Original software publication

MultiVehicle Simulator (MVSim): Lightweight dynamics simulator for multiagents and mobile robotics research

José-Luis Blanco-Claraco [a],[*], Borys Tymchenko [b], Francisco José Mañas-Alvarez [c], Fernando Cañadas-Aránega [d], Ángel López-Gázquez [d], José Carlos Moreno [d]

[a] *Engineering Department, University of Almería, Agrifood Campus of International Excellence (ceiA3), and CIESOL, 04120 Almería, Spain*
[b] *Deci AI, Tel Aviv, Israel*
[c] *Department of Computer Sciences and Automatic Control, Universidad Nacional de Educación a Distancia (UNED), Madrid, Spain*
[d] *Department of Informatics, University of Almería, Agrifood Campus of International Excellence (ceiA3), and CIESOL, 04120 Almería, Spain*

ARTICLE INFO

ABSTRACT

Development of applications related to closed-loop control requires either testing on the field or on a realistic simulator, with the latter being more convenient. To ease that need, this work introduces MVSim, a simulator for multiple vehicles or robots capable of running in real time dozens of agents in simple scenarios, or a handful of them in complex scenarios. MVSim employs realistic physics-grounded friction models for tire–ground interaction, and aims at accurate and GPU-accelerated simulation of most common modern sensors employed in mobile robotics and autonomous vehicle research, such as depth and RGB cameras, or 2D and 3D LiDAR scanners. All depth-related sensors are able to accurately measure distances to 3D models provided by the user to define custom world elements. Efficient simulation is achieved by means of focusing on ground vehicles, which allows the use of a simplified 2D physics engine for body collisions while solving wheel–ground interaction forces separately. The core parts of the system are written in C++ for maximum efficiency, while Python, ROS 1, and ROS 2 wrappers are also offered. A custom publish/subscribe protocol based on ZeroMQ (ZMQ) is defined to allow for multiprocess applications to access or modify a running simulation. This simulator enables and makes easier to do research and development on vehicular dynamics, autonomous navigation algorithms, and simultaneous localization and mapping (SLAM) methods. An experimental performance benchmarking is provided against other state-of-the-art simulators showing significant less CPU usage. The project source code is freely available online under the BSD 3-clause license in https://github.com/MRPT/mvsim.

Code metadata

|	
Current code version	v0.6.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00123
Code Ocean compute capsule	
Legal Code License	BSD 3-clause License
Code versioning system used	git
Software code languages, tools, and services used	C++, python
Compilation requirements, operating environments & dependencies	C++17 compiler, GNU/Linux/Windows/OSX, MRPT, Google Protobuf
If available Link to developer documentation/manual	https://github.com/MRPT/mvsim/blob/develop/README.md, https://mvsimulator.readthedocs.io/
Support email for questions	jlblanco@ual.es

1. Motivation and significance

Research and development in mobile robotics or autonomous vehicles have final goals related to physical prototypes, which leads to expensive and laborious field experiments to tune and
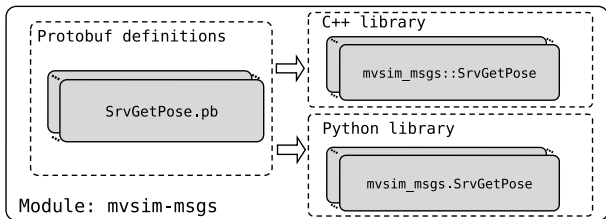
* Corresponding author.
E-mail address: jlblanco@ual.es (José-Luis Blanco-Claraco).

José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez et al.

