UNIVERSITY OF OSLO

CONTROL OF MOBILE ROBOTS

UNIK4490

Odometry and Posture Regulation for a 4 by 4 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 a desired pose, and the posture regulator tries to drive the error towards 0.

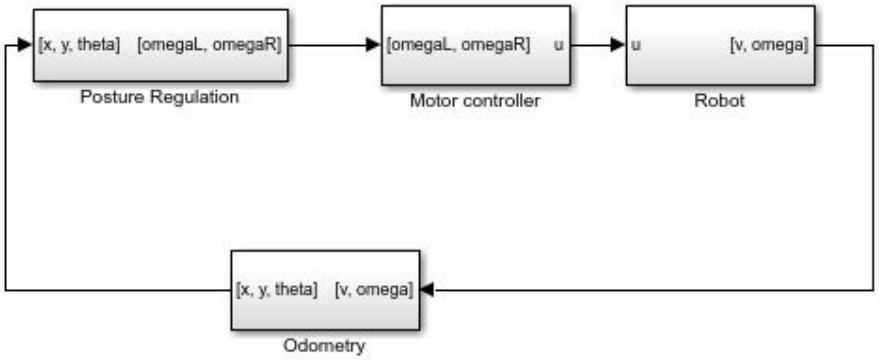


Figure 1: System with Control loop

2.1 Kinematic Model

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. Here v is the driving velocity and ω is the steering velocity.

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

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. In the end 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

$$x_{k+1} = x_k + \frac{v_k}{\omega_k}(sin\theta_{k+1} - sin\theta_k) \tag{2}$$

$$y_{k+1} = y_k + \frac{v_k}{\omega_k}(cos\theta_{k+1} - cos\theta_k) \tag{3}$$

$$\theta_{k+1} = \theta_k + \omega_k T_s \tag{4}$$

We do this by subscribing to the topic */est_vel* which is published from the motordriver.

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. It is assumed that the desired variables are $q_d = [0, 0, 0]^T$ and the error from q_d is represented by

$$\rho = \sqrt{x^2 + y^2} \tag{5}$$

$$\gamma = Atan2(y, x) - \theta + \pi \tag{6}$$

$$\delta = \gamma + \theta \tag{7}$$

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:

$$v = k_1 \rho cos(\gamma) \tag{8}$$

$$\omega = k_2\gamma + k_1 \frac{sin(\gamma)cos(\gamma)}{\gamma}(\gamma + k_3\delta) \tag{9}$$

In our implementation of the controller we get $\vec{q}$ from the odometric module and output 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 (3) and (4) equal to equation (1) and (2) respectively,

$$v = \frac{r(\omega_R + \omega_L)}{2} \tag{10}$$

$$\omega = \frac{r(\omega_R - \omega_L)}{d} \tag{11}$$

and then solving for ω_R and ω_L. This yields:

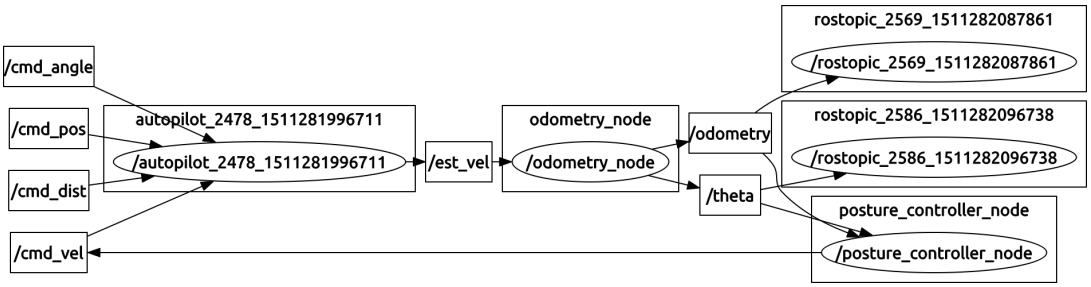
$$\omega_R = \frac{2k_1\rho cos(\gamma)}{2r} + \frac{k_2 d\gamma}{2r} + \frac{k_1 dsin(\gamma)cos(\gamma)(\gamma + k_3\delta)}{2r\gamma} \tag{12}$$

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

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 woth noting that in equation (6) we are using atan2, 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 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 coding exactly the controller derived from the litterature, which we then later understood already are proven asymptotically stable given the lyapunov candidate function

3 Testing and Results

When we got all the nodes together our system looked like this;



And as expected one can see assemblance to figure 1. Here you can see the motordriver labled as the node */autopilot*.

