

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-Shepp 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]



Fig. 1: OpenPodcar test drive.

I. INTRODUCTION

A. Open source hardware

The use of OSH allows for more effective and accessible sharing and collaboration among researchers [2], as well as more accessible entry-level projects such as examples from Oxer and Blummings (2009). OSH brings a number of benefits to a research community, as a common context for results and accessible platform with which to test new ideas allows for more engagement, leading to a more mature field of study.

The precise meaning of open-source hardware (OSH) remains widely debated, unlike open-source software which has become widely accepted to be defined by standard documents such as the GNU General Public License and BSD licence. The OSH issues debated stem from the differing design process between hardware and software, as the development of hardware is much more expensive to iterate and has a much higher entry-level skill requirement [1]. This topic has been augmented by the combination of 3-D printing with open-source microcontrollers [9], however is not yet as widely accepted open-source software.

For software to be open source, its entire stack of *compiler-linked* dependencies must also be open source, while non-linked dependencies need not be open source. For example, an open source program might depend on a non-open operating system which it calls through non-compiler linked shared object libraries (so or dll files). The main debate around OSH definitions is what the equivalent boundary should be for hardware. In conventional product design, a product is a legally defined entity which may, for example, be registered as an entity via a patent or receive safety certification as an entity. Such a product will specify as part of its legal existence that it is comprised from, or manufactured using, other products, which themselves have legal definitions. For example, to obtain a CE safety mark, all the component products must themselves have CE marks. The OSH community does not usually use the term product as this denotes a commitment to certain commercial worldviews, but may use the term design in an analogous way.

It is currently impossible to build almost any large hardware design that does not incorporate non-open designs or make use of non-open designs as tools during construction. So various definitions of OSH of different strengths have been proposed as difference balances between openness and pragmatism:

Deep OSH. The strongest form of OSH, as aimed for but not yet realised in the Open Source Ecology (OSE) project, would build a complete stack of OSH such that there are no non-open components or tools used at any level. While we support the idea of deep OSH and plan to work towards a strong OSH version of our system, this is beyond the scope of our current work.

Weak OSH. The weakest form of OSH is based on conventional product design, and considers the design itself to be the only component in need of openness. Only the design for how to combine it with other components is OSH. This means that the new design could be completely dependent on patented component and tool designs, and on the owners of these designs allowing their use. For example, the Arduino computer is Weak OSH because its component microcontroller, the Atmel AVR, is a patented design and can only be made and sold by its owner.

Shallow OSH. This intermediate view of OSH is that openness refers to a design, which may be built from non-open component designs and tools if and only if the *interface* to these designs and tools is non-patented. This enables multiple suppliers to produce competing alternatives which can be swapped in for these components, including Deep or other OSH alternatives. International standards exist to define many such open interfaces, for example ISO and EU standards. This gives some protection to the life of the new design, making it resilient to suppliers of components going out of business, stopping production, or overcharging for patents. OSE recommends the use of this approach at least as a stopgap measure until Deep OSH alternatives become available. Deep and shallow are thus ends of a continuum, in which the depth of an OSH design can increase as more components are swapped for deeper OSH designs, until a fully deep design is reached.

The podcar is currently only Weak OSH, like the Arduino, because it depends on the use of a patented donor vehicle. It is intended as a basis for future work which could replace the patented components with open interfaces to progress it to shallow and the deeper OSH.

B. 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.

C. Donor vehicle base

A Pihsiang TE-889XLSN hard-canopy scooter (branded in UK as Shoprider Traverso, [13]) 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 ($10k\Omega$) to choose speed mode high or low; A speed dial knob via a variable resistor ($20k\Omega$) to choose a maximum speed value; A throttle lever connected with a potentiometer ($5k\Omega$), $2.5k\Omega$ to $2.6k\Omega$ for each side to speed or brake.

D. Open source software base

ROS, is ‘an open source operating system for robots based on a publish-subscribe pattern [10], which is the robotics communitys standard interface. Building on top of ROS, the ‘ROS ecosystem is a loosely-defined collection of open-source ROS packages with variously accepted standard message formats which act as conventional interfaces to enable many community packages to work together and to act as swappable alternatives for one another.

ROS drivers are packages in the ecosystem which wrap robot hardware with standard ROS interfaces.

