Gazebo simulations of contract-based design for an automated valet parking system

Tom Andersson

Mentor: Richard M. Murray
Co-mentor: Josefine Graebener

September 18, 2020

Contents

1 Introduction

1.1 Background

The increased complexity in cyber-physical systems calls for a modular system design that guarantees safe, correct and reliable behavior, both in normal operation, in the presence of dynamic scenario changes and in failure scenarios [7]. One way of achieving a modular system is to use contract-based design and split up the system into layers, where each layer have specific tasks. Information is sent between the layers according to the rules set up by the contracts. This system architecture not only simplifies the system by splitting it up into several subsystems but in a failure scenario this design also makes it clear in which layer the failure occurred. Other layers can then react and adapt to the new restrictions the failures entail.

These advantages match well with several problems within the development of autonomous vehicles since they have to be developed for an environment constantly affected by dynamic scenario changes due to the interaction with other cars, pedestrians and so on. A first step towards full autonomy can be to limit the scope and focus on an automated valet parking system, a system where customers drop of their cars and an autonomous robot picks them up one at a time and drives them to a free parking spot. The research group at Caltech has developed python simulations of an automated valet parking system, modularly designed using contracts between the system layers. The simulations demonstrate the ability to control several cars that can be parked and recalled within two minutes in the absence of failure scenarios and in the presence of a simulated undesired stop of a car in the system. The project is stored in this Github repository [5]. Figure 1 shows the layout of the valet parking in the simulations [7].

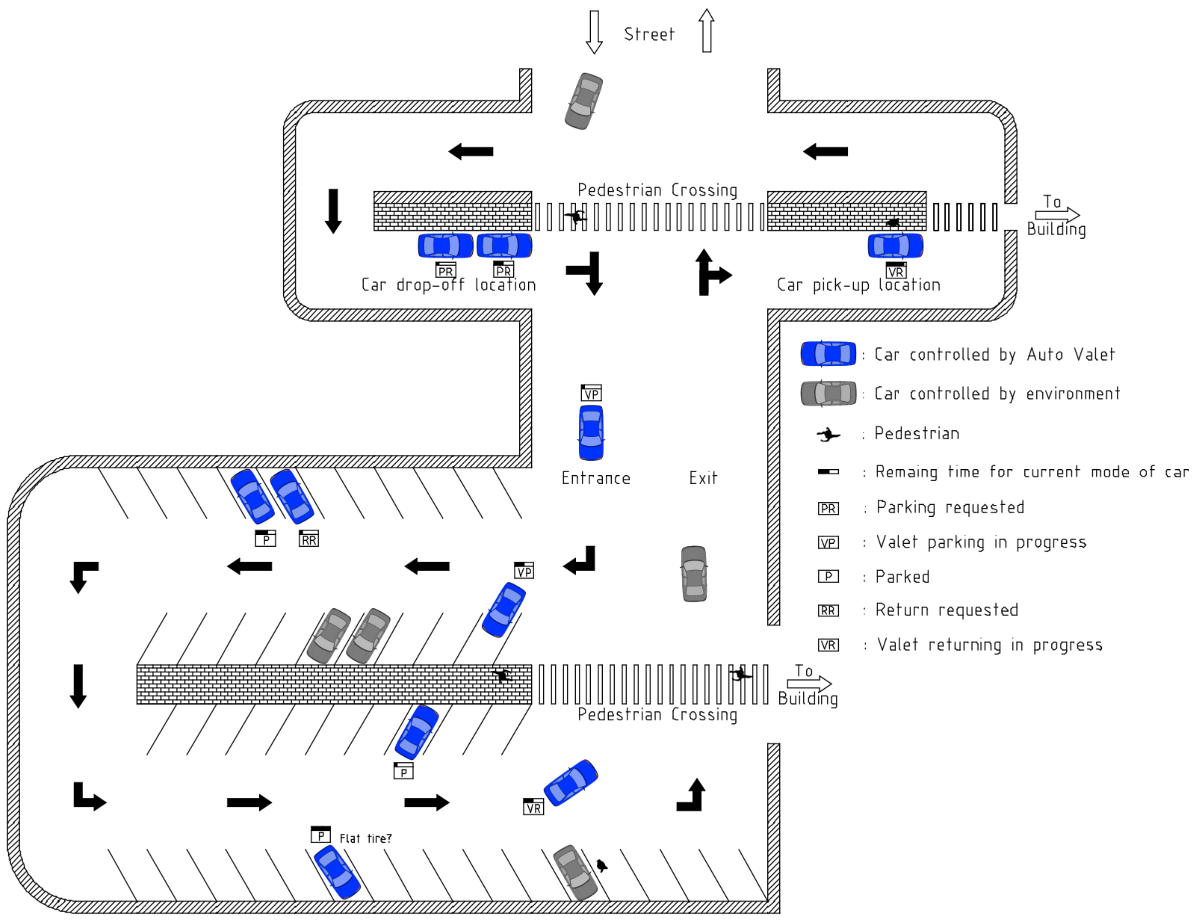


Figure 1: The layout of the valet parking in the python simulations [7].

1.2 Goal

The goal of this project is to prepare the python simulations for a small scale test that would incorporate real life aspects and the possibility to physically separate components in the system by constructing a simulation environment in Gazebo and then integrate this with the main AVP-simulations. The simulations shall include up to 6 robots as well as pedestrians and static obstacles.

1.3 Objectives

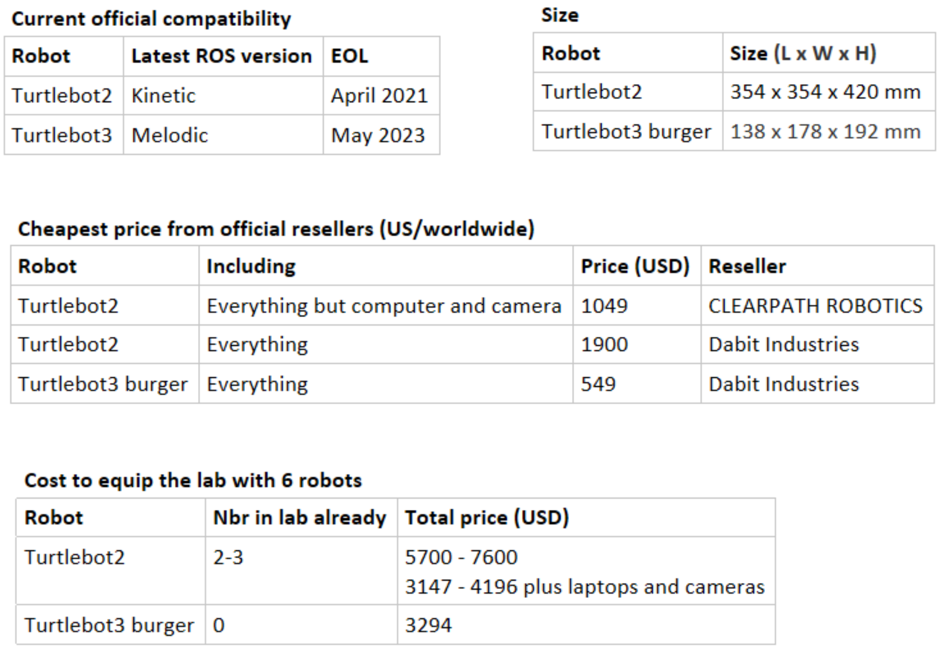
- Become familiar with the work so far done by the research group.
- Learn about Gazebo and choose appropriate robots to simulate. Perform simulations by using commands to set velocities and read velocities and position.
- Build a model of the valet parking environment in Gazebo including a ground plane of the parking lot, walls, the robots, traceable static obstacles and controllable and traceable pedestrians.
- Implement the functionality to dynamically launch a chosen number of robots, pedestrians and obstacles at chosen initial positions in the Gazebo world.
- Integrate the Gazebo simulations with the main AVP simulations.
- Finalize a written report.

2 Constructing the simulation environment

This section describes how the simulation environment including the communication with the environment was developed. More detailed information on each script and function can be found in the corresponding docstrings. The source code can be found on the Github repository in the branch called gazebo_ros [5].