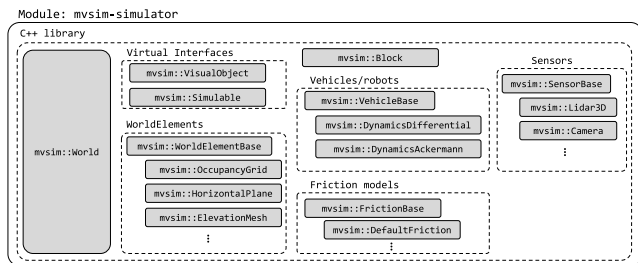SoftwareX 23 (2023) 101443

Fig. 1. Screenshot of the simulator running the warehouse example world. Refer to discussion in Section 3.
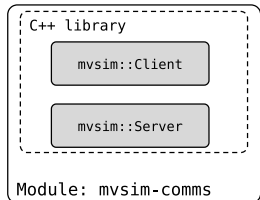
(a) Module: mvsim-comms

(b) Module: mvsim-msgs

(c) Module: mvsim-simulator

Fig. 2. Overview of the modules provided by MVSIM: (a) The communications (messages transport) module, (b) the messages definitions module, and (c) the simulation module.

debug new implementations. In the industry, this turns out into long development cycles, increasing the time-to-market of new products or services. In particular, the present work has a great potential in addressing three scientific problems: (i) simultaneous localization and mapping (SLAM), (ii) autonomous navigation, and (iii) steering control. SLAM is the problem of a mobile agent equipped with exteroceptive sensors to incrementally build a map of its environment as it moves around, autonomously or teleoperated [1,2]. Autonomous navigation is the problem of safely moving a vehicle or robot in its environment, typically from an initial pose to a final one. Typically, this large research field distinguishes between global path planners (find a path given a map) and local planners (trying to follow a predefined path, avoiding unexpected and dynamic obstacles) [3]. Finally, steering control and modeling of ground vehicles [4] address the issue of finding accurate mathematical models for the dynamics and kinematics of steering in such vehicles, and all related applications as state-observers [5] or control [6].

Table 1
Qualitative comparison of MVSim with other state-of-the-art simulators.

	Gazebo	Webots	MVSim
Creation	2011	1996	2014
Physics engine	DART	ODE	Hybrid[a]
3D engine	OGRE	Custom	MRPT
Interfaces	C++ Python ROS	C++ Python ROS MATLAB Java	C++ Python ROS
World files	SDF	VRML	XML
Robotics sensors	Yes (plugins)	Yes	Yes
Ground vehicles	Yes	Yes	Yes
UAVs	Yes	Yes	Not yet
Custom tire–ground force models	No	No	Yes
Approximated fast 2D lidar	No	No	Yes
License	Apache-2.0	Apache-2.0	BSD-3

[a]Box2D plus custom physics solvers.

Users can use MVSim in all major operating systems (GNU/Linux, OSX, Windows). The standalone executable `mvsim-cli` is provided as a central hub to most common operations, such as launching a simulation, or listing and inspecting published topics. A python library is also provided to allow users to easily communicate with a running simulation, from the same or a different machine. Finally, a node for ROS 1 and ROS 2 is provided to act as an intermediary between an MVSim simulation and other ROS nodes, using standardized APIs and conventions as `tf` frames [7] or sensor message types [8].

There exist other software projects with similarities to MVSim. The most notorious are: Player/Stage, Gazebo, and Webots. Player/Stage [9] is an open-source project consisting of a set of software tools used for multi-robot systems (MRS) control. Two main parts are considered: Player and Stage. Player is a device server providing a transparent access to the robot, and Stage is the simulator for moving and sensing multiple robots in a 2D indoor environment. Gazebo [10], developed to tackle complex environments, is an open-source and freely available MRS simulation backend as Stage, but operating in a 3D world and simulating physical interactions between objects. However, it lacks flexibility in the selection of friction models between vehicle tires and the ground, which may be specially important depending on the type of robot and world simulated. Webots [11], as Gazebo, is an open-source robot simulator capable of working with all kind of robots (UGV, automobiles, UAV, bipeds, manipulators...), and MRS, in indoor and outdoor environments, and incorporating physics for each component of the world. A brief comparison of the main actively-maintained robotics simulators is provided in Table 1.

2. Software description

Next, the overall architecture of the software is exposed in Section 2.1 together with some implementation details regarding the physics simulation. Later on, Section 2.3 provides a summary of the functionality offered by the proposal.