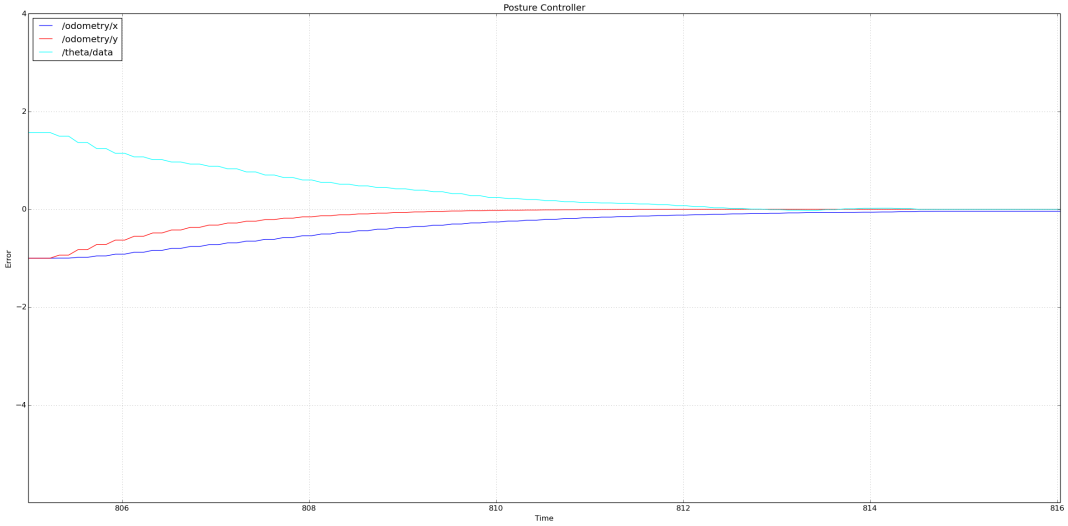
In order to compensate for drift in the odometry, and wrong/estimated velocities (assumption) from the motordriver, we added a factor on the velocity like this;

```
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
```

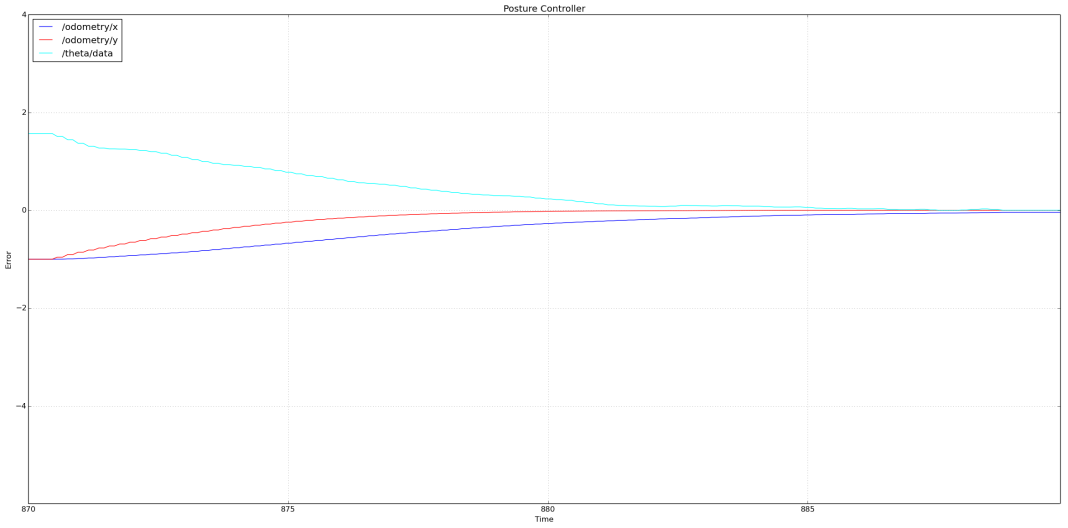
It turned out 1.19 times the velocity gave us a pretty good result (cm tolerances), which confirmed our suspicion regarding the unknown constants in the motordriver.

We started by implementing the reccomended gains (from the litterature) K1= 1 K2 = 2.5 K3=3, but that did not give us 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 2 2.3 1.3 wich was a pretty good regulator, with some accuracy problem. Increasing to 2 4 1.3 gave us a more accurate regulator, which also was quicker.

