mvsim

Nov 25, 2022

Contents

CHAPTER 1

Features

- Lightweight in memory, CPU and library requirements.
- Fully configurable via .xml "world" files.
- World maps:
  - Occupancy gridmaps: input as images or MRPT binary maps (from icp-slam, rbpf-slam, etc.)
  - Elevation meshes.
- Vehicle models:
  - Differential driven (2 & 4 wheel drive).
  - Ackermann steering (kinematic & dynamic steering, different mechanical drive models).
- Sensors:
  - 2D lidar scanners: Robots see each other, their own bodies, etc.
- Interface to vehicles: Choose among:
  - Raw access to forces and motor torques.
  - Twist commands (using internal controllers).

CHAPTER 2

Installing

2.1 From ROS1 or ROS2 repositories

This is preferred for users to quickly and easily install MVSIM and all the example files.

Once you have activated your ROS environment (*setup.bash*), just run:

```
sudo apt install ros-$ROS_DISTRO-mvsim
```

See next steps on how to launch demo files.

Note: You can also build MVSIM ROS nodes from sources, by cloning into a catkin or colcon workspace and build as usual (*catkin build* or *colcon build*).

2.2 Build from sources

Dependencies:

- A decent C++17 compiler.
- CMake >= 3.9
- MRPT (>=2.0.0): In Windows, build from sources or install precompiled binaries.
- Box2D (>=2.4): It will use an embedded copy (git submodule) if no (or too old) system version is found.

In Ubuntu, this will install all requirements:

```
sudo apt install \
  build-essential cmake g++ \
  libbox2d-dev \
  libmrpt-opengl-dev libmrpt-obs-dev libmrpt-maps-dev libmrpt-tclap-dev \
```

(continues on next page)

(continued from previous page)

```
  libmrpt-gui-dev libmrpt-tfest-dev \
  protobuf-compiler \
  libzmq3-dev \
  pybind11-dev \
  libprotobuf-dev \
  libpython3-dev
```

Compile as usual:

```
mkdir build
cd build
cmake ..
make
#make test
```

CHAPTER 3

First steps

After installing or building from sources, you are ready to test the standalone simulator applications to get used to MVSIM.

3.1 1. Manual vehicle teleop

```
mvsim launch mvsim_tutorial/mvsim_demo_2robots.world.xml
```

You should see the GUI of a demo world with two robots equipped with a 2D lidar, scanning a model defined by means of an occupancy grid map and a couple of boxes.

Use the keyboard to teleop the vehicle: `w/s` to increase/decrease the PI controller setpoint linear speed, and `a/d` to turn the Ackermann steering to the left/right. Use the spacebar as a brake.

Select the active robot by pressing the keys `1` or `2`.

3.2 2. ROS launch files

Once the ROS1 or ROS2 mvsim package is installed, you should be able to launch one of the many demo files, for example:

```
# For ROS1:
roslaunch mvsim mvsim_demo_jackal.launch

# For ROS2:
ros2 launch mvsim mvsim_demo_jackal.launch.py
```