2.1. Software architecture

The project comprises (C++ and Pyhton) libraries and executables, with the formers split into three modules as shown in Fig. 2.
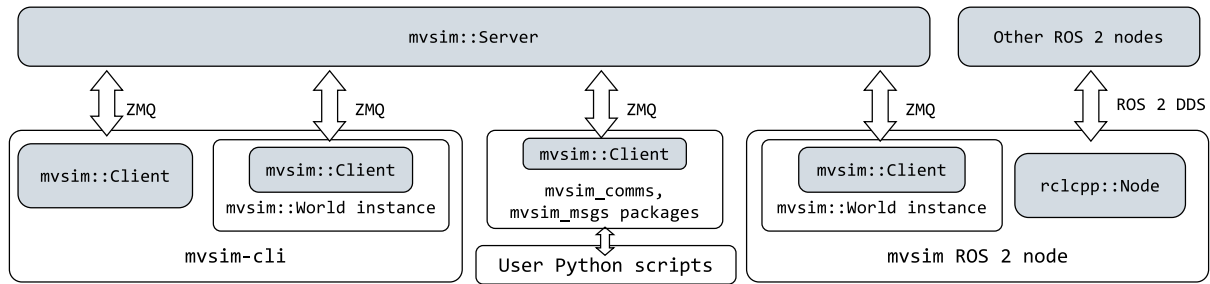
José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez et al.

SoftwareX 23 (2023) 101443

Fig. 3. High-level diagram of potential connections between different processes using MVSIM.

The `mvsim-comms` module is responsible of providing top-level client–server functionality for publish–subscribe and remote service invocation. Data transport is implemented by means of ZeroMQ sockets for their portability, robustness, and efficiency [12]. Data interchange is done by means of messages, defined in the `mvsim-msgs` module by an interface definition language (IDL), in particular, Google Protobuf. IDL definitions are compiled into C++ and a Python libraries. Finally, the `mvsim-simulator` module holds the main functionalities of the software: loading and parsing world definition files, running simulations, exposing a graphical interface to the user, and simulating and publishing sensors.

These libraries can then be used in different ways, as illustrated in Fig. 3. Firstly, the `mvsim-cli` program is provided as a central hub to most non programming-related functionality, such as: launching a simulation for a particular world definition file and exploring it via a GUI, listing and getting details on connected communication clients and advertised topics, or analyzing the rate at which a topic is being published. Note that invoking the ''mvsim-cli launch'' subcommand spans a communication server, to which the client within `mvsim::World` connects to. Another way to run a simulation (and spanning a server) is launching the provided ROS node. Note that, in that case, the MVSim simulation is accessible simultaneously via our custom Python library, and the standard ROS Pub/Sub mechanism. Headless execution of the simulation is also possible (including GPU-accelerated sensor simulation), enabling the usage of MVSim within containerized Continuous Integration (CI) pipelines.

Regarding physics simulation, efficiency is achieved through a hierarchical approach: on a first higher level, rigid objects (named "blocks" in MVSim and which may represent fixed or mobile obstacles) and vehicles (which include wheels as their unique mobile parts) are modeled as 2D entities using the Box2D C++ library [13]. For vehicles or objects with custom 3D meshes, MVSim evaluates their equivalent 2D collision polygon. The Box2D engine then provides collision detection and runs physics simulation on the basis of forces and torques applied to each rigid body, whose calculation is described below. Simulation runs forward in time by numerical integration on the state vector of all mobile objects with a fixed time step. Before running each such step, the Algorithm 1 is applied to account for the second (lower) layer of the hierarchical physics simulation. Next we provide details on the most relevant parts of this algorithm to explain the underlying mechanical model. Kinematics (Ackermann vs. differential-driven vehicles) and PID controllers to achieve closed-loop control of wheels velocity are already handled in `invoke_motor_controllers()` in Algorithm 1. Note that closed-loop velocity control is achieved taking simulated odometry readings as the controller input, thus wheels slippage will lead to wrong odometry estimations, accurately replicating the consequences of slippage on real vehicles.

