**Multi-agent System for non-Holonomic Racing (MuSHR)**

**SLAM-based techniques:**

Gmapping: http://wiki.ros.org/gmapping

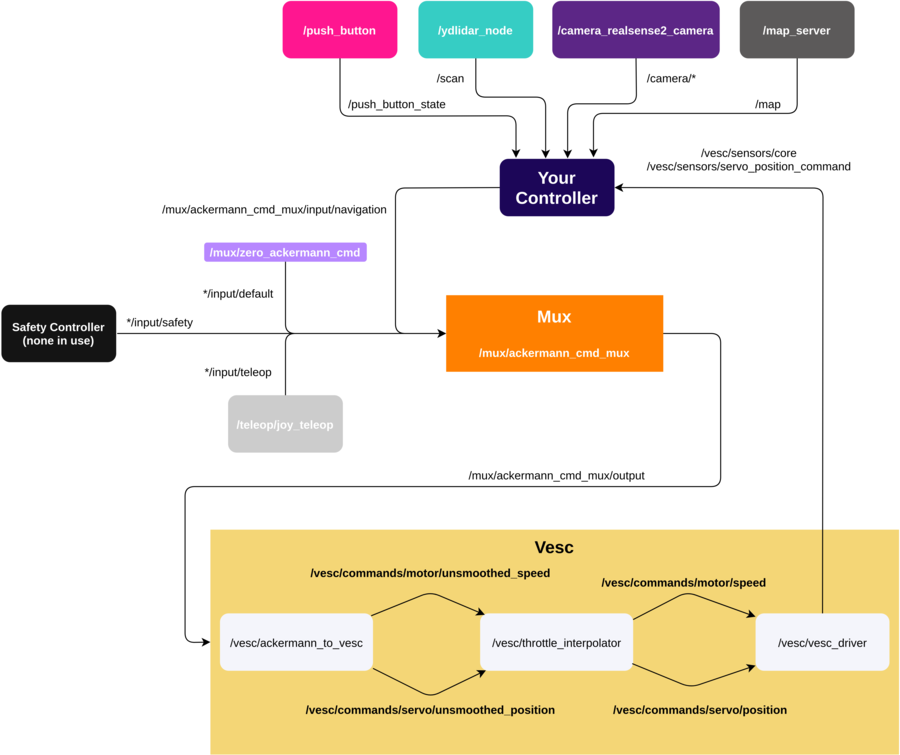
Google’s Cartographer: [https://google-cartographer-ros.readthedocs.io/en/latest/#](https://google-cartographer-ros.readthedocs.io/en/latest/)

**Hardware Overview**

This diagram shows all the components and their data &amp; power connections.

* **Chasis (Redcat Racing Blackout SC 1/10) :** The car chasis. It has adjustable 4x4 suspension and non-flat tires. It is also used for mounting things to it, including the housing which contains the computer and sensors.
* **Computer (Jetson Nano):** This the computer that runs the software on the car. You can connect to the computer in 3 ways primarily: ssh through local network to static IP, connecting to car’s network and using ssh, or plugging a monitor, keyboard, and mouse into the computer directly.
* **Motor (Jrelecs F540 3930KV):** The single DC motor that powers all four wheels. The motor makes the car move and is controlled by the VESC.
* **Servo (ZOSKAY 1X DS3218):** A servo is another type of motor but is better at going to specific angles along its rotation than rotating continuously. The servo’s job is to steer the front wheels by taking in steering angle commands. The servo is also controlled by the VESC.
* **VESC (Turnigy SK8-ESC):** The VESC is responsible for taking high level control commands like steering angle and velocity and converting that to power/angle commands for the motor/servo. The VESC has an associated ROS node. This node takes your ROS ackermann\_msgs/AckermannDriveStamped message and converts that to VescStateStamped (power), and Float64 (steering angle) messages. These messages are something the physical VESC can use to control the motor and servo.
* **NiMH Battery (Redcat Racing HX-5000MH-B):** There are 2 batteries. One to power the motor and in turn the VESC (because the motor power flows through the VESC). And a second to power the computer and sensors. Many of the issues (particularly with the VESC) can stem from one of these batteries having a low battery, so it is good to check them regularly.
* **Power Converter (DZS Elec LM2596):** This power converter converts the higher voltage of the battery to the necessary 5V max for the computer.
* **RGBD Camera (Realsense D435i):** This Intel Realsense camera can publish both rgb and depth. It has a associated node so you can just subscribe to the topic to use the images! OpenCV even has a function for converting ROS Image messages to something OpenCV can use. Note depth cameras are different from stereo cameras (how humans do it). They both provide the same output, but depth cameras project a IR pattern and use a IR camera to see the deformation of that pattern to compute depth. Depth cameras will also sometimes use stereo in addition to make their measurements more accurate. It publishes to the /camera topics. This camera can also publish IMU measurments, but they currently are not being used.
* **Laser Scanner (YDLIDAR X4):** This 360 degree sensor works by having a 1D laser range finder spin around. With each distance reading their is a associated angle the range finder was at. It publishes an array of distance and angle measurements to the /scan topic.
* **Wireless Controller (Logitech F710):** This provides controls to the cars! It doesn’t just have to be use for teleop it can be used by your programs for most anything. It publishes on the /joy topic.
* **Bumber Switch (Vex Bumper Switch):** This is a button on the front of the car that you can use to indicate a collision. It publishes a binary signal to **/push\_button\_state**.
* **Micro SD Card:** Storage for the OS and any logs. What’s great about this is if you want to switch cars you can simply switch SD cards.

