UNIVERSITY OF OSLO

CONTROL OF MOBILE ROBOTS

UNIK4490

Odometry and Posture Regulation for a 4 by 4 differential drive mobile robot

Autumn 2017

Authors
Eirik KVALHEIM, Daniel Sander ISAKSEN and Torgrim R. NÆSS

Supervisors
Dr. Kim MATHIASSEN and Magnus BAKSAAS

1 Introduction

The main goal of this project was to implement motor control, posture regulation and odometric localization in order to get the robot to move to a desired pose. We were working on an existing software stack running ROS (Robot Operating System), so getting to know the previous software aswell as ROS as a system, was also a goal in this project.
A significant portion of the project time was spent on reverse engineering the robot to better understand the system in order to implement our own solutions. In the end we had also spent a lot of time on tuning the regulator parameters, aswell as compensating for odometric faulties. Most of the work was done in collaboration with another group that were working on an identical robot with the same software stack.

2 The System

Figure 1 illustrates our implementation of the system with control loop. It consists of posture regulaton, motor controller, kinematic model, and odometry. Here we are setting an actual pose, with reference to the coordinate system of the desired pose. The posture regulator tries to drive this error towards 0.

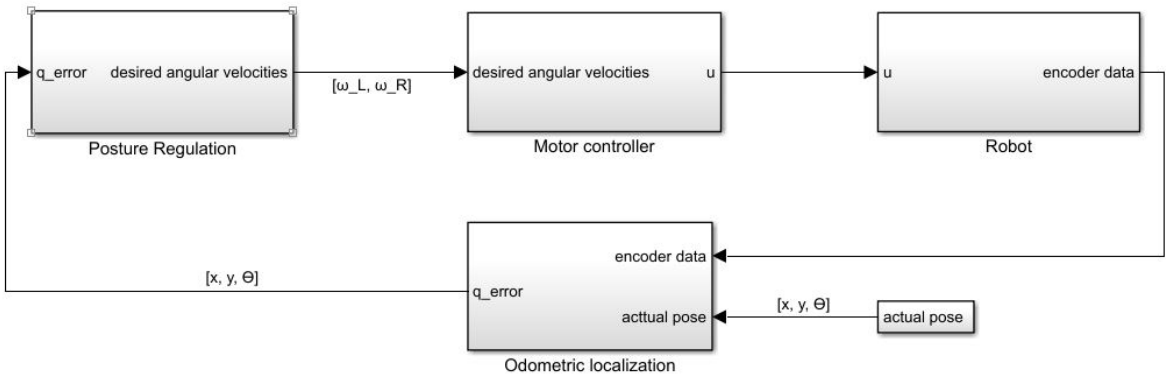


Figure 1: System with Control loop

2.1 Kinematic Model

Our systems configuration is completely described by $\vec{q} = [x, y, \theta]^T$, where (x,y) are the Cartesian coordinates of the center of our robot, and θ is the orientation of the wheels with respect to the x axis. We used the kinematic model of a unicycle derived in chapter 11.2.1 in "Robotics - Modelling, Planning and Control" by B. Siciliano et.al. This can be used because we have a 4 by 4 differential drive robot. Thus it can be mathematically viewed as 4 unicycles, which are translated into a "right wheel" and a "left wheel", and then the configuration variables of the centre of the rover is a function of the actual values of "right wheel" and "left wheel".

The kinematic model of the unicycle is given by v as the driving velocity and ω as the steering velocity, is given in its canonical form as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta \\ sin\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{1}$$

As mentioned earlier the driving and steering velocities v and *omega* are expressions for the centre of the rover, and are thus expressed as a function of the actual velocity inputs, i.e., the angular speeds ω_R and ω_L of the right and left wheels, respectively:

$$v = \frac{r(\omega_R + \omega_L)}{2} \tag{2}$$

$$\omega = \frac{r(\omega_R - \omega_L)}{d} \tag{3}$$

2.2 Motor Control

The implementation of motor control for each wheel was already available in the software stack. We started our project by reverse engineering the code in order to get the motor driver to work. This motordriver/controller had an internal pid regulator, and uses encoder data to calculate wheel velocities. As we were not familiar with the system, and since there were no documentation or comments in the code, we faced challenges on elementary problems as; communication with the robot, running the motordriver and deducing the reason behind the constants in the controller. Regarding the problem with the motordriver, it turned out that there was an usb port which needed to be taken in and out every time the motordriver runs the first time. This was the usb connecting the teensy that reads the encoders. There are still unknown constants in this code that has an effect on the published velocity, so we cannot know if this motordriver is publishing correct speeds or not. We ended up assuming that it was correct and moved on to derive the odometry and posture regulator.