ROS launch files are stored configurations of ROS nodes that can be run using one command from a terminal. These are used widely by ROS packages to quickly launch the nodes required for a certain task at the same time.

tf is a ROS ecosystem library for 3D transforms. Transformas are a substantial part of ROS that allows for sensor input and other systems that require spatial frames to be contextualised by the sensors relative position within the robot. These are published on the tf topic, as explained by Quigley et al (2009), and create a dynamic transformation tree which relates all frames of reference in the system .

Gazebo [6] is a robotics 3D physical simulation and 3D graphics display tool, separate from, but designed to integrate with ROS, and which is the robotics community standard simulator. An example of one such project that makes use of Gazebo in conjunction with ROS is reported by Qian et al 2014) which presents an example of a project in which the hardware and software are developed in parallel, making use of ROS (for providing the software interface) and Gazebo (for testing hardware that doesnt yet exist in the real world).

E. Path planning and control

Path planning means the selection of an entire desired trajectory for a robot to get from a current pose to a desired pose, where a pose is a position together with a heading direction. Path planning may be split into high level and local planning, where high level makes large scale decisions such as which way to go around obstacles, while local planning refines these choices into specific curves with optimal properties. Path control (or path following) is then the real-time process of executing a path plan by interactively monitoring the robots state, sending commands to motors, and trying to make the actualized path as close to the desired path as possible.

Movebase (wiki.ros.org/move_base) is a ROS community standard which defines a set of message formats and a framework of ROS nodes for path planning and control. This framework allows different algorithms to plug in to it. Timed Elastic Band (TEB) [12] is a local planner algorithm which has been integrated as a movebase plugin (wiki.ros.org/teb_local_planner).

Planning and control for car-like vehicles with a steering column is more complex than for differential drive (skid steer) vehicles which turn by having the wheels on one side rotate faster than the other. This is because differential drive vehicles are holonomic, meaning that in the absence of obstacles, they can move greedily from any pose to any other, while steered vehicles and non-holonomic, meaning that greedy motion is not guaranteed to reach a target pose and longer-term plans must be considered. The shortest possible path for a forward-driving steered vehicle is the Dubins path [3], which always consists of two circle arcs connected by a straight line, which is generalized to the Reeds-Shepp path when reversing is also allowed [11]. Parallel parking is a classic difficult path planning task for Reeds-Shepp vehicles.

F. Simultaneous Localisation and Mapping

Simultaneous Localisation and Mapping (SLAM) [14] is the robotic task of inferring the robots location at the same time as building a map of its environment, which is a classic ‘chicken and egg’ problem as the two subtasks depend on one another. Solving SLAM is an NP-hard problem but many standard approximations exist. gmapping [15] is a ROS implementation of a Rao-Blackwellized Particle Filter (RBPF) for 2D SLAM, in which each particle carries an individual map of the environment. The information carried by each particle overlaps, and an estimation of a map can be built based on these relationships. As the robot moves around the environment, these estimations are stored, and when a feedback loop is closed, the estimations cascade into a portion of the completed map. These maps take the form of 2-D occupancy grids, and can be used later by the navigation stack to plan paths around the environment.

Performing SLAM in 3D is more computationally challenging, but has recently become possible due to hardware acceleration provided by parallel GPU computers. OrbSLAM is a ROS 3D SLAM based on the Octomap algorithm, which can operate with mono visual features, or lidar or stereo camera point clouds. It maintains a voxel based map of the environment (similar to maps in the game Minecraft) and is a 3D analog of gmapping.

NDT SLAM (used in Autoware and ILIAD) is another 3D SLAM implementation and algorithm based on Normal Distribution Transforms [4].

G. Related systems

SMART is a design to modify an existing donor golf cart vehicle for automation research, developed by MIT and Singapore. The vehicle is a similar size and power to podcar but is not OSH. Beetlebot [7] is a weak OSH system similar to podcar but for last-mile freight delivery, without human

carrying capacity, and including ROS integration. Open Source Ecology (OSE) is an ambition project which ultimately aims to develop Deep OSH vehicles including a car and tractor, via a process of progressively deeper Shallow OSH designs. OSE is optimised for reliability and for users in developing countries so uses hydraulic power rather than electric as used in podcar. Several functioning Weak OSH designs for large cars exist, including BARC (www.barc-project.com/), OSVehicle Tabby (www.openmotors.co/tabbyevo/). Of these, only BARC includes ROS integration. It is difficult to classify what constitutes a Weak OSH design because, for example, any small modification made to an existing donor vehicle and sufficiently document – such as by a simple tutorial video on how to update a standard cars music system, or the conversion of donor cars into racing vehicles – technically falls under the Weak OSH definition. Autoware [5] is a heavyweight open source software project to construct a ROS based automation stack for large on-road cars with power on-board computers. OSH and OSS are inherently cooperative design processes and legal structures, so it is possible that components from all of the above and from podcar will be interchanged and evolve together in the future.