**Software Components**



- The sensor nodes take the raw input, does some processing, and convert it to ROS friendly topics:

- The map server converts a .yaml map file to a ros topic.

- Your controller (orange) takes in sensor topics coming in from the each sensors' ROS node and the map. It then outputs some command to drive the car.

- That command is put into the mux which listens on multiple /mux/ackermann\_cmd\_mux/input channels and selects the highest priority.

\*/input/default is a zero throttle/steering command that is passed whenever your controller and teleop are not publishing.

- Currently, MuSHR does not have an explicit safety controller publishing to the \*/input/safety topic. The mux priorities can be found mushr\_base/ackermann\_cmd\_mux/param/mux.yaml and they are listed in order of priority below.

1. Safety

2. Teleop

3. Navigation

4. Default

Once the highest priority command is output it goes to the VESC. The VESC smoothes the command by clipping the min/max of the steering/throttle so we don’t try to turn the wheels 180 degrees for example. It then provides that to the VESC driver which directly controls the motors.

**Simple Trajectory Planning**

Create a simple ROS node to read commands (velocity and steering angle) from a file line by line, and send them to the simulator to be applied to the simulated car. Each line will denote a command to be applied for one second. The first line is a message to send as the “starting pose” of the car.

The input files will be of the form:

0,0,0.0

2.0,0.09

3.0,-0.15

The first line is the initial position, of the form x, y, theta, where x and y are the starting coordinates in the map, and theta is the initial angle of the car. The following two lines describe commands that tell the car how fast to go, and at what steering angle. The first says to run at 2.0 meters per second, with a steering angle of 0.09 radians. The second, to run at 3.0 meters per second, with a steering angle of -0.15 radians. We will be applying each command for 1 second. Note that a positive steering angle corresponds to a left turn and a negative steering angle corresponds to a right turn.