2.3 Odometric Localization

The implementation of any feedback controller requires the availability of the robot configuration at each time instant. Since we are not using any external sensors for doing SLAM (Simultaneous Localization And Mapping), our only sensors for knowing where we are is the encoders located on each wheel. In order to use this information to know where we are located and heading (in a world/reference coordinate frame), we need to calculate the *Odometry*. The method used is odometry by exact integration, and can be found in chapter 11.7 in "Robotics - Modelling, Planning and Control" by B. Siciliano et.al. Here we obtain the following equations in chained form:

$$x_{k+1} = x_k + \frac{v_k}{\omega_k}(sin\theta_{k+1} - sin\theta_k) \tag{4}$$

$$y_{k+1} = y_k + \frac{v_k}{\omega_k}(cos\theta_{k+1} - cos\theta_k) \tag{5}$$

$$\theta_{k+1} = \theta_k + \omega_k T_s \tag{6}$$

Where θ_{k+1} is the angle for the next time step relative to the reference frame, T_s is the time step, and x_{k+1} and y_{k+1} are the Cartesian position of the robot for the next time step, relative to the reference frame.

v_k and ω_k is obtained by the equations (2) and (3), and thus giving us the complete odometry. The angular velocities ω_R and ω_L is obtained by subscribing to the topic */est_vel* which is published from the motordriver.

We initialize our odometry node by calculating where our robot is in the reference frame, and sending these variables as an input to the node. Assuming perfect posture control, the robot will then move towards zero position in the reference frame. One case could look like this;

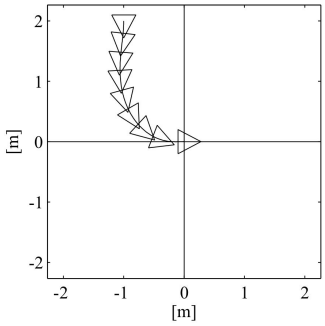


Figure 2: Localization from start to finish

2.4 Posture Regulation

We have implemented posture control from chapter 11.6.2 in "Robotics - Modelling, Planning and Control" by B. Siciliano et.al. In general the posture regulation controller takes in the configuration vector $q = [x, y, \theta]^T$ (Cartesian position and vehicle orientation), and outputs v and ω. We obtain the input vector $\vec{q}$ by subscribing to the topics */odometry* and */theta*, published by the odometry node. It is assumed that the desired variables are $\vec{q}_d = [0, 0, 0]^T$ and the error from $\vec{q}_d$ is represented by

$$\rho = \sqrt{x^2 + y^2} \tag{7}$$

$$\gamma = Atan2(y, x) - \theta + \pi \tag{8}$$

$$\delta = \gamma + \theta \tag{9}$$

where $\rho = ||\vec{e_p}||$ is the distance between current point (x, y) and desired point $(0, 0)$, γ is the angle between $\vec{e_p}$ and the sagittal axis of the vehicle and δ is the axis between $\vec{e_p}$ and the x-axis. v and ω are found by:

$$v = k_1 \rho cos(\gamma) \tag{10}$$

$$\omega = k_2\gamma + k_1 \frac{sin(\gamma)cos(\gamma)}{\gamma}(\gamma + k_3\delta) \tag{11}$$

In our implementation of the controller we get $\vec{q}$ from the odometric module and output a desired ω_R and ω_L to the motor controller. This is done by publishing the angular velocities to the topic */cmd_vel*, which the motordriver subscribes on in order to control the velocities. Equations for ω_R and ω_L expressed by error variables ρ, γ and δ, can be found by setting equation (2) and (3) equal to equation (10) and (11), and then solving for ω_R and ω_L. This yields:

$$\omega_R = \frac{2k_1\rho cos(\gamma)}{2r} + \frac{k_2 d\gamma}{2r} + \frac{k_1 d sin(\gamma)cos(\gamma)(\gamma + k_3\delta)}{2r\gamma} \tag{12}$$

$$\omega_L = \frac{2k_1\rho cos(\gamma)}{2r} - \frac{k_2 d\gamma}{2r} - \frac{k_1 d sin(\gamma)cos(\gamma)(\gamma + k_3\delta)}{2r\gamma} \tag{13}$$