2.2. Dynamics model

Each vehicle or robot (V) is modeled as a rigid body ("chassis") with a given mass and inertia tensor, whose SE(3) pose [14,15] with respect to a global world origin O can be defined as an homogeneous transformation matrix ${}^{O}\mathbf{T}_V$, comprising a translation vector ${}^{O}\mathbf{t}_V$ and a 3×3 rotation matrix ${}^{O}\mathbf{R}_V$. Note the use of 3D coordinates and orientation, SE(3), instead of the equivalent planar SE(2) poses, despite the use of an underlying 2D physics engine. That is because MVSim can provide certain degrees of freedom for vehicles to move up and down and tilt as they transverse digital terrain elevation models, for example.

Let $\mathbf{v}_V$ be the velocity vector of the vehicle V reference point (which may or may not coincide with the center of mass), and $\boldsymbol{\omega}_V$ the vehicle angular velocity, as illustrated in Fig. 4(a). Then, rigid body kinematics leads to the following expression for the velocity vector of a wheel (W) center point, expressed in global coordinates (O):

$$\mathbf{v}_W = \mathbf{v}_V + \boldsymbol{\omega}_V \times {}^{V}\mathbf{t}_W \tag{1}$$

with $\times$ the cross product and ${}^{V}\mathbf{t}_W$ the translational part of the relative pose ${}^{V}T_W$ of the wheel W with respect to the vehicle reference frame V. For planar motion, the cross product has the optimized form shown in line 8 of Algorithm 1. To evaluate the tire–ground friction model the relevant components of the velocity vector of the wheel center point are those in local coordinates with respect to the wheel, i.e. with the local x axis always pointing forward along the longitudinal wheel axis. Let θ be the rotation of the wheel with respect to the vehicle longitudinal axis, as in Fig. 4(a). Then, the velocity vector in the wheel frame W is related to the velocity in global coordinates from Eq. (1) by ${}^{W}\mathbf{v}_W = \mathbf{R}_z(\theta)^{-1}\mathbf{v}_W$ (line 9 of Algorithm 1) with $\mathbf{R}_z(\cdot)$ the rotation matrix for rotations around the z axis. Once we know the chassis weight W loaded on each wheel axis, the motor torque τ_m exerted at each such axis, and the instantaneous local velocity vector (v_x, v_y) of the wheel center point, we can apply a friction model to determine the resulting longitudinal (F_x) and lateral (F_y) forces that the wheel is exerting on the vehicle chassis, effectively driving the vehicle motion. Different friction models are implemented in MVSim, but at present all of them rely on the mechanical model depicted in Fig. 4(b). Note that C denotes here an optional damping factor, modeling velocity-proportional losses in the applied torque, ω is the instantaneous wheel spinning angular velocity, and F_R is the longitudinal friction force (essential to prevent wheel slippage). In friction-related mechanical problems we have upper bounds for the absolute value of friction forces, but determining whether those limits are hit or not requires two-steps: (i) to solve equilibrium equations assuming friction is large enough to fulfill the no-slippage condition, then (ii) clamp the friction force if needed and solve again if slippage do happen. Note that static equilibrium equations can be split into three separable axis. Firstly, vertical forces equilibrium reveals that $F_z = W$, with W here the sum of the partial chassis weight
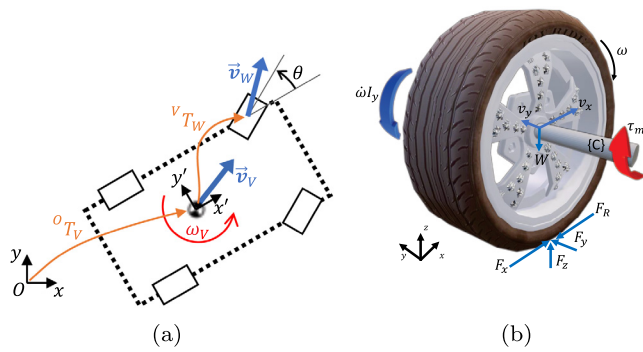
José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez et al.