II. MECHANICAL MODIFICATION

A. Steering

To automate steering, a Gimson GLA750-P 12V DC linear actuator with position feedback is mounted between an anchor 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.

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 metric 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.

III. ELECTRONICS AND FIRMWARE

New vehicle electronics are added on an acrylic board mounted under the seat of the vehicle, as shown in fig. 6



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

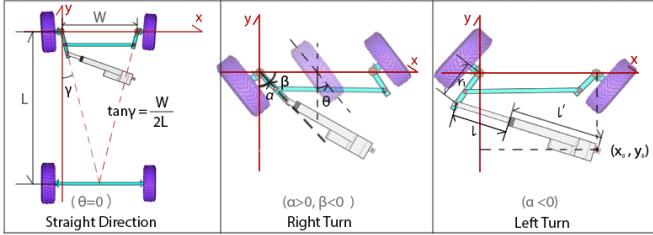


Fig. 3: The bottom view of front wheels steering relationship including geometric coefficients

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.

A. Steering control system

The front wheels are steered by a Pololu JRK21v3 PID controller, which takes serial port desired 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



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

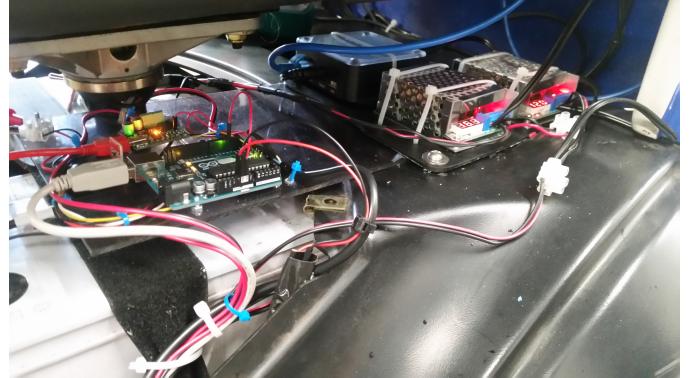


Fig. 5: Below-seat area inside cabin, with newly added boards..

actuator is,

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

$$\beta = \alpha - \frac{\pi}{2} \quad (2)$$

$$x = r_1 * \cos(\beta) \quad (3)$$

$$y = r_1 * \sin(\beta) \quad (4)$$

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

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 Pololu 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.

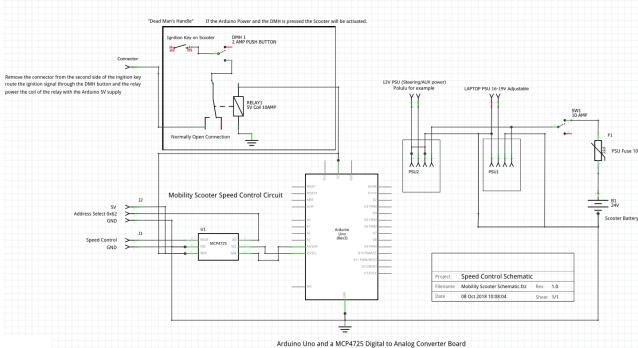


Fig. 6: Circuit diagram for electronic modifications.

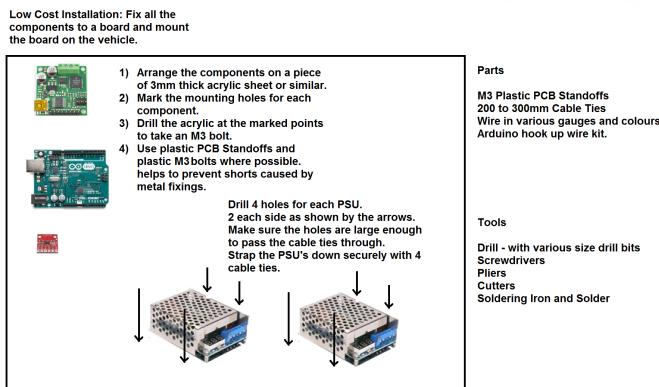


Fig. 7: Drilling.

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) *Safety system:* 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. Given the development platform nature of the OpenPodCar, a safety mechanism which stops the vehicle under fault conditions is an especially important part of the design. The design thus requires a series of positive heartbeat signals to be present and timely and also a dead-mans handle (DMH) to be depressed by a human experimenter at all times, in order for a hardware relay to actively continue to supply power from the vehicles batteries to all other systems. The relay will naturally cut out if these signals are absent for any reason, including failures in the safety systems themselves.

