Robotica (2016) volume 34, pp. 791–822. © Cambridge University Press 2014
doi:10.1017/S0263574714001866

Survey and comparative study of free simulation software for mobile robots

M. Torres-Torriti†*, T. Arredondo‡ and P. Castillo-Pizarro‡

†*Department of Electrical Engineering, Pontificia Universidad Católica de Chile, Santiago, Chile*
‡*Department of Electronics, Universidad Técnica Federico Santa María, Valparaiso, Chile*

(Accepted June 16, 2014. First published online: July 15, 2014)

SUMMARY
In robotics, simulation has become an essential tool for research, education, and design purposes. Various software tools for mobile robot simulation have been developed and have reached different levels of maturity in recent years. This paper presents a general survey of mobile robot simulation tools and discusses qualitative and quantitative aspects of selection of four major simulators publicly available at no cost: Carmen, Player-Stage-Gazebo, Open Dynamics Engine, and Microsoft Robotics Developer Studio. The comparison of the simulators is aimed at establishing the range of applications for which these are best suited as well as their accuracy for certain simulation tasks. The simulators chosen for detailed comparison were selected considering their level of maturity, modularity, and popularity among engineers and researchers. The qualitative comparison included a discussion of relevant features. The quantitative analysis entailed the development of a detailed dynamical model of a mobile robot on a road with varying slope. This model was used as benchmark to compare the accuracy of each simulator. The validity of the simulated results was also contrasted against measurements obtained from experiments with a real robot. This research and analysis should be very valuable to educators, engineers, and researchers who are always seeking adequate tools for simulating autonomous mobile robots.

KEYWORDS: Mobile robots; Simulation software; Comparative survey; Mobile robot dynamics; Physics engine.

1. Introduction

Nowadays simulation plays a central role in the off-line programming of robotic systems, the performance evaluation of industrial robotic cells, and the evaluation of advanced control algorithms.[17,28,55,56] The design and feasibility testing of novel and complex robotic systems, ranging from underwater mobile manipulators[33] to sophisticated closed kinematic chain robots,[50] humanoids,[39] and haptic systems,[44] would not be possible without proper simulation tools. In addition to the research and engineering of robotic systems, simulation is an invaluable tool in the development of virtual environments for operator training as well as the teaching and learning of fundamental physics and underlying concepts of robotic systems. This knowledge would otherwise be very hard to grasp without actual experimentation and visualization.[6,31] On the other hand, robotic systems can be costly, and therefore simulation provides an excellent method of testing and experimenting while avoiding the costs of accidents and the difficulty of controlling disturbances present in the physical world.[6]

The progress in affordable and more powerful computing technology that was made during the 1990s contributed significantly to the development of several publicly available open-source software tools for robotic manipulator modeling and simulation, such as XAnimate,[31] the Robotics Toolbox for Matlab,[8] and the Robotica package for Mathematica.[37] At the turn of the 21st century, the monotonic

* Corresponding author. E-mail: mtorrest@ing.puc.cl

increase in computational power allowed researchers to focus on modeling and simulation problems of growing complexity, such as robot manipulation and grasping,[34] multibody robotic systems with discontinuous contact interactions,[44] and mobile robotics.[48] As reported in the introductory survey by the authors[4] and two other recent brief surveys,[21,29] the number of tools for simulating and interfacing with various mobile robot platforms has grown considerably during the last five years. Among the open-source tools for mobile robot simulation that have achieved some degree of sustained development it is possible to mention Breve,[38] Carnegie Mellon Robot Navigation Toolkit (Carmen),[58] the Open Dynamics Engine (ODE),[65] OpenRave,[43] OpenSim,[68] the Player-Stage-Gazebo (PSG) project,[20,70] RoboCode,[27] Simbad,[23] and UsarSim.[3] There are also a number of mobile robot simulation tools that are publicly available at no cost for personal or academic use but include proprietary closed-source components such as the Microsoft Robotics Developer Studio (MRDS),[25] anyKode Marilou,[57] and V-Rep.[41,75] The last two rely on ODE, which is open-source, while MRDS relies on Nvidia's PhysX® physics engine, which is closed-source. Although this paper focuses on publicly available feeless simulation software, some completely proprietary (closed-source and non-free) tools deserve mention in order to provide a complete view of the wide spectrum of alternatives. Among these tools for mobile robot simulation, it is possible to find Aria®,[9] Cogmation robotSim®,[59] Webots®,[30] Energid Actin Toolkit for Robot Simulation®,[60] and RTI SimCreator®, just to name a few.

Choosing the most adequate simulation tool suitable to one's research, development or education purposes can prove to be a challenge not only because the list of tools is long but also because one may not be aware of all available options. The decision may be further complicated because it is not obvious which tools are more user-friendly than others, which involve only kinematic models and are intended for developing and testing autonomous agent motion planning strategies, or which include accurate dynamical models that facilitate the performance analysis of motion controllers and the computation of forces and power consumption. Moreover, many tools depend on different external packages, which make it more complex for the user to understand the details of all the components involved in the simulation; not to mention that the reusability of the developed code may be affected by changes in external libraries, thus causing the software to be hard to maintain and update.[4]

In view of the variety of mobile robot simulation tools and the importance of identifying their particular benefits, this paper extends the introductory survey presented by the authors in ref. [4] by providing a broader and more in-depth discussion about the existing free of charge, publicly available tools. This paper also presents a detailed qualitative and quantitative comparison of four selected simulation tools: Carmen, PSG, ODE, and MRDS. The choice of simulators to be compared in detail was not simple. However, there are some aspects that are arguably the most important across a wide audience of researchers, programmers, and system engineers alike, namely that the code be stable, well documented, adequately supported, preferably open-source, and produce physically correct results in reasonable time. A review of the current open-source mobile robot simulators and a detailed discussion of the fundamental qualitative differences between the selected simulators are presented in Sections 2 and 3 respectively. These sections provide the main grounds and detailed reasons on which the selection of the simulators was made. However, considering that currently one of the main challenges is to build robots that can interact autonomously with its surrounding environment, which in turn requires tools to design and evaluate mobile manipulator–object interactions, vehicle–terrain interactions, mobile robot–human interactions, among others, simple kinematic models and simulations are insufficient, and therefore the capabilities of the simulators to produce not just visually realistic but also phyisically accurate results must be evaluated. Thus, quantitative results obtained with Carmen, PSG, ODE, and MRDS® are presented in Section 4. The quantitative comparison is based on a set of simulation experiments used to test the fundamental physics compliance of the simulators for open-loop longitudinal motion and closed-loop turning control. The robot position, velocity, and pitch resulting from simulations is not only compared with the ideal trajectory in downhill-roll and ramp-jump experiments but also with measurements obtained from equivalent experiments performed with a real robot. Prior to the discussion of longitudinal motion experiments, the longitudinal motion dynamics is presented in detail considering robot–ground contact interaction. The latter is not extensively found in the current robotics literature, and we believe it should also be of reference and tutorial value to a broad audience who are interested in using or developing robotic simulation systems. Finally, the conclusions are presented in Section 5. Appendix A provides an introductory background of key architectural aspects of simulator paradigms, which may provide

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

further insight into the operating characteristics of each simulator. Readers not familiar with the paradigms of simulator design are suggested to peruse this section before reading the complete paper. It is to be noted that robot middleware software and programming environments, such as Robot Operating System (ROS) or MRPT, offer functionality to integrate simulators, and similarly Carmen, MRDS, and PSG offer middleware functionality that allows to integrate algorithms and hardware with the simulation. In this paper, we briefly discuss middleware functionality in Section 3 and focus on simulator capabailities. Readers interested in a survey of robotics middleware are referred to Elkady and Sobh.[15]

2. Mobile Robot Simulators Overview

The following is an overview of the main open-source mobile robot simulators that were found to be publicly available and free of charge at the time of the writing of this paper:

- *Breve*[26] is a package which allows the creation of 3D simulations of multi-agent systems and artificial life using Python scripts. Breve employs OpenGL to generate visually realistic simulations and includes experimental support for articulated body physical simulation and collision detection with static and dynamic friction. Breve's emphasis is placed on visual realism rather than physically accurate simulations, and the physics engine is a work-in-progress.[26]
- *Carmen*[58] is a modular library that provides algorithms and codes necessary to accomplish all tasks involved in autonomous mobile robot navigation, such as localization based on odometry and laser-scanning, obstacle avoidance, path planning, and mapping. The Carmen library seems to have started as a robot programming API to unify and simplify the task of commanding different brands of robots. However, Carmen now includes a simulation module that can be used to test navigation algorithms. Further details about Carmen's features will be discussed in Section 3.
- *Dynamechs*[33] and *RobotBuilder*[40] are respectively a library and a software application based on the same library for simulating the dynamics of robotic articulations in open or closed-kinematic loops. Dynamechs includes the efficient articulated-body algorithm by Featherstone,[18] which provides accurate joint or body accelerations and forces if models are built correctly. In addition to this capability, Dynamechs incorporates hydrodynamic effects that allow users to simulate underwater vehicles with multiple chains (manipulators or legs). This valuable feature is not found in other open-source dynamic simulation libraries. Dynamechs was last updated in 2001, thus users may need some time to port it to current operating systems. On the other hand, RobotBuilder has a simple to use Windows®-based graphical interface, and can be very useful in introductory dynamics and robotics courses, as well as research, by enabling users to focus on control issues while reducing modeling time. However, some drawbacks are that RobotBuilder is only distributed in a pre-compiled format, its last release dates in 2003, and has less world modeling capabilities than more recent simulators. Also, although RobotBuilder runs on Windows XP®, it relies on the WorldToolKit graphic library, which is no longer supported. This may complicate the release of an enhanced application without rewriting several parts of the code in future.
- *Moby*[64] is a multi-body dynamics simulation library written in C++ and oriented to the simulation of robotic systems with a special emphasis on dynamical accuracy. Unlike most of the physics engines, Moby not only supports maximal-coordinate representations of articulated bodies but also handles reduced-coordinate representations employing the efficient $O(n^2)$ Composite Rigid Body algorithm[49] or the $O(n)$ Articulated Rigid Body algorithm.[18] Moby also includes continuous collision detection, which is a useful feature not common in other dynamic simulation libraries. Moby is not supported in Windows, and requires several external Linux packages and Python to build. This can be a drawback for people without enough programming knowledge and those wishing to port the system to Windows.
- *Open Dynamics Engine* (ODE)[65] is an open-source physics engine, under development by Russell Smith since 2001,[65] which enables the simulation of dynamics of articulated rigid bodies. It includes a basic rendering engine (*drawstuff*) that can be easily replaced by other full-featured alternatives. ODE is very popular in the implementation of robotics simulation applications,[13] both commercial applications, such as anyKode Marilou,[57] Webots,[30] and V-Rep,[75] and free applications, such as Gazebo and several simulators intended specifically for simulating RoboCup soccer and humanoid robots (see references in Drumwright *et al.*[13]). ODE solves simultaneously the Newton–Euler

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

dynamic equations and the contact (collision) constraints by posing the equations governing the systems of rigid bodies as a linear complementarity problem (LCP)[1,45,46] in the maximal coordinate system (see also discussion in Appendix A on the simulators' architecture) to solve for impulses satisfying the constraints imposed by joints connecting bodies and contact points. The LCP in ODE is solved with the successive over-relaxation variant of the projected Gauss–Seidel (PGS) method. Recent successful efforts to extend ODE and improve its simulation time, the handling of viscous joint-dampening, the friction cone approximation, and its solution of non-interpenetration and joint constraints have been reported by Drumwright *et al*.[13] These improvements have been possible by: (i) reordering the terms in the original LCP matrices in such a way that the new LCP is always a simpler convex problem, and (ii) considering the complete friction cone instead of a linearized version of it in the formulation of a nonlinear convex-optimization problem that yields the joint and contact point impulses that minimize kinetic energy, and ensure the fulfillment of non-interpenetration, friction, and motion constraints. Additional details about ODE will be discussed in Section 3.

- *OpenRave*[66] is focused on autonomous motion planning and high-level scripting rather than low-level control and message protocols. To this end, OpenRave provides a plugin-based framework which integrates programming interfaces to services, such as physics, collision detection, kinematics, and robot control. It is based on well-established packages such as the Bullet Physics Package and ODE for physics simulation, ROS and its own modules, which include kinematic models for various types of robots, sensor models, and motion planners. OpenRave is a promising project still under active development.[12] A drawback of this tool is that getting started is not very simple, since it requires knowledge of several different external packages that are necessary to achieve OpenRave's full functionality.
- *OpenSim*[68] is a simulator for mobile robots based on ODE. It includes a component to solve the inverse kinematics of redundant manipulators called IKOR, and integrates Open Scene Graph, Demeter, and OpenGL to provide terrain modeling and rendering capabilities. OpenSim also includes pre-built models of mobile manipulators. OpenSim for mobile robot simulation[68] should not be confused with another simulator that has the same name, but which was created for modeling and simulating the motion dynamics of musculoskeletal systems with biomechanics applications in mind.[11] OpenSim for musculoskeletal biomechanics is an active project, unlike OpenSim for mobile robot simulation, which has not been active since 2006.
- *Player-Stage-Gazebo*[20,70] is an open-source project that consists of the Stage (2D) and Gazebo (3D) simulators, and the Player network server. In PSG, application programmers may use different application programming interfaces (APIs) and associated libraries depending on whether the users want to connect to Player, Stage, or Gazebo and the language they want to use. PSG depends on ODE and Bullet for simulating rigid body dynamics, and on Object-oriented Graphics Rendering Engine (OGRE) as a rendering engine. The PSG project aims for Portable Operating System Interface (POSIX) compliance and runs on most of the UNIX-like operating systems.[4] It is to be noted that PSG has been selected as a simulation library of the ROS project, which is rapidly growing in popularity since its release.[72] Additional details about PSG will be discussed in Section 3.
- *Simbad*[73] is a 3D robot simulator employed for research and education purposes. It is intended for users who want to study machine learning and artificial intelligence algorithms in the context of autonomous agents. It emphasizes readability, simplicity, and user-friendliness over accurate real-world simulations, and therefore it only utilizes a simplified physics model.[23]
- *UsarSim*[74] stands for Unified System for Automation and Robot Simulation. It is a high-fidelity robot and environment simulation tool based on the Unreal Tournament game engine by Epic Games Inc. In November 2009, Epic released a free version of the game engine under the name of Unreal Development Kit (UDK). According to UDK's current end-user license agreement, developers can use it at no cost in non-commercial applications, otherwise a fee must be paid. Like MRDS, UsarSim is oriented at macroscopic world simulations and employs Nvidia's PhysX® as its physics engine through UDK, which relies on PhysX®. Another similarity between UsarSim and MRDS is that they hide the finer level of details from the user. This can be advantageous when the user is only interested in simulating robots at the macroscopic level. In fact, UsarSim is employed in the RoboCup Rescue Virtual Robots Competition and the IEEE Virtual Manufacturing and Automation Competition. However, hiding the finer level of details can be a disadvantage for users wishing to easily obtain information about robot's internal forces or the interaction forces between

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

the robot and its environment. Although MRDS and UsarSim would target similar audiences, one difference is that UsarSim relies on the Player project[20,70] as a network server to provide a unified interface for controlling the robots and communicating with sensors and actuators. MRDS instead uses a proprietary API implemented in C# for concurrency and communication; see discussion in Section 3.3. Hence, UsarSim can be employed under more operating systems, e.g. Linux, Windows, and Mac than MRDS. One of UsarSim's drawbacks is that it may not be very user-friendly yet as its first production version 1.0 was released in August 2011. Moreover, UsarSim depends on several external packages that the user must understand before being able to develop simulations in UsarSim.