Where d is the distance between the outer gripping point of the wheels (almost width of the rover), and k_1, k_2 and k_3 are the controller gains. It is worth noting that in equation (8) we are using $atan2(x, y)$, which is a function that is undefined in $x = y = 0$, therefore γ and δ is also undefined for $x = y = 0$. To avoid this problem we designed the controller to accept the position and orientation if each element of $\vec{q}$ had less then 0.05 error. At first, while trying to adjust controller gains we also tried to modify the controller further than the litterature did, to account for the weaknesses. We had no luck with this and ended up with implementing exactly the controller derived from the litterature. The funny thing was that we later understood that the controller already were proven asymptotically stable given the lyapunov candidate function

$$V = \frac{1}{2}(\rho^2 + \gamma^2 + k_3\delta^2) \tag{14}$$

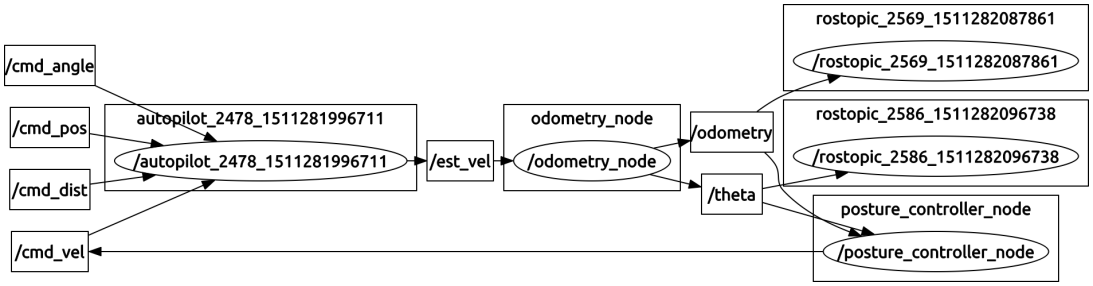
wihich yiealds the time derivative

$$\dot{V} = -k_1 cos^2(\gamma)\rho^2 - k_2\gamma^2) \tag{15}$$

where $\dot{V}$ is negative semi-definite. Further analysis of the closed-loop system leads to conclude that the system trajectory for the equilibrium posture is also tending to 0, and thus the controller is stable.

3 Testing, Results and Conclutions

After implementing the system and getting the ROS nodes to communicate, our system looked like this;



As expected, one can see resemblance to figure 1. Here you can see the motordriver labeled as the node */autopilot*.

We started by implementing recommended gains from the litterature, where $k_1 = 1$, $k_2 = 2.5$, and $k_3 = 3$, but hese gains did not provide a good regulator. Reflecting on the equations (8-13) and discussing which gains we needed to increase with regards to how the robot behaved, we came up with $k_1 = 2$, $k_2 = 2.3$, and $k_3 = 1.3$ wich was a pretty good regulator, with only a small accuracy problem. Increasing to $k_1 = 2$, $k_2 = 4$, and $k_3 = 1.3$ provided an even more accurate regulator, which conveniently happend to also be quicker.
We also found another set of gains where $k_1 = 1$, $k_2 = 6$, and $k_3 = 7$ that gave a very slow and steady regulator, with very good accuracy at distances < 2m. Over 2m the "2, 4, 1.3" regulator was clearly better on minimizing the error.
Blelow is a plot of the "2, 4, 1.3" regulator;

