TEK4030 - Mandatory Assignment 3

Bergflødt, Adrian Bårdevik, Lars Kristian

November 29th 2021

Contents

1 Introduction

1.1 Aim

The purpose of this project was to implement a posture regulation method for a differential-drive mobile robot. The aim of a posture regulation method is to guide a robot from an initial pose (i.e., a position and orientation) to a desired pose. In other words, we wanted the mobile robot to asymptotically reach a desired pose by following a control law suitable for the regulation problem and for the kinematic constraints imposed by the mechanical design of the robot.

The figure below is an adaptation of a drawing from one of the lectures that introduced the project task, and is meant to illustrate the problem.

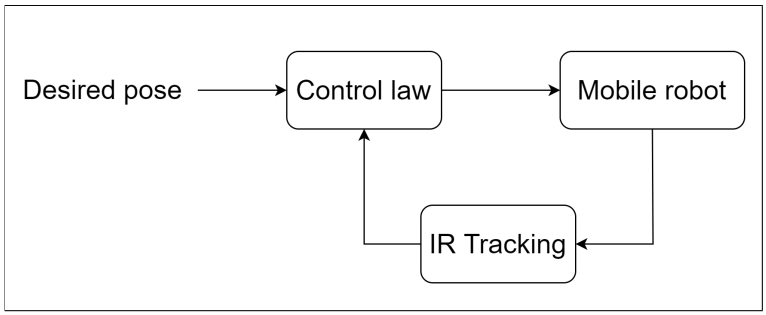

Figure 1: Overall process

We see in this figure, that the desired pose is first an input to the control law, which then generates a control input for the mobile robot to follow. A tracking system delivers measurements in a feedback loop and the control law generates new inputs in a such a way that the robot eventually reaches the desired posture.

1.2 Project Tools

1.2.1 Software

This project used the Robot Operating System (ROS) framework, for implementation of a control node and for communication between that node, a tracking system and the mobile robot. The control node was implemented in C++. ROS uses a publisher-subscriber architecture which enables publishing and subscribing on topics through a master-node. The master-node was a laptop running Ubuntu which had been set up in advance of the project. The Qualisys motion capture system was used to track the position and orientation of the robot.

1.2.2 Turtlebot3 Waffle Pi

The Turtlebot3 Waffle Pi is a differential-drive mobile robot with two fixed wheels in front and a support in the back that slides on the floor. The robot has a non-holonomic constraint as it is not capable of moving instantaneously along its wheel axis of rotation in a straight path towards the goal - it has to change its orientation first.

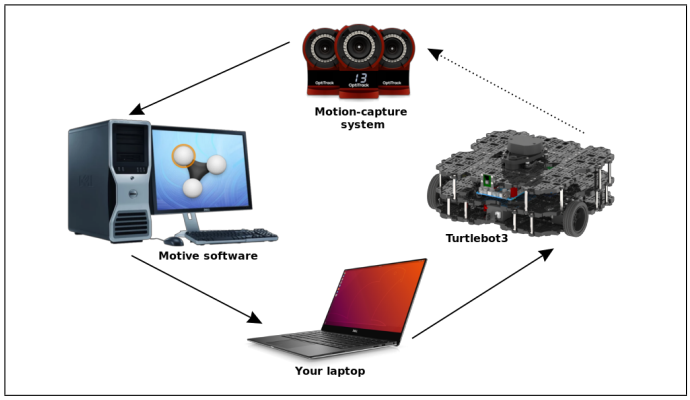

Figure 2: Project setup (Stensrud, 2019)

2 Control Law

For posture regulation of a unicycle, the following control law may be used (from Siciliano et al., 2009, Chapter 11.6):

$$v = k_1 \rho cos\gamma \tag{11.81}$$

$$\omega = k_2\gamma + k_1 \frac{sin\gamma cos\gamma}{\gamma}(\gamma + k_3\delta), \tag{11.82}$$

where

$$\rho = \sqrt{x^2 + y^2}$$
$$\gamma = Atan2(y, x) - \theta + \pi$$
$$\delta = \gamma + \theta$$

If x and y are the coordinates of the midpoint of the wheels of the robot, and θ is the angle of rotation along the z-axis of a body-attached frame with the origin attached to the midpoint of its fixed wheels, then the kinematic model for a unicycle, presented in Siciliano et al., 2009, Chapter 11.2, is also applicable to the differential-drive robot that was used in this project (from Siciliano et al., 2009, Chapter 11.2). This suggests that the control law above may be used for posture regulation problems involving the Turtlebot3 Waffle Pi - the control law was derived for the unicycle model. Given this simple formulation, the desired position is the origin, v is the driving velocity in the x-direction of the robot's body frame (i.e., its sagittal axis), and ω is the steering velocity along the z-direction of the frame. γ is the angle between a vector going from the center of the body to the origin of the Cartesian frame that is fixed with respect to the moving robot (we may refer to this as the inertial frame), and the x-direction of the body-frame of the robot. δ is the angle between the vector going from the center of the robot to the origin of

the fixed Cartesian frame and the x-axis of the frame. ρ is the distance from the center of the robot to the origin of the fixed Cartesian plane. As shown in Siciliano et al., 2009, Page 513, given the candidate Lyapunov function:

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

then

$$\dot{V} = -k_1 cos^2\gamma\rho^2 - k_2\gamma^2$$

which is negative semi-definite, suggesting a bounded system state - as mentioned in Siciliano et al., 2009, Chapter 11.6, the system, ρ, γ and δ converges towards zero and the system is continuous except at the origin of the fixed Cartesian plane. Using the control law in (11.81) and (11.82), our desired setpoint was the origin of the fixed Cartesian plane, and we considered the problem solved when the robot could be placed arbitrarily in a room (where the tracking system was set up) and follow the control law until ρ was approximately zero and the x-axis of the body-frame of the robot was aligned with the x-axis of the inertial frame (or the fixed Cartesian plane).

3 ROS Nodes and Topics

This project utilized pre-created nodes for the motion tracking system and for the Turtlebot. A node was also created for implementing the control law explained in Chapter 2 called *controller*. Code for this node is available in Appendix A. This node subscribes to the */pose* topic which is published to by the *qalisys_stream* node, messages on this topic is of type *geometry_msgs/Pose Message* containing position and rotation in the form of quaternions relative to the calibrated origin of the motion capture room. The topic publishes velocity control commands in the form of a Twist message to the Turtlebot on the */cmd_vel* topic. Messages on this topic is of type *geometry_msgs/Twist Message* containing linear and angular velocity commands, that the turtlebot interprets and completes. A diagram displaying the relation between the nodes and the Turtlebot is shown in Figure 3.

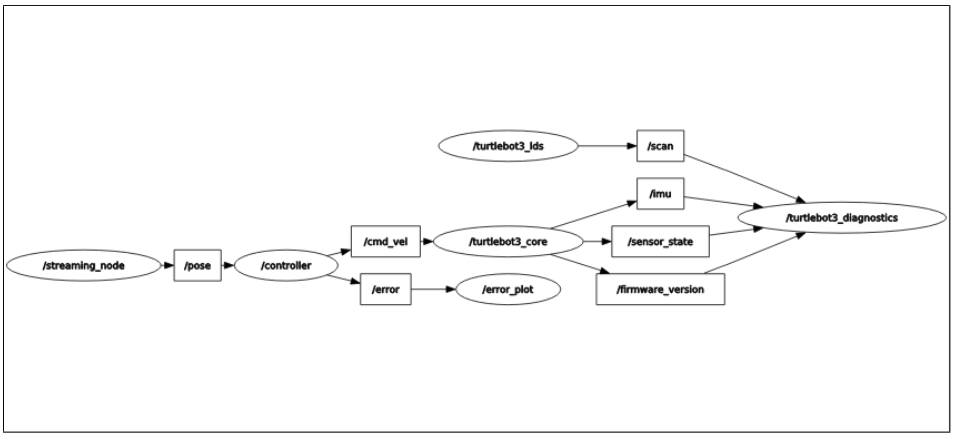

Figure 3: Overview of nodes and topics (generated with rqt_graph)

4 Results

The plot in Figure 4 shows how ρ varied with time for a single run (i.e., how the distance between the robot and the origin of the fixed Cartesian plane was reduced with time).

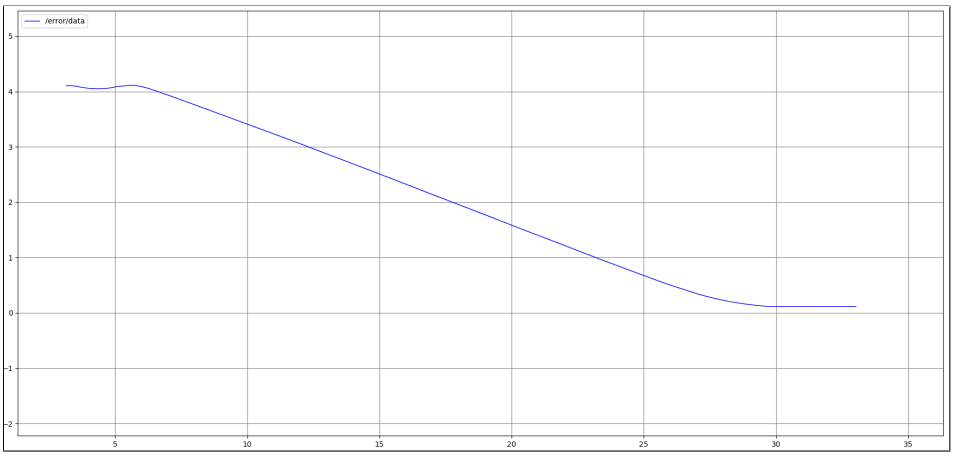

Figure 4: Plot of ρ against time (generated with rqt_plot)

We see in the figure above that the value of ρ starts to decrease, but then increases before it decreases towards zero - this is an example of a reorientation manoeuvre. The robot was placed initially with the front away from the origin of the Cartesian plane. The manoeuvre is shown more clearly in Figure 5, which shows the variation of x-position with time, and Figure 6, which shows the variation of y-position with time.