There exists other mobile robot simulators such as EyeSim[61] (for EyeRobot) and RoboCode[71] (for virtual robot tank battles), which are conceived to simulate only one type of robot under some specific scenarios, and therefore cannot be considered completely versatile simulation libraries or packages. There are also a number of projects whose original purpose was to provide a unified programming interface for robot programming. Some of these tools have grown to include models, control algorithms, some simulation capabilities, or the possibility of communicating with external programs that can provide simulation capabilities. Since the primary objective of these tools is to serve as robot programming interfaces or operating systems, rather than mobile robot simulation libraries, they will not be discussed in this paper. However, the most important ones deserve to be mentioned because of their growing chances of interaction with existing simulators. Following are the main robot hardware programming interfaces: Player[70] (client–server infrastructure for application component communication), ROS,[72] JDE,[63] OpenRDK,[67] OROCOS,[69] and the YARP[76] middleware API. The reader may found an in-depth review of the existing robotics middleware in the recient survey.[15]

3. Selected Simulators' Qualitative Comparison

This section reviews Carmen, PSG, ODE, and MRDS in terms of their main features, including architectural aspects, flexibility, and modularity issues. For step-by-step installation and usage notes, we created a web site which includes this information[62] as well as the full source code of the simulations presented in this paper. See also Appendix B for a brief discussion about installation and usage aspects. A comparison summary of the tools and their most relevant features are presented in Tables I and II, which respectively present the main physics modeling capabilities and middleware functionality of simulators. The user interface is an optional feature in most simulators and is not required to perform the experiments in our present study, hence a detailed discussion is not present in Table II. In any case, user intereface information for each simulator is provided in each associated subsection where appropriate.

3.1. Carmen: Carnegie Mellon robot navigation toolkit

Carmen[58] is an open-source C++ library that provides basic navigation functions for mobile robot control, sensors reading and logging, localization, path planning, mapping, obstacle avoidance and motion simulation. Carmen was developed by researchers at Carnegie Mellon's Robotics Institute and sponsored by DARPA's MARS Program. The development of Carmen's first beta 0.1 versions dates back to October 2004, version 0.6 was made publicly available on SourceForge in April 2006. The current version is beta 0.7.4 and was released on October 2008.

The main features of Carmen include: support for different mobile robot platforms: iRobot's ATRV, B21R, ActivMedia Pioneer robots, Nomadic Technologies Scout and XR4000 and Segway; support for various sensors: Sick LMS and PLS laser scanners, Hokuyo IR PB9 obstacle detection sensor, NMEA GPS data streams; simulation in 2D only; navigation algorithms for localization, path planning, mapping; written in C language with some support for Java; currently it runs under Linux only and has a highly modular design relying on the well-established and supported Inter-Process Communication (IPC) library developed since 1994 at CMU by Reid Simmons.

The main modules of Carmen are the robot base interface, the sensors interface, the control module, the graphical user-interface (optional) module and the simulator module. Using the IPC library, each module subscribes to certain messages, which are managed by the central server as shown in Fig. 1. The modules can be queried by the central server, which then publishes the messages

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

Table I. Comparison of main physics modeling capabilities of selected robot simulation tools.

	Carmen	MRDS	PSG	ODE
World space	2D	3D	3D	3D
Type of motion model supported	Kinematic	Dynamic	Dynamic	Dynamic
Motion update algorithm	Kinematics specified by the user	PhysX[a]	ODE[b]	NE+LCP[c]
Collision handling method	No	Yes	Yes[d]	Yes[e]
Built-in geometries	No	Spheres, boxes, planes, and meshes	Spheres, boxes, cylinders	Spheres, capsules, boxes, cylinders, planes, rays, and meshes.
Built-in joint models	No	Revolute (hinge), slider (prismatic), cylindrical (piston), spherical (ball & socket), 6-DOF configurable	Same as ODEs	Revolute (hinge), slider (prismatic), cylindrical (piston), spherical (ball & socket), universal (Cardan joint), caster wheel-like, intermittent unilateral contact, 2D-planar, and other user-defined combinations.
Built-in actuators models	No	Same as joint models	Same as ODEs	Rotary and linear, and other user-defined.
Built-in robot models	Pioneer robots, no robot arms.	Pionner-2DX, Pioneer-3AT, Lego Mindstorms, iRobot Create, Segwat RMP, Festo Robotino, ABB and Kuka robot arms, and other.	Pionner-2DX, Pioneer-3AT, PR2, quadrotor, robot arm, Segway RMP, Turtlebot, YouBot, and many other[f]	Generic cart example.
Built-in sensor models	Odometry, GPS, laser scanner	Odometry, GPS, laser scanner, Kinect camera	Odometry, GPS, laser scanner, camera, stereo camera, IMU, force and torque, contact, wireless transceiver	None
On the fly model configuration and update while the simulation is running	No	No	Yes	Yes

[a]Nvidia's PhysX engine implements several algorithms for physics particles and rigid multi-body objects. The library is a proprietary and there are little details on its current algorithms. However, in the case of rigid multi-body objects, it possibly still retains algorithms implemented by NovodeX prior to being purchased by Ageia in 2004 and later by Nvida in 2008. The original NovodeX physics engine solved the Newton–Euler equations for each body in maximal coordinates using the impulse-velocity time-stepping formulations by Stewart–Trinke[45,46] and Anitescu–Potra,[1,2] who pose the problem as an LCP to solve the motion update equations and the motion constrains for all bodies simultaneously.

[b] Motion simulation is based on ODE; see the following footnote. In January 2014, Gazebo announced that in addition to ODE it now supports three other physics engines: Bullet, DART, and Simbody. The last two employ the reduced coordinate approach by Featherstone and optimized for kinematic chains (see discussion in Appendix A) in contrast to Bullet and ODE, which are maximal coordinate solvers optimized for performance over many independent models. Each physics engine is motivated by a particular application domain from gaming (Bullet) to simplified robot dynamics (ODE) to biomechanics (Simbody) to computer graphics and robot control (DART).

[c]The Newton–Euler equations for each body are formulated in maximal coordinates and solved simultaneously as an LCP. The LCP is solved by the successive over-relaxation variant of the PGS method; see also the discussion in Appendix A and refs. 1, 2, 45, 46.

[d] Gazebo has traditionally relied on ODE, and thus solved collisions using `OPCODE` (see next footnote), but at the time of writing this paper it has added support for the `RAPID` library.

[e]Uses Mirtich's approach[35] and several libraries, which include `libccd` for collision detection between convex shapes using the Gilbert–Johnson–Keerthi algorithm, the Expand Polytope Algorithm (EPA), and the Minkowski Portal Refinement (MPR) algorithm. The user can select whether ODE uses functions from the `Gimpact` or the `OPCODE` library, which support collision detection for trimeshes in particular concave shapes and deformable meshes.

[f] Full list in the Gazebo model's database https://bitbucket.org/osrf/gazebo_models/src.

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

Table II. Comparison of main algorithms and software features provided by the selected robot simulation tools.

	Carmen	MRDS	PSG	ODE
High-level robot control and reasoning algorithms				
Path planning and motion control	Shortest-path planner based on set of way-points with PD controller	None	See footnote[a]	None
Localization	Classical Monte Carlo localization using particle filter and laser scans	None	See footnote[a]	None
Mapping	Scan matching	None	See footnote[a]	None
Software features				
Supported OS	Linux	Windows	Linux	Linux & Windows
Compiler/ Language	GCC/C	Visual Studio .Net/C# & MS Visual Programming Language	GCC/C	GCC/C
Open-source	Yes	No	Yes	Yes
Simulator architecture	Distributed (IPC)	Distributed (CCR+DSS)	Distributed (sockets/shared memory)	Monolithic
Installation complexity	High	Low	High	Low
User-interface	Command-line	Command-line & graphical	Command-line	Command-line
Visualization and rendering	2D	3D/DirectX	3D using `OGRE` library/OpenGL	3D using `drawstaff` library/OpenGL
Middleware functionality	Yes	Yes	Yes[b]	No
Initial release year and references	2004[58]	2006[25]	2001[20,70]	2001[13,65]

[a]PSG does not have its own set of high-level robot control and reasoning algorithms, but a large set of path planning, motion control, simultaneous localization, and mapping algorithms that are available in ROS. These algorithms can be invoked in PSG using the communication application programming interfaces that support the ROS–PSG integration.

[b]Player provides hardware abstraction, low-level device control, and inter-process communication functionality. In addition, it is based on a graph architecture where processing takes place in nodes that may receive, post, and multiplex sensor, control, state, planning, actuator, and other messages. Through the ROS–PSG integration, PSG can take advantage of the ROS components that provide robot abstraction, software and hardware integration, and other common application-independent functionality that allow to simplify the composition of subsystems that form part of a robotic system.

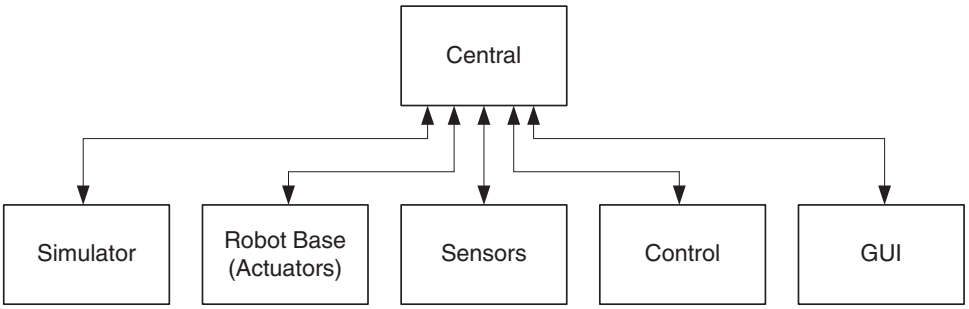


Fig. 1. Carmen IPC architecture based on TCP sockets for communication between several processes and a central server.

to the subscribing modules. Although this architecture can be slower than using a shared-memory approach, it allows running the modules on different host machines over a network. This distributed architure approach is advantageous because it allows to control or simulate systems with multiple agents.[55] Moreover, in many real systems some navigation tasks may require a dedicated processor due to their complexity, thus making it necessary to distribute the whole process over a network.

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

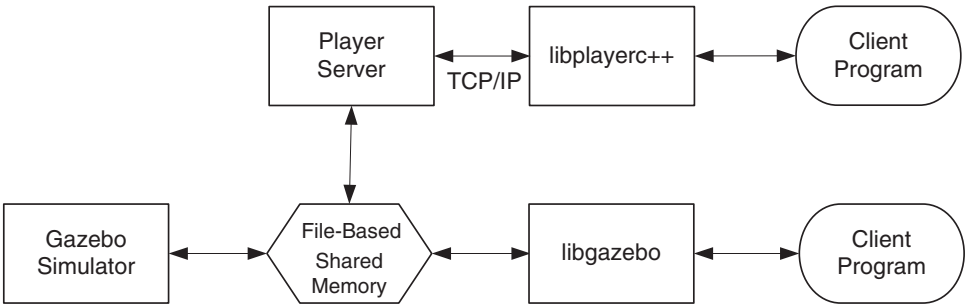


Fig. 2. PSG architecture.

One of the outstanding features of Carmen is that it includes state-of-the-art mapping and navigation algorithms not available in other simulators. However, some drawbacks of Carmen when compared with other simulators are that its documentation is partially complete and does not include examples on how to add new modules or program navigation strategies. Also, the fact that Carmen only considers 2D kinematic models can be a disadvantage. Significant programming efforts would be necessary to furnish Carmen with additional 3D simulation capabilities. It is fair to say that Carmen is a well-thought-out and compact simulator, and can be a great tool for people especially interested in testing novel simultaneous localization and mapping algorithms in 2D environments with possibly multiple robots.

3.2. Player-Stage-Gazebo

As mentioned in Section 2, PSG consists of two multi-robot simulators: Stage (2D) and Gazebo (3D), and the network server called Player. The architecture of PSG is presented in Fig. 2, which shows that client programs can communicate with the simulator using either the *libplayerc++* API of the Player server or the *libgazebo* API. It is to be noted that Stage or Gazebo implement standalone simulators that can communicate with client applications through the *libgazebo* API, without requiring the Player server, since the *libgazebo* API provides message passing functionality by means of filed-based memory sharing (a fast and reliable message passing mechanism that has less time variability and latency than TCP/IP communication). The Player server is useful for running simulations over a network using its standard set of device interfaces for sensors and actuators. Currently, Player, Stage, and Gazebo versions are 3.0.2, 4.0.1, and 0.10.0 respectively. Further details about these components are discussed in the following sections, and in greater depth in ref. [70].

3.2.1. Player. The Player server has an equivalent role to that of the central server of Carmen, but in addition to providing its own TCP/IP socket-based inter-process communication functionality, it also includes other transports, such as JINI (though not widely used) for message passing.

In typical PSG configurations, robots and sensors (real or simulated) are controlled using a standard set of interfaces provided by the Player server. In principle, client programs cannot tell difference between real or simulated devices when using the Player server. Thanks to the hardware abstraction provided by Player, users developing software with the Stage or Gazebo simulators should be able to quickly start controlling real robots after only a few configuration changes. It should also be possible to switch to different brands or models of kinematically equivalent robots after minor configuration changes.

Player is normally run in the same machine as the simulator or the robot that is being controlled. The user program controlling the robot connects as a client to the Player server. Client programs can connect to many servers running concurrently. On the other hand, each server can respond to many clients connected to it because Player does not implement exclusive access to sensors or actuators. The most commonly used client libraries (APIs to Player) are *libplayerc* and *libplayerc++* for C and C++ respectively. Client libraries in Python and other languages are also available.

Each device interface provided by the Player server specifies different ways to interact with the device (real or simulated). The interface/driver model implemented in Player groups devices by logical functionality so that devices which approximately do the same task are abstracted and appear to be the same from user's perspective. An interface is a specification for the contents of the data

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

stream, so an interface for a robotic device maps input streams into sensor readings, output streams into actuator commands, and Input Output Control commands (IOCTLs) into device configurations.

The software that converts between a device's native formats and the interface's required formats is called a driver. For example, the Player *position2d* interface covers ground-based mobile robots, allowing them to accept motion commands (either position or velocity targets) and to report their state (current position and velocity). Many drivers support the *position2d* interface, some examples are: *p2os* (i.e. Pioneer), *obot*, and *rflex* (i.e. RWI Robots), each of these controls a different kind of robot. The job of the driver is to make the robot support the standard interface.[48]

Unfortunately, using Player is only practical for non-real-time experiments with slow moving, statically stable wheeled robots, as Player was designed to support update rates of the order of 5–100 Hz.[19,48]

3.2.2. Stage and Gazebo. Gazebo has 3D simulation capabilities and allows to implement computationally intensive simulations of rigid-body physics and realistic sensor feedback, but can handle a small robot population. In contrast, Stage provides fairly simple and computationally inexpensive models for many devices, rather than attempting to emulate them with great fidelity. The latter can be useful for simulating large populations of robots when dynamical accuracy is not sought.