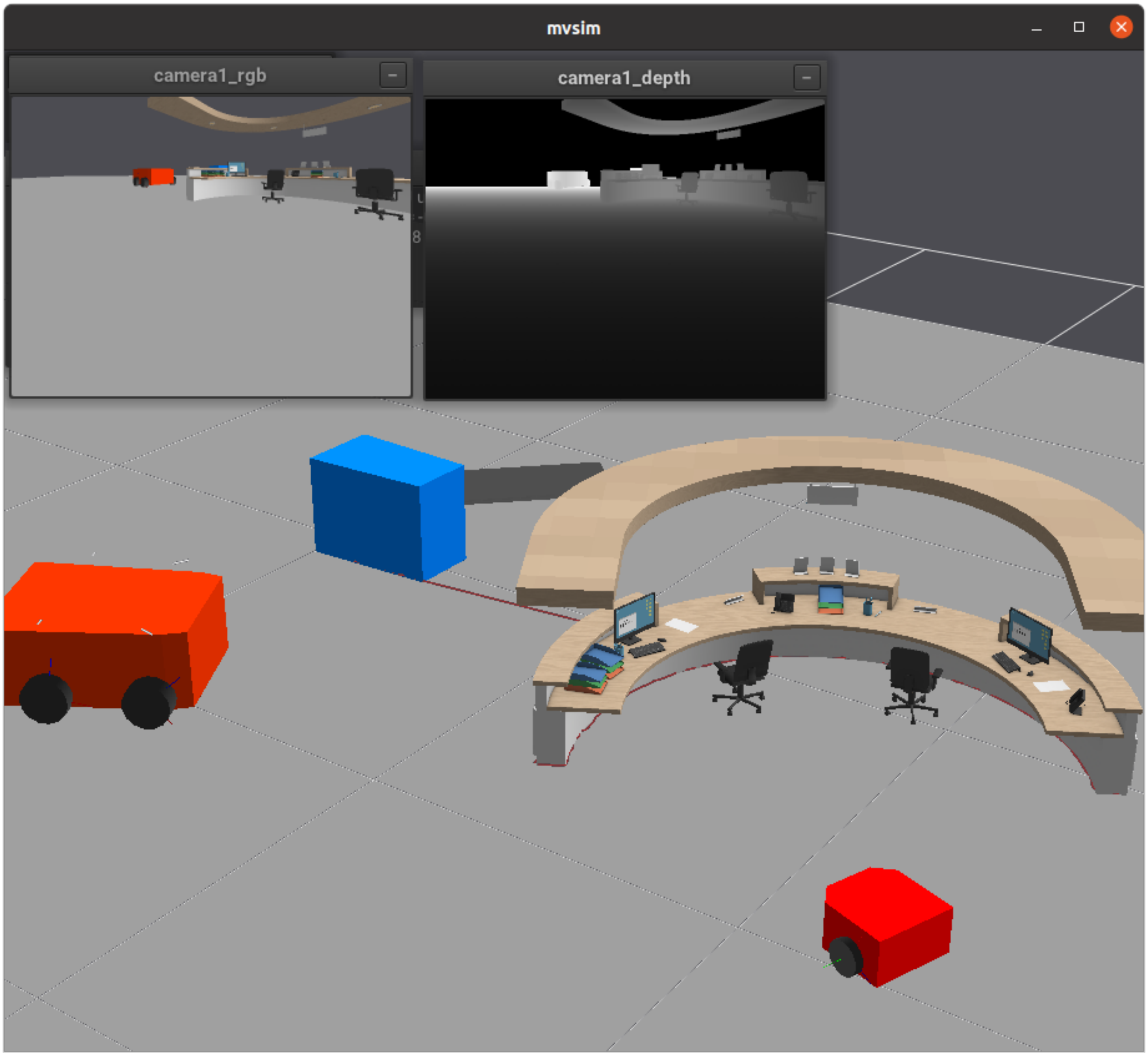


Fig. 1: Screenshot of the `mvsim_demo_2robots` example world.

Fig. 2: Screenshot of the `mvsim_demo_jackal` ROS demo.

CHAPTER 4

Architecture

The project comprises:

- A C++ library: `libmvsim`
- A ROS1 node. It can be run standalone.
- `mvsim-server`: A standalone program to run the simulation and, optionally, displaying a GUI live view of the world, accept keyboard/mouse orders, etc. It also uses ZMQ+protobuf as a communication system for user programs to interact with the simulation (for example, from a C++ or Python program).

CHAPTER 5

Simulated world definition

Simulation happens inside a World object. This is the central class for usage from user code, running the simulation, loading XML models, managing GUI visualization, etc. The ROS node acts as a bridge between this class and the ROS subsystem.

Simulated worlds are described via configuration files called "world" files, defined in the XML file format.

Many examples can be found in the mvsim_tutorial directory.

The next sections explain possible XML elements in a world file.

