OpenPodCar: an open source vehicle for self-driving car research

Chris Waltham[†], Fanta Camara[*†],David Churchill[†], Charles W. Fox[*†]
[*]Institute for Transport Studies, University of Leeds, UK
[†] School of Computer Science, University of Lincoln, UK

***Abstract*—Open source software for autonomous vehicle (self driving car) control is now available but there is a need for a corresponding open-source hardware platform to enable researchers to build standard setups and share research in this field. Here we are constructing such a platform. The Lincoln Podcar is a physical vehicle based on a low cost off the shelf, hard canopy, mobility scooter. It is large enough to transport one person at speeds up to 15kmh, for example for use as a last-mile autonomous taxi service from the train station to the office; or to transport up to three Deliveroo-sized freight containers similarly around a city center. But it is also small enough to make serious injury to pedestrians unlikely in the event of a collision. Together with its low costs it thus forms an ideal balance between real world utility, safety, cost and research convenience. The open source platform consists of (a) CAD designs, sample bill of materials, and build instructions to modify the off the shelf donor vehicle for electronic control; (b) Arduino, ROS and Gazebo control and simulation software files which provide standard ROS interfaces and simulation of the vehicle; and (c) higher-level ROS implementations of standard medium-level robot control, including the movebase interface which enacts command to get the vehicle from one pose to another. The vehicle uses Ackermann steering and Reeds-Sheppe curves. Its total build cost in 2020 is around 7000USD in total, including sensors and computing equipment.**

***Index Terms*—Autonomous Vehicles, Pedestrian - Vehicle Interaction [3 maximum]**

I. Requirements

Specific requirements for such a platform are that it needs to be as low cost as possible, and easy to build. This is to enable the community to reproduce and use it. Consumer levels of safety and reliability are not required, preferring to minimise cost, through research standards of safety and reliability are required.

II. Donor vehicle

A Pihsiang TE-889XLSN hard-canopy scooter (branded in UK as Shoprider Traverso) is used as the podcar platform. It is an electric mobility scooter powered by two 12V batteries connected in series to provide 24V operating voltage and containing 75Ah. In its standard configuration, the donor vehicles steering is controlled by a human operated loop handle bar. The speed and braking systems are both powered by an electric motor and an electric brake via the trans-axle assembly, controlled by an AC2 digital controller receiving different voltage signals to drive forward or brake. The manual speeding and braking systems are controlled by three buttons connected in series on the handle bar. A toggle switch in parallel with a resistor (10kΩ) to choose speed mode high or low; A speed dial knob via a variable resistor (20kΩ) to choose a maximum speed value; A throttle lever connected with a potentiometer (5kΩ), 2.5kΩ to 2.6kΩ for each side to speed or brake.

This project has received funding from EU H2020 interACT: Designing cooperative interaction of automated vehicles with other road users in mixed traffic environments under grant agreement No 723395. Thanks to Yao Chen for making figures X and Y, and Gabriel Walton for scoping simulation tools.

Fig. 1: OpenPodcar test drive.

III. Mechanical modification

A. Steering

To automate steering, a GLA750-P 12V DC linear actuator with position feedback is mounted between an archor on the underside of the chassis and the car's front axle via bearings. This actuator has a 8mm/s full load (750N) speed and 250mm stroke length (installation length is 390mm).

To access the underside of the vehicle, use two axle stands as in fig. 2. There is an existing hole in the right front wheel axle. Mount the linear actuator via rear hole to the left side of the front chassis and connect it through the front hole of the actuator with the hole in the car's right front wheel axle via bearings as shown in fig. 3 and fig. 8.

Fig. 2: Tilting the vehicle using two axle stands, to enable access to the underside. (Note also lidar mounted to roof.)