Gazebo and Stage do not simulate specific devices such as a Sick® LMS-200 laser scanners or Pioneer robots from MobileRobots®. Instead they use abstract models, e.g. "laser" and "position." These abstract models are defined and configured in a world file. Often a model imitates a real device, for example, a "laser" specified with a 180° field of view and angular resolution equal to that of Sick® LMS-200. Available sensor models in Gazebo include sonars, scanning laser range-finders, Global Positioning System (GPS) and inertial measurement units (IMU), and monocular and stereo cameras. Robots models available include most commonly used robot types, such as Pioneer 2-DX, Pioneer 2-AT, and Segway® RMP.

3.3. Microsoft Robotics Developer Studio

MRDS[25] is a development system that provides functions to command some common mobile robotic platforms, e.g. iRobot, Lego NXT, MobileRobotics, as well as manipulator arms, e.g. Kuka and Lynx, among others. MRDS also provides simulation and virtual world visualization capabilities built on top of Nvidia's PhysX (ex-Ageia PhysX) dynamics engine. MRDS includes a tool called Visual Programming Language (VPL), which allows non-programmers to create applications by dragging and dropping blocks that represent services.

The programming infrastructure of MRDS is based on Microsoft's Coordination and Concurrency Runtime (CCR) and the Decentralized Software Services (DSS) component. The CCR provides a message-passing framework for asynchronous programming required to coordinate multiple sensors and actuators. It frees the programmer from having to manually implement threads, locks, and semaphores. DSS extends CCR's functionality by offering resources to coordinate services across a network and write applications made up of loosely coupled components.

3.4. Open Dynamics Engine

In addition to the features of ODE discussed in Section 2, it is possible to mention that ODE supports interactive and real-time simulation to the extent that it gives the user complete freedom to change the structure of the simulated system even while the simulation is running. ODE also has highly stable integrators so that simulation errors will not exceed expected limits or loose physical meaning. Other important features that have been key to its success are that it is supported in Linux, Mac, and Windows, and is licensed under the GNU LGPL and BSD models. The latter allows developers to create both free open-source programs and closed-source commercial applications. Furthermore, ODE can be programmed in C, C++, and in some other languages, such as Python, using appropriate wrappers, appealing to a wider audience of developers.

Open Dynamics Engine works through the interaction of four essential kinds of structures: World, Space, Body, and Geom. The "World" structure manages gravity and performs time integration, while the "Space" structure is contained in the "World" structure and optimizes collision detection. Structures of type "Body" are also contained in the "World" structure and have physical properties. Finally, structures of type "Geom" define collision detection and the rendering of "Body" structures.

A minor drawback of ODE is that determining contact points for collisions of objects modeled in terms of 3D triangle mesh structures, called "trimesh," is not easy to implement. Thus, using primitive objects, e.g. spheres, boxes, and cylinders, to construct more complex object is preferable. Another issue is that as a low-level physics engine, it can be integrated with different projects, but for users who need to develop complex worlds with ease, ODE has no built-in higher level features.

4. Selected Simulators' Quantitative Evaluation

Accuracy of the simulators is tested in terms of two experiment sets. The first set considers the fundamental physics compliance for purely longitudinal motion. The second set of experiments considers completing a square reference trajectory on a plane using an open-loop control command sequence of fixed duration.

The purpose of the first set of experiments is to determine whether the simulator is capable of producing results that are consistent with the theoretical longitudinal dynamics of a wheeled robot. Only Gazebo, MRDS, and ODE are compared with the ideal model trajectories. These simulators are also compared with the measurements obtained from the same experiment performed with a real robot. Carmen is not included in this first set of experiments because it only allows for 2D motion models. For the sake of a clear exposition, the longitudinal dynamics equations are presented in full detail in Section 4.1 before discussing the simulation results in subsequent sections.

The goal of the second set of experiments is to assess the simulators' accuracy and repeatability in the context of a simple trajectory following problem in which the robots are required to complete a square path. This experiment is carried out for the four simulators considered in this review.

It is to be noted that different simulators were tested using the same model parameters or as similar as possible whenever the simulator did not include an equivalent configuration option. For all simulators, the robot model employed corresponds to that of the MobileRobots® Pioneer 2-DX. Its dimensions (length, width, and height) are approximated to $0.4 \times 0.30 \times 0.20$ m and the robot's body weight is 21.5 kg. Distance between the wheels is 0.3 m and the wheels' diameter is 0.15 m.

4.1. Longitudinal motion dynamics of wheeled mobile robots

The equations describing the longitudinal motion dynamics of wheeled mobile robots or standard two-axle vehicles can be obtained by the application of Newton's Second Law for forces and moments. If the terrain along which the robot moves has varying slopes and possible discontinuous jumps (as illustrated in Fig. 4 in the experiments section), the equations must consider the piecewise combination of three motion situations: (a) when front and rear wheels are in contact with the terrain, (b) when front wheels have lost contact with the terrain, and (c) when the mobile robot is free falling. The vehicle model that we have developed considers the following standard facts and symplifying assumptions:[24,51]

1. The robot has the geometric mass and inertia parameters that are summarized in Table III and Fig. 3.
2. The robot–terrain interaction constraints can be expressed as contact, traction, and geometry constraints:
   (a) *Contact constraint*: The robot's suspension and wheels are rigid (non-deformable) and the terrain cannot be deformed nor penetrated. Furthermore, at points at which the terrain slope has a sudden change, the collision between the wheels and the dihedral formed by tangent planes at the point of change in slope is a perfectly inelastic collision (i.e. coefficient of restitution equal to zero), allowing the robot to track the terrain without bouncing back.
   This constraint implies that while the robot is in contact with the terrain, the vertical acceleration and velocity are zero, i.e. $\dot{v}_z = 0$ and $v_z = 0$, and that the robot tracks road slopes, i.e. the robot inclination angle θ is equal to the slope angle of the terrain. This assumption can be viewed as the robot being in static equilibrium along the terrain's normal, which is coincident with robot's $\mathbf{z}^r$-axis.
   (b) *Traction constraint*: The terrain reacts with wheel forces with a traction force that can be modeled using ideal Coulomb friction, in which the force is proportional to the friction coefficient μ and the down-force or load of the robot on the terrain (the robot's weight

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

Table III. Notation for motion equations of mobile robot.

Symbol	Description
Robot parameters	
m	Robot's mass
I	Robot's inertia moment
r	Wheel radius
h	Height of robot's center of gravity
l_f, l_r	Horizontal distances from the center of gravity to axles
$L = l_f + l_r$	Distance between robot axles
Robot local variables (body frame)	
$\mathcal{O}^r = \{\mathbf{x}^r, \mathbf{y}^r, \mathbf{z}^r\}$	System of longitudinal, lateral, and vertical robot axes
v_x, v_z	Longitudinal and vertical velocities in robot's body frame
θ	Robot pitch angle/road slope angle
ω	Angular velocity, i.e. $\omega = \frac{d\theta}{dt} = \dot{\theta}$
T_f, T_r	Front and rear traction forces
N_f, N_r	Front and rear normal forces
$F_{ax} = c_x \dot{v}_x$	Aerodynamic friction force along $\mathbf{x}^r$
$F_{az} = c_z \dot{v}_z$	Aerodynamic friction force along $\mathbf{z}^r$
$\tau_{a\theta} = c_\theta \omega$	Aerodynamic friction torque turning about θ
τ_{mf}, τ_{mr}	Front and rear wheels' motor torque
$\dot{\phi}_f, \dot{\phi}_r$	Front and rear wheels' angular velocity
μ_s	Wheel-ground static friction coefficient
μ_v	Wheels' viscous friction torque coefficient
Robot world variables (inertial frame)	
$\mathcal{O}^w = \{\mathbf{x}^w, \mathbf{y}^w, \mathbf{z}^w\}$	System of longitudinal, lateral, and vertical world axes
g	Gravity acceleration
γ	Road curvature angle
R	Road curvature radius
$\beta = \gamma/2$	Robot-to-ground traction angle

minus upward centrifugal acceleration forces). If the net load force becomes zero or negative, it can be assumed that the robot has lost contact with the terrain, and thus the traction force is zero. The traction force is also assumed to be zero if the wheels start to slip. The wheels can be assumed to be slipping when their torque is such that the tangential wheel-to-road force exceeds the road-to-wheel traction force. The slipping condition is maintained until the wheels' torque produces a wheel-to-road force below the nominal traction force. In other words, the kinetic friction is assumed to be zero independent of the sliding velocity. Modeling the transition from static to a non-zero kinetic friction is left out, because the simulators do not take into account kinetic friction, and because precise slipping models require in turn complex wheel models considering contact patch geometry and wheel deformation, among other aspects.

(c) *Geometry constraint*: If the road is parametrized as a curve $c(s) \stackrel{\text{def}}{=} (f(s), g(s)) \equiv (x, y(x)) \in \mathbb{R}^2$, where s is the free parameter, the road slope can be obtained as $\theta = \text{atan}(\frac{\partial g}{\partial s} / \frac{\partial f}{\partial s}) = \text{atan}(dy/dx)$. Thus, $\omega = \dot{\theta} = [1/(1 + (dy/dx)^2)] \cdot (d^2y/dx^2) \cdot (dx/dt)$. For the vehicle rolling on the curve, its longitudinal velocity v_x satisfies $v_x = |dc/dt| = dx/dt * |(1, dy/dx)|$, therefore $dx/dt = v_x/\sqrt{1 + (dy/dx)^2}$. Hence, $\omega = v_x/R$, where $R = (1 + (dy/dx)^2)^{3/2}/(d^2y/dx^2)$ is the road curvature.

3. Due to relatively low speeds, aerodynamic forces only consider drag, while aerodynamic lift forces and their effect on traction can be neglected.

The next three sections present the derivation of motion equations taking into account the stated assumptions.

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

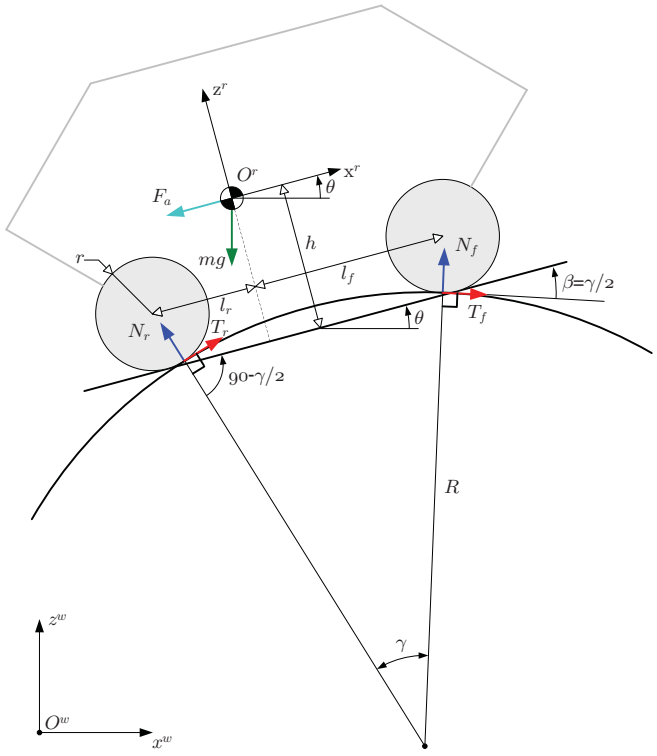


Fig. 3. Forces acting on mobile robot moving along an inclined non-flat road with local slope θ and curvature R.

4.1.1. Standard longitudinal motion. The summation of forces along the $\mathbf{x}^r$ and $\mathbf{z}^r$ axes, and moments about the center of gravity are given by

$$\begin{aligned}\sum F_x: \; m\dot{v}_x - m\omega v_z \;&=\; T_f\cos(\beta) + T_r\cos(\beta)\\ &\quad -\delta N_f\sin(\beta) + \delta N_r\sin(\beta) - F_{ax} - mg\sin(\theta), \qquad (1)\\ \sum F_z: \; m\dot{v}_z + m\omega v_x \;&=\; \delta T_f\sin(\beta) - \delta T_r\sin(\beta)\\ &\quad +N_f\cos(\beta) + N_r\cos(\beta) - mg\cos(\theta), \qquad (2)\\ \sum M_y: \; I\dot{\omega} \;&=\; T_f\cos(\beta)h + T_r\cos(\beta)h - \delta T_f\sin(\beta)l_f - \delta T_r\sin(\beta)l_r\\ &\quad -\delta N_f\sin(\beta)h + \delta N_r\sin(\beta)h + N_f\cos(\beta)l_f - N_r\cos(\beta)l_r, \qquad (3)\end{aligned}$$

where $\delta = \mathrm{sgn}(R)$ is the sign of the road's curvature radius R. Since for a forward moving vehicle on a dip $v_x > 0$ and $\omega > 0$, $R = \frac{v_x}{\omega} > 0$ and $\delta = 1$, whereas $v_x > 0$, $\omega < 0$, $R < 0$ and $\delta = -1$ for a vehicle on a crest. It is to be noted that $\dot{v}_x$ and $\dot{v}_z$ on the left-hand side of Eqs. (1) and (2), respectively, correspond to the inertial acceleration components of the robot's body frame $\mathcal{O}^r$ relative the stationary frame $\mathcal{O}^w$ (expressed along the coordinate axes of the reference frame $\mathcal{O}^r$), while the terms containing $-\omega v_z$ and ωv_x are the components of centrifugal acceleration, which must be considered because the robot's body frame $\mathcal{O}^r$ is a rotating (non-inertial) frame. In fact, denoting the standard unit basis vectors of the rotating frame $\mathcal{O}^r$ as $\mathbf{i}^r$, $\mathbf{j}^r$, and $\mathbf{k}^r$, the robot's angular velocity can be expressed as $\Omega = -\omega\mathbf{j}^r$, while its translational velocity is expressed as $\mathbf{v}^r = v_x\mathbf{i}^r + v_z\mathbf{k}^r$, and thus the centrifugal acceleration is $\Omega \times \mathbf{v}^r = -\omega v_z\mathbf{i}^r + \omega v_x\mathbf{k}^r$.

Equations (1)–(3) can be simplified by noting that the robot-to-ground traction angle $\beta \approx 0$ when $R \gg L = l_f + l_r$. Using this approximation, which is valid for regular wheeled robots on standard

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

roads,[24] the total traction is obtained from (1) as:

$$T_f + T_r = m\dot{v}_x - m\omega v_z + F_{ax} + mg\sin(\theta). \tag{4}$$

Replacing the total traction force into (3) with the assumption that $\beta = 0$ yields:

$$N_f l_f - l_r N_r = I\dot{\omega} - (m\dot{v}_x - m\omega v_z + F_{ax} + mg\sin(\theta))\, h. \tag{5}$$

Equation (2) simplified with $\beta = 0$ can be employed together with (5) to solve for the normals N_f and N_r:

$$N_f = \frac{1}{L}\left[mg\cos(\theta)l_r - (m\dot{v}_x - m\omega v_z + F_{ax} + mg\sin(\theta))h + m(\dot{v}_z + \omega v_x)l_r + I\dot{\omega}\right] \tag{6}$$

$$N_r = \frac{1}{L}\left[mg\cos(\theta)l_f + (m\dot{v}_x - m\omega v_z + F_{ax} + mg\sin(\theta))h + m(\dot{v}_z + \omega v_x)l_f - I\dot{\omega}\right] \tag{7}$$

The previous equations for the wheels' normals can be further simplified under the additional assumption that the angular acceleration as well as the vertical velocity and acceleration are negligible, i.e. $\omega = 0, v_z = 0, \dot{v}_z = 0$ on regular roads without abrupt bumps or potholes, and due to the previously stated assumptions. Replacing $\omega = v_x/R$, where R is the signed curvature radius, in the simplified equations for the normals makes it clearer that the normal forces decrease due to the centrifugal acceleration acting along the positive $\mathbf{z}^r$-axis for a robot cresting on a hill ($R < 0$), while the normal forces increase for a dipping robot on a slope with positive curvature radius ($R > 0$). It is also to be noted that during acceleration the load is transferred from the front to the rear axle in proportion to the acceleration.