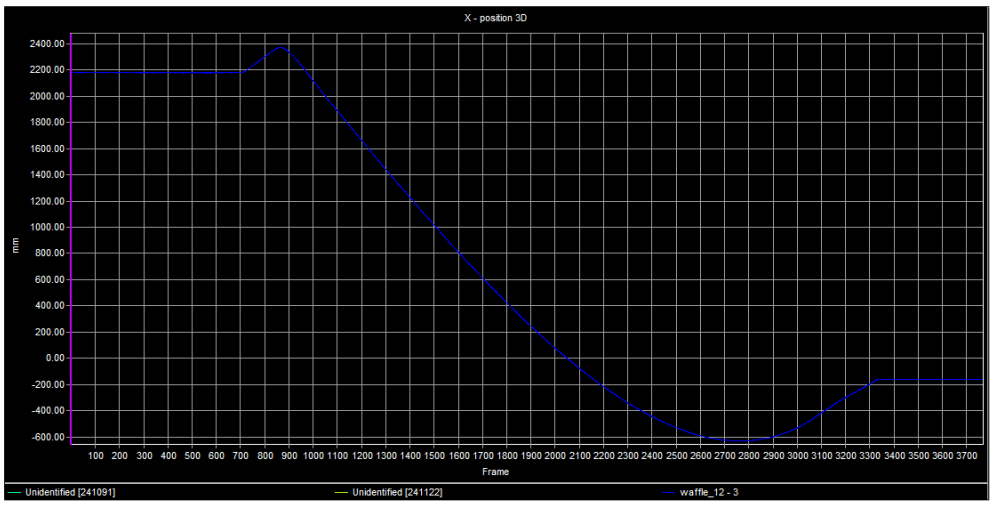

Figure 5: Variation of x-position with time (generated using Motive)

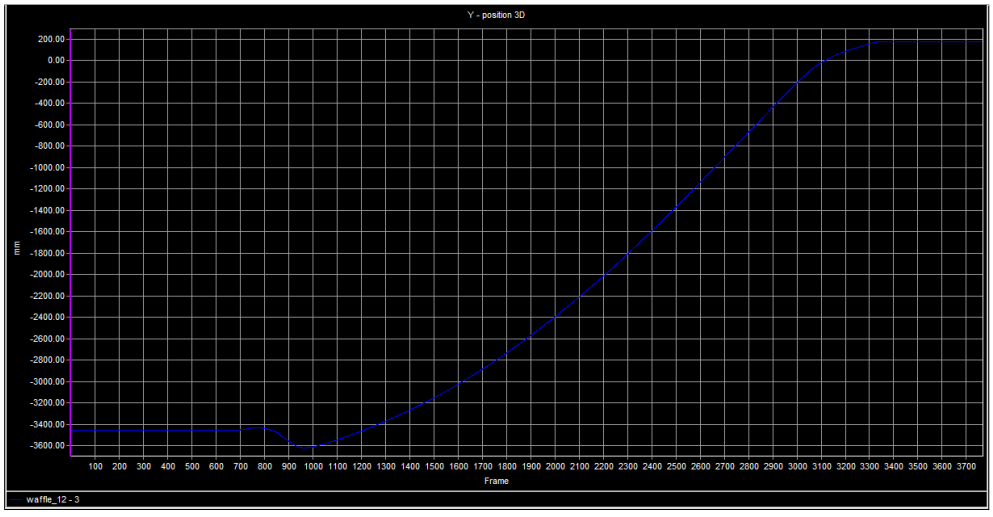

Figure 6: Variation of y-position with time (generated using Motive)

Figures 7 and 8 shows images of the Turtlebot's starting and stopping pose in a test run. We see that the robot is oriented away from the goal (shown in Figure 8), and that the robot's x-axis is aligned with the x-axis of the inertial frame (shown approximately by the white tape on the floor beneath the robot).