5.1 1. Global XML tags

World definition begins with tag **<mvsim_world>**. To define simulation timestep, use **<simul_timestep>** with *float* value specified in seconds.

```
<mvsim_world version="1.0">
...
        <!-- General simulation options -->
        <simul_timestep>0</simul_timestep> <!-- Simulation fixed-time interval for␣
↪numerical integration [s], or 0 to auto-determine -->
...
</mvsim_world>
```

5.2 2. GUI options

GUI options are specified with tag *gui*. *gui* has several nested tags:

```
<mvsim_world version="1.0">
...
        <!-- GUI options -->
```

(continues on next page)

(continued from previous page)

```
        <gui>
                <!-- Is camera orthographic or projective? -->
                <ortho>false</ortho>

                <!-- Show reaction forces on wheels with lines -->
                <show_forces>true</show_forces>
                <force_scale>0.01</force_scale> <!-- (Newtons to meters draw scale) --
↪>

                <!-- default camera distance in world units -->
                <cam_distance>35</cam_distance>

                <!-- camera vertical field of view in degrees -->
                <fov_deg>35</fov_deg>

                <!-- <follow_vehicle>r1</follow_vehicle> -->
        </gui>
...
</mvsim_world>
```

5.3 3. "World elements"

Scenario defines the "level" where the simulation takes place.

<element class="occupancy_grid"> depicts MRPT occupancy map which can be specified with both image file (black and while) and MRPT grid maps. **<file>** specifies file path to image of the map.

<element class="ground_grid"> is the metric grid for visual reference.

<element class="elevation_map"> is an elevation map (!experimental). Mesh-based map is build of elevation map in simple bitmap where whiter means higher and darker - lower.

This tag has several subtags:

- **<elevation_image>** - path the elevation bitmap itself
- **<elevation_image_min_z>** - minimum height in world units
- **<elevation_image_max_z>** - maximum height in world units
- **<texture_image>** - path texture image for elevation bitmap. Mus not be used with *mesh_color* simultaneously
- **<mesh_color>** - mesh color in HEX RGB format
- **<resolution>** - mesh XY scale

5.4 4. Vehicle class descriptions

Tag **<vehicle:class>** depictd description of vehicle class. The attribute *name* will be later referenced when describing vehicle exemplars.

Inside **<vehicle:class>** tag, there are tags **<dynamics>**, **<friction>** and exemplars of **<sensor>**.

5.4.1 Vehicle dynamics

At the moment, there are three types of vehicle dynamics implemented. Refer [vehicle_models] for more information.

<dynamics> with attribute *class* specifies class of dynamics used. Currently available classes:

- differential
- car_ackermann
- ackermann_drivetrain

Each class has specific inner tags structure for its own configuration.

5.4.2 Common

Every dynamics has wheels specified with tags **<i_wheel>** where i stand for wheel position index (r, l for differential drive and fr, fl, rl, rr for Ackermann-drive)

Wheel tags have following attributes:

- *pos* - two floats representing x an y coordinate of the wheel in local frame
- *mass* - float value for mass of the wheel
- *width* - float value representing wheel width [fig:wheel_forces]
- *diameter* - float value to represent wheel diameter [fig:wheel_forces]

Ackermann models also use **<max_steer_ang_deg>** to specify maximum steering angle.

<chassis> is also common for all dynamics, it has attributes:

- *mass* - mass of chassis
- *zmin* - distance from bottom of the robot to ground
- *zmax* - distance from top of the robot to ground

5.4.3 Controllers

There are controllers for every dynamics type [sec:controllers]. In XML their names are

- raw - control raw forces
- twist_pid - control with twist messages
- front_steer_pid - [Ackermann only] - control with PID for velocity and raw steering angles

Controllers with *pid* in their names use PID regulator which needs to be configured. There are tags **<KP><KI><KD>** for this purpose. Also they need the parameter **<max_torque>** to be set.

Twist controllers need to set initial **<V>** and **<W>** for linear and angular velocities respectively.