2.1 The robots

After the initial process of learning about the current state of the project the choice of robot to simulate was analysed through a comparison between turtlebot2 and turtlebot3 burger. These two robots were chosen because the research group already had access to 2-3 turtlebot2 robots and turtlebot3 burger is its successor. This comparison was made to optimise for a small scale test at a later point in time. The collected data can be found in figure 2. This comparison shows that turtlebot3 burger is a good choice when it comes to price, official compatibility and size, a smaller size reduces the size of the physical area needed during a small scale test. In parallel with this comparison, the process of learning about the Gazebo simulator was ongoing by going through tutorials and running demos with turtlebots in Gazebo.



Current official compatibility

Robot	Latest ROS version	EOL
Turtlebot2	Kinetic	April 2021
Turtlebot3	Melodic	May 2023

Size

Robot	Size (L x W x H)
Turtlebot2	354 x 354 x 420 mm
Turtlebot3 burger	138 x 178 x 192 mm

Cheapest price from official resellers (US/worldwide)

Robot	Including	Price (USD)	Reseller
Turtlebot2	Everything but computer and camera	1049	CLEARPATH ROBOTICS
Turtlebot2	Everything	1900	Dabit Industries
Turtlebot3 burger	Everything	549	Dabit Industries

Cost to equip the lab with 6 robots

Robot	Nbr in lab already	Total price (USD)
Turtlebot2	2-3	5700 - 7600 3147 - 4196 plus laptops and cameras
Turtlebot3 burger	0	3294

Figure 2: A comparison between turtlebot2 and turtlebot3 burger.

With the help from these tutorials, [8], a launch file for the turtlebot3, one_robot.launch, one launch file for launching multiple turtlebots at specific positions, robots.launch, and one main launch file for launching the turtlebots in an empty world, main.launch was constructed. Figure 3 shows 2 turtlebots launched in an empty world.

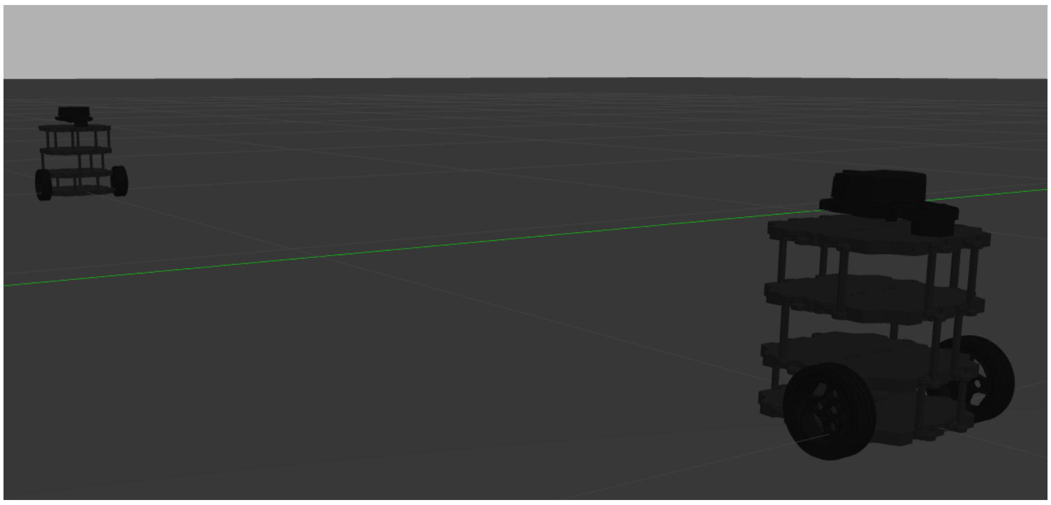


Figure 3: 2 turtlebots launched in an empty world.

The next step was to control the turtlebots through ROS. The turtlebots can be controlled by setting linear and rotational velocity. To keep going, the turtlebots need to continue to receive new data, if the data stops coming the turtlebot will stop. This was the belief throughout the project but after the project had ended and the student continued studying ROS it was realized that topics that are published before the communication link is fully established will be ignored without any warnings, this might have been what happened here but since the project had already ended it was not analysed within the project. The robots also publishes a ROS topic that describes the current states of the robot, i.e position, orientation, velocity etc. A python program, communication, that publishes robot commands and subscribes to the states of the robots was written. To change the commands that are published and request information of the states of the robots, the class RobotCtrl was written. This class will only receive and send topics when asked to while communication does it at a given frequency. To test this functionality, a test program was written. The described ROS-structure is shown in figure 4. One can choose how many robots to simulate by changing a global variable and change the number of robots in the robots.launch file.

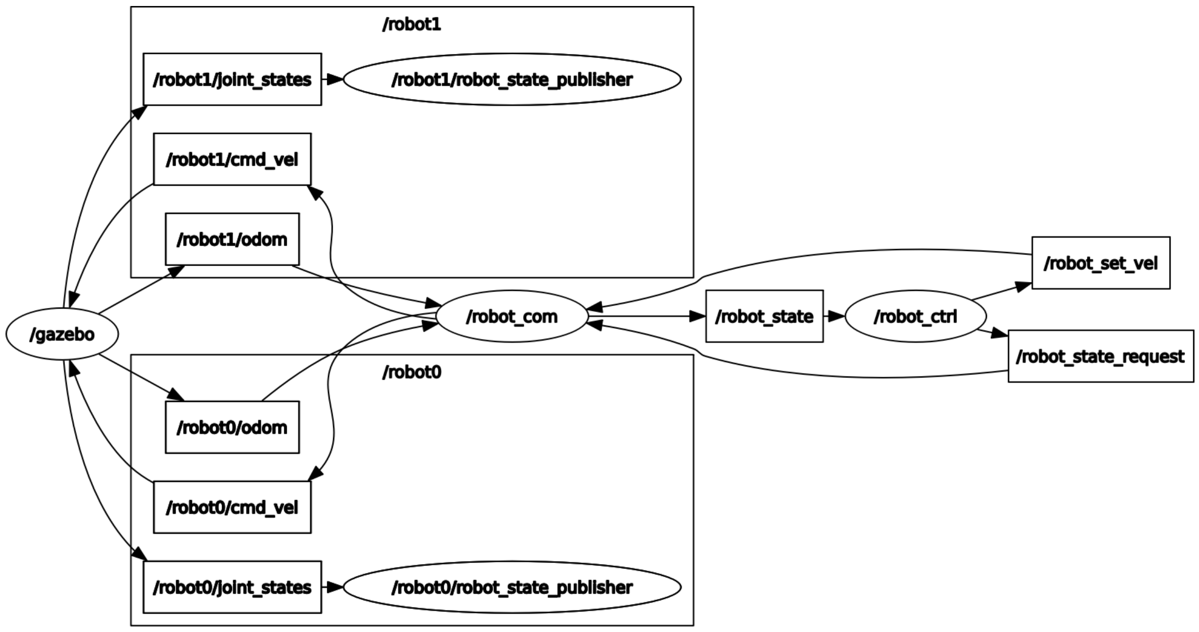


Figure 4: The ROS structure when 2 robots are launched.

The shape of the turtlebots is not similar to the shape of a normal car and the shape of the cars used in the pure python simulations. Therefore, the base model for the turtlebots was altered by adding a hollow box with no bottom, see figure 5. This created a shape more similar to normal cars while keeping the increased weigh to a minimum.

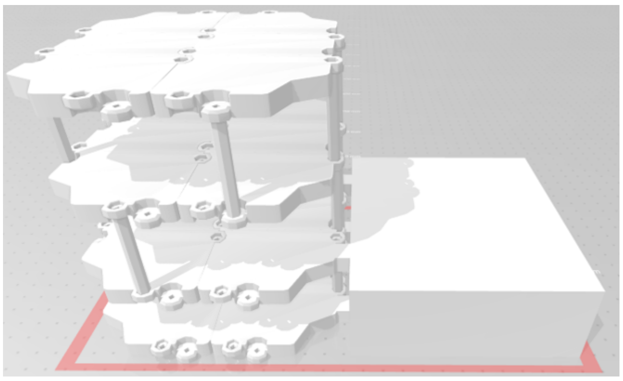


Figure 5: The altered appearance of the turtlebots.