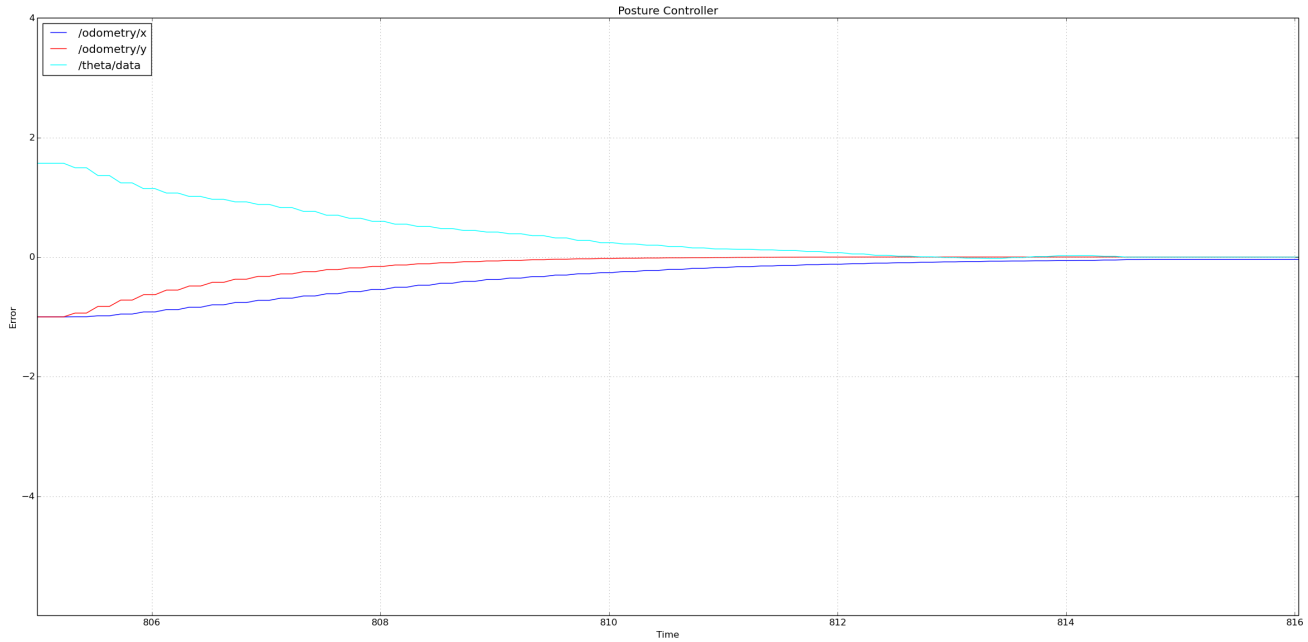


Figure 3: Regulator with gains $k_1 = 2$, $k_2 = 4$, and $k_3 = 1.3$

Where x and y starting position was -1m, and the robot started in $\theta = \frac{\pi}{2}$. Below is the exact same starting conditions for the "1, 6, 7" regulator;