Steer controllers need to set initial **<V>** and **<STEER_ANG>** for linear velocity and steering angle respectively.

5.4.4 Ackermann-drivetrain model

needs a differential type and split to be configured. For this purpose there is a tag **<drivetrain>** with argument *type*. Supported types are defined in [sec:ackermann_drivetrain]. In XML their names are:

- open_front

- open_rear
- open_4wd
- torsen_front
- torsen_rear
- torsen_4wd

<drivetrain> has inner tags describing its internal structure:

- **<front_rear_split>**
- **<front_rear_bias>**
- **<front_left_right_split>**
- **<front_left_right_bias>**
- **<rear_left_right_split>**
- **<rear_left_right_bias>**

which are pretty self-explanatory.

5.4.5 Friction

Friction models are described in [sec:friction_models] and defined outside of **<dynamics>**. The tag for friction is **<friction>** with attribute *class*.

Class names in XML are:

- wardiagnemma
- default

Default friction [sec:default_friction] uses subtags:

- **<mu>** - the friction coefficient
- **<C_damping>** - damping coefficient

In addition to **default**, **Ward-Iagnemma** friction includes subtags:

- **A_roll**
- **R1**
- **R2**

that are described in [sec:wi_friction].

5.4.6 Sensors

Sensors are defined with **<sensor>** tag. It has attributes *type* and *name*.

At the moment, only laser scanner sensor is implemented, its type is *laser*. Subtags are:

- **<pose>** - an MRPT CPose3D string value
- **<fov_degrees>** - FOV of the laser scanner
- **<sensor_period>** - period in seconds when sensor sends updates
- **<nrays>** - laser scanner rays per FOV

- **<range_std_noise>** - standard deviation of noise in distance measurements
- **<angle_std_noise_deg>** - standatd deviation of noise in angles of rays
- **<bodies_visible>** - boolean flag to see other robots or not

5.5 5. Vehicle instances

For each vehicle **class**, an arbitrary number of vehicle **instances** can be created in a given world.

Vehicle instances are defined with the **<vehicle>** tag that has attributes *name* and *class*. *class* must match one of the classes defined earlier with **<vehicle:class>** tag.

Subtags are:

- **<init_pose>** - in global coordinates: x, y, γ (deg)
- **<init_vel>** - in local coordinates: v_x,v_y, ω (deg/s)

5.6 6. "Obstacle block" classes

Write me!

5.7 7. "Obstacle block" instances

Write me!

5.8 8. Vehicles and blocks parameters

Vehicles and obstacles blocks share common C++ `mvsim::Simulable` and `mvsim::VisualObject` interfaces that provide the common parameters below.

Note: The following parameters can appear in either the {vehicle,block} class definitions or in a particular instantiation block, depending on whether you want parameters to be common to all instances or not, respectively.

5.8.1 Related to topic publication

Under the `<publish> </publish>` tag group:

- **publish_pose_topic**: If provided, the pose of this object will be published as a topic with message type `mvsim_msgs::Pose`.
- **publish_pose_period**: Period (in seconds) for the topic publication.

Example:

```
<publish>
  <publish_pose_topic>/r1/pose</publish_pose_topic>
  <publish_pose_period>50e-3</publish_pose_period>
</publish>
```

5.8.2 Related to visual aspect

Under the `<visual> </visual>` tag group:

- **model_uri**: Path to 3D model file. Can be any file format supported by ASSIMP, like `.dae`, `.stl`, etc. If empty, the default visual aspect will be used.
- **model_scale**: (Default=1.0) Scale to apply to the 3D model.
- **model_offset_x**, **model_offset_y** , **model_offset_z**: (Default=0) Offset translation [meters].
- **model_yaw**, **model_pitch**, **model_roll**: (Default=0) Optional model rotation [degrees].
- **show_bounding_box**: (Default=‘‘false‘‘) Initial visibility of the object bounding box.