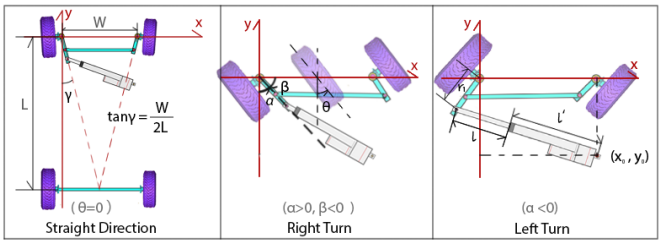

Fig. 3: The bottom view of front wheels steering relationship including geometric coefficients

B. Sensor installation

1) Lidar option: A Velodyne16 lidar sensor is mounted on the vehicle roof using a small optical tripod (TODO give model). Note that optical devices have unusual physical mounting standards which use Imperial rather than metrics units. This system is used by optics research based on optical table sizing. The lidar screws onto the tripod. The tripod is cabled-tied to the vehicle roof via drilled holes at locations in fig. TODO. It is mounted at a 10 degree tilt downwards (to allow pedestrians to be most clearly seen in the 16 scan lines).

2) Depth camera option: A lower cost alternative to lidar is to use a stereo camera for point cloud sensing. In this option, a StereoLabs ZedCam is mounted similarly on the vehicle roof.

3) TODO Odometry: TODO – we need some Hall effect or other shaft encoders to measure odometry.

IV. Electronic Modification

New vehicle electronics are added on an acrylic board mounted under the seat of the vehicle, as shown in fig. 6 and 8 and mounted as in fig. fig:actuatorMounted. Details are described below. Where different voltage power supplies are required, DCDC converters are used and mounted in the same way.

Fig. 4: Underside with linear actuator added for steering.

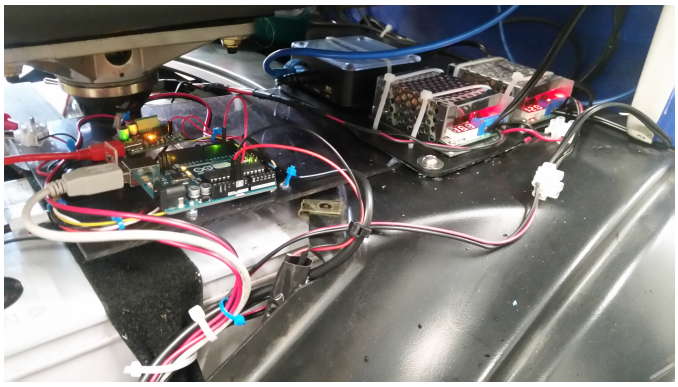

Fig. 5: Below-seat area inside cabin, with newly added boards..

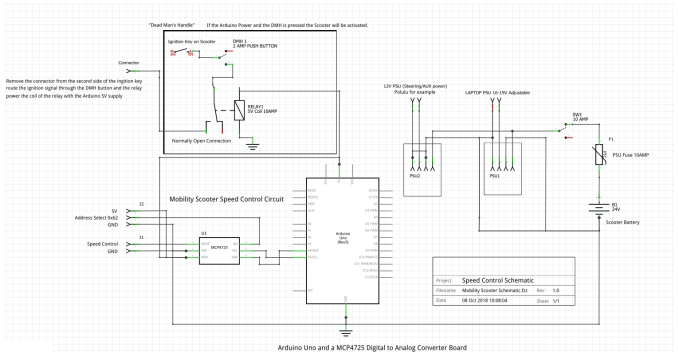

Fig. 6: Circuit diagram for electronic modifications.