Calculating the normal forces N_f and N_r is necessary to determine whether the wheel slipping condition has been reached. A general theory that could accurately predict the relationship between the tractive efforts and the wheel slipping has not yet been fully developed,[51] and on the other hand, the simulators considered in this paper only employ the classic Coulomb's static friction model. Hence, under this model the *net traction force* is given by:

$$T_i = \begin{cases} \frac{1}{r}\left(\tau_{mi} - \mu_v \frac{v_x}{r}\right), & \frac{1}{r}\left(\tau_{mi} - \mu_v \frac{v_x}{r}\right) \le \mu_s N_i \\ 0, & \frac{1}{r}\left(\tau_{mi} - \mu_v \dot{\phi}_i\right) > \mu_s N_i \end{cases}, \quad i = f, r, \tag{8}$$

where τ_{mi} is the motor torque on the wheel, $\mu_v \frac{v_x}{r}$ is the loss of torque due to viscous friction, and $\dot{\phi}_i$ is the wheel angular velocity. Note that $\dot{\phi}_i = \frac{v_x}{r}$ when the wheels are rolling without slipping, while $\dot{\phi}_i$ can be larger than $\frac{v_x}{r}$ when the wheels are slipping. The model for the driving traction force in (8) basically states that the force moving the robot is the tangential force exerted by the wheels' torque as long as this force is less than the maximum friction force $\mu_s N_i$. When the maximum friction force is exceeded, the only force affecting the rate of change of robots' longitudinal velocity v_x is the external aerodynamic drag force F_{ax} and the force of gravity if the robot is on a road with slope $\theta \neq 0$. For other available models considering the nonlinear nature of the tractive effort and longitudinal slip, the reader is referred to refs. [24, 51].

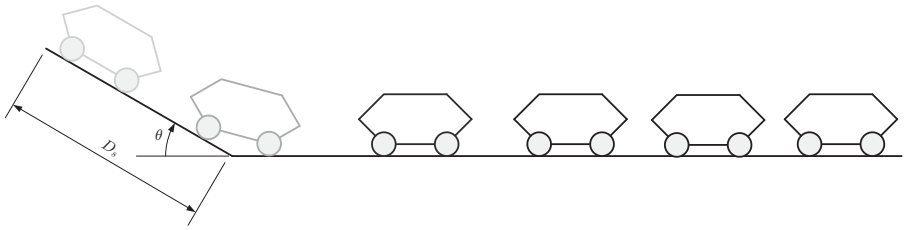
4.1.2. Transition from standard to free fall motion. If the front wheels loose contact with the ground due to a sudden change in the terrain, then $N_f = 0$, $T_f = 0$, and the robot will behave as an inverted pendulum with a pivot at the rear wheels' axle. Hence, when the front wheels are not in contact with the terrain, the equations of motion (1)–(3), with the standard assumption that $\beta \approx 0$, reduce to:

$$\sum F_x : \; m\dot{v}_x = T_r - F_{ax} - mg\sin(\theta) + m\omega v_z, \tag{9}$$

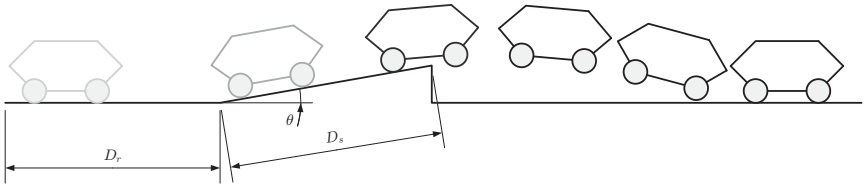
$$\sum F_z : \; m\dot{v}_z = N_r - mg\cos(\theta) - m\omega v_x, \tag{10}$$

$$\sum M_y : \; I\dot{\omega} = T_r h - N_r l_r. \tag{11}$$

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press



(a) Downhill-roll experiment.



(b) Ramp-jump experiment.

Fig. 4. Longitudinal motion experiments.

Rearranging the previous equations, then

$$T_r = m\dot{v}_x + F_{ax} + mg\sin(\theta) - m\omega v_z, \tag{12}$$

$$N_r = m\dot{v}_z + mg\cos(\theta) + m\omega v_x, \tag{13}$$

$$I\dot{\omega} = (m\dot{v}_x + F_{ax} + mg\sin(\theta) - m\omega v_z)h - (m\dot{v}_z + mg\cos(\theta) + m\omega v_x)l_r. \tag{14}$$

4.1.3. Free fall motion. Immediately after the robot looses ground contact, e.g. after jumping off a ramp, it becomes airborne and its motion is governed by the equations of a free falling body. Under this situation, the only external forces are the force of gravity and the aerodynamic friction, thus the force and moment balance equations in the robot reference frame are given by:

$$m\dot{v}_x = -mg\sin(\theta) - F_{ax} + m\omega v_z, \tag{15}$$

$$m\dot{v}_z = -mg\cos(\theta) - F_{az} - m\omega v_x, \tag{16}$$

$$I\dot{\omega} = -\tau_{a\theta}, \tag{17}$$

where $F_{ax} = c_x v_x^r$, $F_{az} = c_z v_z^r$, and $\tau_a = c_\theta \omega$ are the aerodynamic resistance forces.

Robot velocities in the coordinates of inertial frame can be obtained by transforming robot velocities in body coordinates using the standard rotation matrix $\mathbf{R}_\theta = [\cos(\theta), -\sin(\theta); \sin(\theta), \cos(\theta)]$.

4.2. Experiment set 1: longitudinal motion

This first set of longitudinal motion experiments is divided into the following: (i) a ramp descent experiment (Fig. 4(a)), and (ii) a ramp-jump experiment (Fig. 4(b)). The results obtained with each simulator are compared with the results of the theoretical model derived in the previous section and the measurements obtained with a real robot.

4.2.1. Results obtained with the simulators. The purpose of the ramp descent experiment is to confirm the consistency of the implementation of frictional and gravitational forces in the simulator. In this experiment, the robot brakes are released and the robot is allowed to roll freely downhill until frictional forces stop it. The ramp slope is $\theta = -30^\circ$, and its length is $D_s = 2$ m (see Fig. 4(a)). This experiment was repeated 30 times and the averaged results are presented in the first column of Fig. 5 and Table IV. The downhill-roll experiment does not include results for MRDS, because, as indicated in online forums, for that simulator it is not possible to set robot's wheels to rotate freely. For MRDS, if the motors are turned off, then their state is blocked and will not turn. If the motors are turned on

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

Table IV. Results of the downhill-roll experiment.

Software	Average final position (m)	deviation deviation
Ideal model	3.53	–
Gazebo	2.63	0.05
ODE	3.53	9.03×10^{-16}

Table V. Results of the ramp-jump experiment.

Software	Average final position (m)	Standard deviation
Ideal model	6.73	–
Gazebo	6.41	0.02
ODE	6.72	9.0×10^{-16}
MRDS	4.78	1.0×10^{-6}

and wheel velocities are set to zero, an internal control loop prevents the wheels from turning despite the torque being set to zero. Hence, the simulated robot in MRDS stays fixed at a location on the ramp, as shown in Fig. 6(a).

The first column in Fig. 5 shows the time variation of longitudinal and vertical positions of robot in world coordinates x^w, z^w, and its pitching angle θ. The last row in Fig. 5 shows robot's position in the $\overline{\mathbf{x}^w\mathbf{z}^w}$ plane, which corresponds to robot's traversed trajectory over the ramp, and thus the curve follows the geometry of downhill ramp. The resulting longitudinal position obtained with ODE for the downhill moving robot follows closely to that of the ideal model, unlike the one obtained with Gazebo. This variation is due to the different friction model in Gazebo and the non-uniform time-stepping produced by communication latencies, which, in turn, affect the solution of dynamic equations. In fact, the variability of the solutions obtained with Gazebo is quite large compared with that of the solutions obtained with other simulators, as shown in Table IV.

The descent curve of z^w versus time in Fig. 5(b) shows that the force of gravity driving the robot downhill is consistent in all simulators, since the time at which the bottom point of the ramp is reached is about 1 s. It is easy to verify that this is consistent with a simplified point mass model without friction. If the only acceleration acting on the body is that of gravity, and the body is constrained to move on an inclined plane of 2-m length with slope $\theta = -30^\circ$, then the time at which the body reaches the ground is given by $t = \sqrt{\frac{4}{g\cos(\pi/3)}} = 0.9$ s. If viscous friction acts on wheels' axles, then the time at which the ground is reached will naturally be greater than the last value.

While the position accuracy resulting with ODE is acceptable, it is to be noted that robot's pitching angle θ does not follow road's slope very well. None of the simulators performed very well in this aspect, one possible cause for this result is that the methods employed by the simulators to check collision and geometric constraints are adapted for fast visually realistic computations rather than accurate surface following. Improving this aspect can be crucial for accurate studies of robot–ground interaction and the effects on the stability of the transported load.

The second longitudinal motion experiment comprises driving the robot with maximum torque and making it jump a 1-m long ramp that has a 10° slope (see Fig. 4(b)). Once the robot looses contact with the ground, the motors are turned off and allowed to rotate freely. The choice of a 10° slope is of practical nature. First, regular research robots do not have much power to climb steeper inclines with full payload. Second, a 10° slope corresponds to an almost 20% slope, which is more than twice the standard slope of regular roads. Current standards vary widely, but in general maximum highway grades must be below 10%, while driveways can have grades of as much as 30% for short distances. The purpose of the jump experiment is to evaluate simulators' performance under a discontinuous process and their ability to manage various modes of dynamics explained in Section 4.1.

The averaged results for 30 runs of the ramp-jump experiment are summarized in the second column of Fig. 5, and in Table V. The results in the second column of Fig. 5 include the ideal (theoretical) solution computed using the model equations of Section 4.1.1, and a simplified solution,

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

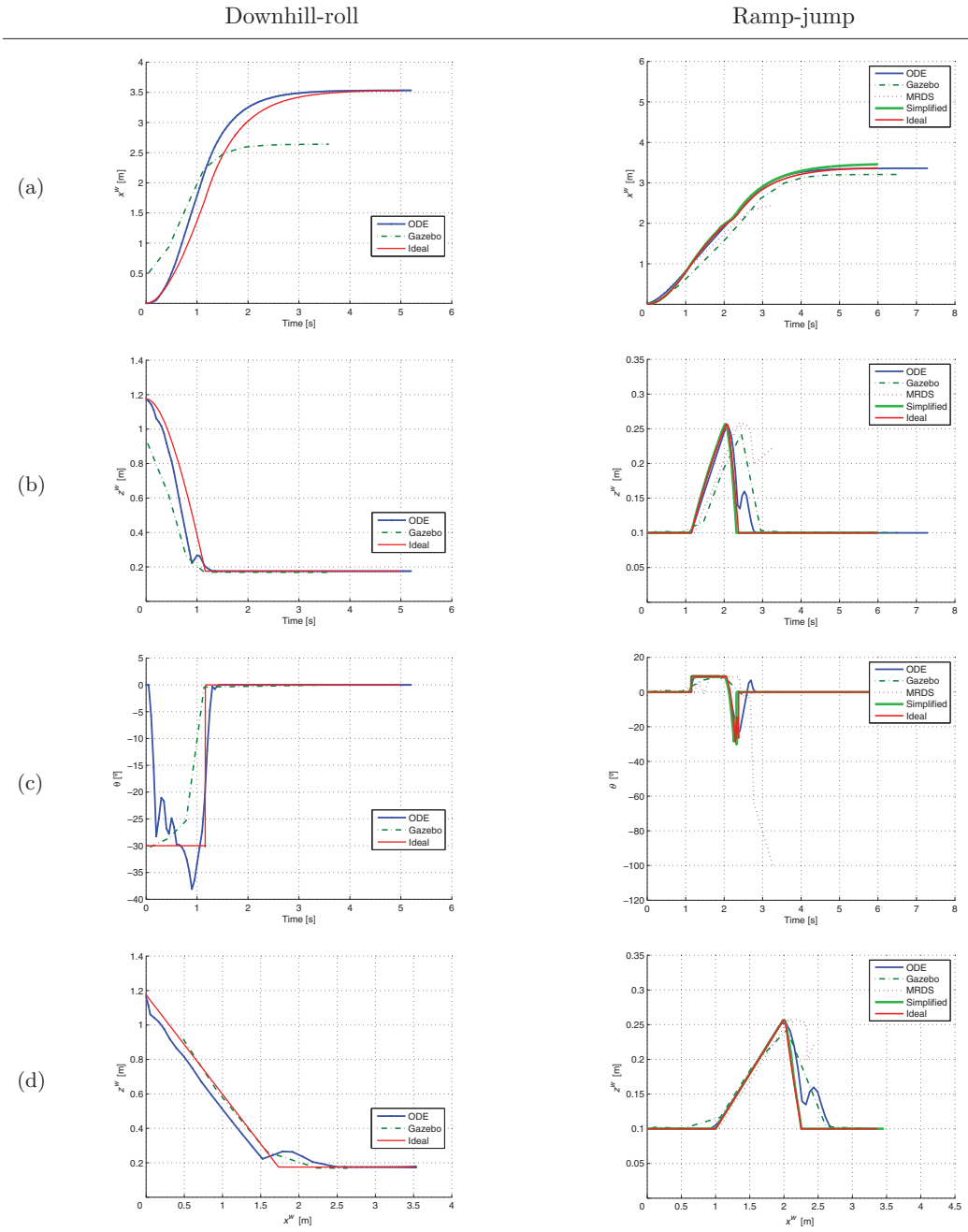


Fig. 5. Downhill-roll and ramp-jump experiments: (a) x^w vs. t, (b) z^w vs. t, (c) θ vs. t, (d) z^w vs. x^w.

which corresponds to the solution obtained by ignoring the Coriolis forces. Neglecting these forces is not correct from the modeling point of view, but when the angular velocity ω and the velocity v_z are small, the Coriolis forces have a negligible effect. It is to be noted that while the robot is in contact with the ground, $v_z = 0$, thus the Coriolis term ωv_z is zero and the resulting ideal and simplified trajectories differ slightly after the jumping point, as can be seen in Fig. 5. This difference becomes more significant when the falling height and motion speed increase. The MRDS pitch angle θ in Fig. 5 shows that the robot rolls over and stops moving after the jump. This behavior is not present in other simulators. The most likely causes of this behavior are related to slight differences in the location of the center of mass of robot's representation and differences in terrain friction and collision models in spite of efforts to make all parameters as equivalent as possible. It is also to be noted that with the exception of the MRDS simulation, in which the robot rolls over shortly after it falls from

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

the ramp, in all other simulators the robot comes to a stop at about 3.2–3.4 m due to the action of aerodynamic and viscous friction forces acting on it (see Fig. 5). Another drawback of MRDS is that robot's command speed is limited to 1 m/s. Setting higher speeds in MRDS would be possible, but is not straightforward because it would require that the user creates a new robot model component by rewritting part of the functions available with MRDS.