To further mimic the kinematics of a car, the class RobotCtrl was extended with a function to set velocity and steering angle using equation 1.

$$\phi = v\frac{tan(\delta)}{l} \tag{1}$$

$$\begin{aligned} \phi &= rotational\ velocity\ [rad/s] \\ v &= linear\ velocity\ [m/s] \\ \delta &= steering\ angle\ [rad] \\ l &= wheelbase\ [m] \end{aligned}$$

2.2 The world

After the control of the robots was implemented a parking lot environment was built in a Gazebo world. The environment was reduced in size, compared to figure 1, to be better fitted for simulations with only 6 robots and to take up less space during a future physical test. This world was built by adding an image of a reduced parking lot as a ground plane in the world. This image was scaled so that the turtlebot3 burger would fit in the parking spots. Thereafter, walls were added on top of the ground plane image. See figure 6. The next task will be to add pedestrians and static obstacles into the Gazebo world.

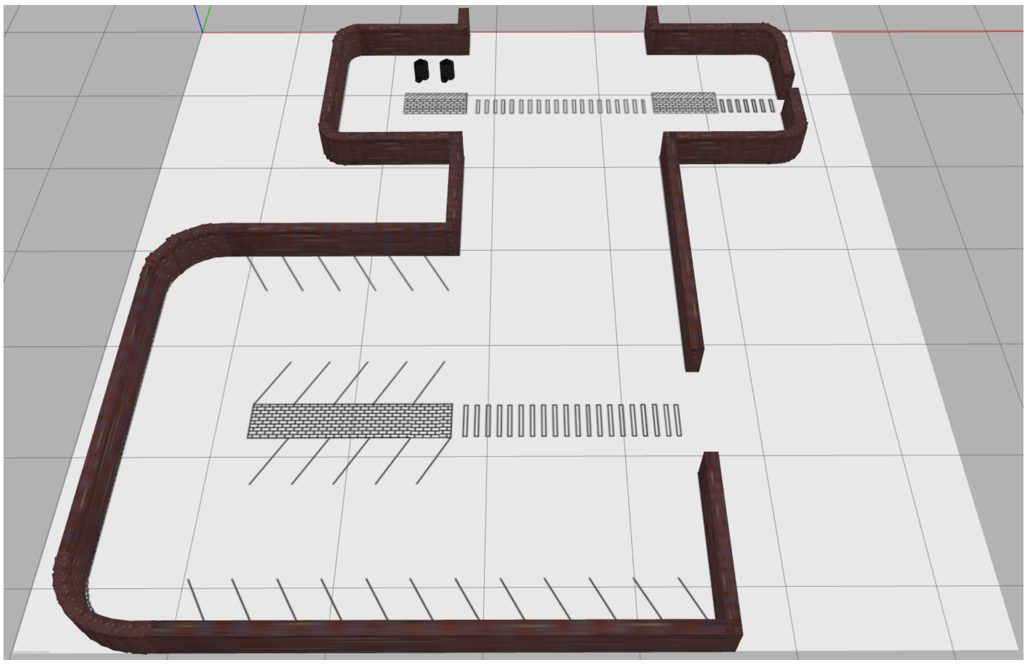


Figure 6: The parking lot environment implemented in a Gazebo world.

2.3 Pedestrians and static obstacles

By taking models from the main Gazebo library and reducing their size to fit the parking lot environment as well as reducing the pedestrians friction in the x- and y-direction to allow for easier control, model descriptions files for static obstacles and pedestrians were created in the format sdf. By following a tutorial, [2], a launch file for launching sdf-files into the Gazebo world was written. To get the state of a model, Gazebo publishes a ROS-topic called model_states at a certain rate. The script communication subscribes to this topic. When requested through another topic, the script will publish the relevant states, i.e x- and y-position, yaw angle and, for pedestrians, the corresponding velocities, for a specified model.

Another tutorial was used to learn how to control models in a Gazebo world [1]. The tutorial describes that models are controlled using plugin programs written in C++. Each model description file is linked to a specific plugin file and the plugins are compiled together with the entire ROS-workspace. As an example, for launching two pedestrians one would need two model description files and two plugin files. This makes it complicated to allow the user to choose any number of pedestrians to launch without having to add more files and recompile the workspace, which is time consuming if it had to be done every single time. A solution to that is to limit the system to work with a maximum of a certain number of pedestrians and have that many model files and plugin files stored. This solution was implemented with five as the maximum number. Another solution could be to look into the more complex description files that are used for robots and write such description files for the pedestrian models or to use some kind of robot with existing description files as pedestrians. Since the system is meant to simplify for a future real world test, the later of those options would make sense to implement but due to time restraints it was chosen to not look into that. Each plugin is setting the linear and rotational velocity of its corresponding pedestrian model when it receives a ROS-topic containing the wanted velocities.

Similar to the class RobotCtrl, the class ModelCtrl was written. It sets velocities for pedestrian by publishing the ROS-topics that the plugins are subscribed to and receives the states of the models by requests to the communication script. Two obstacles and two pedestrians that are launched in the Gazebo world can be seen in figure 7.

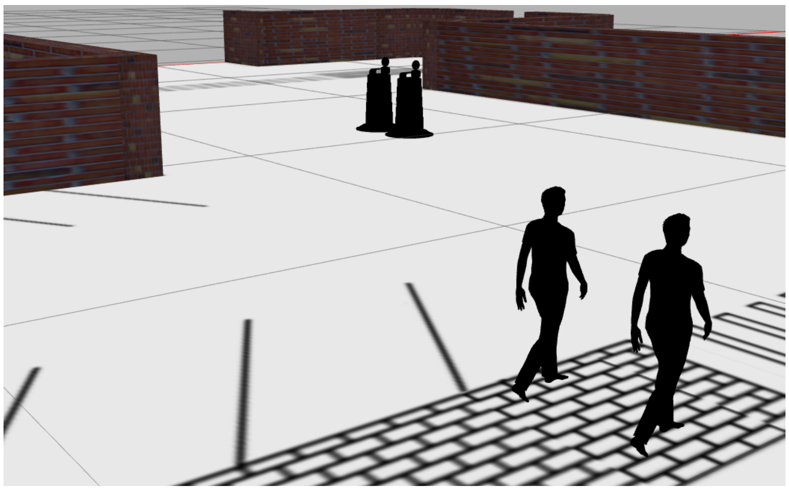


Figure 7: Pedestrians and static obstacles launched in the Gazebo world.

2.4 Dynamic launching

Up to this point, the number of robots, pedestrians and obstacles as well as their initial positions were hard coded into the launch files. Therefore, the user would have perform these kind of changes by changing the launch files manually. A more dynamic solution was now going to be implemented. This was done by changing the structure of the launch file into 4 separate files, one for launching the world, one for launching a robot, one for launching an obstacle and one for launching a pedestrian. Except for the launch file for the world, each launch file is launched together with arguments describing the initial position and rotation as well as an identification number. For launching these files the python script gazebo_launcher was written. It is subscribed to a ROS-topic containing the initial positions of robots, pedestrians and obstacles. Upon receiving this topic the script launches the files. When the script receives a topic without any data in it, everything that has been launched is terminated. This script was written by following yet another tutorial. [6]. To publish the ROS-topic, a class named GazeboCtrl was written. To change the number of robots, pedestrians or obstacles that are being launched, the user still has to change the global variables representing these numbers manually. This could be considered tedious and a solution could be to let the GazeboCtrl class set these values once a world is being launched. However, since several other scripts are using these variables one would also have to change those programs so that the variables aren't used before GazeboCtrl has given them their values. Implementing this is considered to improve the program but due to time limitations it has been chosen to not do that unless there is time left by the end of the project.

2.5 System summary

The ROS-structure, previously seen in figure 4 for a world only containing robots, has with the inclusion of obstacles, pedestrians and dynamic launching evolved into the ROS-structure seen in figure 8. In figure 8, the node named /test_ctrl works as a main script that creates an instance of each of the RobotCtrl, ModelCtrl and GazeboCtrl classes. These could have been split up into different nodes but for the sake of simplicity they were kept in the same node in this example.