A. Steering control system

The front wheels are steered by a Pololu (TODO give model number) PID controller, which takes serial port desired

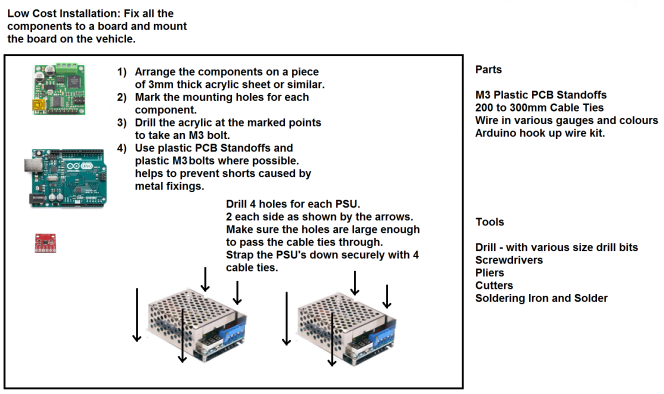

Fig. 7: Drilling.

positions as input. It also takes feedback position information as an analog voltage from the linear actuator as an input. It outputs analog high-power voltages to the linear actuator.

TODO give PID settings.Windows app is needed to set them once only.

The relationship between the required central turning angle θ of the pair of front wheels and extending length l of linear actuator is,

$$\theta = \alpha - \arctan(\frac{W}{2H}) \tag{1}$$

$$\beta = \alpha - \frac{\pi}{2} \tag{2}$$

$$x = r_1 * \cos(\beta) \tag{3}$$

$$y = r_1 * \sin(\beta) \tag{4}$$

$$l = \sqrt{(x_0 - x)^2 + (y_0 - y)^2} - L + l_0 \tag{5}$$

Where r_1, x_0, y_0, W, H and L are the geometric coefficients shown in Fig.1. Among them, the value of y_0 is negative. l_0 is the initial value of the linear actuator position feedback.

Diagnostic test commands can be passed to the Polulo using the commands provided in /tools/cmdSteer. *Do not give commands outside the range 1000-25000 as they have damaged the vehicle.*

TODO – WRITE SAFETY WRAPPER TO LIMIT THIS RANGE..

A non-ROS test of the C API for the Pololu is provided in /tools/pololuTestCSerial.

FA:cmd	Effect
2500	max right
1900	center
1000	max left

Do not give commands outside this range as they have mechanically destroyed the system.

B. Speed control system

1) Dead mans handle - IMPORTANT SAFETY INFORMATION: Autonomous vehicles can present a significant hazard to people and the environment in which they operated. Damage to surroundings and possible injury to operators and bystanders could result from inappropriate use or malfunction. It is essential that a suitable emergency stop system is implemented in all autonomous vehicles.

Implementing and testing this safety system should be undertaken with the drive wheels of the vehicle raised off of the ground, allowing for checks to be made of the DMH without the risk of the vehicle speeding off out of control.

Given the development platform nature of the OpenPodCar, a safety mechanism which stops the vehicle under fault conditions is a very important part of the design. During the build several methods of implementing this safety system where checked, the one described below is the current solution.

The speed controller on this particular mobility scooter has a dead zone at around 2.5V on its speed input signal which corresponds to drive wheels stopped. Above the dead zone and up to 5V being forward control values and 0V to below the dead zone being reverse control voltages.

These values were checked using a multi meter prior to implementing the Arduino controller and DAC. This means that if the DAC output from the Arduino where to fail and 0V for example is applied to the mobility scooters speed controller input, the vehicle would respond by spinning the drive wheels backwards at full throttle. This being a dangerous eventuality, it is very important in terms of safety to ensure this situation can not arise.

A two stage approach is used to reduce this risk. Refer to the schematic diagram DMH section in conjunction with this description.

a) Stage 1 Relay.: A relay is used which interrupts the mobility scooters key ignition circuit. If the relay is not energised by the presence of a 5V supply to the Arduino, the vehicles movement is disabled. This effectively ensures that if the Arduino is non-functional, for example its power supply has failed or it has been unplugged from the USB port of the control PC and there is a danger that the DAC is not producing the control systems required voltage, the scooter is automatically disabled by effectively switching it off.

b) Stage 2 DMH Switch.: A sturdy push button is used which also interrupts the vehicles key ignition circuit. If the PodCar operator detects any abnormality in operation during operation, he/she simply releases pressure from the DMH switch and the vehicles movement is disabled. The DMH switch is wired in series with the relay in the key ignition circuit ensuring that if both the relay contacts and the DMH switch are closed, this is the only condition where the PodCar movement is active.