*Examples:*

plans/straight\_line.txt

0,0,0

2,0.0

3,0.0

4,0.0

5,0.0

6,0.0

6,0.0

6,0.0

5,0.0

4,0.0

3,0.0

2,0.0

plans/figure\_8.txt

0,0,0.785

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

2.0,-0.09

*Code:* src/path\_publisher.py

Explanation: https://mushr.io/tutorials/intro-to-ros/

|  |
| --- |
| #!/usr/bin/env python  import rospy  from ackermann\_msgs.msg import AckermannDrive, AckermannDriveStamped  from geometry\_msgs.msg import (  Point,  Pose,  PoseWithCovariance,  PoseWithCovarianceStamped,  Quaternion,  )  from tf.transformations import quaternion\_from\_euler  def run\_plan(pub\_init\_pose, pub\_controls, plan):  init = plan.pop(0)  send\_init\_pose(pub\_init\_pose, init)  for c in plan:  send\_command(pub\_controls, c)  def send\_init\_pose(pub\_init\_pose, init\_pose):  pose\_data = init\_pose.split(",")  assert len(pose\_data) == 3  x, y, theta = float(pose\_data[0]), float(pose\_data[1]), float(pose\_data[2])  q = Quaternion(\*quaternion\_from\_euler(0, 0, theta))  point = Point(x=x, y=y)  pose = PoseWithCovariance(pose=Pose(position=point, orientation=q))  pub\_init\_pose.publish(PoseWithCovarianceStamped(pose=pose))  def send\_command(pub\_controls, c):  cmd = c.split(",")  assert len(cmd) == 2  v, delta = float(cmd[0]), float(cmd[1])  dur = rospy.Duration(1.0)  rate = rospy.Rate(10)  start = rospy.Time.now()  drive = AckermannDrive(steering\_angle=delta, speed=v)  while rospy.Time.now() - start < dur:  pub\_controls.publish(AckermannDriveStamped(drive=drive))  rate.sleep()  if \_\_name\_\_ == "\_\_main\_\_":  rospy.init\_node("path\_publisher")  control\_topic = rospy.get\_param("~control\_topic", "/car/mux/ackermann\_cmd\_mux/input/navigation")  pub\_controls = rospy.Publisher(control\_topic, AckermannDriveStamped, queue\_size=1)  init\_pose\_topic = rospy.get\_param("~init\_pose\_topic", "/initialpose")  pub\_init\_pose = rospy.Publisher(init\_pose\_topic, PoseWithCovarianceStamped, queue\_size=1)  plan\_file = rospy.get\_param("~plan\_file")  with open(plan\_file) as f:  plan = f.readlines()  # Publishers sometimes need a warm-up time, you can also wait until there  # are subscribers to start publishing see publisher documentation.  rospy.sleep(1.0)  run\_plan(pub\_init\_pose, pub\_controls, plan) |

## Navigation Stack Overview

At the highest level MuSHR’s navigation stack consists of three principal components:

1. **Receding Horizon Controller (RHC) Node:** This node is responsible for motion planning and generating controls(steering, speed) for the car. The implementation shipped with the car uses a Model Predictive Controller (MPC) to generate control signals which are sent to the car’s motor controller (VESC).

RHC is a model predictive contoller that plans to waypoints from a goal (instead of a reference trajectory). This controller is suitable for cars that don't have a planning module, but want simple MPC.

**librhc Layout**

librhc (mushr\_rhc\_ros/src/librhc) is the core MPC code, with the other source being ROS interfacing code. The main components are:

* Cost function (librhc/cost): Takes into account the cost-to-go, collisions and other information to produce a cost for a set of trajectories.
* Model (librhc/model): A model of the car, currenly using the kinematic bicycle model.
* Trajectory generation (librhc/trajgen): Strategies for generating trajectory libraries for MPC to evaluate.
* Value function (librhc/value): Evaluation of positions of the car with respect to a goal.
* World Representation (librhc/workrep): An occupancy grid based representation for the map.

**mushr\_rhc\_ros ROS API**

#### Publishers

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /rhcontroller/markers | [visualization\_msgs/Marker](http://docs.ros.org/api/visualization_msgs/html/msg/Marker.html) | Halton points sampled in the map (for debugging purposes). |
| /rhcontroller/traj\_chosen | [geometry\_msgs/PoseArray](http://docs.ros.org/api/geometry_msgs/html/msg/PoseArray.html) | The lowest cost trajectory (for debuggin purposes). |
| /car/mux/ackermann\_cmd\_mux/input/navigation | [ackermann\_msgs/AckermannDriveStamped](http://docs.ros.org/api/ackermann_msgs/html/msg/AckermannDriveStamped.html) | The lowest cost control to apply on the car. |

#### Subscribers

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /map\_metadata | [nav\_msgs/MapMetaData](http://docs.ros.org/api/nav_msgs/html/msg/MapMetaData.html) | Uses dimension and resolution to create occupancy grid. |
| /move\_base\_simple/goal | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | Goal to compute path to. |
| /car/car\_pose | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | When using simulated car pose Current pose of the car. |
| /car/pf/inferred\_pose | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | When using particle filter for localization Current pose of the car. |