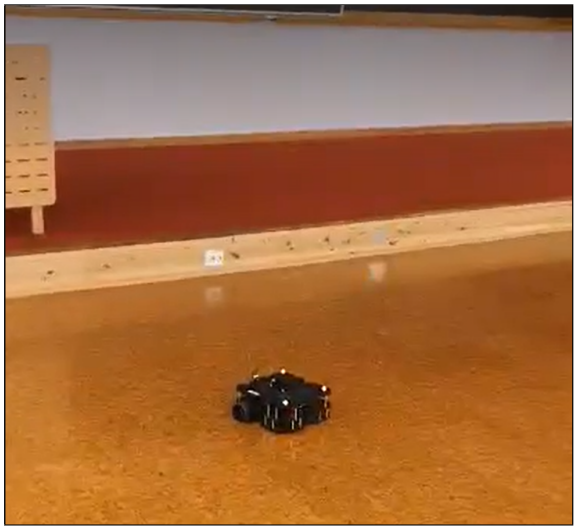

Figure 7: Turtlebot initial (rest) pose

Figure 8: Turtlebot desired (rest) pose

References

Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2009). *Robotics: Modelling, planning and control*. Springer.

Stensrud. (2019). Finished_setup [[Online; accessed December 02, 2021]]. https://github.uio.no/TEK4030/tek4030_turtlebot3/blob/master/images/finished_setup . png ? fbclid = IwAR2XnKaPfmp_hyeZPymRqopDYb_RNIQs6a9qkgEfylMUg6ob4xl-5kumID4

A controller.cpp

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "geometry_msgs/PoseStamped.h"
#include "Eigen/Eigen"
#include <geometry_msgs/Vector3.h>
#include <cmath>
#include <std_msgs/Float64.h>
#include <iostream>
#include <sstream>

ros::Publisher* twist_pub_glob = NULL;
ros::Publisher* error_pub_glob = NULL;

static double smallestDeltaAngle(const double& x, const
    double& y)
{
  // From https://stackoverflow.com/questions/1878907/the-
    smallest-difference-between-2-angles
  return atan2(sin(x - y), cos(x - y));
}

void callback(const geometry_msgs::Pose::ConstPtr& msg){
    ROS_INFO_STREAM("Recieved: " << msg);
    Eigen::VectorXd pos(3,1); //Insert the angle here too
    pos(0) = msg->position.x;
    pos(1) = msg->position.y;
    pos(2) = msg->position.z;

    Eigen::Quaterniond quat;
    quat.x() = msg->orientation.x;
    quat.y() = msg->orientation.y;
```

```
30      quat.z() = msg->orientation.z;
31      quat.w() = msg->orientation.w;
32
33      //convert from quaternion to rotation matrix
34      Eigen::Matrix3d DCM = quat.normalized().toRotationMatrix
        ().transpose();
35
36      //get rotation angle around x-axis
37      double theta = atan2(DCM(1,0), DCM(0,0));
38
39      //calculate controller variables
40      double rho = sqrt(pow(pos(0), 2)+pow(pos(1), 2));
41      double gamma = smallestDeltaAngle(atan2(pos(1), pos(0)),
        theta+ M_PI);
42      double delta = gamma + theta;
43
44      // Gains
45      double k_1 = 0.5;
46      double k_2 = 2.5;
47      double k_3 = 2.5;
48
49      // Control law
50      double v = k_1*rho*cos(gamma);
51      double omega = k_2*gamma + k_1*((sin(gamma)*cos(gamma))/
        gamma)*(gamma+k_3*delta);
52
53      //Publish
54      std_msgs::Float64 error_msg;
55      geometry_msgs::Twist twist;
56
57      error_msg.data = rho;
58
59      //stop if close enough
```

```
60      if(rho <= 0.12 )
61      {
62        v = 0;
63        omega = 0;
64      }
65
66      //Limit linear velocity
67      if (v > 0.26)
68      {
69        v = 0.26;
70      }
71
72      twist.linear.x = v;
73      twist.linear.y = 0.0;
74      twist.linear.z = 0.0;
75      twist.angular.x = 0.0;
76      twist.angular.y = 0.0;
77
78      //Limit angular velocity
79      if (omega > 1.82)
80      {
81        omega = 1.82;
82      }
83
84      twist.angular.z = omega;
85
86      error_pub_glob->publish(error_msg);
87      twist_pub_glob->publish(twist);
88  }
89
90  int main (int argc, char **argv){
91      ros::init(argc, argv, "sample");
92      ros::NodeHandle n;
```

```
93      ros::Rate loop_rate(10);
94
95      ros::Subscriber sub = n.subscribe("/pose", 1000, callback
        );
96      ros::Publisher pub_twist = n.advertise<geometry_msgs::
        Twist>("/cmd_vel", 1000); //waffle2/cmd_vel
97      twist_pub_glob = &pub_twist;
98
99      ros::Publisher pub_error = n.advertise<std_msgs::Float64
        >("/error", 1000);
100     error_pub_glob = &pub_error;
101
102     ros::spin();
103
104     return 0;
105 }
```

Listing 1: Controller node