SoftwareX 23 (2023) 101443

Fig. 4. (a) Kinematics model of a wheel W on a vehicle V. (b) Free body diagram of the mechanical problem involved in determining reaction forces at the wheel–ground contact point. Here, x is always the wheel local forward-facing direction and z points up, $\dot{\omega}$ is the instantaneous angular acceleration of the wheel spinning motion, and I_y the inertia tensor component related to axial (y) rotation.

Algorithm 1 `pre_timestep()` algorithm

1: **for all** v in vehicles **do**
2:     # Evaluate chassis weight **W** on each wheel:
3:     **W**[]   ← v.weight_on_wheels()
4:     # Evaluate motor torques **T**:
5:     **T**[]   ← v.invoke_motor_controllers()
6:     **for all** i,w in enumerate(v.wheels) **do**
7:         # Wheel center point velocity vector:
8:         $\mathbf{v}_w \leftarrow \mathbf{v}_g + (-\omega * w.y, \omega * w.x)$ ▷ See Eq. (1)
9:         $\mathbf{v}_{local} \leftarrow$ rotate_z($\mathbf{v}_w, \theta_i$)
10:        $\{F_x, F_y\} \leftarrow$ v.friction_model($\mathbf{v}_{local}, \mathbf{T}[i], \mathbf{W}[i]$)
11:        # Apply force to vehicle chassis at wheel center:
12:        v.apply_force($\{F_x, F_y\}, \mathbf{t}_W$)
13:     **end for**
14: **end for**

---

and the wheel weight itself. Therefore, the maximum friction force using a simple dry-friction model with friction coefficient μ is $F_{max} = \mu W$. Secondly, lateral equilibrium requires determining the friction force F_y required to avoid lateral slippage induced by the velocity component v_y: the lateral acceleration required to reduce v_y to zero for a simulation time step dt is $a_y = -v_y/dt$, thus given Newton's second law of motion the required force is $F_y = -\frac{v_y}{dt}\frac{W}{g}$ with g the gravity acceleration. This force is clamped to the range $[-F_{max}, F_{max}]$, leading to lateral wheel slippage when the bounds are hit. Thirdly, equilibrium of longitudinal forces and moment on the local axis y finally gives us the value for F_x. In this case, the kinematic constraint (rolling without sliding) would imply changing the wheel angular speed from w to $\omega^\star = v_x/R$ with R the wheel effective radius, hence imposing a desired angular acceleration of $\dot{\omega}^\star = (\omega^\star - \omega)/dt$. Taking moments from the wheel axis in Fig. 4(b) leads to the equilibrium equation:

$$\tau_m - \dot{\omega}I_y + RF_R - RF_x = 0 \qquad (2)$$

where F_R is an optional rolling friction (e.g. MVSim implements the realistic Ward–Iagnemma model in [16]). Once the friction force F_x is determined, if saturation happens due to the existence of the upper bound F_{max}, Eq. (2) is evaluated again with $F_x = \pm F_{max}$ to determine the actual spinning angular acceleration $\dot{\omega}$ given that longitudinal slippage was unavoidable.

Finally, MVSim supports vehicles or robots with 3 or more wheels to move on terrain models defined by elevation maps. This is accomplished by determining the elevation of each wheel–ground contact point with respect to the reference level, then applying the quaternion Horn method [17] to solve for the optimal vehicle attitude.
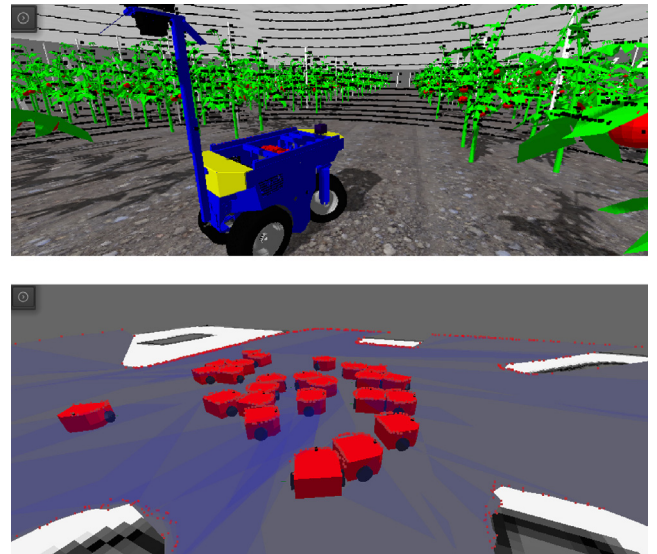

Fig. 5. Screenshots of shipped demo worlds.

