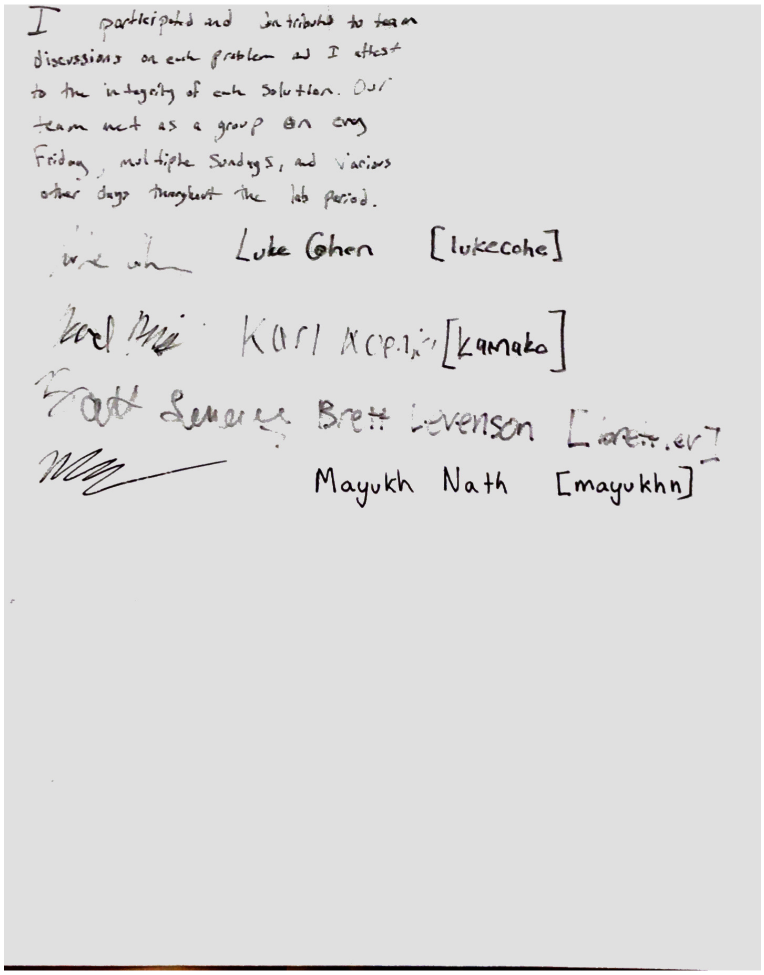


Figure 10. Error between current pose and target pose

In our best trial that we had on test day, we were within less than a centimeter of the final setpoint. We achieved this by increasing our tolerance for the distance for all the intermediate setpoints and decreasing the tolerance for our final setpoint. This way, our robot still drove over all the setpoints, but not necessarily perfectly until the final setpoint for which it had tighter constraints. This allowed us to achieve our desired precision without spending too much time trying to hit intermediate setpoints perfectly. We noticed some non-ideal behaviour from our robot. The most prominent example is that it would often drift to the left while seemingly attempting to drive straight. We initially thought this was a due to a physical difference between the motors, but when we ran some test code to simply drive straight, the robot worked perfectly. We eventually realized that the problem was in the transition from the turn state to the drive state. While turning the robot's right wheel would spin forward, and the left wheel would spin backwards. Upon switching to the drive state, the robot's right wheel would continue to spin forward, but the left wheel would have to rapidly switch directions from backwards to forwards, inducing wheel slippage, causing the left side to briefly lag behind the right and causing the robot to turn to the left, our scheme accounted for this since once the heading of the robot drifted past our tolerance, it would stop and turn until it was once again facing in the right direction. This did, however, waste a significant amount of time.

4. Conclusions

In this lab we successfully navigated the M-bot to traverse a square path, while minimizing the total time of running to 50 seconds and the resulting error at each vertex to almost zero. To achieve this we localized the robot using SLAM, and then augmented the gaps in updates with odometry readings. These two inputs allowed our robot to have a strong estimate of the true pose of the robot, which we then plugged into our PID controller, which would command the robot the correct distance to the goal. By starting off altering our angle before the translational error, we always ensure the robot travels closer to the target. Using this control system, the robot can traverse the square efficiently and accurately.



I participated and contributed to team discussions on each problem and I attest to the integrity of each solution. Our team met as a group on every Friday, multiple Sundays, and various other days throughout the lab period.

[illegible] Luke Cohen [lukecohe]

[illegible] Karl [illegible] [kamako]

[illegible] Brett Levenson [illegible]

[illegible] Mayukh Nath [mayukhn]