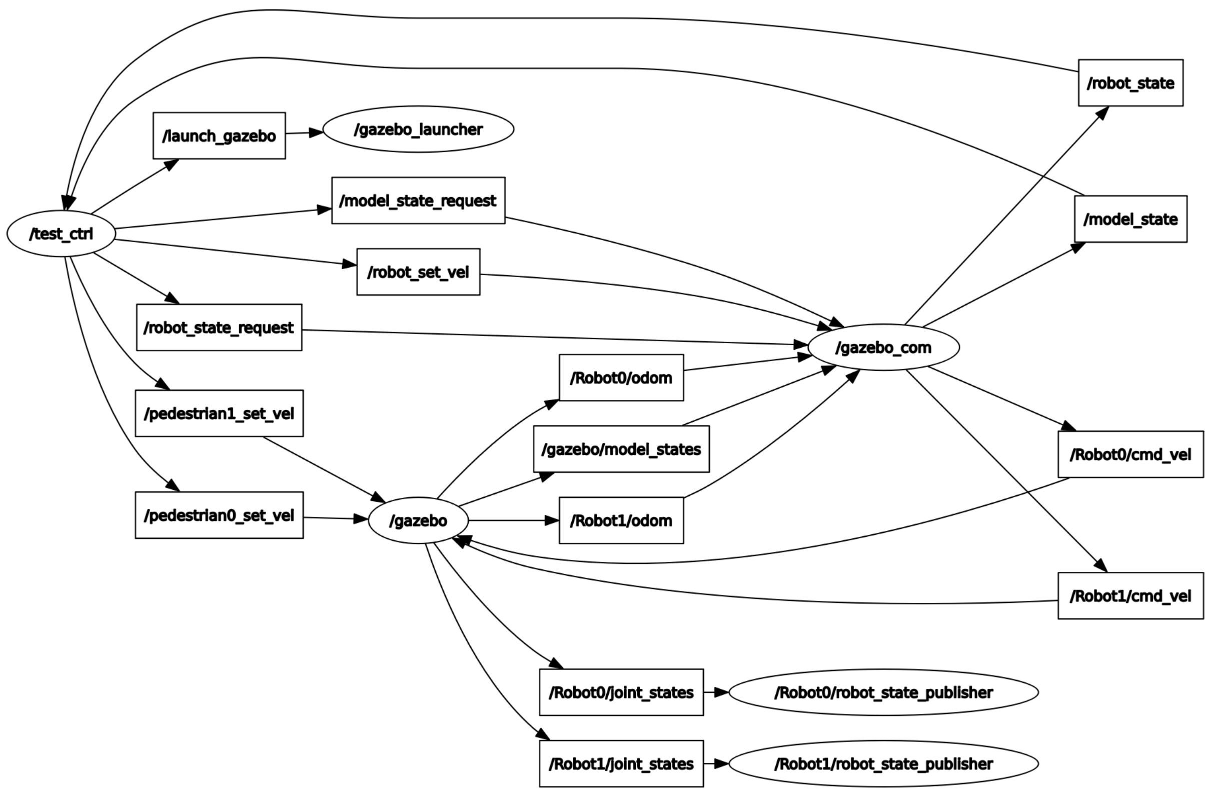


Figure 8: The ROS structure with robots, pedestrians, static obstacles and dynamic launching.

3 Integration

3.1 Scaling down the environment

Since the parking lot environment in the gazebo world was smaller than the one in the original simulations some adaptations had to be made. The trajectory planner used image files to determine the area available for the cars and coordinates for the location of the parking spots, these image files were replaced with images of the new environment, see motionplanning/imglib in the branch gazebo_ros.

The coordinates of different areas within the environment such as the parking spots, the different lanes etc was stated as global variables, these were altered to fit the new layout. Finally, the coordinates and distances in the simulation was described both in pixels and in meters, therefore the scaling factor between pixels and meters had to be adjusted to fit the size of the turtlebots. See the folder variables in the branch gazebo_ros.

3.2 Implementing the trajectory controller

Initially the trajectory control was not described as a major part of the project. However, when it was time to integrate this feature with the turtlebots it was suggested to not use the controller used in the original simulations due to its complexity. Therefore, different control options were looked into.

First, Lyapunov's method was studied, [12], but no valid Lyapunov function could be found. This result makes sense since the distance to the goal position cannot be strictly decreasing if the direction the car is facing is perpendicular to the goal position. Thereafter, the maximum principle within optimal control techniques was studied, [13], but as the method was explored in relation to the control problem in question the complexity grew to a point where it was decided to not continue but instead look for a simpler solution. The third approach consisted of linearizing the system and use a state feedback controller with feedforward, [3], but this approached also encountered problems because the linearized system matrix turned out to be uncontrollable at certain points of linearization, for example when the velocity of the turtlebot was equal to zero.

Finally, it was decided to use the same controller as in the original simulations after all. This was a linear model predictive controller [11]. The idea of MPC's is to use a model of the process and predict the reactions of the real process by iteratively applying control inputs to the model. The method also allows for setting boundaries on states, control signals and outputs [14]. The original car model in this controller used acceleration and steering angle as control inputs. Therefore the functionality to set acceleration to the turtlebots had to be implemented. This was done by step wise increasing the speed at a specified sample rate and recalculating the angular velocity to obtain the desired steering angle accordingly. The already integrated MPC controller used the predicted states, position in x and y, velocity and yaw angle, as the true outcome at each sample. This was changed so that the MPC used the measured states from the turtlebot while also setting the output to the turtlebot. Furthermore, the MPC was altered so that it ran at a specified rate, this was only simulated in the original version and had to be altered since the predictions rely on a specified rate.

To test this the main simulations were limited to only accept one car into the system, the gazebo world was set to be launched together with the main simulations and a turtlebot was given the initial position of the car in the main simulations. During the test, the turtlebot started following the given trajectory. However, the path tracking was oscillating at a considerable level which caused the turtlebot to deviate from its trajectory until finally, the MPC problem could not be solved or the turtlebot had hit a wall and got stuck. It was noticed that the computing time was higher than the set sampling time which made the predictions inaccurate. The sampling time was increased to match the computing times but at this level it was so high that the approximations due to the discretization destabilized the system. The one step prediction error, i.e the difference between the predicted states and measured states one sample step into the future was measured during a test round and it was indeed considerably high, see figure 9, 10, 11 and 12.

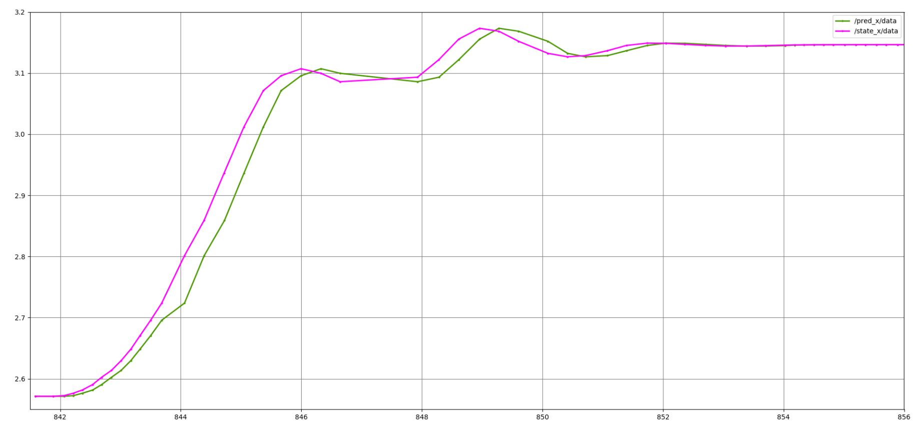


Figure 9: The predicted and measured x-position when using the original controller with acceleration and steering angle as control signals.