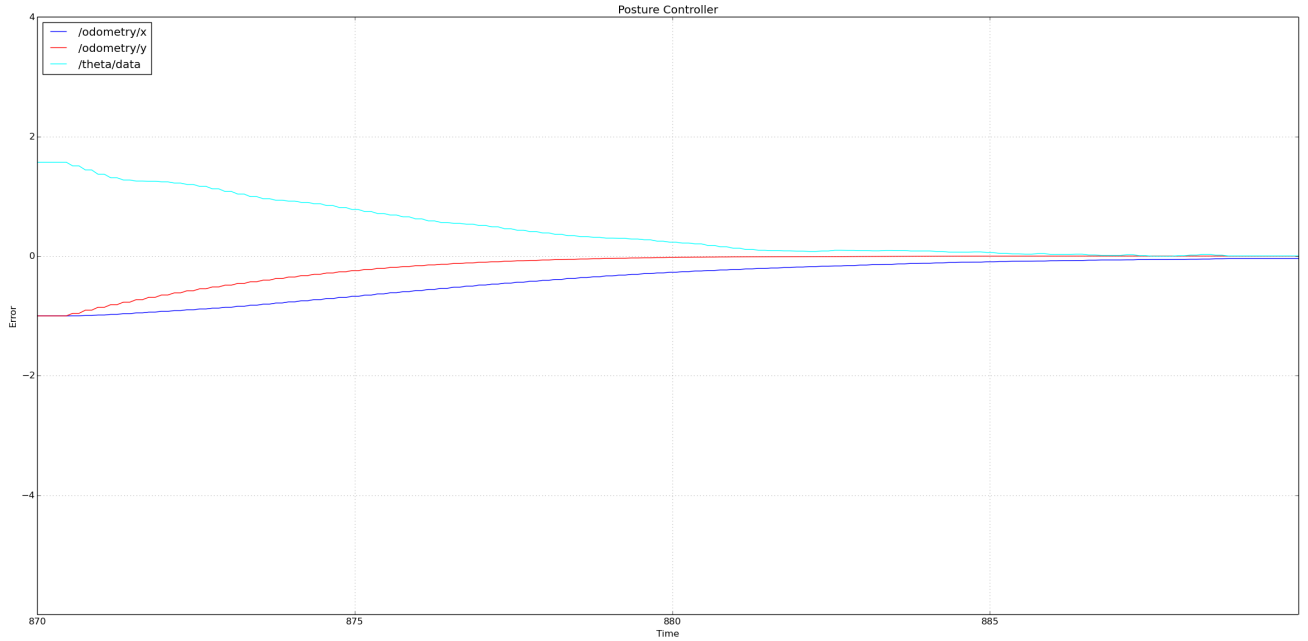


Figure 4: Regulator with gains $k_1 = 1$, $k_2 = 6$, and $k_3 = 7$

We can see that the "1, 6, 7" regulator (in the same case) uses approximately 10 seconds longer, but achieves the same goal.

We concluded that the "2, 4, 1.3" regulator was the best one overall, and tested it further on other starting positions;