Once again, the results show that ODE follows the ideal model trajectory with a surprising degree of accuracy, especially as far as the longitudinal position in time is concerned. The curves for z^w versus time show that the simulators produce results according to the expected theoretical behavior. In the case of MRDS and ODE, a bounce is also observed. This is explained by the fact that MRDS and ODE also consider collision effects, while the simplified ideal model assumes inelastic collisions with coefficients of restitution equal to zero, i.e. the robot effectively stops on the surface without bouncing at all. The poorest performance is exhibited by Gazebo, especially considering that the evolution of the pitching angle in time does not correspond to the true pitch when the robot is on the ramp and after it falls from the ramp.

Another interesting aspect of the jump experiment is the observed behavior of robot as its front wheels loose contact with the ground. When this occurs, the robot starts to rotate about the rear wheel, pitching down toward the ground. Without aerodynamic friction, the robot would permanently spin until hitting the ground, provided that the distance to the ground is large enough. If the distance to the ground is not so large (e.g. 0.15 m as in this simulation), then robot's pitch angle θ decreases by almost 40° and levels to zero when the robot hits the ground. For a small robot such as the Pioneer 2-DX, hitting the ground at an angle of 40° may be enough to damage it. In this sense, an accurate simulation of the robot motion becomes a valuable tool to help determine the kind of roads that the robot can safely traverse.

4.2.2. Results obtained with the real robot. To validate the simulated results, the same downhill-roll and ramp-jump experiments were repeated for 12 times each with a real robot (see Fig. 6(b)) to obtain a statistically significant amount of data. The downhill ramp considered a length $D_s = 2$ m and a slope angle $\theta = -30°$ as in the case of simulations, while the ramp-jump experiment was carried out on an incline of $D_s = 1$ m and a slope angle $\theta = 10°$. This ramp has the same slope angle as the one used in the simulations, but was shortened in length to reduce the height of the fall from 34.7 cm to 17.3 cm in order to minimize further damages to robot's wheel axles and gearheads which occurred during the experiments.

Robot's position was measured using a calibrated camera and a visual tracking system that was programmed using OpenCV to perform foreground segmentation of a moving object and follow robot's centroid. The visual tracking system yielded measurements at 29 frames per second with a resolution of 7.4 mm per pixel and an error with a standard deviation of 68.7 mm, which is comparable with that of industrial laser trackers in the range of US$10–20k and is an acceptable error level for the purpose of validation. The robot was also equipped with a combined gyroscope and inertial measurement unit (gyro-IMU) Crossbow RGA-300CA. The gyro-IMU measurements provided more reliable measurements of robot's pitch orientation than those obtained with the visual measurement system. The gyro-IMU measurements were also integrated to compute odometric estimates of robot's position and velocity as additional comparison variables, in spite of the fact that gyro-IMU biases and shock-induced disturbances on internal MEMs of gyro-IMU result in cumulative errors that render the odometric position estimates less reliable than the measurements directly obtained from the visual measurement system.

Comparing the visual and gyro-IMU measurements presented in Figs. 7 and 8 with the simulated results in Fig. 5, it is possible to observe from the visual measurements in Fig. 7 that the behavior of the real robot is consistent with that of the simulated results. In fact, the averaged visual position measurements curve shows that the robot's vertical position z^w (Fig. 7(b)) reaches the floor in approximately 1.2 s closely matching the time predicted by the ideal model. The horizontal position traveled by the robot after 2 s is about 3.7 m (Fig. 7(a)), while the ideal model shows a traveled distance of about 3.0 m. In the experiments with the real robot it was observed that it would come to a stop at about 5.5 m, while the theoretical model and the simulations show that the robot stops at 2.5–3.5 m. This is due to the fact that we estimated from the motors and bearing data sheets larger friction coefficients than the ones that the real robot seems to have. The trajectory in the $\overline{\mathbf{x}^w\mathbf{z}^w}$ plane shown in Fig. 7(c) is consistent with ramp's geometry and the simulated z^w versus x^w trajectory. The

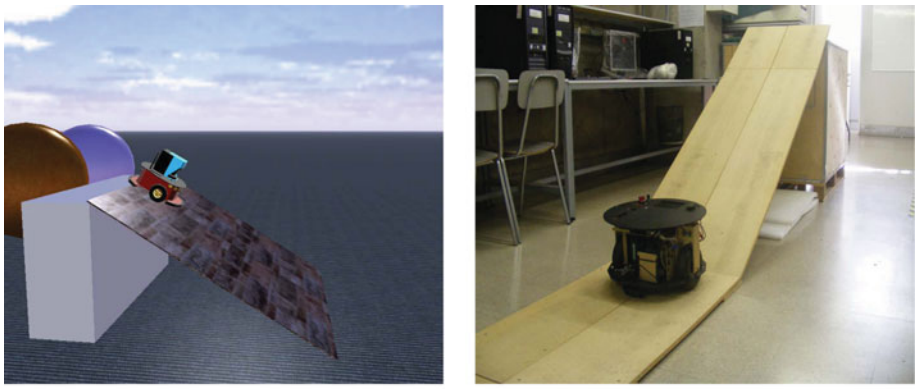
https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

curve in Fig. 7(c) shows a spike of almost 20 cm at position $x^w = 2.3$ m. Significantly, the downhill curves of the ODE simulator in Figs. 5(b) and (d) also show a similar bounce when the robot impacts the ground plane. In this sense, ODE is more consistent physically than other simulators with respect to what actually happened during the experiments.

The results in the second column of Fig. 7 for the ramp-jump experiment are also congruent with the simulated results. The apparently high level of noise in the ramp-jump measurements is due to the robot bouncing on the ground rather than pure noise in electronics. There is also an increase of variability in the position measurements just before the robot starts to climb the ramp, as can be seen in Figs. 7(b) and (c). Again, this is produced by the collision of robot with ramp and the fact that the slope abruptly changes from 0° to 10°. Moreover, the springiness of the air-filled tires accentuates the observed acceleration spikes before and after the ramp (see Figs. 7 and 8(a)). Once again ODE shows partially the bouncing effect after the robot falls from the ramp, but no significant changes are visible prior to climbing the ramp. The visual measurements were also used to compute the velocity of the robot in both experiments, yielding values that were consistent with the expected values obtained from the simulation. The real robot moves downhill at a speed around 2 m/s, which is similar to robot's speed in the corresponding simulation. Similarly, the simulated ramp-jump experiment and the simulation results show that the robot takes roughly 1.5 s to climb the ramp (between time instant 1 s and 2.5 s in the experiment with the real robot; see Fig. 7(c)) and an extra second to roll off the jump (between time instant 2.5 s and 3.5 s).

The measured pitch angles shown in Fig. 8(a) are comparable with the ones shown in the simulation results in Fig. 5(c). The pitch angle measurements by the gyro-IMU unit can also be estimated considering that the acceleration vector components in the robot's coordinate frame are given by $\dot{v}_x \approx a_f + \omega v_z - g\sin(\theta)$, $\dot{v}_z \approx -\omega v_x - g\cos(\theta)$, where g is the gravity acceleration and a_f is the net tractive acceleration. Since $|a_f| \ll |g\sin(\theta)|$ whenever the robot's friction and velocity are small, in the case of this robot a_f can be assumed to be negligible and the ratio $(\dot{v}_x - \omega v_z)/(\dot{v}_z + \omega v_x) \approx \sin(\theta)/\cos(\theta)$ allows to compute the pitch angle as $\theta \approx \text{atan}((\dot{v}_x - \omega v_z)/(\dot{v}_z + \omega v_x))$. Furthermore, for most of the constant slope ramp $\omega = 0$ and the pitch angle simplifies to $\theta \approx \text{atan}(\dot{v}_x/\dot{v}_z)$. This approach is employed by the gyro-IMU because the pitch curve plotted using the linear accelerations $\dot{v}_x$ and $\dot{v}_z$ turned out to be the same as the one obtained directly from the sensor output, which is shown in Fig. 8(a). The relation between θ and the robot accelerations $\dot{v}_x$, $\dot{v}_z$ explains why the pitch measurements obtained with IMU show a negative spike in θ before the robot starts climbing the 10° slope. The deceleration in the longitudinal axis, i.e. $\dot{v}_x < 0$, which slows down the robot when it hits the inclined plane, creates artifact in the pitch curve. Some other spikes in the measured pitch curves are attributed to robot's bounce. These sudden changes in the measured θ, either due to the way the pitch is computed or because of robot's bounces, are one of the factors that spoil the computation of robot's velocities and positions in world coordinates by integrating its acceleration components projected on the $\mathbf{x}^w$ and $\mathbf{z}^w$ axes. These acceleration components are given by $a_x^w = \dot{v}_x\cos(\theta) - \bar{\dot{v}}_z\sin(\theta)$, $a_z^w = \dot{v}_x\sin(\theta) + \bar{\dot{v}}_z\cos(\theta)$, where $\bar{\dot{v}}_z$ is calculated from the measured vertical acceleration $\dot{v}_z$ as $\bar{\dot{v}}_z = \dot{v}_z + g\cos(\theta)$, in order to suppress the gravitational acceleration component in the gyro-IMU measurement that for the most part of robot's trajectory is cancelled by the ground plane restriction. In other words, for most of the flat road with $\beta = 0$ and $\omega = 0$, the robot's weight and the normals at the points of contact cancel each other, and therefore any acceleration orthogonal to the ground plane whose magnitude differs from $g\cos(\theta)$ is caused by a change in slope or temporary loss of ground contact (see Eqs. (1)–(3)). To the contrary, the gravitational acceleration component should not be subtracted from the gyro-IMU measurment when the robot is in free fall motion or has temporarily lost ground contact due to bumps that make it bounce. The difficulty in correctly discriminating the situation when the gravity acceleration component that is orthogonal to the ground plane does not produce motion from the situation when it has to be integrated is one of the causes for an inaccurate odometric measurement of robot's positon. This especially occurs along the $\mathbf{z}^w$ axis of the world coordinate frame due to errors in the estimation of robot's true motion-producing vertical acceleration, and consequently of the projected acceleration component along the $\mathbf{z}^w$ axis. Other sources of errors that deteriorate the odometric estimates are the well-known sensor biases and drifts, which cannot be fully eliminated despite calibration, and the errors produced by the internal construction characteristics of the MEMs-based gyro-IMU. The MEMs-based gyro-IMU employed in the experiments exhibits cross-coupling between the measurements of its three supposedly independent orthogonal axes, thus a strong shock or collision along one of its axes appears as a measurement disturbance in the remaining

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press



(a) (b)

Fig. 6. Downhill-roll experiment: (a) visualization in MRDS, (b) real robot in the laboratory.

axes due to imperfect alignment of internal MEMs. In contrast with the the visual measurements, which were very consistent with the simulated curves, the odometric measurements obtained with the gyro-IMU are inaccurate but provide valuable insight into the effects of mechanical shock due to the non-smooth interaction between the robot and the ground, as is particularly clear in the pitch measurements in Fig. 8(a).

4.3. Experiment set 2: planar square trajectory

In this experiment, the robot was programmed to perform a 3×3 m square-shaped trajectory under open-loop control in a simulated world area of 5×5 m. To this end, the translation velocity was set to 0.1 m/s and the rotation velocity was set to $\omega = (v_R - v_L)/d_w$, where $v_R = -v_L = 0.1$ m/s are the right and left wheel translational velocities, and $d_w = 0.4$ m was the distance between the wheels (i.e. rotation about robot's $\mathbf{z}^r$-axis at 0.5 rad/s). In each case, the robot was instructed to run for a set amount of time until it reached the first corner of the square shape, then it performed a rotation for a set amount of time until a 90° turn to the left was completed, and then it was instructed to run to the next corner and so on. For each robot we performed 30 runs, from these we computed and plotted the trajectories shown in Fig. 9.

The simulators are compared in terms of trajectory accuracy, simulation precision, and computation time, which is the CPU time required to complete the trajectory of the second experiment. The time is due to the CPU load caused by the simulator and the experiment being executed. In all the simulations, as well as with the real robot, the sampling period was programmed to be as close as possible to 0.1 s, while the simulator integrators time-stepping was fixed at 0.01 s, in order to ensure similar computational burdens and a fair comparison. However, it is to be noted that the sampling period uniformity cannot be fully guaranteed across simulators because of the differences in their architecture and implementation aspects ranging from collision detection and handling schemes to higher level message passing and interprocess communication functionality. The computation time is compared with the actual time; it would take an ideal robot to complete the 3×3-m square trajectory with a linear acceleration limit of 0.1 m/s^2, a maximum angular acceleration of 5 rad/s^2, and the specified reference velocities. With these parameters, the ideal robot should take about 134.2 s to complete the trajectory. In comparison, the real robot took on average 133.4 ± 2.1 s to complete the square path according to our timing of each lap of the experiment. The ratio between the computation time and the time the real system takes to perform the same simulated action is the so-called real-time factor, which is a common metric for measuring the capability of a system to perform and respond in time to the data requests and commands that must be issued to the real system. The trajectory accuracy is measured in terms of end-point error. When traversing a square path, cumulative simulation errors become evident due to two facts. First, the rotation angles which should be 90° have some error, which causes the path to not be perfectly square. Second, the traversed distance along each side of the square is not exactly 3 m. Moreover, when the robot moves in one direction, the length of the path is not the same as that when the robot moves in the opposite direction, thus resulting in a final position, which