The addition of the Relay and the DMH Switch are essential for safe operation, especially where new unproven autonomous control systems are in development.

A photograph of the installed system is shown in fig. 8.

Deadmans handle to ignition. Replace key with relay (TODO model) and dead-mans-handle button switch (TODO model) on a 10m, 2-core cable. TODO: are any other console functions changed?

Fig. 8: Steering console showing newly added relay (with lit LED) to dead mans handle, replacing the ignition switch (only) in the circuit.

TODO edited version of wiring from shoprider manual. And photo of the real wires.

2) Speed controller: TODO how to set the console switches to correct fixed settings.

An Arduino UNO is used to send electric signals to the vehicle's motor controller in place of the donor vehicles paddle controllers potentiometer. An Adafruit MCP4725 DAC is connected to the Arduino as in fig. TODO CIRCUIT, and used to send clean analog speed command voltages to the donor vehicles internal controller.

Arduino code is supplied in the distribution (/Arduino/ThrottleControlSerial.ino). When uploaded to the Arduino (using the standard Arduino IDE running on the laptop), it provides a simple serial port API running at 112000baud (TODO other erial options.) It receives commands of the form FA:210 as speed commands. The test scripts /tools/zeroSpeed.py and /tools/testSpeed.py can be used to send example commands for debugging.

FA:cmd	Voltage	Effect
0	0	stupid fast reverse
80	0	v fast reverse (ros limit)
132	1.82	slowest reverse motion
		stop - dead zone - allows signition
201	2.71	slowest forward motion
240	2.78	v fast forward (ros limit)
255	?	stupid fast forward

To start the ignition, the car safety system requires the control voltage to be in the dead range. A problem is that this doesnt correspond precisely to any fixed speed bytes, due to the USB power issues. But if we pick a number solidly in the center of the dead zone, such as 164, this will work for most USB supplies. (i.e. when the vehicles battery is flatter, the voltages provided to USB power by it are lower. For example, we might send 164 and get 1.9V instead of the usual 2.26V.) this may result in vehicle not starting - maybe by design or accident. get beep as is outside start zone. thoery: the ardunio is getting lower power eg max 4.9V instead of 5V, which gets divided by the DAC value. To deal with these instabilities, we added a potential divider at the battery to check the voltage and control the podcar accordingly.

- insert Figure 2 about the wiring diagram of speed buttons and micro-controller.
- talk about safety (dead-man button).And speed code part with details.

V. Physical vehicle software interface (ROS)

A ROS interface to and from the physical vehicle is provided, for ROS Kinetic and Ubuntu 16.04 (xenial), as illustrated in fig. 12 and described below.

A. Control system interface

The system expects to hear two incoming ROS control messages: speedcmd_meterssec and wheelAngleCmd, which contain single floats representing the desired speed in meters per second, and the desired front wheel orientation in radians respectively.

These two messages are received by ROS nodes speedm2arduino and wheelAngle2Pololu, which are ROS drivers for the Arduino speed controller and the Polulo steering controller respectively.

B. Manual joystick control

Converters from a standard ROS USB joystick driver node to the speed and angle command interface messages are provided, by joystick2speedms and joystick2wheelAngle. These use the y axis of a joystick for speed and x for steering.

C. Lidar option

Velodyne works with the velodyne ROS package. But it needs to be set up so we can talk to the Velodyne over ethernet. The laptop must be on wired network, not wifi. The IPs must be configured as in the velodyne docs TODO details/refs. The lidar IP is 192.168.1.201.

D. Depth camera option

VI. Simulation

A robotics simulation of the vehicle is provided for use in Gazebo 8.6.0 underROS Kinetic and Ubuntu 16.04 (Xenial).