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.

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 vehicle's 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?

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

TODO how to set the console switches to correct fixed settings.

2) *Speed controller:* An Arduino UNO [8] 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. 6, and used to send clean analog speed command voltages to the donor vehicles internal controller.



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

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 serial 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 doesn't 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 vehicle's 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 the vehicle not starting - maybe by design or accident. (This produces a beep as is outside the start zone.) The Arduino 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, as in . 9. We provide a BV command in the Arduino serial protocol which allows callers to request the current battery voltage. This can then be used by the higher level (Python) systems to decide what speed bytes to send, including compensating for the floating dead zone.

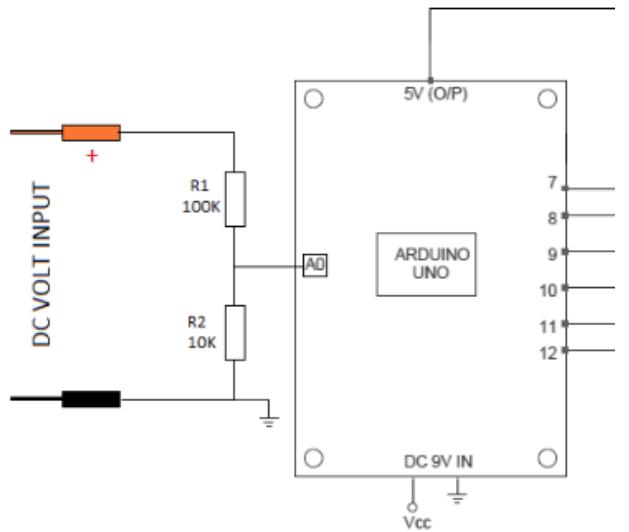


Fig. 9: Potential divider linked to the battery

IV. PHYSICAL VEHICLE SOFTWARE INTERFACE (ROS)

A ROS interface to and from the physical vehicle is provided as illustrated in fig. 12 and described below. This runs on a Panasonic CF-19 Toughbook computer mounted on-board the vehicle (and powered from a DCDC converter from the vehicle battery) running Xubuntu 16.04 (Xenial) and ROS Kinetic.

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 Polulu 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 a 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.

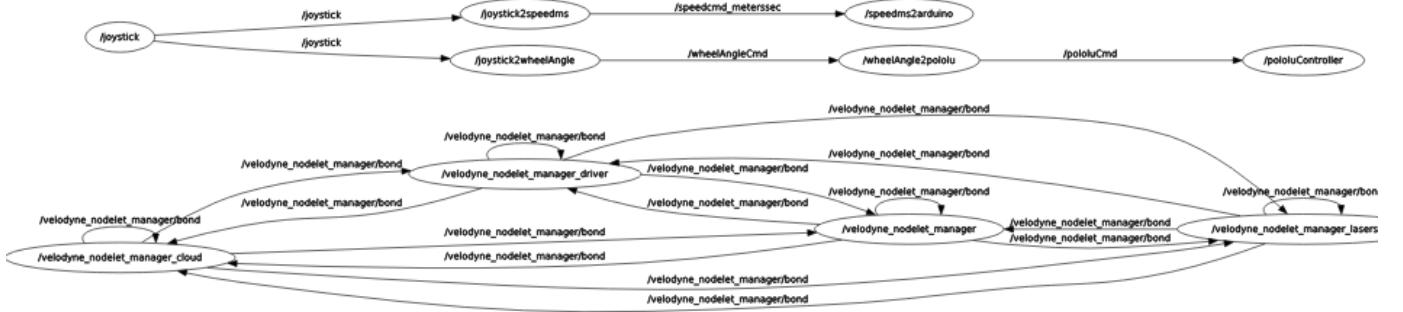


Fig. 10: ROS nodes and messages for low level control.