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

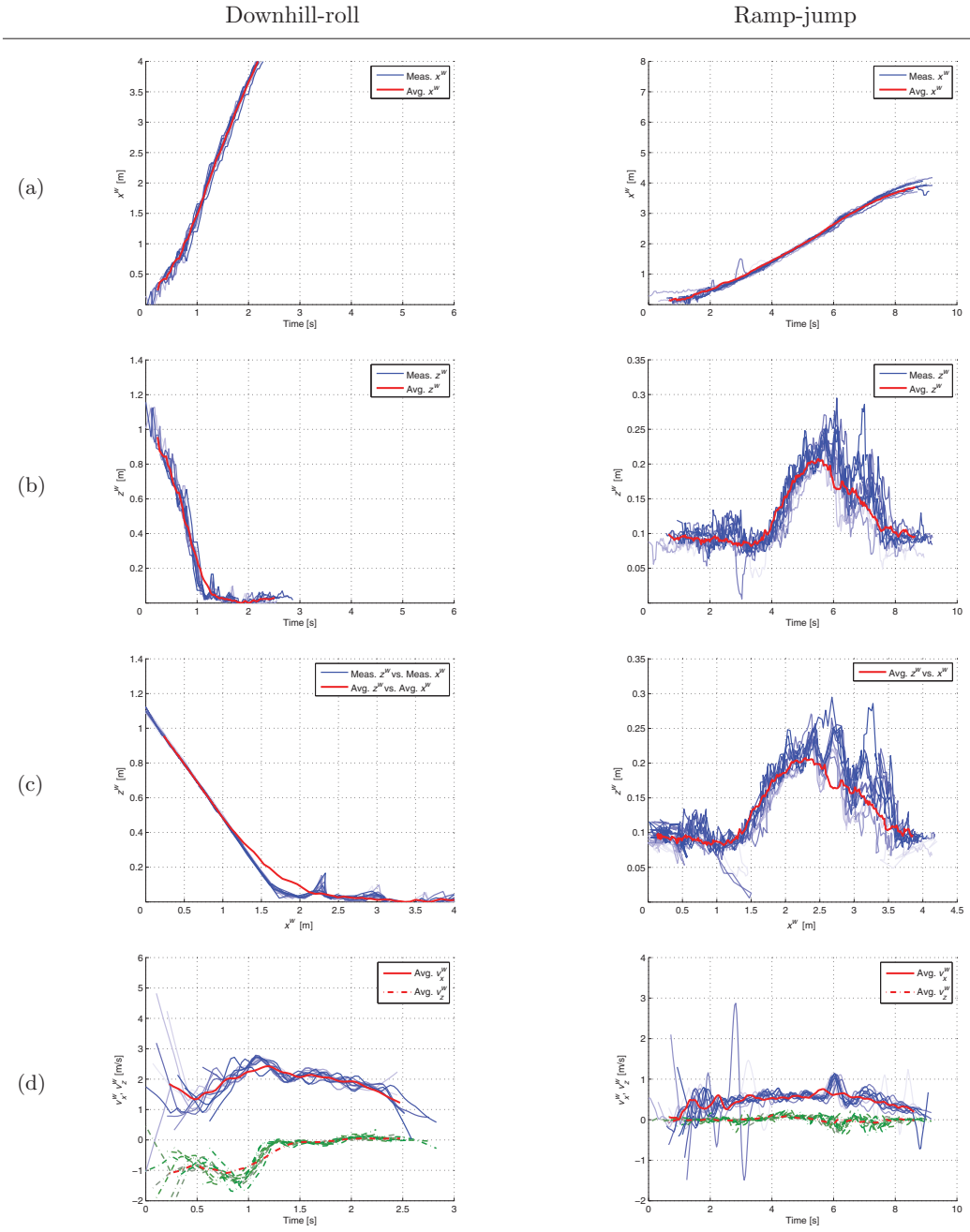


Fig. 7. Visual measurements of the downhill-roll and ramp-jump experiments with a real robot: (a) x^w vs. t; (b) z^w vs. t; (c) z^w vs. x^w; (d) v_x^w, v_y^w vs. t. Red curves correspond to smoothed averages using a 0.3-s moving window.

does not exactly match the starting point. The larger the cumulative errors in translation and rotation, the larger the error between the final and initial points of trajectories. Therefore, the motion model and simulator accuracy can be assumed to be proportional to the end-point error and simulation precision can be computed directly from the standard deviation of the simulated trajectories endpoints. It is to be noted that in the simulations we are considering a disturbance free world and an open-loop controller, therefore the end-point error should not be interpreted as controller's failure, but rather how well wheel slippage and dynamics effects are reproduced by the simulator relative to the real robot behavior. As a matter of fact, the real robot trajectories in Fig. 9(e) show that the real robot end-point error is very consistent with the values computed by Gazebo, ODE, and Carmen. On the

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

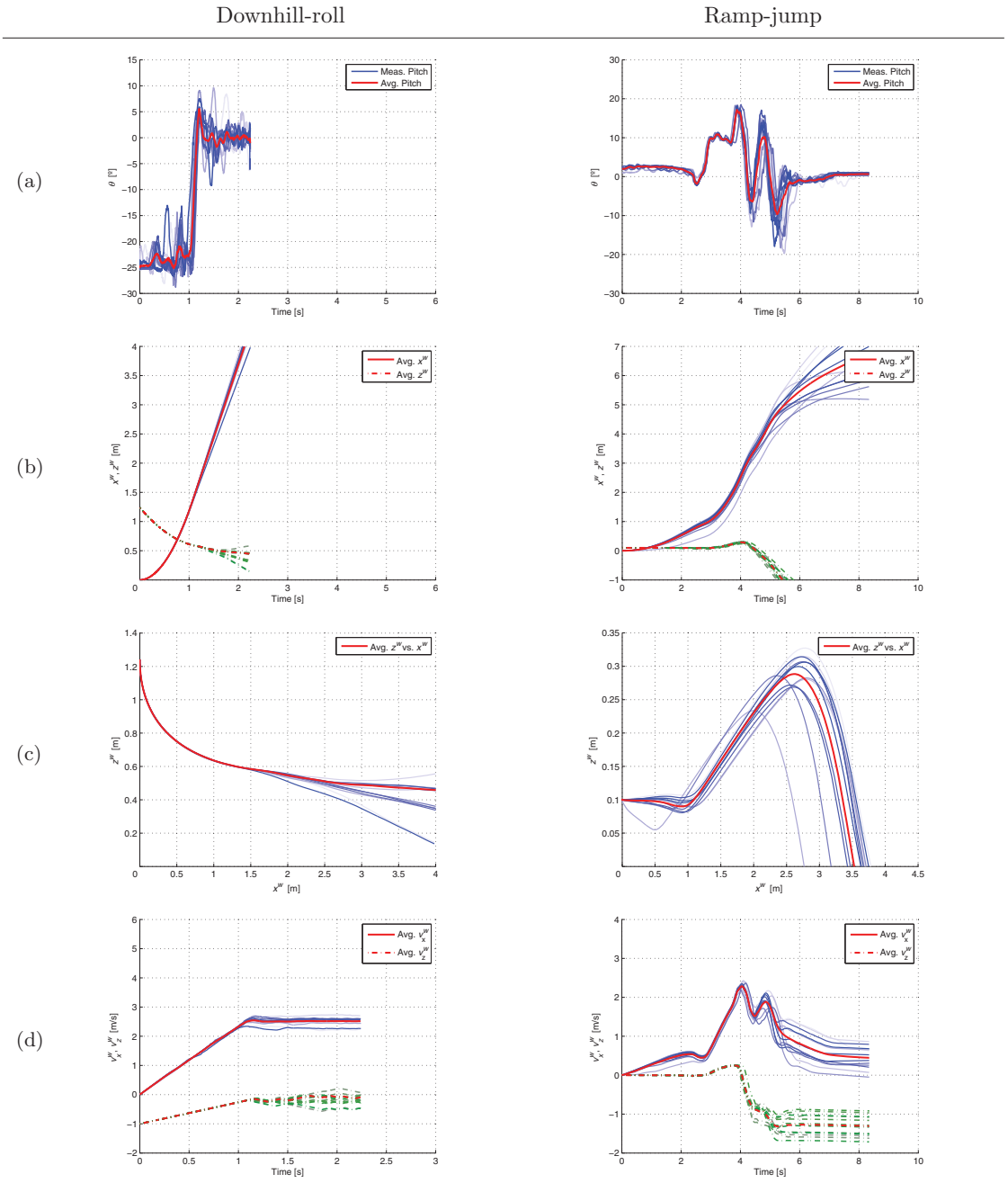


Fig. 8. Gyro-IMU odometric measurements of the downhill-roll and ramp-jump experiments with a real robot: (a) θ vs. t; (b) x^w, z^w vs. t; (c) z^w vs. x^w; (d) v_x^w, v_y^w vs. t. Red curves correspond to smoothed averages using a 0.3-s moving window.

other hand, computing the standard deviation of the end-point error provides a measure of simulator's repeatability. Low standard deviations under the same simulation conditions imply that the simulator results are more reliable and not affected by the way the different aspects are internally handled by each simulator. Since the simulations are carried out under controlled constant conditions, the standard deviation should be ideally close to zero for the simulators, unlike the real robot which is subject to actuators and sensors' noise, and more noticeably, the wheel slippage disturbances, which combined with the measurement noise of the visual tracking system yield a moderate end-point standard deviation for the 30 runs of the experiment.

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

Table VI. Results for the square trajectory experiment.

Software	Endpoint position RMS error (m)	Endpoint std. dev. σ_x (m)	Endpoint std. dev. σ_y (m)	Endpoint std. dev. σ_θ (°)	Processing time (s)	real-time factor
Carmen	0.2900	0.0089	0.0115	3×10^{-17}	141.79	1.06
Gazebo	0.2407	0.1142	0.1719	2.3616	71.47	0.53
MRDS	0.0376	0.0102	0.0065	0.0054	60.80	0.45
ODE	0.2079	5.6836×10^{-9}	7.0×10^{-15}	3.0×10^{-15}	81.30	0.61
Real robot	0.2260	0.1563	0.1582	3.1208	–	–

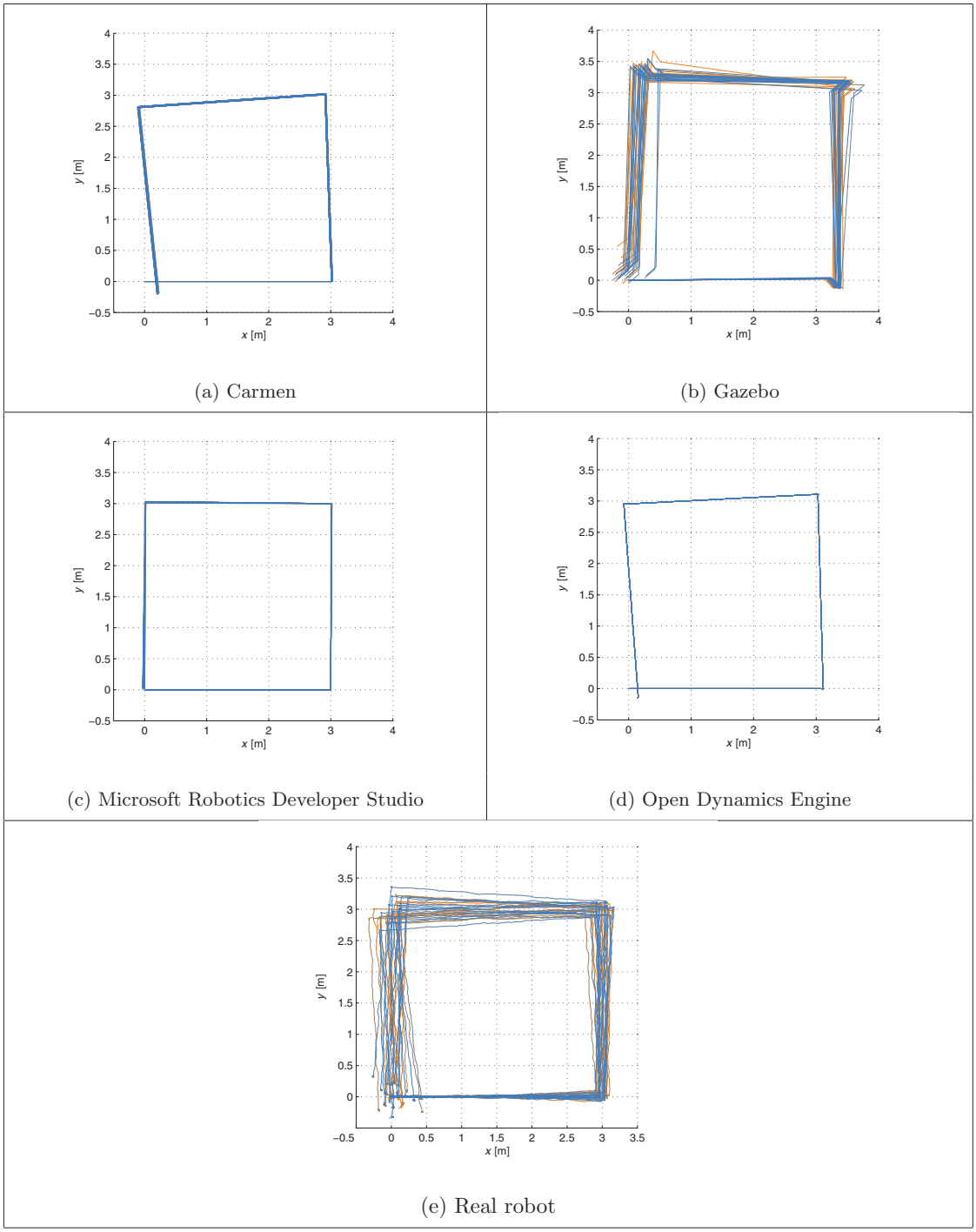
As seen in Table VI, the results obtained show that Carmen has the largest end-point position error (29 cm), followed by Gazebo (24 cm), ODE (21 cm), and finally MRDS (4 cm). Compared with the real robot's end-point RMS error (23 cm), the simulators error values seem reasonable. However, considering that the simulations modeled a noiseless and disturbance-free, the end-point error should have been very small, unlike the real robot's error. Therefore, it is not possible to conclude that the similarity in error magnitudes is due to high physical fidelity of the simulators. In fact, Fig. 9(e) shows that the real robot has a larger viability in the traversed path than the simulated paths in Figs. 9(a)–(d), which exhibit higher repeatabilities. In terms of precision, MRDS and Carmen have similar standard deviation values better than 1 cm (0.3% if computed over 3 m, the length of square's side) for the end-point location, but ODE has the lowest standard deviation with values of the order of 10^{-9} or smaller for the final position. The precision of Carmen is better than 1 cm (0.3% if computed over 3 m, the length of the square's side). Gazebo exhibits the worst precision, with a standard deviation of the resulting endpoint as 11 and 17 cm for x and y respectively (a relative error of almost 0.7%). In terms of processing time, Carmen takes longer time because it uses a TCP socket-based communication scheme between different processes as opposed to Gazebo's shared memory approach and ODE's implementation, which results in a single executable file after compilation. The real-time factor presented in the last column of Table VI shows that MRDS, PSG, and ODE are able to compute the solution in about 50% time it would take the real robot to complete the trajectory. It is to be noted, however, that if the robot would move twice faster, none of the simulators would be able to perform in real time. In fact, none of the simulators has been conceived for real-time performance, and most likely for many applications that require integration with middleware simulations-in-the-loop would be a bottleneck limiting the operation speed of the whole system.

5. Conclusions

This paper presented a comparative survey of the existing free software tools for mobile robot simulation. Four publicly available simulators were selected for detailed comparison considering their level of maturity, modularity, and to some extent their popularity: Carmen, PSG, ODE, and MRDS. The comparison is presented in both qualitative and quantitative terms. Qualitative aspects include relevant features and ease of learning aspects, while the quantitative comparison is carried out considering the simulators' accuracy and precision in modeling the motion of mobile robots taking into account real-world physics. The simulated results were compared with those obtained from analytically derived equations of longitudinal motion for a mobile robot on a road of varying slope. Experiments using a real robot endowed with a gyro-IMU and an external visual tracking system were also carried out to compare the results predicted by the models with the results observed in practice.

The qualitative and quantitative comparison of the selected simulators shows that none of them is yet sufficiently general and indisputably superior to the other, even considering the great efforts behind the development of each tool to respond to the needs of a broad community of researchers and developers of robotic applications. Each simulation tool is motivated by particular application domains that may share many things in common, but seek different attributes that result in tools of diverse flavors. For example, Carmen, MRDS, and PSG grew in part from the need of developing middleware and application programming interfaces that could provide hardware abstraction and allow to speed

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press



(a) Carmen

(b) Gazebo

(c) Microsoft Robotics Developer Studio

(d) Open Dynamics Engine

(e) Real robot

Fig. 9. Comparison of square trajectories.

up the development of robot navigation strategies. On the other hand, physical simulation accuracy was not as important as in ODE, which has been for a long time a *de facto* tool for simplified robot dynamics simulation, but which does not provide any type of middleware functionality. Currently, the PSG tool has received further support from the community behind the development of ROS, with the integration of other physics engines in addition to ODE, which are common in game development (Bullet), biomechanics (Simbody), and computer graphics and control (Dart). This recent push to PSG, which adds many algorithms for autonomous robotics, makes it potentially more attractive than MRDS and Carmen. Moves in a similar direction have been taken in the development of V-REP which

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

has recently included Bullet (in addition to ODE) and improved its model database and middleware interfaces. In turn, Bullet just announced that it has implemented the reduced coordinate algorithms of Featherstone to provide physical accuracy. Undeniably, robot simulation tools are evolving quickly, with clear trends toward the tighter integration of robot middleware and simulation, and the increase in physical accuracy which is necessary for the development of grasping, robot-environment interaction, mobile manipulator, and robot–human interaction simulations, to name a few. Many simulation tools available, such as Carmen, were motivated by problems in motion planning and navigation, and hence conceived as tools for the development of algorithms in these areas. Motion planning and navigation are mainly kinematic problems and therefore do not demand the same physical realism at the dynamic level. On the other end of the spectrum, users seeking physical accuracy that is needed for optimal design of the robot mechanical hardware and control laws, may find tools, such as ODE, Bullet, or Featherstone's Spatial Toolbox, more adequate, even if they do not provide sets of built-in robot models or middleware functionality.