Example:

```xml
<visual>
  <model_uri>robot.obj</model_uri>
  <model_scale>1.0</model_scale>
  <model_offset_x>0.0</model_offset_x>
  <model_offset_y>0.0</model_offset_y>
  <model_offset_z>0.0</model_offset_z>
</visual>
```

CHAPTER 6

Simulation execution

Simulation executes step-by-step with user-defined Δt time between steps. Each step has several sub steps:

- Before time step - sets actions, updates models, etc.
- Actual time step - updates dynamics
- After time step - everything needed to be done with updated state

6.1 Logging

Each vehicle is equipped with parameters logger(s). This logger is not configurable and can be rewritten programmaticaly.

Logger are implemented via **CsvLogger** class and make log files in CSV format which then can be opened via any editor or viewer.

Loggers control is introduced via robot controllers, each controller controls only loggers of its robot.

Best results in visualizing offers QtiPlot [fig:qtiplot_example1].

At the moment, following characteristics are logged:

- Pose $(x, y, z, \alpha, \beta, \gamma)$
- Body velocity $(\dot{x}, \dot{z}, \dot{z})$
- Wheel torque (τ)
- Wheel weight (m_{wp})
- Wheel velocity (v_x, v_y)

Loggers support runtime clear and creating new session. The new session mode finalizes current log files and starts to write to a new bunch of them.

6.2 Limitations

- A limitation of box2d is that no element can be thinner than 0.05 units, or the following assert will be raised while loading the world model:

CHAPTER 7

Physic models

7.1 1. Wheel dynamics

We introduce wheels as a mass with cylindrical shape (Figure [fig:wheel_forces]). Each wheel has following properties:

- location of the wheel as to the chassis ref point [m,rad] in local coordinates $L_w = \{x_w, y_w, \Phi\}$
- diameter d_w [m]
- width w_w [m]
- mass m_w [kg]
- inertia I_{yy}
- spinning angular position ϕ_w [rad]
- spinning angular velocity ω_w [rad/s]

Thus, each wheel is represented as $W = \{L_w, d_w, w_w, m_w, I_{yy}, \phi_w, \omega_w\}$

Fig. 1: Wheel forces

7.2 2. Friction models

7.2.1 Friction models base

Friction model base introduces *Friction input* structure, that incorporates forces of wheel

- weight on this wheel from the car chassis, excluding the weight of the wheel itself w [N]
- motor torque τ [Nm]

- instantaneous velocity

$$\nu = \begin{bmatrix} \nu_x \\ \nu_y \end{bmatrix}$$

in local coordinate frame

7.2.2 Default friction

At the moment, there is only one basic friction model available for vehicles. Default friction model evaluates …

Default friction evaluates forces in the wheel coordinate frame:

$$\nu_w = \begin{bmatrix} \nu_{wx} \\ \nu_{wy} \end{bmatrix} = R(\Phi_w) \cdot \nu$$

To calculate maximal allowed friction for the wheel, we introduce partial mass:

$$m_{wp} = \frac{w_w}{g} + m_w$$

$$F_{f,max} = \mu \cdot m_{wp} \cdot g$$

Where μ is friction coefficient for wheel.

Calculating latitudinal friction (decoupled sub-problem):

$$F_{f,lat} = m_{wp} \cdot a = m_{wp} \cdot \frac{-\nu_{wy}}{\Delta t}$$

$$F_{f,lat} = max(-F_{f,max}, min(F_{f,lat}, F_{f,max}))$$

Calculating wheel desired angular velocity:

$$\omega_{constraint} = \frac{2\nu_{wx}}{d_w}$$

$$J_{desired} = \omega_{constraint} - \omega_w$$

$$\omega_{desired} = \frac{J_{desired}}{\Delta t}$$

Calculating longitudinal friction:

$$F_{f,lon} = \frac{1}{R} \cdot (\tau - I_{yy} \cdot \omega_{desired} - C_{damp} \cdot \omega_w)$$