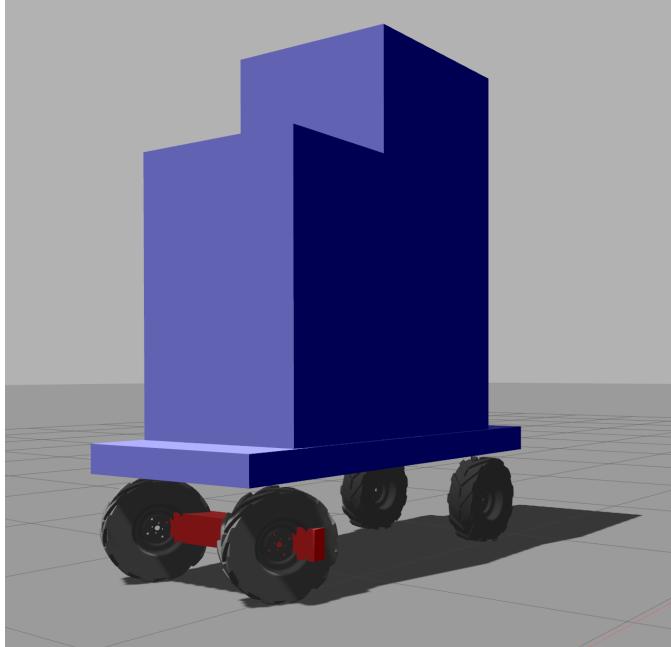


Fig. 11: Physical simulation of vehicle.

D. Depth camera option

TODO Dom Li is working on this...

V. SIMULATION

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

The physics simulation is based on a simplified vehicle geometry with two large cuboids containing the vehicles mass. 11 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.



Fig. 12: Visual mesh simulation of vehicle.

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 Gazebo's 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 12. 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 shape of the vehicle.

A basic 3D world containing the podcar and various test objects from Gazebo libraries is provided by default as shown in Ffg. 14.

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

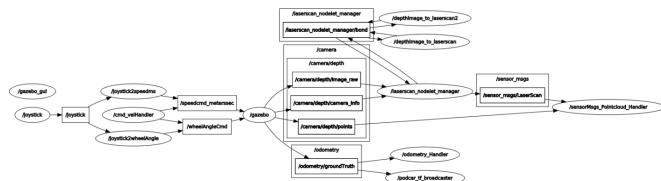


Fig. 13: ROS nodes used in simulation under manual joystick control.

VI. 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

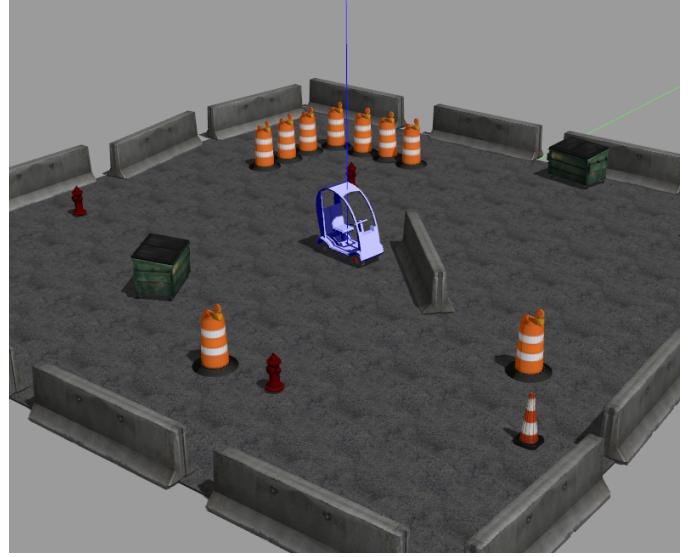


Fig. 14: Gazebo simulation world

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. Dubins 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 aren't 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

- [10] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [11] James Reeds and Lawrence Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific journal of mathematics*, 145(2):367–393, 1990.
- [12] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Efficient trajectory optimization using a sparse model. In *2013 European Conference on Mobile Robots*, pages 138–143. IEEE, 2013.
- [13] Shoprider. Shoprider flagship luxury scooter model: Te-889xsln user manual. *Manual*, 2016.
- [14] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.
- [15] Barry Loh Tze Yuen, Khairul Salleh Mohamed Sahari, and Zubaidi Faiesal Mohamad Rafaai. Improved map generation by addition of gaussian noise for indoor slam using ros. *Journal of Robotics, Networking and Artificial Life*, 4(2):118–123, 2017.

APPENDIX A: USER GUIDE

A. Vehicle

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: roscl podcar

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

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

Type: ros-launch podcar podcar.launch

Use the joystick to control steering and speed for manual drive.

B. Simulation

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
ros-launch podcar podcarsim.launch
```

C. Automation

TODO should be exactly the same as simulation but connect to real vehicle.

APPENDIX B: TROUBLESHOOTING GUIDE

D. 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 the battery is nearly flat.

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

E. 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.

F. 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.