Concerning the physical accuracy of the simulators, the results in Section 4 show that ODE is the most physically consistent simulator in the sense that it reproduces with considerable similarity what was observed and measured in the experiments. For example, the simulated robot with a reference speed of 2 m/s over a 2-m slope of 10° jumps a height slightly higher than $2\sin(10\pi/180) = 0.35$ m and travels a distance of approximately 7.7 m in 6 s (see right column of Fig. 5), while the experiment using a real robot with a reference speed of 1 m/s jumping over a 1-m slope of 10° jumps a height slightly higher than $\sin(10\pi/180) = 0.18$ m and travels a distance of 3.8 m in 6 s (see right column of Fig. 7). This clearly shows that the actual robot driven at half the speed on a ramp of half the length compared with that of the simulation jumps half the height and travels half the distance in the same amount of time. Furthermore, ODE shows that the pitch angle becomes positive after the jump due to the robot bouncing against the floor (see Fig. 5(c)), which is very similar to the bounce measured with the gyro-IMU sensor (see Fig. 8(a)). The bouncing effect is not predicted by other simulators, not even by the ideal model, which for simplicity assumes a perfectly inellastic collision, i.e. a zero coefficient of restitution. While it is not possible to claim that ODE or any other simulator is 100% physically accurate, the results show that ODE is physically consistent and has the potential of offering high physical accuracy compared with other simulators, provided that the user specifies the correct friction and collision parameters of real robot, and employs a sufficiently small integrator stepping period. All the simulators employed the same physical constants and parameters, but unfortunately, controlling their behavior was not free from tuning of simulator parameters, which were inevitably different due to their internal design. This is a problem that has been noted by other authors comparing complex simulations of biped robots implemented in Matlab and the commercial physics simulator Adams,[53] who show that in spite of all parameter tuning efforts, it is not possible to obtain exactly the same simulated robot behavior.

MRDS could be potentially more accurate than what was obtained in the experiments because it is based on detailed physics models provided by Nvidia's PhysX physics engine, but unfortunately the programming framework makes it difficult to access the simulation engine's state and get the data at arbitrary rates. In fact, MRDS makes an attempt to raise the programming of robotic applications to a higher level of abstraction, freeing the programmer from having to write code for the management of low level simulation tasks. However, this higher level of abstraction comes at a cost of giving up the control of message-passing and task scheduling activities to the framework, thus making the system less flexible. For example, extracting torques and forces is not a trivial task as this information is masked away from the user. In this sense, MRDS can be a good tool to test and visualize high-level navigation and robot–world interaction strategies, but is not the best tool available for rapid robot and mechatronic systems prototyping.

The user-friendliness of each of the compared simulators is different, notwithstanding that they all involve similar programming, compilation, and command line execution tasks. ODE is well documented, and easy to install and learn. ODE is very flexible given that its primary objective is to serve as a physics engine, but it also includes some visualization components to achieve simulation's 3D views. A disadvantage is that it requires developing code from scratch specific to robot navigation. Another disadvantage is that it does not include a generic programming interface to command standard robotic systems. However, this aspect is arguable, since the robot interfacing tools in Carmen, PSG, and MRDS still require the user to set specific parameters for the connected hardware. On the other hand, there are many projects relying on ODE, such as V-REP, which include some ready-made

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

mechanisms, sensor, and actuator models that can be reused. Carmen, MRDS, and PSG provide a framework conceived specifically for simulation of mobile robots in multi-agent scenarios. The major difference between Carmen, MRDS, and PSG is that MRDS and PSG include 3D dynamical models, while Carmen considers only 2D kinematic models.

ODE's capability to generate accurate simulations and its quick learning curve is explained by the fact that it is specifically conceived for physics simulation and has a monolithic structure, unlike Carmen, MRDS, or PSG, which run different process that must communicate with certain latencies. The latter is not necessarily a disadvantage, since a real hardware implementation may also require a distributed process approach. ODE's monolithic structure can be a disadvantage for the implementation of distributed applications. However, there are several libraries for developing concurrent applications that could be used together with ODE if necessary; see for example Zhen *et al.*[55] and references therein. ODE is written in C and can be compiled on a wide variety of systems running Linux or Windows, unlike the rest of the simulators which are specific to certain operating systems (see Table II).

In contrast to MRDS or ODE, both Carmen and Gazebo in their current distributions are more complex to install because of required libraries and other dependencies. We created a site,[62] with detailed installation notes and easy-to-follow simulation examples to help new users of these tools to hopefully get their systems up and running more quickly, and with much less trouble.

In relation to commercial simulators, some of them, such as anyKode Marilou, V-REP Pro, and Webots, employ ODE as their physics engine and therefore a similar performance to that achievable by directly building the simulation in ODE or PSG is expected. Other commercial simulators that allow to model robotic systems but have been created aiming at the problem of multi-body mechanical systems simulation use their own physics engines to achieve high fidelity physics simulations. These engines are part of industrial grade products oriented to the manufacturing industry and CAE applications. Although accurate statistics are hard to obtain, commercial high fidelity physics engines are mentioned for fewer times in the research literature than the open source ones. Among the most advanced commercial products delivering high physical accuracy are Adams by MSC Software Corporation, Vortex by CM Labs, AgX Dynamics by Algoryx Simulation AB, and SimMechanics toolbox for MATLAB/Simulink by MathWorks Inc. Due to its long development history, Adams is one of the most widely used multibody dynamics simulation and analysis tools. Vortex Dynamics is comparatively newer with respect to Adams, not as general as Adams, but has specialized on mechanical motion and contact dynamics simulation of mobile robotic applications, thus is being adopted by many companies and in some research projects for the design of mobile robots. As a matter of fact, V-REP has recently included Vortex and Bullet as alternatives to ODE to deliver high physics fidelity, especially using Vortex. A limitation of these simulation tools is that their development is motivated by applications requiring high physics accuracy, and therefore provide little or no middleware functionality to integrate real robotics hardware, model sensors, and test high level robot perception and guidance algorithms, which would have to be developed or integrated into the simulation by the user using other tools. In this context, V-REP with its recent integration of Vortex seems to have the potential of becoming a general tool for robot development that could allow to simulate both mechanical aspects and high level control algorithms while offering middleware functionality through ROS. On the other hand, the recent addition of new physics engines to PSG mentioned above clearly shows the efforts in the development of simulation tools that can cover a wide spectrum of needs from physical accuracy for mechanical design and structural stress testing to maneuverability analysis, grasping strategies analysis and development, and many other aspects that must be considered when conceiving autonomous robots that must interact with unknown and changing environments, for which purely kinematic models and simulations do not provide all the information needed to characterize the robot behavior.

The qualitative comparison and experimental analysis of the simulators presented in this paper should be very useful to educators, engineers, and researchers alike who are seeking suitable tools for simulating autonomous mobile robots. Introductory background aspects about simulation processes and simulator architectures were discussed in Appendix A. This information can be of tutorial value to researchers and developers interested in the simulation of robotic systems.

Acknowledgment
This project has been supported by the National Commission for Science and Technology Research of Chile (Conicyt) under Fondecyt Grant 1110343, and by UTFSM under DGIP grant 231356.

References

1. M. Anitescu and F. A. Potra, "Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems," *Nonlinear Dyn.* **14**, 231–247 (1997).
2. D. Baraff, "Linear-Time Dynamics Using Lagrange Multipliers," *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, New Orleans, LA, USA (Aug. 4–9, 1996) pp. 137–146.
3. S. Carpin, M. Lewis, J. Wang, S. Balakirsky and C. Scrapper, "USARSim: A Robot Simulator for Research and Education," *Proceedings of the IEEE International Conference on Robotics and Automation, 2007*, Roma, Italy (Apr. 10–14, 2007) pp. 1400–1405.
4. P. Castillo-Pizarro, T. Arredondo-Vidal and M. Torres-Torriti, "Introductory Survey to Open-Source Mobile Robot Simulation Software," *Proceedings of the 2010 Latin American Robotics Symposium and Intelligent Robotic Meeting (LARS)*, Sao Bernardo do Campo, Brazil (Oct. 2010) pp. 150–155.
5. A. Chatterjee and A. Ruina, "A new algebraic rigid-body collision law based on impulse space considerations," *J. Appl. Mech.* **65**(4), 939–951 (1998).
6. L. E. Chiang, "Teaching robotics with a reconfigurable 3D multibody dynamics simulator," *Comput. Appl. Eng. Educ.* **18**(1), 108–116 (2010).
7. J. D. Cohen, M. C. Lin, D. Manocha and M. Ponamgi, "I-COLLIDE: Qn Interactive and Exact Collision Detection System for Large-Scale Environments," *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (I3D '95), Monterey, CA, USA (Apr. 9–12, 1995) pp. 189–196.
8. P. I. Corke, "A robotics toolbox for MATLAB," *IEEE Robot. Autom. Mag.* **3**(1), 24–32 (Mar. 1996).
9. I. G. Daza, L. M. B. Pascual, M. A. S. Vazquez, E. L. Guillen, R. B. Navarro and L. B. Vazquez, "Low Level Control in States Space for the Pioneer," **In**: *Proceedings of the the 2005 International Conference on Computer as a Tool (EUROCON 2005)*, Vol. 1, Belgrade, Serbia and Montenegro (Nov. 21–24, 2005) pp. 322–325. IEEE, New Jersey, USA.
10. J. García de Jalón and E. Bayo, *Kinematic and Dynamic Simulation of Multibody Systems: The Real-Time Challenge* (Springer-Verlag, Berlin, Germany, 1994).
11. S. L. Delp, F. C. Anderson, A. S. Arnold, P. Loan, A. Habib, C. T. John, E. Guendelman and D. G. Thelen, "Opensim: Open-source software to create and analyze dynamic simulations of movement," *IEEE Trans. Biomed. Eng.* **54**(11), 1940–1950 (Nov. 2007).
12. R. Diankov and J. Kuffner, "Openrave: A Planning Architecture for Autonomous Robotics," *Technical Report* CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA (Jul. 2008).
13. E. Drumwright, J. Hsu, N. Koenig and D. Shell, "Extending Open Dynamics Engine for Robotics Simulation," **In**: *Simulation, Modeling, and Programming for Autonomous Robots*, Lecture Notes in Computer Science, Vol. 6472 (N. Ando, S. Balakirsky, T. Hemker, M. Reggiani and O. von Stryk, eds.) (Springer, Berlin, Germany, 2010) pp. 38–50.
14. D. H. Eberly, *Game Physics*. (*The Morgan Kaufmann Series in Interactive 3D Technology*), 2nd ed. (Elsevier, New York, NY, Apr. 2010).
15. A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *J. Robot.* **2012**, 15 (2012), Article ID 959013.
16. K. Erleben, "Module-Based Design for Rigid Body Simulators," *Technical Report* DIKU 02/06, Department of Computer Science, University of Copenhagen, Denmark (2002).
17. M. Eroglu, "Computer simulation of robot dynamics," *Robotica* **16**(6), 615–621 (1998).
18. R. Featherstone, *Rigid Body Dynamics Algorithms* (Springer, New York, NY, 2008).
19. E. Folgado, M. Rincón, J. R. Álvarez and J. Mira, "A Multi-Robot Surveillance System Simulated in Gazebo," **In**: *Proceedings of the 2nd International Work-conference on Nature-Inspired Problem-Solving Methods in Knowledge Engineering: Interplay Between Natural and Artificial Computation, Part II (IWINAC '07)* (Springer-Verlag, Berlin, Germany, 2007) pp. 202–211.
20. B. P. Gerkey, R. T. Vaughan and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," *Proceedings of the 11th International Conference on Advanced Robotics*, Coimbra, Portugal (Jun. 30–Jul. 3, 2003) pp. 317–323.
21. A. Harris and J. M. Conrad, "Survey of Popular Robotics Simulators, Frameworks, and Toolkits," *Proceedings of IEEE SoutheastCon, 2011*, Nashville, TN, USA (Mar. 17–20, 2011) pp. 243–249.
22. E. J. Huag, *Computer-Aided Kinematics and Dynamics of Mechanical Systems: Basic Methods*, Allyn and Bacon Series in Engineering (Prentice Hall, Upper Saddle River, NJ, 1989).
23. L. Hugues and N. Bredeche, "Simbad: An Autonomous Robot Simulation Package for Education and Research," **In**: *SAB*, Lecture Notes in Computer Science, Vol. 4095 (S. Nolfi, G. Baldassarre, R. Calabretta, J. C. T. Hallam, D. Marocco, J.-A. Meyer, O. Miglino and D. Parisi, eds.) (Springer, New York, NY, 2006) pp. 831–842.
24. R. N. Jazar, *Vehicle Dynamics: Theory and Application* (Springer, New York, NY, 2008).
25. K. Johns and T. Taylor, *Professional Microsoft Robotics Developer Studio* (Wrox, Indianapolis, IN, 2008).