$$F_{f,lon} = max(-F_{f,max}, min(F_{f,lon}, F_{f,max}))$$

Simply composing friction forces to vector:

$$F_f = \begin{bmatrix} F_{f,lat} \\ F_{f,lon} \end{bmatrix}$$

With new friction, we evaluate angular acceleration *(code says angular velocity impulse, but the units are for acceleration)* of the wheel:

$$\alpha = \frac{\tau - R \cdot F_{f,lon} - C_{damp} \cdot \omega_w}{I_{yy}}$$

Using given angular acceleration, we update wheel's angular velocity:

$$\omega_w = \omega_w + \alpha \cdot \Delta t$$

7.2.3 Ward-Iagnemma friction

This type of friction is an implementation of paper from Chris Ward and Karl Iagnemma **:raw-latex:‘\cite{ward-iagnemma-friction}‘**.

Rolling resistance is generally modeled as a combination of static- and velocity-dependant forces [17], [21]. Authors propose function with form similar to Pacejka’s model **:raw-latex:‘\cite{pacejka_tire_model}‘** as a continuously differentiable formulation of the rolling resistance with the static force smoothed at zero velocity to avoid a singularity. The rolling resistance is

$$F_{rr} = sign(V_{fwd}) \cdot N \cdot (R_1 \cdot (1e^{A_{roll}|V_{fwd}|}) + R_2 \cdot |V_{fwd}|)$$

Where A_{roll}, R_1, R_2 are the model-dependent coefficients. The impact of these coefficients is shown at figure [fig:wi_rr] taken from original paper.

This force F_{rr} is then added to $F_{f,lon}$.

Default constants were chosen as in reference paper and showed good stability and robust results. In addition, they can be altered via configuration file.

7.3 3. Vehicle models

Vehicle models are fully configurable with world XML files.

7.3.1 Vehicle base class

Vehicle base incorporates basic functions for every vehicle actor in the scene. It is also responsible for updating state of vehicles.

It has implementation of interaction with world. Derived classes re-implement only work with torques/forces on wheels.

At the moment, no model takes into account the weight transfer, so weight on wheels is calculated in this base class.

Vehicle base class also provides ground-truth for velocity and position.

$$p_w = \frac{p_{chassis}}{N_w}$$

- Before time step:
  - Update wheels position using Box2D
  - Invoke motor controllers (reimplemented in derived classes)
  - Evaluate friction of wheels with passed friction model
  - Apply force to vehicle body using Box2D
- Time step - update internal vehicle state variables q and $\dot{q}$
- After time step - updates wheels rotation

Center of mass is defined as center of Box2D shape, currently there is no +Z mobility.

7.3.2 Differential driven

A differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels and hence does not require an additional steering motion.

Odometry-based velocity estimation is implemented via Euler formula (consider revising, it doesn't include side slip):

$$\omega_{veh} = \frac{\omega_r \cdot R_r - \omega_l \cdot R_r}{y_r - y_l}$$

$$\nu_x = \omega_l \cdot R_l + \omega \cdot y_l$$

$$\nu_y = 0$$
$$Where: math: `\omega_{veh}`isangularvelocityoftherobot,$$

R_i - radius of the wheel, y_i is the y position of the wheel, ω_i is the angular velocity of the wheel. All calculations in the robot's local frame.

Nothing more interesting here.

7.3.3 Ackermann driven

Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or other vehicle designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

Ackermann wheels' angles are computed as following:

$$\alpha_{outer} = atan(cot(|\alpha| + \frac{w}{2l})$$

$$\alpha_{inner} = atan(cot(|\alpha| - \frac{w}{2l})$$
$$where: math: `\alpha`isthedesiredequivalentsteeringangle,$$

w is wheels distance and l is wheels base. Outer and inner wheel are defined by the turn direction.