The physics simulation is based on a simplified vehicle geometry with two large cuboids containing the vehicles mass. **??** Parallel steering linkage is simulated via ODE joints forming a tracking rod. Rear drive motor and the steering linear actuator are simulated via ODE PID controllers. Wheel geometry was measured from the physical vehicle. Wheel forwards and lateral Coulomb friction coefficients were tuned by trial and error to produce realistic motion and steering. The model is represented in SDF format, which is easy portable for use in other simulation and CAD systems.

There are some key differences when it comes to actuating the tracking rod between simulation and real life. On the physical robot, a linear actuator is mounted on the underside of the vehicle and attached to the front axle, which can

extend/retract to angle the wheels. However, in simulation, the pivot joints that join the front wheels and the tracking rod can have a force applied directly to them without the need for a complicated actuator. This reduces the complexity of the model without affecting its behaviour, and is still appropriately accurate to real life.

There are also some similarities, for example both steering solutions implement a PID (Proportional, Integral, Derivative) controller, that uses a proportional gain, integral function and differential function to determine exactly how much force to apply at any given time in order to keep the wheels at the desired angle. The parameters for the existing robots PID controller have not been published, however trial and error found a proportional gain of 2, differential weight of 1 and integral weight of 0 were adequate.

The simulation implements the same ROS interface as the physical vehicle system to enable plug and plug interoperability between them. In the simulation, vehicle control messages are received, and vehicle sensor messages are sent by, a Gazebo plugin (podcarGazeboROSPlugin.cpp) which calls Gazebos simulation functions.

Desired vehicle control commands, as for the physical vehicle, consist of (speed, angle) pairs, to command the speed of the rear wheels in m/s (speedcmd_meterssec message) and the steering angle (wheelAngleCmd message) of the front wheels in radians. The plugin and Gazebo implement these commands using simulated rotary and linear PID controllers and actuators, to rotate the rear wheels and linearly actuate the tracking rod respectively.

The plugin then publishes sensor messages corresponding to those from the physical vehicle. These include: /odometry/groundTruth, a (noiseless) standard odometry format estimate for the vehicles pose; and /camera/image_raw and /camera_depth, the depth camera data.

A detailed graphical mesh model of the vehicle is provided for display, rather than physical simulation, purposes **??**. This may be important for experiments such as virtual reality interactions between the simulated vehicle and human subjects, e.g. [**?**] as their psychological responses may depend on the apparent size and share of the vehicle.

Fig. 13 shows the complete ROS node configuration used during simulation, under manual joystick control.

To install and run the simulation:

```
cd <install location>/catkin_ws
catkin_make
source devel/setup.bash
cd src/podcar/models/plugins
cmake .
make
export GAZEBO_PLUGIN_PATH=$GAZEBO_PLUGIN_PATH`pwd`:
cd ..
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH`pwd`:
cd ../../../
rosdep install --from-paths src --ignore-src -r -y
roslaunch podcar podcarsim.launch
```

VII. Path planning and control (movebase)

Front-wheel-steering means that the front two wheels are used to steer the vehicle, with the rear wheels trailing and-or powering the vehicle. Ackermann steering is a special case of front-wheel-steering, in which the front wheels are connected to a tracking rod via two shorter steering arms arranged so that the two steering arms form a triangle with the center point of the vehicles rear axle when the wheels are facing forwards. This results (TODO cite proof) in the front wheels forming the correct angles (which are different from each other) needed for the vehicle to drive in circular segments, with the circle radius determined by the steering angle. Vehicles whose possible driving motions are of this form are called Dubins Cars if they can drive only forwards, or Reeds-Shepp cars if they can drive forwards and reverse. ubins and Reeds-Shepp cars have possible driving trajectories from pose A to pose B which consist of at most one circle arc segment and two straight lines to and from it. The donor podcar is a Reeds-Shepp car with Ackermann steering.

The values for parameters such as minimum turning radius were calculated from the technical specifications of the base vehicle [**?**]SHOPRIDERSPECS).