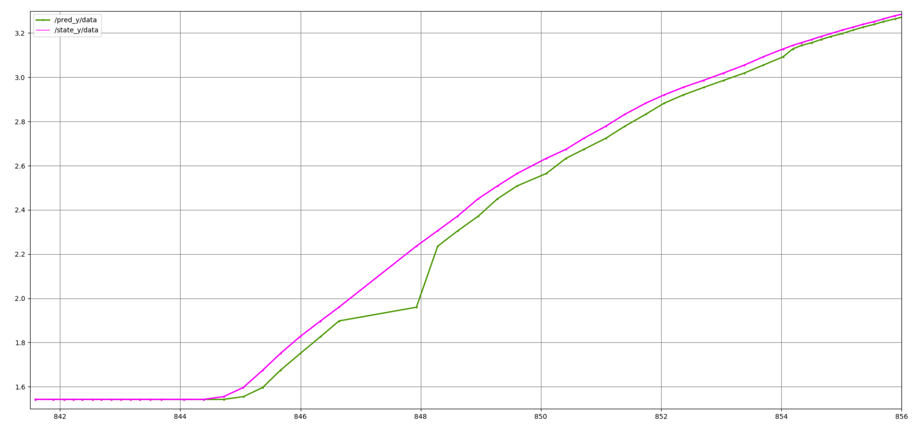


Figure 10: The predicted and measured y-position when using the original controller with acceleration and steering angle as control signals.

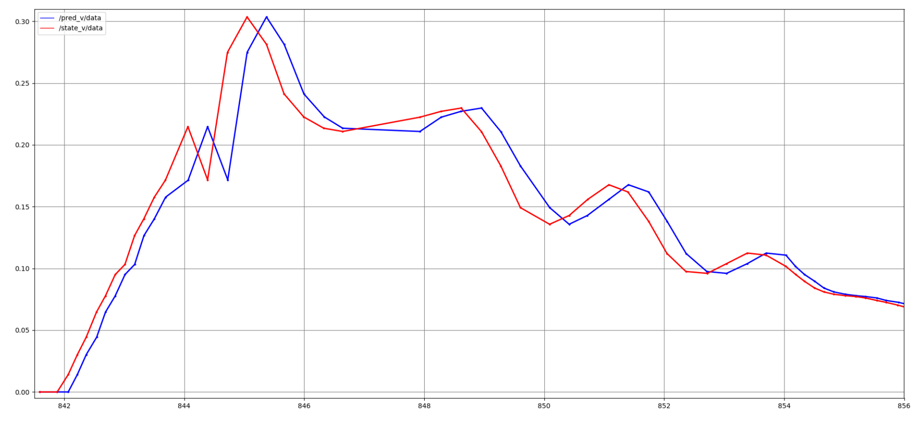


Figure 11: The predicted and measured velocity when using the original controller with acceleration and steering angle as control signals.

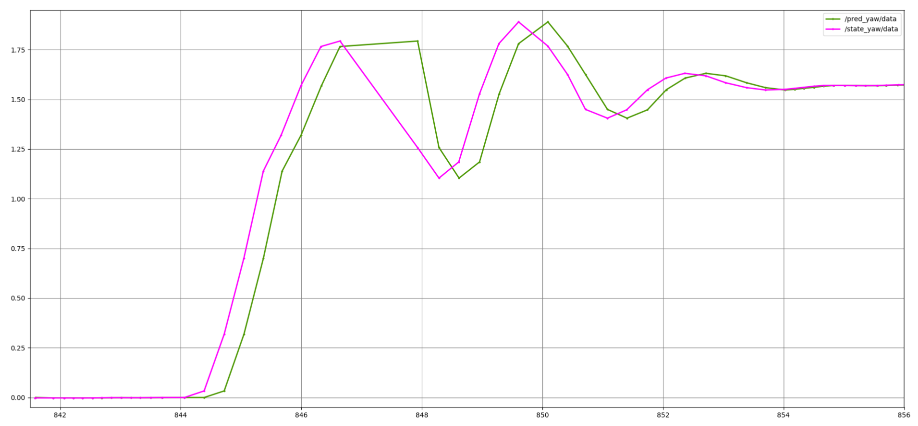


Figure 12: The predicted and measured yaw angle when using the original controller with acceleration and steering angle as control signals.

It was checked whether the computing times were impacted significantly by context switches during the MPC computations but no difference compared to the previous tests could be seen when not allowing any context switches during the computations.

In efforts to lower the computing times and the prediction error, new models were derived to check weather another model could perform better than the original model. The original model had position in x and y, velocity and yaw angle as states and acceleration as well as steering angle as control signals. It was discredited using forward Euler [10]. All of the new models, derived from the turtlebot model, were discretized using zero order hold, see equation 2, 3, 4, , 6, 7, 8 and 9. In the case of the models that have the velocity both as a state and a control signal, the state representing the velocity was added manually after the discretization rather than being included from the start.

$$\dot{z} = f(z(t), u(t)) \tag{2}$$

$$A = \frac{d}{dz} f(z^*(t), u^*(t)) \tag{3}$$

$$B = \frac{d}{du} f(z^*(t), u^*(t)) \tag{4}$$

$$\dot{z} \approx Az + Bu \tag{5}$$

$$\Phi = e^{Ah} = I + Ah + \frac{A^2h^2}{2} + \underbrace{\cdots}_{\substack{=0 \\ \text{for equation} \\ 13,\ 21 \text{ and } 29}} \tag{6}$$

$$\Gamma = \int_0^h e^{As} ds B \tag{7}$$

$$z_{k+1} \approx \Phi z_k + \Gamma u_k \tag{8}$$

$$y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} z \tag{9}$$

$$a = acceleration\,[m/s^2]$$
$$\phi = rotational\ velocity\,[rad/s]$$
$$v = linear\ velocity\,[m/s]$$
$$v_a = rotational\ velocity\,[rad/s]$$
$$\delta = steering\ angle\,[rad]$$
$$l = wheelbase\,[m]$$
$$z = states$$
$$u = control\ signals$$
$$h = sample\,period\,[s]$$

One model used the velocity and and steering angle as control signals giving the results shown in equation 10, 11, 12, 13, 14, 15, 16 and 17.

$$z = \begin{pmatrix} x \\ y \\ \phi \end{pmatrix} \tag{10}$$

$$u = \begin{pmatrix} v \\ \delta \end{pmatrix} \tag{11}$$

$$\dot{z} = \begin{pmatrix} vcos(\phi) \\ vsin(\phi) \\ v\frac{tan(\delta)}{l} \end{pmatrix} \tag{12}$$

$$A = \begin{pmatrix} 0 & 0 & -v^*sin(\phi^*) \\ 0 & 0 & v^*cos(\phi^*) \\ 0 & 0 & 0 \end{pmatrix} \tag{13}$$

$$B = \begin{pmatrix} cos(\phi^*) & 0 \\ sin(\phi^*) & 0 \\ \frac{tan(\delta^*)}{l} & \frac{v^*}{l*cos^2(\delta^*)} \end{pmatrix} \tag{14}$$

The velocity was added as a state in the linearized verion

$$z = \begin{pmatrix} x \\ y \\ v \\ \phi \end{pmatrix} \tag{15}$$

$$\Phi = \begin{pmatrix} 1 & 0 & 0 & -v^*sin(\phi^*)h \\ 0 & 1 & 0 & v^*cos(\phi^*)h \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16}$$

$$\Gamma = \begin{pmatrix} cos(\phi^*)h - \frac{sin(\phi^*)tan(\delta^*)h^2v^*}{2l} & -\frac{sin(\phi^*)h^2v^{*2}}{2lcos^2(\delta^*)} \\ sin(\phi^*)h + \frac{cos(\phi^*)tan(\delta^*)h^2v^*}{2l} & \frac{cos(\phi^*)h^2v^{*2}}{2lcos^2(\delta^*)} \\ 1 & 0 \\ \frac{tan(\delta^*h)}{l} & \frac{hv^*}{lcos^2(\delta^*)} \end{pmatrix} \tag{17}$$

The prediction error was measured during a test round but it was still considerably high and the behaviour was similar to that of the original model, see figure 13, 14, 15 and 16.