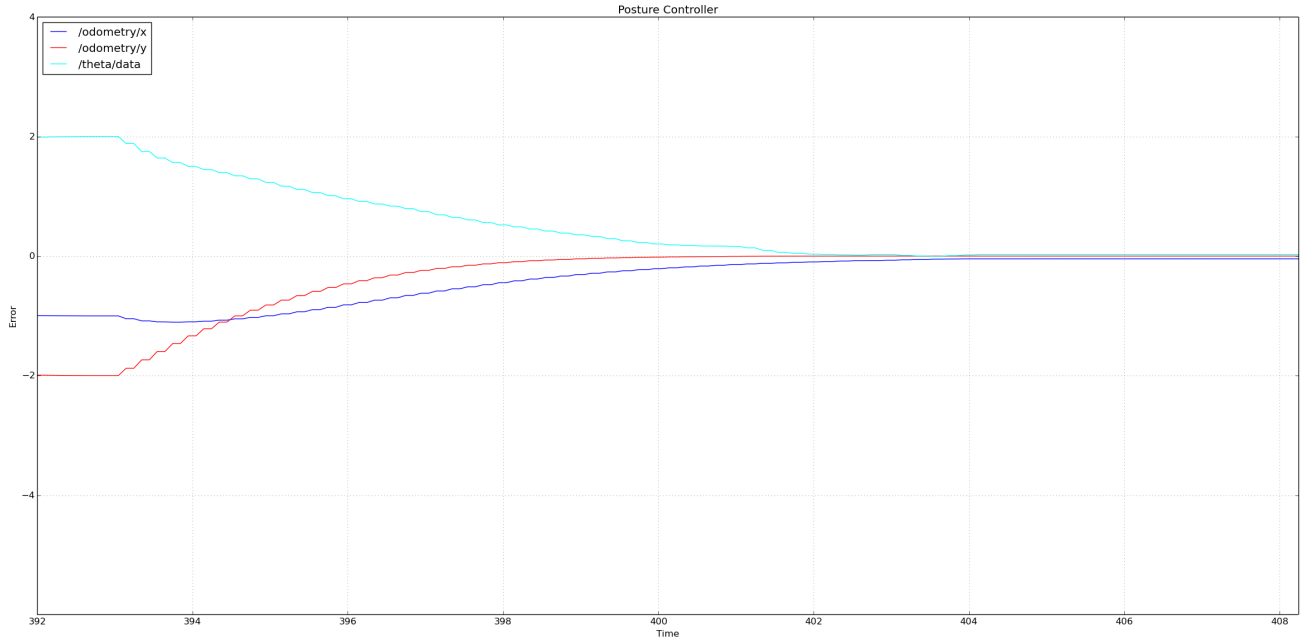


Figure 5: $x = -1, y = -2, \theta = 2$

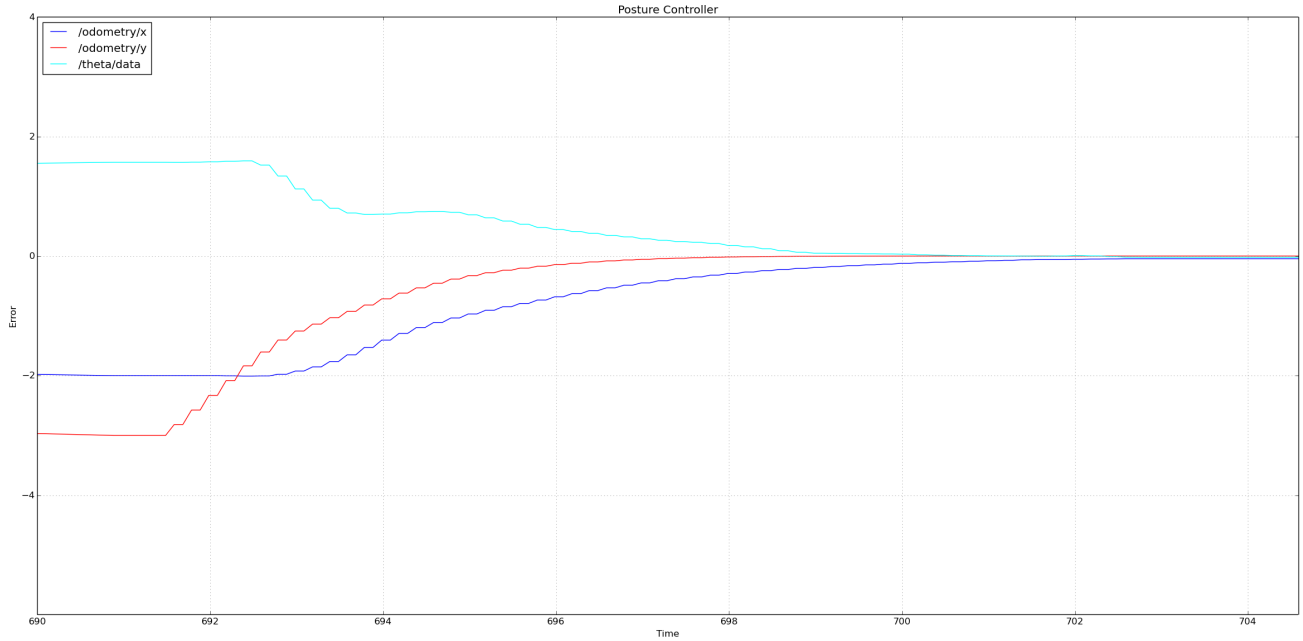


Figure 6: $x = -2, y = -3, \theta = \frac{\pi}{2}$

We see that if we set very high errors (have the desired pose far away from where the robot is going), θ tends to be more unstable. We can also clearly here see the time sampling intervals as small jumps

in the graph. The fact that θ tends to be more unstable with more difficult odometric localizations is expected. The robot may need to turn suddenly in order to get into the right path. It would have been interesting to see if lower sampling time would have had a practical difference on the odometry and the controller.
We noticed that the odometry always was off by something that looked like a scaling factor. In fact, this factor was one of the first things we started investigating on day one of our project. We investigated this mathematically by making an expression for how long one should drive at 100% speed in order to get to a position, but we did not get enough data to calculate it, because there was trouble with the rovers wifi. We assumed that the reason why the odometry was off must be because of that */est_vel* were publishing wrong velocities. In order to compensate for the wrong/estimated velocities from the motordriver, we added a factor on the velocity like this;

```
20          // Calculating variables to know where the rover are in the world coordinate frame
21          // The method used is odometry by exact integration, and can be found in chapter 11.7
22          void odometry_calculations(rover2_motordriver::Velocity vel){
23
24              // Calculating velocities with regards to world coordinates
25              velocity = ((Wr-Wl)/2)*1.19; // (wheel_radius*(Wr-Wl))/2; <- Old, taken away becau
26
```

It turned out 1.19 times the velocity gave us a pretty good result (cm tolerances), which confirmed our suspicion regarding the unknown constants in the motordriver.
You can see two short videos of the robot driving with this new compensation in our github repository.

So in the end we've got accurate results from our system, with only a few errors. It is worth noting that we implemented the odometry under the assumption of constant velocity inputs within the sampling interval. That is definetly not the case because of the pid regulator in the motordriver, and that could explain some of the errors. We also concluded that the motordriver sends out angular velocities in m/s, so we edited our code to compensate for that. We suspect that somewhere in the motordriver they do something wrong when trying to calculate velocities in m/s, and for further development of the system we would advise to calculate and send out rad/s, so that everything agrees with the formulas in the odometry by default.

Another interesting experiment is based on the observation that the "2, 4, 1.3" regulator had higher speed going towards its goal, and therfore hit more in the middle of the ± 0.05 gap, while the "1, 6, 7" regulator was so slow that it stopped immediately after crossing the limit. It would have been interesting to lower the ± 0.05 limit and then try the "1, 6, 7" controller on longer distances, because it was a more "stable" controller.