Move Base is a set of ROS nodes that are standard for controlling robots via ROS. It enforces conventions for being able to send messages to controllers, for example the use of Twist messages. These messages are made up of 6 floats, the first 3 representing linear velocity in each axis (x, y and z; forward, left and up) with the latter 3 representing angular velocity in the same space. Move Base requires the robot to move with these given velocities when a Twist message is published to the /cmd vel topic (Marder-Eppstein, 2018). These standards ensure compatibility between packages, making the addition of existing packages to a project much easier.

Twist messages, however, dont apply to all robots. For example, the vast majority of robots arent able to move directly up in the world, so the linear z component of the message is ignored. A similar problem arises when considering car-like steering, as they are unable to inflict an angular velocity. In this case, the angular z component is instead taken to refer to the desired angle of the steerable wheels, which resolves the incompatibility problem provided all nodes involved are aware of the difference. Alternatively, an intermediary node could be introduced that calculates the wheel angle needed to make the robot turn at a given rate, however this would be unnecessarily complex and wouldnt have the desired effect if the robots linear velocity was 0.

Fig. 13 shows the complete ROS node configuration used during movebase, with the simulation backend. When using the physical vehicle, simulation nodes are replaced by the physical vehicle nodes as described above.

VIII. Localisation and mapping system

gmapping – current

TODO:

OrbSLAM – based on Octomap, adding mono visual features. Octomap can also be used with any other point cloud data eg lidar.

NDT SLAM – used at LCAS - ILIAD.

IX. User Guide

A. Power up

Check that the vehicles original lever for auto-manual is set to auto (DOWN). It is on the main motor, under the vehicle at the rear left, colored red. Requires some force to move it.

Power on the vehicle using the original on-off switch located under the seat on the left. It is marked ON-OFF.

Power on the modified electronics using the new toggle switch. (This lights LEDs on the DCDCs and Pololu, and the lidar makes a whirring sound).

Check that the batteries are charged (use a multimeter across one of the DCDC converters, need to see 24V or over. DO NOT USE THE VEHICLE IF IT IS UNDERCHARGED, THIS IS DANGEROUS.

Power on the laptop using the slider switch on its front right.

Login as user podcar, password TODO.

Type: roscd podcar

Unplug the laptops USB connection and plug it in again. (HACK)

Run the test script podcar/tools/pololuCSerialTest/a.out (HACK)

Type: roslaunch podcar podcar.launch

Use the joystick to control steering and speed.

B. Usage

X. Troubleshooting Guide

A. Vehicle

1) vehicle beeps continuous when press DMH and rear wheels do not move: This is due to a safety mode preventing ignition.

Check: is the manual-auto switch under the rear motor on auto?

Check: are the batteries well charged (must be 24V or over.)

Check: is the control voltage is the dead zone, it should be.

2) Rear wheels do not move, control voltages are correct: Control voltages means the display on the voltmeter LED. Should be above 1.2 or below 1.8 for forward and backwards.

Check: main vehicle battery level, by connecting the vehicle charger and inspecting the battery charge level. Problem occurs if battery is nearly flat.

Check: charger must be disconnected for rear wheels to moe (safety feature).

B. Lidar

1) No velodyne_points message published: Check: laptop must be on wired network, not wifi.

Check: wired network must be configured correctly, see velodyne setup docs. Maybe be interfered if wifi has been used recently. Check connections to velodyne box including power and ethernet.

C. Simulation

1) AF_NET error: If this is thrown by the Gazebo plugin – it may be because Gazebo is being run standalone rather than launched as a ROS node as required.

XI. Related work

About 20 references or more if you can (at least one page) about similar work, SLAM, vehicle control and simulation, open source projects.... Cite this project from UC berkeley: http://www.barc-project.com/

XII. Discussion

We hope that the platform will be useful...