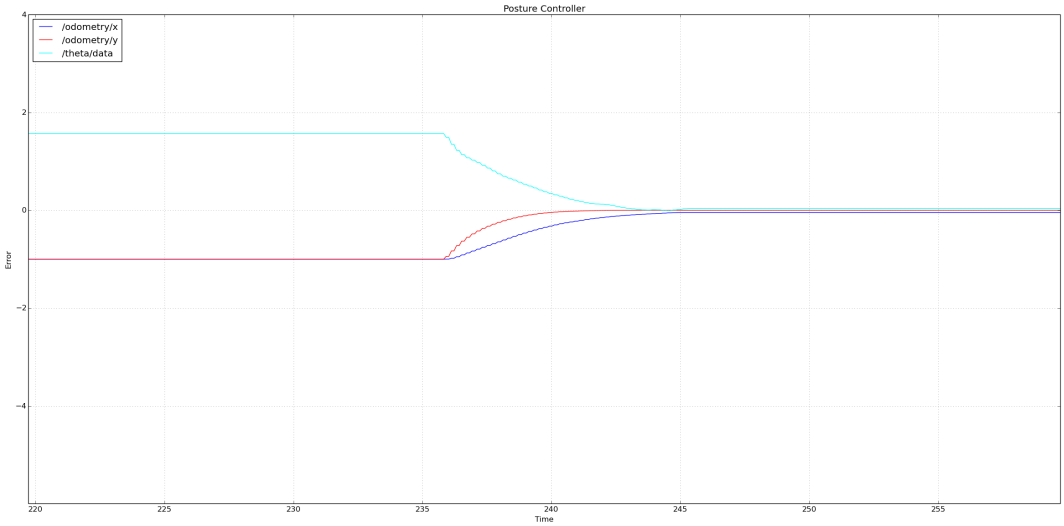


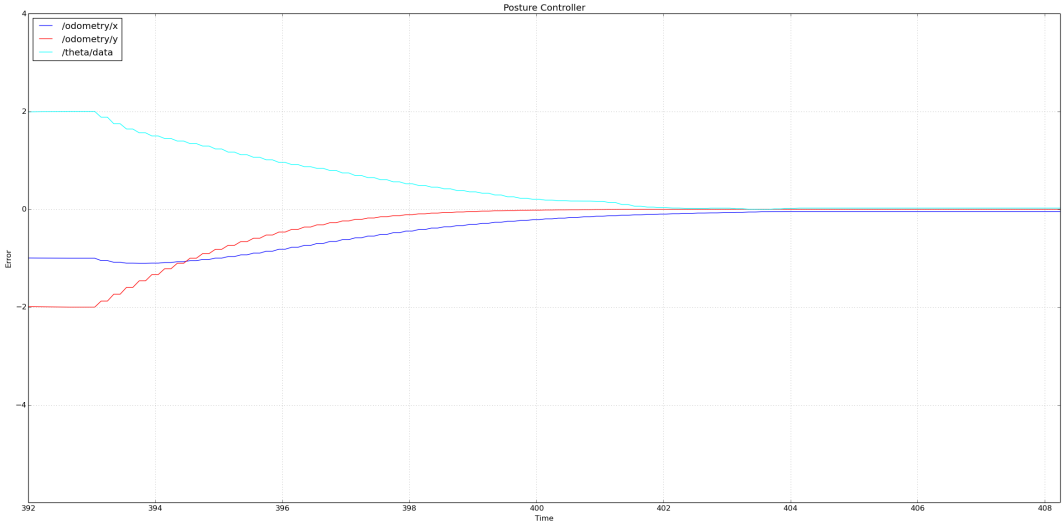
We tried different regulator parameters;

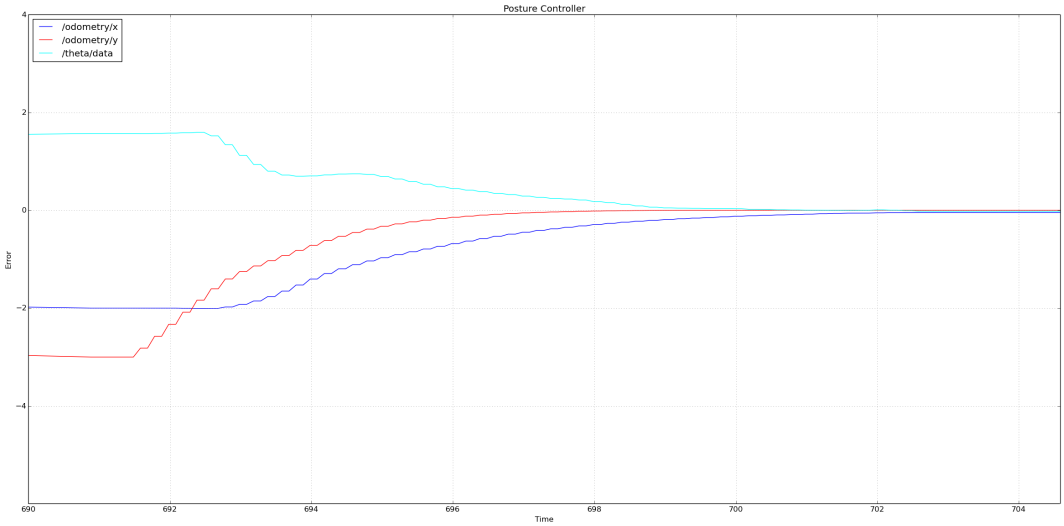


The 1 6 7 regulator was also very accurate, ecpecially at distances < 2m. The 2 4 1.3 regulator was very accurate at all distances and was also very fast, so we turned up going for this regulator.

The 2 4 1.3 regulator preformed like this;


Posture Controller
/odometry/x
/odometry/y
/theta/data
Error
Time
4
2
0
−2
−4
220
225
230
235
240
245
250
255


Posture Controller
/odometry/x
/odometry/y
/theta/data
Error
Time
4
2
0
−2
−4
392
394
396
398
400
402
404
406
408


Posture Controller
/odometry/x
/odometry/y
/theta/data
Error
Time
4
2
0
−2
−4
690
692
694
696
698
700
702
704