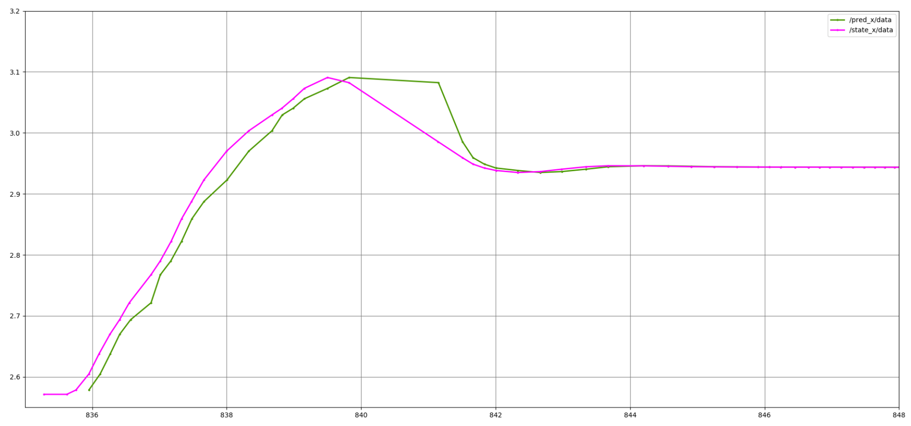


Figure 13: The predicted and measured x-position when using the controller with velocity and steering angle as control signals.

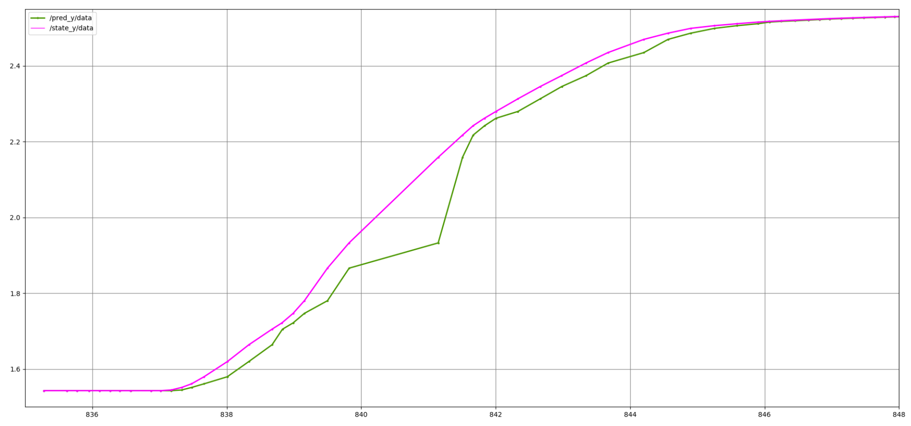


Figure 14: The predicted and measured y-position when using the controller with velocity and steering angle as control signals.

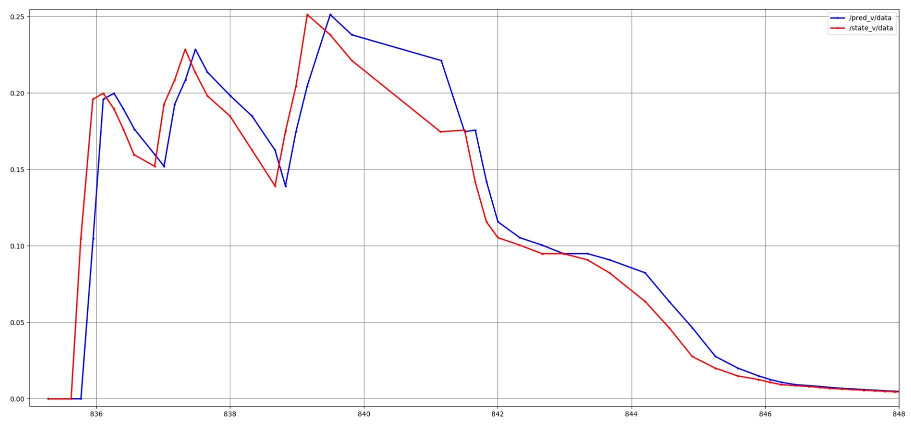


Figure 15: The predicted and measured velocity when using the controller with velocity and steering angle as control signals.

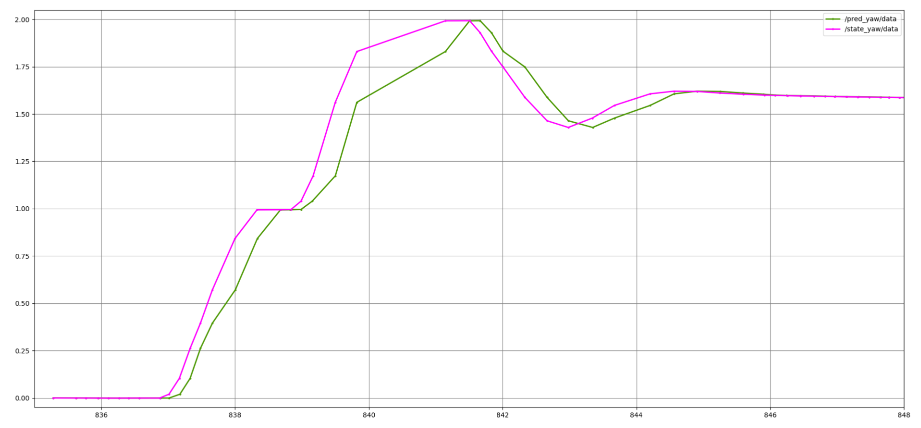


Figure 16: The predicted and measured yaw angle when using the controller with velocity and steering angle as control signals.

One model used the linear and rotational velocity as control signals giving the results shown in equation 18, 19, 20, 21, 22, 23, 24 and 25.

$$z = \begin{pmatrix} x \\ y \\ \phi \end{pmatrix} \tag{18}$$

$$u = \begin{pmatrix} v \\ v_a \end{pmatrix} \tag{19}$$

$$\dot{z} = \begin{pmatrix} vcos(\phi) \\ vsin(\phi) \\ v\frac{tan(\delta)}{l} \end{pmatrix} \tag{20}$$

$$A = \begin{pmatrix} 0 & 0 & -v^* sin(\phi^*) \\ 0 & 0 & v^* cos(\phi^*) \\ 0 & 0 & 0 \end{pmatrix} \tag{21}$$

$$B = \begin{pmatrix} cos(\phi^*) & 0 \\ sin(\phi^*) & 0 \\ 0 & 1 \end{pmatrix} \tag{22}$$

The velocity was added as a state in the linearized verion

$$z = \begin{pmatrix} x \\ y \\ v \\ \phi \end{pmatrix} \tag{23}$$

$$\Phi = \begin{pmatrix} 1 & 0 & 0 & -v^* sin(\phi^*)h \\ 0 & 1 & 0 & v^* cos(\phi^*)h \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{24}$$

$$\Gamma = \begin{pmatrix} cos(\phi^*)h & -\frac{v^*}{2} sin(\phi^*)h^2 \\ sin(\phi^*)h & \frac{v^*}{2} cos(\phi^*)h^2 \\ 1 & 0 \\ 0 & h \end{pmatrix} \tag{25}$$

The prediction error was measured during a test round again but it was still considerably high and the behaviour was similar to that of the original model, see figure 17, 18, 19 and 20.

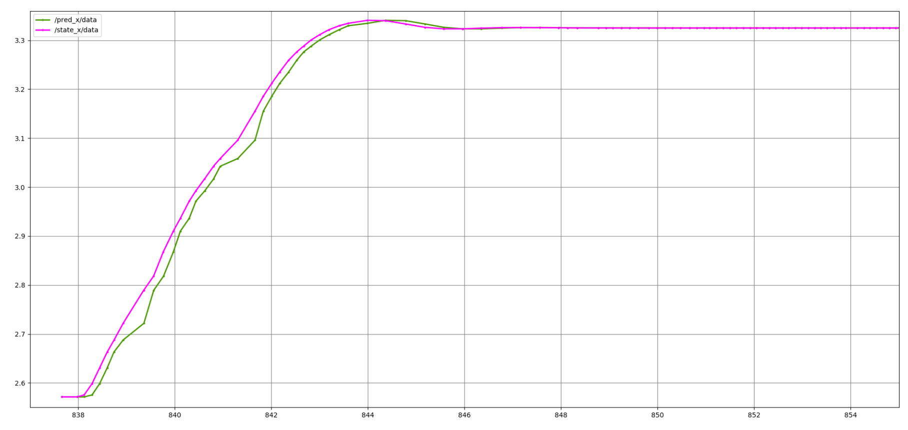


Figure 17: The predicted and measured x-position when using the controller with linear and rotational velocity as control signals.