2.3. Software functionalities

MVSim allows running simulations of one or more vehicles observing collisions, realistic rigid body kinematics and wheel–ground friction models, as well as sensor simulation. Sensor data is accessible via either our custom ZMQ-based pub/sub communication system (C++ and Python interfaces) or via ROS 1 [18] or ROS 2 [19] standard sensor messages. Simulations can run in real wall-clock time or at a different rate. For example, faster than real-time runs can be used in headless instances to support simulation-based unit tests of mobile robotic software. Implemented sensors at present include pin-hole cameras, RGBD (depth) cameras, 2D and 3D LiDARs. In all these cases, sensor simulation exploits GPU acceleration via an OpenGL pipeline to achieve generation of millions of 3D LiDAR points per second in real-time. Each vehicle is equipped with a configurable PID controller to generate the motor torques required to accomplish the desired vehicle linear and angular speed set-points. Additionally, vehicles have an optional high-speed logging system to log all the internal physics details (frictions, forces, torques, etc.) to CSV files for posterior debug and analysis. World models are defined by means of XML files whose format is detailed in the on-line project documentation page.[1] There is support for including other XML files within a given file, parsing and replacement of environment variables, mathematical expression evaluation, and flow control-like structures such as `for` loops at the XML document level.

3. Examples and experimental benchmark

The project repository contains a `mvsim_tutorials` subdirectory with several example files illustrating the different possibilities of the simulator and the XML-definition language. As qualitative example worlds, a warehouse with a Jackal robot is shown in Fig. 1, a greenhouse with a custom Ackermann-like vehicle is shown in Fig. 5(a), and a demonstration of how to run dozens of robots driving autonomously from Python scripts is shown in Fig. 5(b). We claim that not needing to provide an URDF

---

[1] Example XML world files are provided in https://github.com/MRPT/mvsim/tree/develop/mvsim_tutorial.

José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez et al.
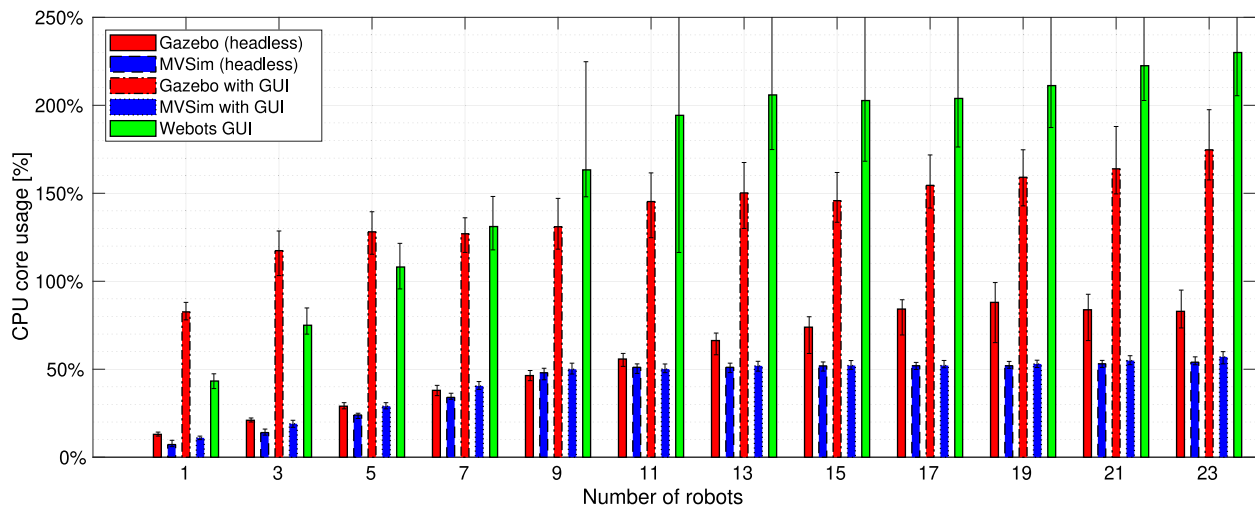
SoftwareX 23 (2023) 101443