#### Services

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /rhcontroller/reset/hard | [std\_srvs/Empty](http://docs.ros.org/api/std_srvs/html/srv/Empty.html) | Creates a new instance of the MPC object, redoing all initialization computation. |
| /rhcontroller/reset/soft | [std\_srvs/Empty](http://docs.ros.org/api/std_srvs/html/srv/Empty.html) | Resets parameters only, not redoing initialization. |

1. **Localization Node:** In order for the controller to know where it is, and therefore also whether it is in the proximity of known obstacles in the map, it must know its location. Solving this problem is called “localization”. The Localization Node is implemented using a method called Particle Filtering which relies primarily on a data stream from the laser scanner.

**mushr\_pf: Particle Filter**

Particle filter (pf) for localization using the laser scanner. The pf requires a map and laser scan.

**Publishers**

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /car/pf/inferred\_pose | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | Particle filter pose estimate |
| /car/pf/viz/particles | [geometry\_msgs/PoseArray](http://docs.ros.org/api/geometry_msgs/html/msg/PoseArray.html) | Partilcle array. Good for debugging |
| /car/pf/viz/laserpose | [geometry\_msgs/PoseArray](http://docs.ros.org/api/geometry_msgs/html/msg/PoseArray.html) | Pose fo the laser |

**Subscribers**

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /map | [nav\_msgs/OccupancyGrid](http://docs.ros.org/melodic/api/nav_msgs/html/msg/OccupancyGrid.html) | Map the robot is in |
| /car/scan | [sensor\_msgs/LaserScan](http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html) | Current laserscan |
| /car/vesc/sensors/servo\_position\_command | [std\_msgs/Float64](http://docs.ros.org/api/std_msgs/html/msg/Float64.html) | Current steering angle |
| /car/vesc/sensors/core | [vesc\_msgs/VescStateStamped](https://github.com/prl-mushr/vesc/blob/master/vesc_msgs/msg/VescStateStamped.msg) | Current speed |

1. **Planner Node** This node generates a plan that the RHC controller will follow. The planner does not consider dynamic obstacles when constructing its plan; rather it uses a static map of the environment. It chains the car’s motion primitives together into a plan, using a search algorithm (such as A\*).

**mushr\_gp: Global Planner**

This ROS package hosts the global planner for the MuSHR system. It wraps a search-based planner from [SBPL](https://github.com/sbpl/sbpl) (Search-Based Planning Library). Search based methods have two parts - first, the planning problem and environment must be represented by a graph, and then the graph must be searched from a starting point to a goal.

This MuSHR global planner builds upon the SBPL repo by Search-Based Planning Lab at Carnegie Mellon University. MuSHR specifically makes use of the ARA\* planning algorithm and plans on an implicit graph in a SE2 (x, y, θ) state space.

#### Publishers

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /path\_topic | [geometry\_msgs/Path](http://docs.ros.org/en/melodic/api/nav_msgs/html/msg/Path.html) | The trajectory computed by the planner. |

#### Subscribers

|  |  |  |
| --- | --- | --- |
| **Topic** | **Type** | **Description** |
| /map | [nav\_msgs/OccupancyGrid](http://docs.ros.org/en/melodic/api/nav_msgs/html/msg/OccupancyGrid.html) | Uses the provided occupancy grid as the graph for planning. |
| /goal\_topic | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | Goal to compute path to. |
| /start\_topic | [geometry\_msgs/PoseStamped](http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html) | Starting location of the path being computed. |

NOTE: The reason why there are two planners (mushr\_gp and mushr\_gprm) is because mushr\_gp is too resource intensive to be run on the jetson nano 4GB variant. However, if the desktop/laptop computer remains connected and in range of the MuSHR car, you can run mushr\_gp on the computer instead as they share the same ROS master. If you need the planner to run on the jetson nano, we recommend using the mushr\_gprm package. For the Jetson Xavier NX or when running on the sim exclusively, mushr\_gp will work. Both repositories contain ROS packages that reproduce the desired functionality. You need only concern yourself with each package’s launch files to use them effectively. You can find the launch files in each package’s launch directory.