XIII. Acknowledgements

Thanks to Jacob Lord for creating the graphical mesh model, and Yao Chen for initial research into Dubins path methods.

potential_divider.png

Fig. 9: Potential divider linked to the battery

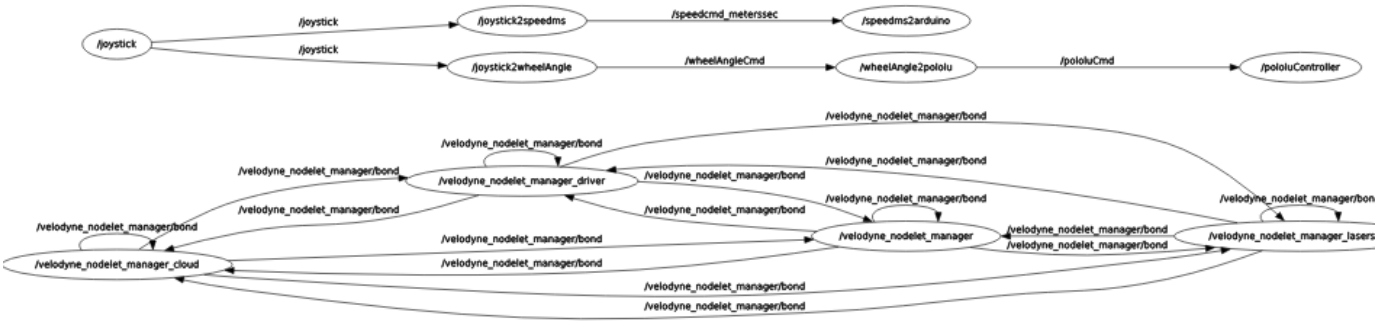

Fig. 10: ROS nodes and messages for low level control.

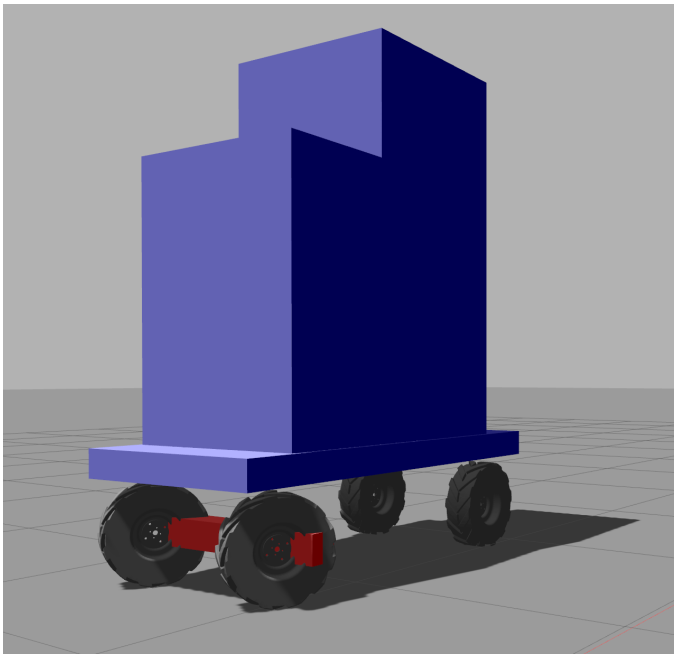

Fig. 11: Physical simulation of vehicle.

Fig. 12: Visual mesh simulation of vehicle.

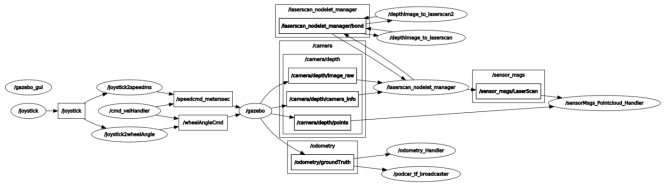

Fig. 13: ROS nodes used in simulation under manual joystick control.