Odometry-based velocity estimation is implemented via Euler formula (consider revising, it doesn't include side slip):

$$\omega_{veh} = \frac{\omega_{rr} \cdot R_{rr} - \omega_{rl} \cdot R_{rr}}{y_{rr} - y_{rl}}$$

$$\nu_x = \omega_{rl} \cdot R_{rl} + \omega \cdot y_{rl}$$

$$\nu_y = 0$$
$$Where: math: `\omega_{veh}`isangularvelocityoftherobot,$$

R_{ri} - radius of the rear wheel, y_{ri} is the y position of the rear wheel, ω_{ri} is the angular velocity of the rear wheel. All calculations in the robot's local frame.

7.3.4 Ackermann-driven with drivetrain

This type of dynamics has the same geometry as simple Ackermann-driven robots. However, its powertrain is completely different.

Instead of one "motor" per wheel, this type of dynamics incorporates one "motor" linked to wheels by differentials.

There are two types of differentials:

- Open differntial
- Torsen-like locking differential **:raw-latex:‘\cite{torsen-whitepaper}‘**

Each type of differential can be linked with following configurations:

- Front drive
- Rear drive
- 4WD

Split is customizable between all axes.

As engine plays controller, whose torque output is then fed into differentials.

For open differential act the following equations:

$$\tau_{FL} = \tau_{motor} \cdot K_{s,f} \cdot K_{s,frl}$$

$$\tau_{FR} = \tau_{motor} \cdot K_{s,f} \cdot (1 - K_{s,frl})$$

$$\tau_{RL} = \tau_{motor} \cdot K_{s,r} \cdot K_{s,rrl}$$

$$\tau_{RR} = \tau_{motor} \cdot K_{s,r} \cdot (1 - K_{s,frl})$$

Where $K_{s,f}, K_{s,frl}, K_{s,rrl}$ are split coefficients between axes.

Different things happen for Torsen-like differentials. As this type is self-locking, its torque output per wheel depends on wheel’s velocity. Here is the function of selecting torque on the next time step based on previous time step velocity. First, introduce the bias ratio - the ratio indicating how much more torque the Torsen can send to the tire with more available traction, than is used by the tire with less traction. This ratio represents the “locking effect” of the differential. By default, it is set to $b = 1.5$

ω_1, ω_2 and t_1, t_2 are the output axles angular velocities and torque splits respectively. K_s is differential split when it is not locked.

$$\omega_{max} = max(|\omega_1|, |\omega_2|)$$

$$\omega_{min} = min(|\omega_1|, |\omega_2|)$$

$$\delta_{lock} = \omega_{max} - b \cdot \omega_{min}$$

$$\delta_t = \begin{cases} \delta_{lock} \cdot \omega_{max}, & \text{if } \delta_{lock} > 0 \\ 0, & \text{if } \delta_{lock} \le 0 \end{cases}$$

$$f_1 = \begin{cases} K_s \cdot (1 - \delta_t) & \text{if } |\omega_1| - |\omega_2| > 0 \\ K_s \cdot (1 + \delta_t) \end{cases}$$

$$f_2 = \begin{cases} (1 - K_s) \cdot (1 + \delta_t) & \text{if } |\omega_1| - |\omega_2| > 0 \\ (1 - K_s) \cdot (1 - \delta_t) \end{cases}$$

$$t_1 = \frac{f_1}{f_1 + f_2}$$

$$t_2 = \frac{f_2}{f_1 + f_2}$$

Torque delivery for 2WD is pretty straightforward. There is one input from “motor” and two outputs to wheels, so wheel torques are:

$$\tau_i = \tau_{motor} \cdot t_i$$

where t_i is the output of Torsen differential for i-th wheel.

With 4WD, torque is first split with Torsen to front and rear parts, each of them is than split independently with another Torsen.

At the moment, there is no model of the engine and thus no feedback of tires torque to engine.

7.4 4. Controllers

Different vehicles have different controllers. At the moment, differential and Ackermann drives have their own controllers.

Controllers are divided into several types:

- Raw forces
- Twist

Ackermann has controller, which controls steering angle and speed.

Controllers' input and output are described by dynamics' classes that they use.