Fig. 6. Performance comparison of Gazebo, MVSim, and Webots for a simple simulation environment with *N* robots. Bars indicate the median over 60 sampled CPU usage values, overlaid with 90% confidence intervals.

file for our simulator makes the design of simulations faster and simpler to learn.

Next we provide a quantitative benchmarking comparing MVSim to other state-of-the-art robotic simulators. We have measured CPU usage of Gazebo, Webots, and MVSim while working within a test ROS 2 application, with and without the simulator native GUI. RViz was disabled in the tests to avoid competition for the GPU. The results, in Fig. 6, shows the CPU usage of the simulators with an increasing number of identical robots (differential-driven, equipped with a 2D LIDAR). In all cases, it is clear how MVSim matches or improves the performance of state-of-the-art simulators. All runs were measured on an Intel(R) Core(TM) i7-8700 CPU (3.20 GHz) with an NVIDIA GeForce GT 710 GPU. Measurements include the sum of all CPU usage by all the processes required to run the simulation itself, publish the robot data to the ROS 2 subsystem, and, for the study cases with a GUI, handling and updating the graphical interfaces. Scripts to reproduce the benchmark are available online for the sake of reproducibility.[2]

4. Impact

The present simulator has great potential to research on SLAM with either 2D LiDAR [20,21], 3D LiDAR [22], depth cameras [23, 24], or pure visual SLAM [25], as well as collaborative SLAM approaches [26] by means of multiple simultaneous robots. Generation of synthetic datasets with precise ground truth trajectories are key to train, test, and validate SLAM solutions. Integration of MVSim with ROS 2 Nav2 [27] autonomous navigation framework has been also successfully tested (example launch files are provided online), allowing easy debugging and testing of navigation algorithms in simulation before moving them to real robots. Additionally, it allows ground-truth data generation for research on vehicle dynamics and related topics, where real-world ground truth data is hard to obtain due to the nature of wheel–ground interactions [5,28].

5. Conclusions

After analyzing the design and features of the presented simulator, we defend that it has potential to become one of the new

tools researchers from different fields might use in their day-to-day work to ease the generation of synthetic datasets, or test and validate new algorithms that require closing a loop between acting and sensing on moving agents. In comparison to existing simulators, writing an MVSim XML world file to define a world and a vehicle or robot requires much less training and effort, lowering the entry barrier to ROS-based simulation to students and researchers while still obtaining quality simulations.

Declaration of competing interest

Authors declare that they have no competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2023.101443.

References

[1] Grisetti G, Kümmerle R, Stachniss C, Burgard W. A tutorial on graph-based SLAM. IEEE Intell Transp Syst Mag 2010;2(4):31–43.
[2] Cadena C, Carlone L, Carrillo H, Latif Y, Scaramuzza D, Neira J, et al. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. IEEE Trans Robot 2016;32(6):1309–32.
[3] Kavraki LE, LaValle SM. Motion planning. In: Springer handbook of robotics. Springer; 2016, p. 139–62.

---

2 URL: https://github.com/ual-arm/robotic-simulators-benchmark.

José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez et al.

SoftwareX 23 (2023) 101443

[4] Mandow A, Martinez JL, Morales J, Blanco JL, Garcia-Cerezo A, Gonzalez J. Experimental kinematics for wheeled skid-steer mobile robots. In: 2007 IEEE/RSJ international conference on intelligent robots and systems. IEEE; 2007, p. 1222–7.