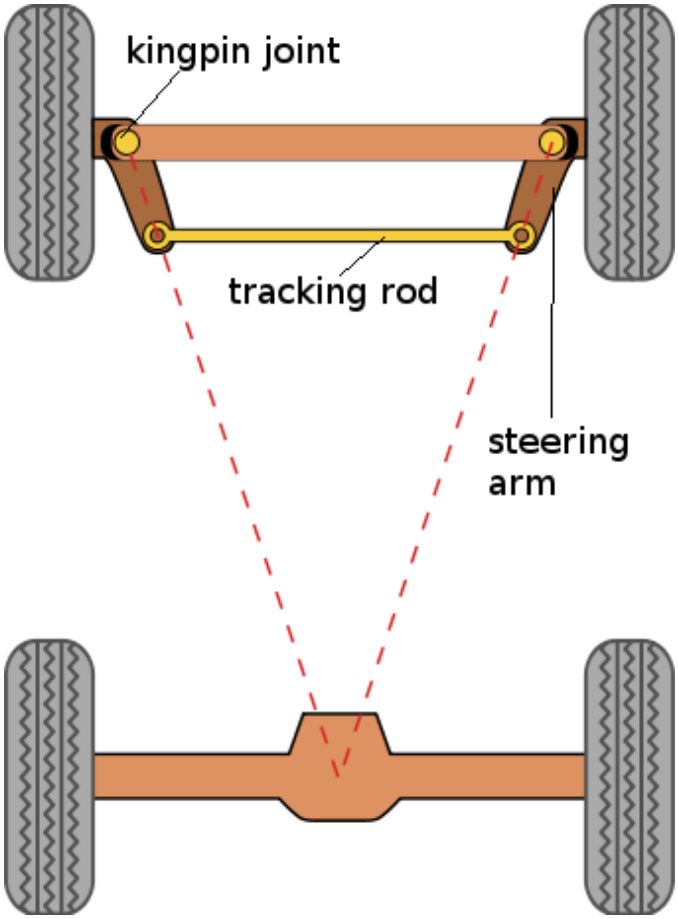

Fig. 14: Ackermann steering (Source: Wikipedia; Ackermann Steering; Creative Commons.

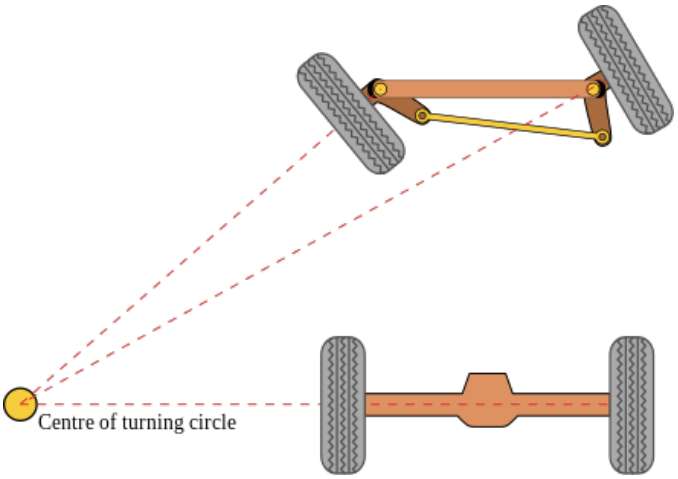

Fig. 15: Ackermann steering (Source: Wikipedia; Ackermann Steering; Creative Commons.

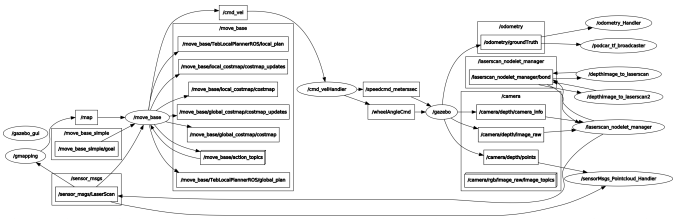

Fig. 16: ROS nodes used in simulation under move$_b asecontrol$.