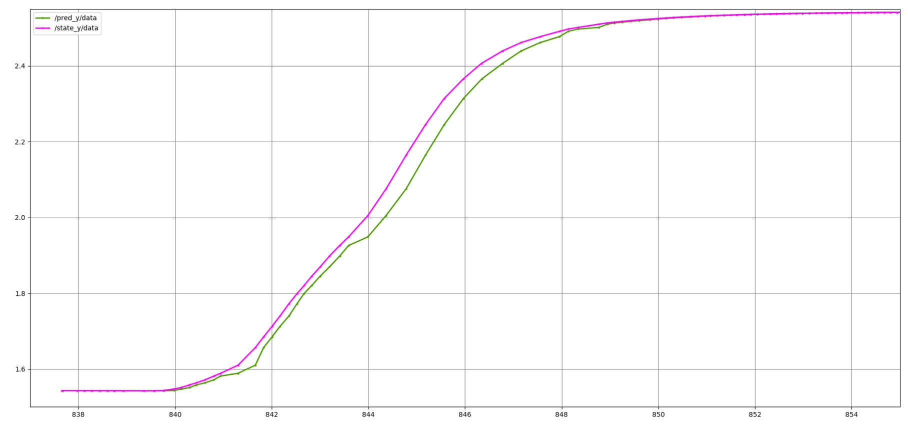


Figure 18: The predicted and measured y-position when using the controller with linear and rotational velocity as control signals.

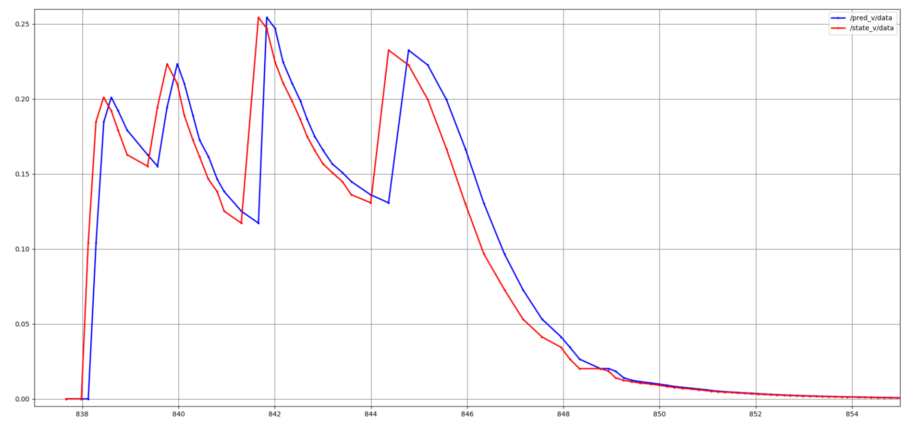


Figure 19: The predicted and measured velocity when using the controller with linear and rotational velocity as control signals.

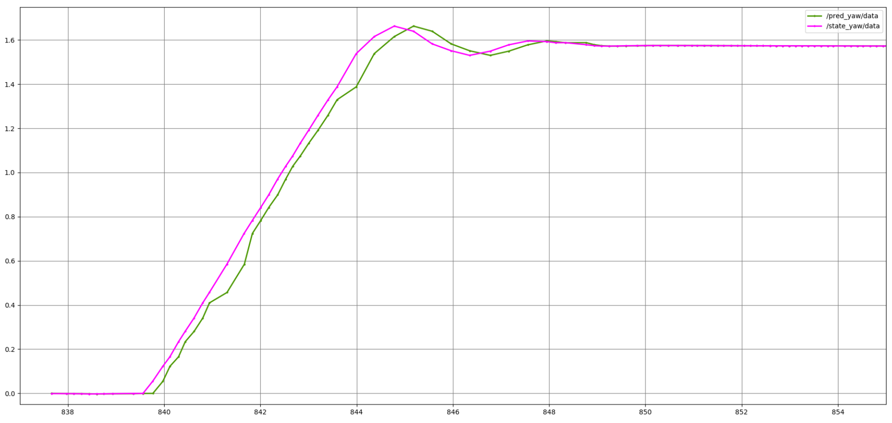


Figure 20: The predicted and measured yaw angle when using the controller with linear and rotational velocity as control signals.

Since the prediction error for the velocity was lowest when acceleration was used as a state and the prediction error for the yaw angle was lowest when rotational velocity is was chosen to combine these two into a controller using acceleration and rotational velocity as control signals, see equation 26, 27, 28, 29, 30, 31 and 32.

$$z = \begin{pmatrix} x \\ y \\ v \\ \phi \end{pmatrix} \tag{26}$$

$$u = \begin{pmatrix} a \\ v_a \end{pmatrix} \tag{27}$$

$$\dot{z} = \begin{pmatrix} vcos(\phi) \\ vsin(\phi) \\ a \\ v_a \end{pmatrix} \tag{28}$$

$$A = \begin{pmatrix} 0 & 0 & cos(\phi^*) & -v^*sin(\phi^*) \\ 0 & 0 & sin(\phi^*) & v^*cos(\phi^*) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{29}$$

$$B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{30}$$

$$\Phi = \begin{pmatrix} 1 & 0 & cos(\phi^*)h & -v^*sin(\phi^*)h \\ 0 & 1 & sin(\phi^*)h & v^*cos(\phi^*)h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{31}$$

$$\Gamma = \begin{pmatrix} \frac{1}{2}cos(\phi^*)h^2 & -\frac{v^*}{2}sin(\phi^*)h^2 \\ \frac{1}{2}sin(\phi^*)h^2 & \frac{v^*}{2}cos(\phi^*)h^2 \\ h & 0 \\ 0 & h \end{pmatrix} \tag{32}$$

Once again, the prediction error was measured during a test round but it was still considerably high and the behaviour was similar to that of the original model, see figure 21, 22, 23 and 24.

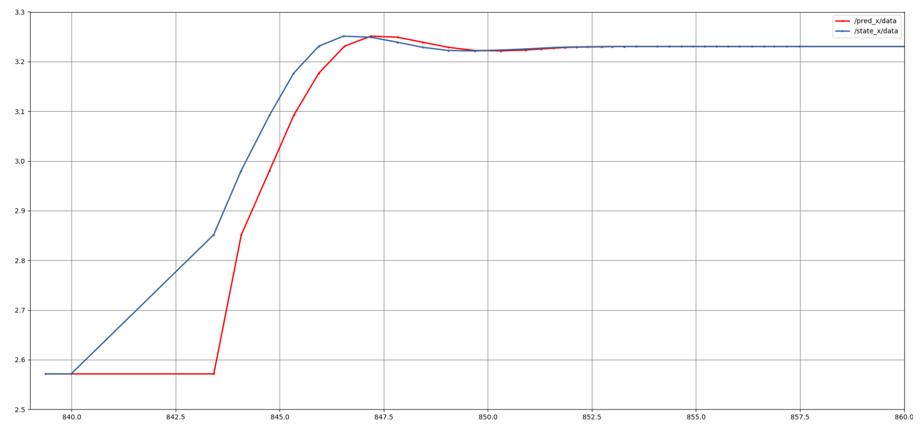


Figure 21: The predicted and measured x-position when using the controller with acceleration and rotational velocity as control signals.

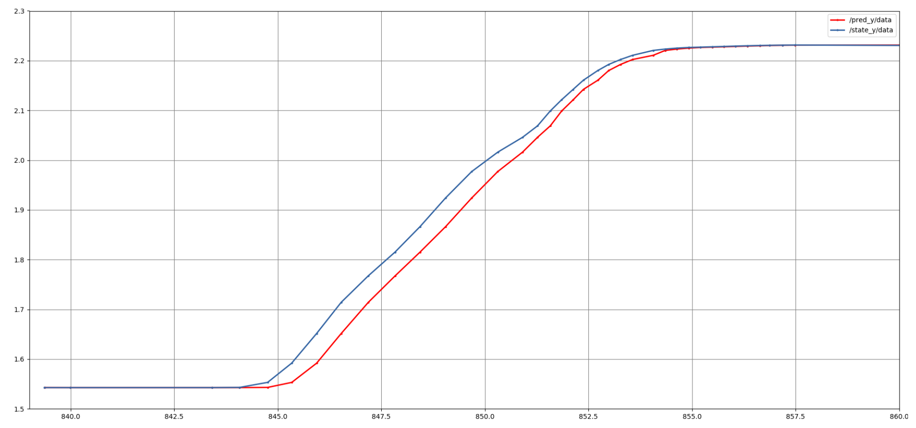


Figure 22: The predicted and measured y-position when using the controller with acceleration and rotational velocity as control signals.