## Running the navigation stack

To operate the navigation stack, we will use foxglove to send pose targets to the vehicle. When operating in the real world, the pose estimate of the car may be incorrect. You can correct this by providing the particle filter with the correct pose estimate using the Set Pose Estimate button on the bottom right of the foxglove window and then using the button to publish clicked points.

**Note**: When publishing a pose, the pose will correspond to the pose at the tip of the arrow and not the base of the arrow.

<https://mushr.io/tutorials/autonomous-navigation/final_vid_stack.mp4>

**Multi-agent Navigation System**

A navigation system enables a robot to move quickly between different poses while avoiding collisions with the environment or other agents. The navigation system in this project uses a slightly modified version of the [NH-TTC system](https://github.com/davisbo/NHTTC). It is a decentralized, multi-agent navigation system that considers the Non-holonomic constraints of the car and the time to collision when finding the optimal control actions. To learn more about how it works, check out the paper [here](http://motion.cs.umn.edu/r/NH-TTC/arxiv-NHTTC.pdf)!

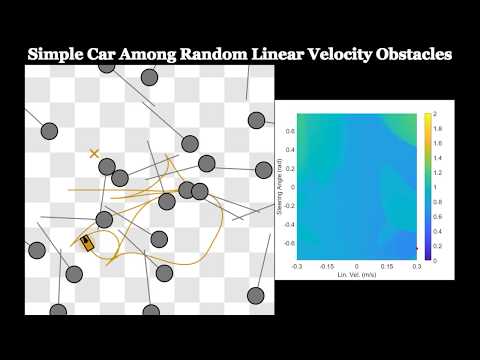
***NH-TTC: A generalized framework for anticipatory collision avoidance***

NH-TTC is a general method for fast, anticipatory collision avoidance for autonomous robots having arbitrary equations of motions. Our proposed approach exploits implicit differentiation and subgradient descent to locally optimize the non-convex and non-smooth cost functions that arise from planning over the anticipated future positions of nearby obstacles. The result is a flexible framework capable of supporting high-quality, collision-free navigation with a wide variety of robot motion models in various challenging scenarios. We show results for different navigating tasks, with our method controlling various numbers of agents (with and without reciprocity), on both physical differential drive robots, and simulated robots with different motion models and kinematic and dynamic constraints, including acceleration-controlled agents, differential-drive agents, and smooth car-like agents. The resulting paths are high quality and collision-free, while needing only a few milliseconds of computation as part of an integrated sense-plan-act navigation loop.

Code: <https://github.com/davisbo/NHTTC>

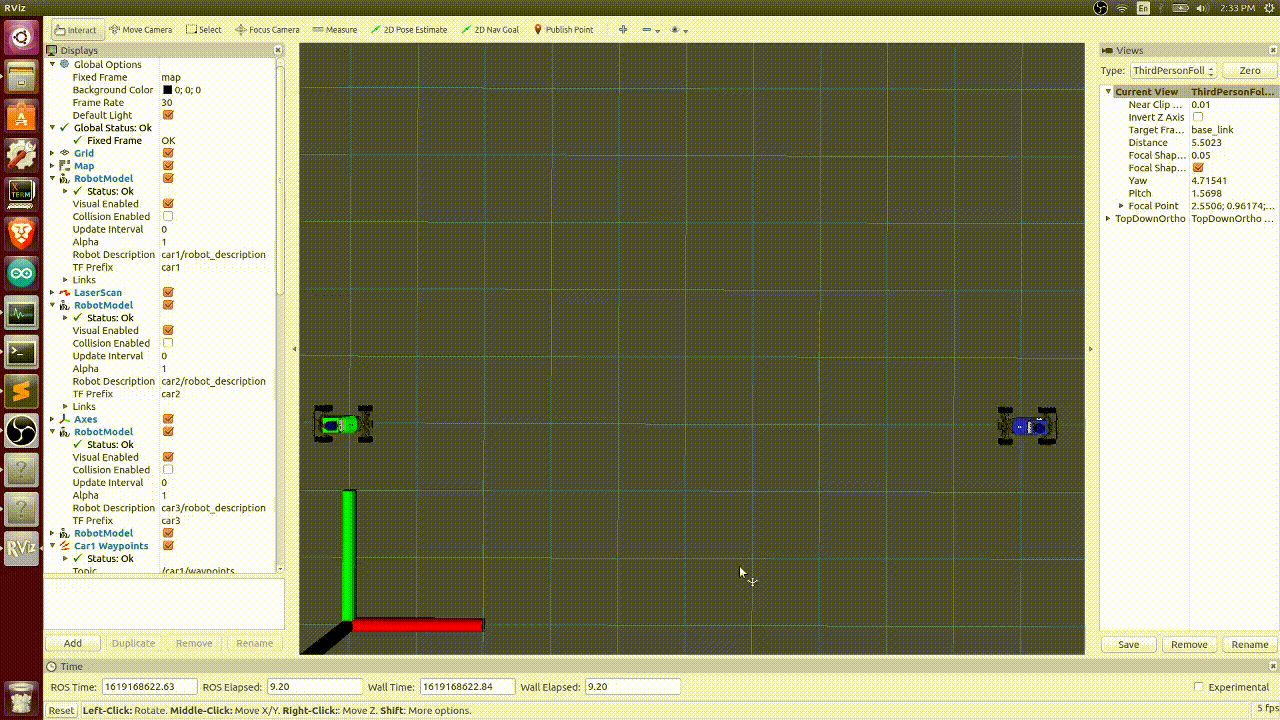
Paper: <http://motion.cs.umn.edu/r/NH-TTC/arxiv-NHTTC.pdf>