7.4.1 Differential raw controller

This type of controller has simple response to user's input integrating wheel torque with each simulation frame.

7.4.2 Differential Twist controller

Differential twist controller uses PID regulator to control linear and angular speed of the robot.

Setpoints for v_r and v_l are calculated as following:

$$v_l = \nu - \frac{\omega}{2} \cdot w$$

$$v_r = \nu + \frac{\omega}{2} \cdot w$$

where ν is desired linear velocity and ω is desired angular velocity.

Inverted formula are suitable to get actual velocities from odometry estimates.

Then, velocity of the wheels is controlled with PID regulator.

7.4.3 Ackermann raw controller

As a raw differential controller, raw Ackermann controller integrates user input and sets wheel torques and steering wheel angle.

7.4.4 Ackermann twist controller

Ackermann twist controller uses PID regulator to control wheel torques responding to angular and linear velocity commands. Turn radius and desired steering angle are calculated:

$$R = \frac{\nu_s}{\omega_s}$$

$$\alpha = atan(\frac{w}{r})$$

Desired velocities for wheels are computed by rotating desired linear velocity to the steering angle. In the same way, actual velocities from "odometry" are computed.

Then, torque of separate wheels is controlled with PID regulators for each wheel.

7.4.5 Ackermann steering controller

Ackermann steering controller takes as input linear speed an steering angle.

Then, it executes Ackermann twist controller to control wheels' torques.

7.4.6 Ackermann-drivetrain controllers

These controllers' steering is identical to Ackermann contollers, however, their torque part is different.

These controllers' output acts like 'engine' for drivetrain. Instead of separate outputs to wheels, it has one torque output to differentials that will split it to separate wheels.

CHAPTER 8

Extensions

It is possible to extend the simulator with custom models for sensors, vehicles, and world elements.

Note: Write me!

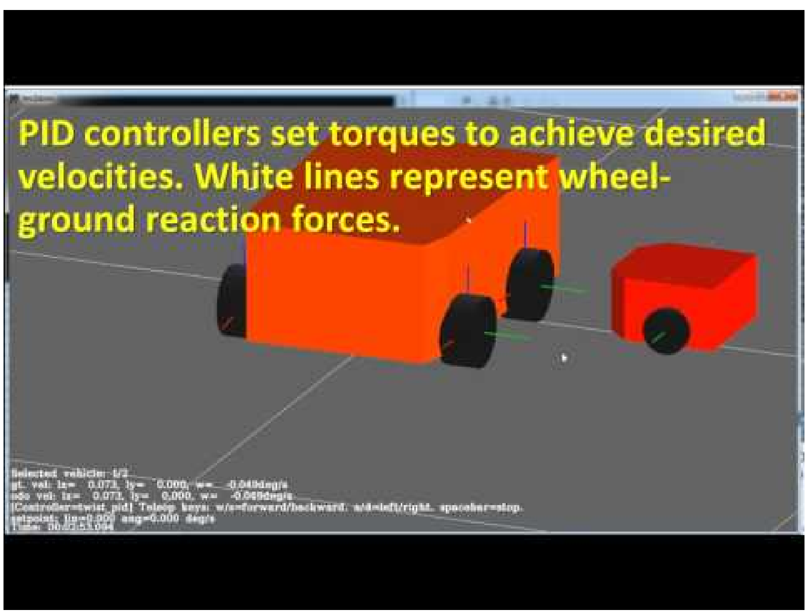
CHAPTER 9

About

Lightweight, realistic dynamical simulator for 2D ("2.5D") vehicles and robots. It is tailored to analysis of vehicle dynamics, wheel-ground contact forces and accurate simulation of typical robot sensors (e.g. laser scanners).

This project includes a C++ library *mvsim*, a standalone app, and a ROS1 node.

Licensed under the permissive 3-clause BSD License.



Documentation credits:

- Borys Tymchenko @spsancti
- Jose Luis Blanco Claraco @jlblancoc

Github repository: https://github.com/MRPT/mvsim