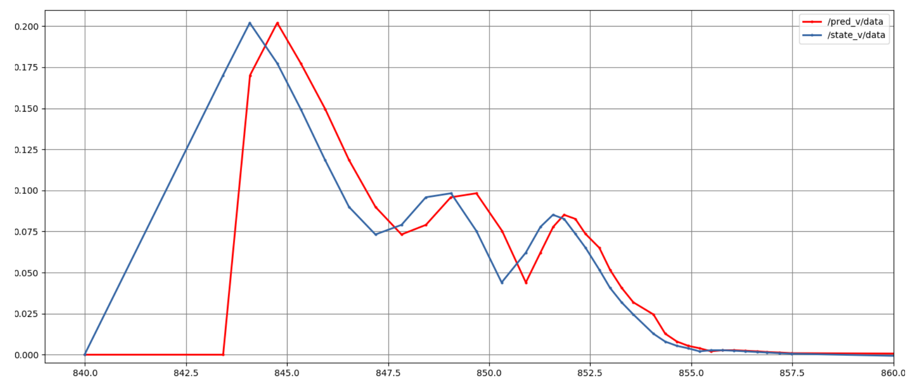


Figure 23: The predicted and measured velocity when using the controller with acceleration and rotational velocity as control signals.

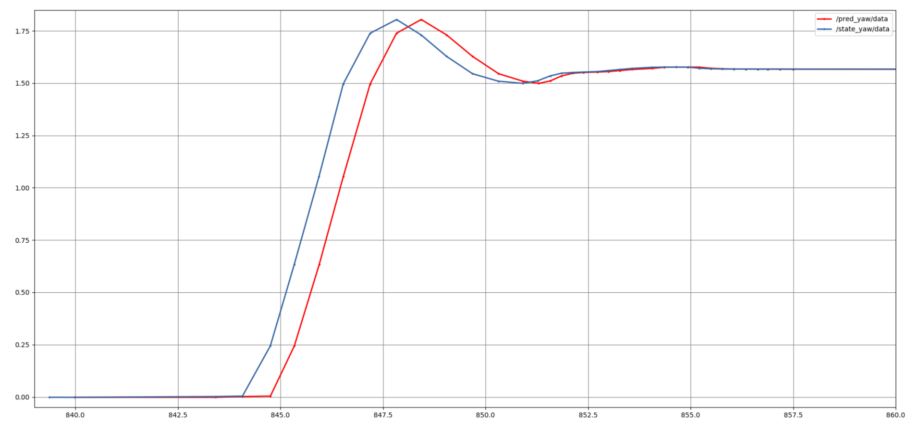


Figure 24: The predicted and measured yaw angle when using the controller with acceleration and rotational velocity as control signals.

Unfortunately the trajectory control could not be solved within the frame of the project. A PHD student at Caltech, Ugo Rosolia, who is experienced with MPC suggested to try his nonlinear MPC instead [9]. However, there was not enough time to try this. Another alternative would be to use a paid online course for learning about navigation with the turtlebot [4]. A final suggestion is to run the simulations on a more powerful computer and see if the computing times could be reduced enough.

3.3 Still to be implemented

It was prioritized to get the trajectory control working and therefore the pedestrians and static obstacles were not integrated with the main simulations. However, to integrate the static obstacles one only need to set the position of the obstacles and include that data when launching the gazebo world. To integrate the pedestrian one would have to set a initial position for the pedestrians and include that when launching the gazebo world as well. Then, to move the pedestrians, one would have to specify their movement and make sure they don't collide with each other, for example by just letting two pedestrians move back and forth on the crossing next to each other. Unlike the turtlebots, the model of the pedestrians does not include the full dynamics and therefore their velocity in the x and y direction can be set without taking disturbances such as friction into account, i.e no complex feedback controller is necessary to make sure a pedestrian stay on its straight path.

4 Challenges

The major challenges has been to identify how ROS, Gazebo and the turtlebot description are supposed to interact with each other. For example, how to launch turtlebots in a gazebo world, how to set velocity of a robot and how to read the states of a robot. This was solved by taking one problem at a time and search the web for demos and tutorials. Another major challenge was to get the plugins working. It took a relatively long time to figure out that the plugins had to be compiled together with the ROS-workspace. Once that had been figured out there were still some problems, just recompiling the workspace was not enough, one had to compile it completely from scratch by deleting the devel and build folder as well as the CMakeLists for the workspace and then compile it. After compiling, one had to manually copy the file setup.sh in the devel folder and paste it into the gazebo folder within devel/share. Why this has to be done manually remains unknown. The time it took to get this working was significantly longer than expected. The implementation of the control system to make the turtlebots follow a given trajectory turned out to be a major challenge that was not solved within the time frame of the project.

References

[1] Miguel Angel. *Gazebo QA 002 – How to create Gazebo plugins in ROS Part 2.* URL: https://www.theconstructsim.com/gazebo-qa-002-create-gazebo-plugins-ros-part-2/ (visited on 07/28/2020).

[2] Miguel Angel. *Gazebo QA 003 – How to spawn an SDF custom model in Gazebo with ROS.* URL: https://www.theconstructsim.com/gazebo-qa-003-spawn-sdf-custom-model-gazebo-ros/ (visited on 07/28/2020).

[3] Karl-Erik Årzén Björn Wittenmark Karl Johan Åström. *Computer Control: An overview.* Educational version. Lund: Department of automatic control, Lund University, 2016, pp. 68–73.

[4] The Construct. *Mastering with ROS: Turtlebot3.* URL: https://www.robotigniteacademy.com/en/course/mastering-ros-turtlebot3/details/ (visited on 09/17/2020).

[5] Josefine Graebener et al. *Automated Valet Parking.* URL: https://github.com/jgraeb/AutoValetParking (visited on 07/01/2020).

[6] David Lu. *roslaunch/API Usage.* URL: http://wiki.ros.org/roslaunch/API%5C%20Usage (visited on 07/28/2020).

[7] Richard M. Murray and Josefine Graebener. *SURF 2020: Hardware Implementation of Contract-Based Design for Automated Valet Parking System.* URL: http://www.cds.caltech.edu/~murray/wiki/index.php?title=SURF_2020:_Hardware_Implementation_of_Contract-Based_Design_for_Automated_Valet_Parking_System (visited on 07/01/2020).

[8] Arif Rahman. *ROS QA 130 – How to launch multiple robots in Gazebo simulator?* URL: https://www.theconstructsim.com/ros-qa-130-how-to-launch-multiple-robots-in-gazebo-simulator/ (visited on 07/01/2020).

[9] Ugo Rosolia. *nonlinearFTOCP.* URL: https://github.com/urosolia/MultiRate/blob/master/python/nonlinearFTOCP.py (visited on 09/17/2020).

[10] Atsushi Sakai. *Path tracking.* URL: https://pythonrobotics.readthedocs.io/en/latest/modules/path_tracking.html (visited on 09/13/2020).

[11] Atsushi Sakai. *Path tracking simulation with iterative linear model predictive control for speed and steer control.* URL: https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathTracking/model_predictive_speed_and_steer_control/model_predictive_speed_and_steer_control.py (visited on 09/13/2020).

[12] Lennart Ljung Torkel Glad. *Reglerteori Flervariabla och olinjära metoder.* 2:8. Lund: Studentlitteratur AB, 2016, pp. 360–366.

[13] Lennart Ljung Torkel Glad. *Reglerteori Flervariabla och olinjära metoder.* 2:8. Lund: Studentlitteratur AB, 2016, pp. 453–468.

[14] Lennart Ljung Torkel Glad. *Reglerteori Flervariabla och olinjära metoder.* 2:8. Lund: Studentlitteratur AB, 2016, pp. 423–432.