Video: <https://youtu.be/9EG48v0r_YY>

[](https://www.youtube.com/embed/9EG48v0r_YY?feature=oembed)

## Running the Simulator example:

In this demo, the blue car is trying to follow the blue arrows and the green car is trying to follow the green arrows. The green car is supposed to stop a small distance after crossing the intersection of the two paths whereas the blue car is supposed to move towards the top after the intersection of the two paths. The two paths coincide to force the navigation system to display its capabilities



The system takes a set of waypoints rather than a single goal point. The message to publish for this is:/car\_name/waypoints of type: [geometry\_msgs/PoseArray](http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/PoseArray.html).

The waypoint can also contain just one waypoint, so it is possible to test the system with single waypoints if you prefer that. The reason why taking an array is preferred is so that the waypoint management code doesn’t have to be written by you (user). You simply pass the set of the waypoints in and the navigation sytem takes care of managing them on it’s own.

The z axis coordinate represents the time difference between 2 waypoints.

Note that a z axis value of 0.001 equals a time difference of 1 unit. This is done so that the waypoints don’t look like they’re floating off the ground when visualized in rviz. The unit of time is equal to the time it takes for the car to cover the distance between two waypoints in a straight line at the rated speed. The reason for this is to allow the system’s speed to be scaled up or down without changing the global plan’s timing itself.

## Tuning the parameters

The multi-agent navigation system *can* work out of the box for most applications, however, it is possible to tune it. The parameters can be changed inside the yaml config file, in this case, the nhttc\_demo.yaml. The navigation system uses a solver which has some level of stochasticity to it, which can lead to slightly different behavior. The demonstrations shown here are to explain the effect of changing the parameters and do not indicate the exact performance. Also note that there may be additional parameters besides the ones specified here. Those are experimental and should be set to false or left as is by the user.