[5] Reina G, Paiano M, Blanco-Claraco J-L. Vehicle parameter estimation using a model-based estimator. Mech Syst Signal Process 2017;87:227–41.

[6] Wang R, Jing H, Hu C, Chadli M, Yan F. Robust H∞ output-feedback yaw control for in-wheel motor driven electric vehicles with differential steering. Neurocomputing 2016;173:676–84.

[7] Meeussen W. Rep 105-coordinate frames for mobile platforms. 2010, Available in: https://www.ros.org/reps/rep-0105.html.

[8] Rico FM. A concise introduction to robot programming with ROS2. CRC Press; 2022.

[9] Gerkey B, Vaughan RT, Howard A, et al. The player/stage project: Tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th international conference on advanced robotics, Vol. 1. Citeseer; 2003, p. 317–23.

[10] Koenig N, Howard A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ international conference on intelligent robots and systems, Vol. 3. IEEE; 2004, p. 2149–54.

[11] Michel O. WebotsTM: Professional mobile robot simulation, vol. 1, no. 1. 2004, p. 39–42, arXiv:0412052.

[12] Hintjens P. ZeroMQ: messaging for many applications. " O'Reilly Media, Inc."; 2013.

[13] Catto E. Iterative dynamics with temporal coherence. In: Game developer conference, Vol. 2, no. 5. 2005.

[14] Sola J, Deray J, Atchuthan D. A micro Lie theory for state estimation in robotics. 2018, arXiv preprint arXiv:1812.01537.

[15] Blanco-Claraco JL. A tutorial on se(3) transformation parameterizations and on-manifold optimization. 2021, arXiv preprint arXiv:2103.15980.

[16] Ward CC, Iagnemma K. A dynamic-model-based wheel slip detector for mobile robots on outdoor terrain. IEEE Trans Robot 2008;24(4):821–31.

[17] Horn BK. Closed-form solution of absolute orientation using unit quaternions. J Opt Soc Amer 1987;4(4):629–42.

[18] Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, et al. ROS: An open-source Robot Operating System. In: ICRA workshop on open source software, Vol. 3, no. 3.2. Kobe, Japan; 2009, p. 5.

[19] Macenski S, Foote T, Gerkey B, Lalancette C, Woodall W. Robot operating system 2: design, architecture, and uses in the wild. Science Robotics 2022;7(66):eabm6074.

[20] Grisetti G, Stachniss C, Burgard W. Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Trans Robot 2007;23(1):34–46.

[21] Kohlbrecher S, Von Stryk O, Meyer J, Klingauf U. A flexible and scalable SLAM system with full 3D motion estimation. In: 2011 IEEE international symposium on safety, security, and rescue robotics. IEEE; 2011, p. 155–60.

[22] Hess W, Kohler D, Rapp H, Andor D. Real-time loop closure in 2D LIDAR SLAM. In: 2016 IEEE international conference on robotics and automation. ICRA, IEEE; 2016, p. 1271–8.

[23] Whelan T, Salas-Moreno RF, Glocker B, Davison AJ, Leutenegger S. Elastic-Fusion: Real-time dense SLAM and light source estimation. Int J Robot Res 2016;35(14):1697–716.

[24] Mur-Artal R, Tardós JD. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Trans Robot 2017;33(5):1255–62.

[25] Campos C, Elvira R, Rodríguez JJG, Montiel JM, Tardós JD. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. IEEE Trans Robot 2021;37(6):1874–90.

[26] Zou D, Tan P, Yu W. Collaborative visual SLAM for multiple agents: A brief survey. Virtual Real Intell Hard 2019;1(5):461–82.

[27] Macenski S, Martin F, White R, Ginés Clavero J. The Marathon 2: A Navigation System. In: 2020 IEEE/RSJ international conference on intelligent robots and systems. 2020.

[28] Leanza A, Reina G, Blanco-Claraco J-L. A factor-graph-based approach to vehicle sideslip angle estimation. Sensors 2021;21(16):5409.