carrot\_goal\_ratio: 1.0

max\_ttc: 6.0

solver\_time: 20

obey\_time: true

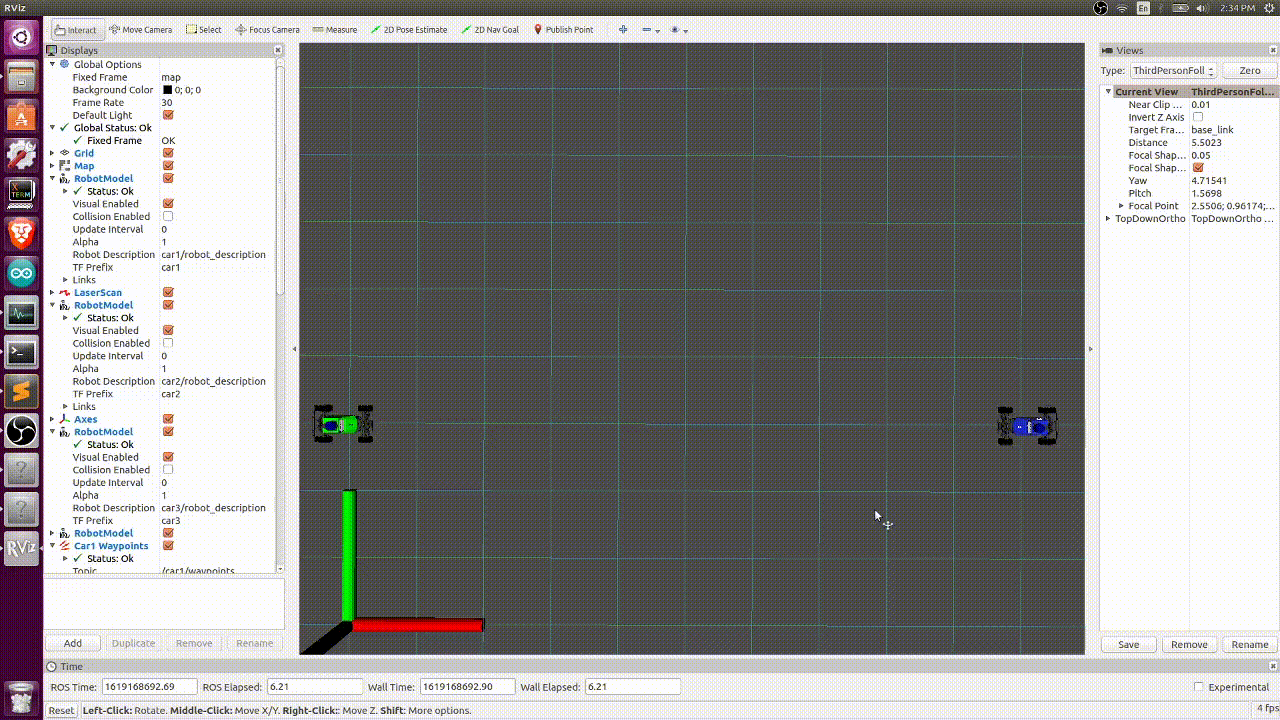
**1) carrot\_goal\_ratio:** The ROS wrapper implements a carrot-goal navigation system where waypoints are selected from a prescribed path. The waypoints are selected such that they are some “lookahead” distance away from the car. Keeping the car aimed at a waypoint farther away prevents it from getting stuck in a local minimum. Keeping this lookahead closer to the car makes sure the car does not deviate too far away from the prescribed path while getting to the point farther down the line. The ratio of this lookahead distance or carrot-goal distance to the turning radius has been defined as the carrot-goal ratio. The reason for that name is that the way the system works is akin to a donkey(MuSHR car) trying to eat a carrot(waypoint) hung from a stick that you(navigation system) are holding while sitting on top of it. The donkey moves where the carrot goes but can never actually reach it:

Text

Description automatically generated

A value of 1.0 means the carrot goal distance is the same as the turning radius. A value of 2 indicates that the carrot goal distance is twice the turning radius. larger numbers result in smoother navigation, however, they come with the drawback of greater path-deviation as the system will tend to “round off” corners a lot sooner. The first figure shows the performance with carrot-goal-ratio of 1.0, and the second figure shows performance with a ratio of 1.5:

A screenshot of a computer

Description automatically generated with medium confidence

If the car tends to get stuck around turns, increase the carrot-goal ratio in increments of 0.1. If the car appears to be rounding off the turns too soon or significantly deviating from the path near turns, causing issues with other agents, reduce the carrot-goal-ratio in decrements of 0.1.

**2) max\_ttc:** Stands for maximum time to collision. This parameter decides which agents to consider and which to not consider when optimizing for the next control action. The time to collision is calculated using the current state (pose as well as twist) of all agents. A larger max\_ttc results in a larger time horizon for considering potential collisions. A larger max\_ttc will make the car respond earlier to other agents but can result in the car deviating from its path too early. A smaller max\_ttc will make the car less sensitive to agents far away but may result in the car responding too late and ending up in deadlocks more often. The first figure shows the performance with a max\_ttc of 3.0 and the second shows the performance with max\_ttc of 6.0 seconds. Notice how the cars start avoiding each other earlier in the second example.