26. J. Klein, "Breve: A 3D Environment for the Simulation of Decentralized Systems and Artificial Life," *Proceedings of the Eighth International Conference on Artificial Life* (MIT Press, Cambridge, MA, 2002), pp. 329–334.
27. K. Kobayashi, Y. Uchida and K. Watanabe, "A Study of Battle Strategy for the Robocode," **In**: *Proceedings of the SICE 2003 Annual Conference*, Vol. 3, Fukui, Japan (Aug. 4–6, 2003) pp. 3373–3376. IEEE, New Jersey, USA.
28. Z. Kovačić, S. Bogdan, K. Petrinec, T. Reichenbach and M. Punčec, "Leonardo – The Off-line Programming Tool for Robotized Plants," *Proceedings of the 9th Mediterranean Conference on Control and Automation*, Jun. 27–29, Dubrovnik, Croatia (Jun. 2001).
29. K. Kumar and P. Singh Reel, "Analysis of Contemporary Robotics Simulators," *Proceedings of the International Conference on Emerging Trends in Electrical and Computer Technology (ICETECT), 2011*, Tamil Nadu, India (Mar. 23–24, 2011) pp. 661–665.
30. B. Magyar, Z. Forhecz and P. Korondi, "Developing an Efficient Mobile Robot Control Algorithm in the Webots Simulation Environment," **In**: *Proceedings of the IEEE International Conference on Industrial Technology, 2003*, Vol. 1, Maribor, Slovenia (Dec. 10–12, 2003) pp. 179–184.
31. D. W. Marhefka and D. E. Orin, "Xanimate: An educational tool for robot graphical simulation," *IEEE Robot. Autom. Mag.* **3**(2), 6 –14 (Jun. 1996).
32. D. W. Marhefka and D. E. Orin, "A compliant contact model with nonlinear damping for simulation of robotic systems," *IEEE Trans. Syst. Man Cybern.* **29**(6), 566 –572 (Nov. 1999).
33. S. McMillan, D. E. Orin and R. B. McGhee, "Object-oriented Design of a Dynamic Simulation for Underwater Robotic Vehicles," *Proceedings of the 1995 IEEE International Conference on Robotics and Automation, 1995*, Vol. 2, Nagoya, Aichi, Japan (May 21–27, 1995) pp. 1886–1893.
34. A. T. Miller and P. K. Allen, GraspIt! A versatile simulator for robotic grasping. *IEEE Robot. Autom. Mag.* **11**(4), 110–122 (Dec. 2004).
35. B. V. Mirtich, Impulse-Based Dynamic Simulation of Rigid Body Systems *Ph.D. Thesis* (University of California, Berkeley, CA, Dec. 1996).
36. R. M. Murray, S. S. Sastry and L. Zexiang, *A Mathematical Introduction to Robotic Manipulation*, 1st ed. (CRC Press, Boca Raton, FL, 1994).
37. J. F. Nethery and M. W. Spong, "Robotica: A Mathematica package for robot analysis," *IEEE Robot. Autom. Mag*. **1**(1), 13 –20 (Mar. 1994).
38. A. Rahim, J. Teo and A. Saudi, "An Empirical Comparison of Code Size Limit in Auto-Constructive Artificial Life," *Proceedings of the IEEE Conference on Cybernetics and Intelligent Systems, 2006*, Bangkok, Thailand (Jun. 7–9, 2006) pp. 1–6.
39. T. Reichenbach, "A dynamic simulator for humanoid robots," *Artif. Life Robot.* **13**, 561–565 (2009).
40. S. J. Rodenbaugh and D. E. Orin, "Robotbuilder User's Guide, ver. 1.0.," *Technical Report*, Department of Electrical Engineering (The Ohio State University, Ohio, May 2003).
41. E. Rohmer, S. P. N. Singh and M. Freese, "V-REP: A Versatile and Scalable Robot Simulation Framework," *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*,Tokyo, Japan, November 3–7, 2013 (Nov. 2013) pp. 1321–1326.
42. J. Ros, R. Yoldi, A. Plaza and X. Iriarte, "Real-time Hardware-in-the-Loop Simulation of a Hexaglide-Type Parallel Manipulator on a Real Machine Controller," **In**: *ECCOMAS Thematic Conference on Multibody Dynamics 2011* (J. C. Samin and P. Fisette, eds.), Brussels, Belgium (European Community on Computational Methods in Applied Sciences (ECCOMAS), 2011) pp. 1–11.
43. R. B. Rusu, A. Holzbach, R. Diankov, G. Bradski and M. Beetz, "Perception for Mobile Manipulation and Grasping Using Active Stereo," *Proceedings of the 9th IEEE-RAS International Conference on Humanoid Robots, 2009*, Paris, France (Dec. 7–9, 2009) pp. 632–638.
44. W. Son, K. Kim, N. M. Amato and J. C. Trinkle, "A Generalized Framework for Interactive Dynamic Simulation for Multirigid Bodies," *IEEE Trans. Syst. Man Cybern.* **34**(2), 912 –924 (Apr. 2004).
45. D. Stewart and J. C. Trinkle, "An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction," **In**: *Proceedings of the ICRA IEEE International Conference on Robotics and Automation, 2000*, Vol. 1, San Francisco, CA, USA (Apr. 24–28, 2000) pp. 162–169. IEEE, New Jersey, USA.
46. D. E. Stewart, "Rigid-body dynamics with friction and impact," *SIAM Rev.* **42**(1), 3–39 (2000).
47. M. Tang, D. Manocha, J. Lin and R. Tong, "Collision-streams: Fast GPU-based Collision Detection for Deformable Models," *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D '11)*, San Francisco, CA, USA (Feb. 18–20, 2011) pp. 63–70.
48. R. T. Vaughan and B. P. Gerkey, "Really Reusable Robot Code and the Player/Stage Project," **In**: *Software Engineering for Experimental Robotics*, Berlin, Germany (2006) pp. 267–289. Springer-Verlag.
49. M. W. Walker and D. E. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *J. Dyn. Syst. Meas. Control* **104**(3), 205–211 (1982).
50. J. Wang, C. M. Gosselin and L. Cheng, "Modeling and simulation of robotic systems with closed kinematic chains using the virtual spring approach," *Multibody Syst. Dyn.* **7**, 145–170 (2002).
51. J. Y. Wong, *Theory of Ground Vehicles*, 4th ed. (John Wiley, Hoboken, NJ, 2008).
52. K. Yamane and Y. Nakamura, "Stable Penalty-Based Model of Frictional Contacts," *Proceedings of 2006 IEEE International Conference on Robotics and Automation (ICRA 2006)*, Orlando, FL, USA (May 15–19, 2006) pp. 1904–1909.

53. T. Ylikorpi, J.-L. Peralta and A. Halme, "Comparing passive walker simulators in MATLAB and ADAMS," *J. Struct. Mech.* **44**(1), 65–92 (2011).
54. J. C. Zagal, J. Ruiz-Del-Solar and P. Vallejos, "Back to Reality: Crossing the Reality Gap in Evolutionary Robotics," *Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles*, Lisbon, Portugal (Jul. 5–7, 2004) pp. 1–9.
55. Z. Zhen, C. Qixin, C. Lo and Z. Lei, "A CORBA-based simulation and control framework for mobile robots," *Robotica* **27**(3), 459–468 (2009).
56. L. Žlajpah, "Simulation in robotics," *Math. Comput. Simul.* **79**(4), 879–897 (2008).
57. anyKode Marilou. Available at: `http://www.anykode.com/` (accessed July 2, 2014).
58. Carmen: Carnegie Mellon Robot Navigation Toolkit. Available at: `http://carmen.sourceforge.net/` (accessed July 2, 2014).
59. Cogmation robotics. Available at: `http://www.cogmation.com/` (accessed July 2, 2014).
60. Energid. Available at: `http://www.energid.com/actin-simulation-advantages.htm` (accessed July 2, 2014).
61. EyerSim. Available at: `http://robotics.ee.uwa.edu.au/eyebot/doc/ sim/sim.html` (accessed July 2, 2014).
62. M. Torers-Torriti, T. Arredondo and P. Castillo-Pizarro, Installation instructions for Carmen, Gazebo and Open Dynamics Engine. Available at: `http://sourceforge.net/apps/mediawiki/arsproject/` (accessed July 2, 2014).
63. JDE project. Available at: `http://jderobot.org/` (accessed July 2, 2014).
64. Moby. Available at: `http://physsim.sourceforge.net/` (accessed July 2, 2014).
65. ODE: Open Dynamics Engine. Available at: `http://www.ode.org/`, `http://opende.sourceforge.net/wiki/index.php/` (accessed July 2, 2014).
66. OpenRAVE. Available at: `http://openrave.org/` (accessed July 2, 2014).
67. OpenRDK. Available at: `http://openrdk.sourceforge.net/` (accessed July 2, 2014).
68. OpenSim. Available at: `http://opensimulator.sourceforge.net/` (accessed July 2, 2014).
69. The Orocos Project. Available at: `http://www.orocos.org/` (accessed July 2, 2014).
70. PSG: Player-Stage-Gazebo. Available at: `http://playerstage.sourceforge.net/`, `http://gazebosim.org/` (accessed July 2, 2014).
71. Robocode. Available at: `http://robocode.sourceforge.net/` (accessed July 2, 2014).
72. ROS. Available at: `http://www.ros.org/wiki/` (accessed July 2, 2014).
73. Simbad. Available at: `http://simbad.sourceforge.net/` (accessed July 2, 2014).
74. USARSim. Available at: `http://sourceforge.net/projects/usarsim/` (accessed July 2, 2014).
75. V-REP. Available at: `http://www.v-rep.eu/` (accessed July 2, 2014).
76. G. Metta, P. Fitzpatrick and L. Natal, YARP: Yet another robot platform. `http://eris.liralab.it/yarp/` (accessed July 2, 2014).

A. Simulators' architecture and dynamics modeling approaches

Understanding the underlying architecture and implementation of simulators for robotic systems is essential in order to compare the various elements that contribute to the simulators' modularity, accuracy, speed, and other functionality aspects, such as the possibility of running simulations distributed over a network of host machines. At the highest level of abstraction, a simulator architecture can be divided into five modules responsible for handling specific tasks in the simulator: (i) time control and stepping module, (ii) motion solver, (iii) constraint solver, (iv) collision handler, and (v) collision detector. For practical or computational efficiency, some of these modules can be intertwined together in such a way that they are actually implemented as a single module. Nonetheless, this conceptual division into modules, also proposed in ref. [16], is very useful as it allows the description of major simulation paradigms and facilitates both understanding and design of simulators. Figure A1 illustrates a general modular simulator architecture. The simulator must return the state $x(t_0 + k\Delta T)$, $k = 1, 2, 3, \ldots$ of the system's components every ΔT seconds. The state can be a list or vector that includes position, velocities, accelerations, and even the current configuration of constraints. As shown in Fig. A1, the operation of simulator in terms of its modules can be broadly explained as follows. The user configures and sends a request to simulate the motion of a multibody system whose description is in some simulation parameter file or specified through some commands in program files. The simulation starts from an initial state $x(t_0)$, which is not only used to initialize the motion solver but also employed in other modules to initialize constraints and contact conditions. The first action of the simulator (1) is taken by the time stepper module, which requests the motion solver to return (8) the state of the system $x(t_0 + \Delta T)$, given the current (initial) state $x(t_0)$. The motion solver updates the position of all bodies in the system typically employing Newton's force laws, taking a step Δt smaller than ΔT. The motion solver must then take into account motion constraints and

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

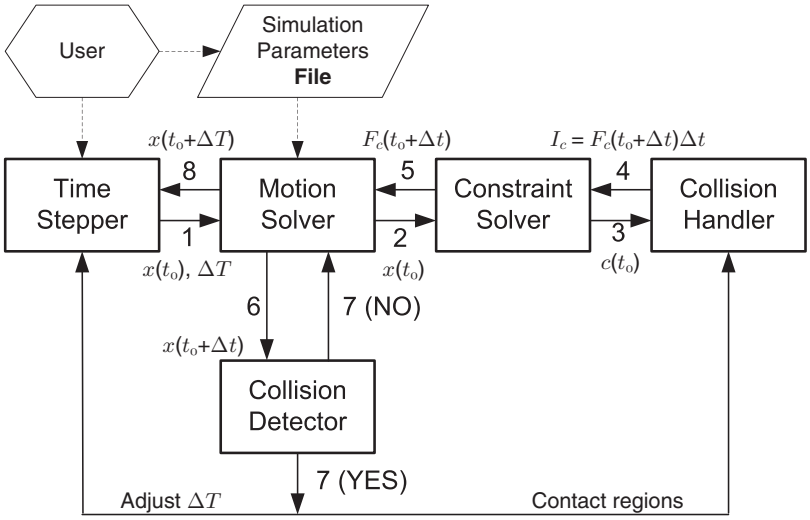


Fig. A1. The simulation process and its main modules.

forces, thus requests (2) the constraint solver to check for active constraints. Motion constraints occur between two or more bodies in *permanent (steady) contact* (e.g. parts connected by joints), between *persistent contacts* (e.g. objects sliding, rolling, and resting), or between *temporary contacts* (e.g. colliding bodies). Thus, the motion solver must call the constraint solver to return (5) the constraint forces F_c between the interacting bodies. Constraints can be bilateral ("=") or unilateral ("≥"), for example, bilateral forces are the forces at joints connecting and keeping together the links in an articulated robot, while contact forces between the body and the surface upon which it rests, e.g. the normal forces between the wheels of a vehicle and the ground arise from unilateral contact or non-penetration constraints. In order for the constraint solver to compute constraint forces, it must invoke (3) the collision handler module to return (4) the impulsive forces I_c associated with collisions producing a change in momentum of the colliding bodies. Finally, once the motion solver has received the constraint forces F_c, it passes (6) the newly estimated intermediate state $x(t_0 + \Delta t)$ for an increment $\Delta t < \Delta T$ of time. The collision detector must verify whether there are any new collisions. If there are none, then (7), the motion solver, takes another step Δt and repeats the cycle (2, 3, 4, 5, 6, 7) until the interval ΔT has been simulated, or the collision detector detects a collision. When the collision detector detects a collision, it must send (7) the list of contact points or regions to the collision handler. The collision detector must also adjust the time interval ΔT for which motion is to be computed in the case of adaptive time stepping methods, which look to find the precise impact time and thus reduce or avoid the possibility of object interpenetration, which increases in fixed time step approaches as objects move at higher speeds and their dimensions are small.[16]

There exist several approaches and algorithms to implement each of the simulator modules shown in Fig. A1. It is important to understand the main options available, since the choices can impact the performance of the simulator. There is probably no optimal general choice, as the selected techniques often depend on the final application of the simulator, e.g. simulation speed is often preferred over physical accuracy in computer graphics, while in mechanism design and control it can be the other way around. Beginning with the time stepping module, basically the choices are fixed time stepping or adaptive time stepping.[16] State-of-the-art approaches employ second-order time steppers with pyramidal friction as proposed by Stewart and Trinkle,[45] who cast the Netwon–Euler equation, non-penetration constraints, and friction model as an LCP in discrete time form. In this method, time stepping is implicit, thus rendering unnecessary the computation of impact time.

The motion solver module must consider the type of coordinate representation of the system (maximal or reduced), and the type of integrator (explicit or implicit). The reduced-coordinate formulation takes a system with m degrees of freedom, and a set of c constraints that remove c degrees of freedom. The remaining $n = m - c$ degrees of freedom are usually known as

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

generalized coordinates, while the original m coordinates are called *maximal coordinates*.[2] Reduced or generalized coordinate methods are also called *relative approaches* because the coordinates often define the relative motion of a body with respect to another in a sequentially ordered fashion. On the other hand, maximal coordinate approaches are also called *Lagrange multiplier methods* because they employ the more generic set of m maximal coordinates, and enforce that motion complies with the constraints by including a set of constraint forces, i.e. the motion update is solved as an optimization problem subject to c constraints, thus requiring a set of c scalar coordinates corresponding to the Lagrange multipliers. Unlike reduced coordinate approaches which rely on relative rotation-translation transformations between the reference frames of each body, maximal coordinate formulations express each body's position and orientation with respect to a general (world) absolute (inertial) frame, thus some refer to the Lagrange or maximal coordinate approaches also as *absolute coordinate* approaches. Regardless of the preferred formulation, the equations of motion of a multibody system can be written as a differential algebraic equation of the general form:

$$\dot{q} = H^{\dagger}(q)v, \tag{A1}$$

$$M(q)\dot{v} = f(q, v, t) + H^{\dagger^T}(q)G^T(q, t)\lambda, \tag{A2}$$

$$g(q, t) = 0, \tag{A3}$$

where $q : \mathbb{R} \to \mathbb{R}^{n_q}$ and $v : \mathbb{R} \to \mathbb{R}^{n_v}$ are the vectors of variables that respectively define the pose and velocity of the components of the multibody system at any instant of time t, thus completely describing the configuration and motion of the multibody system. The matrix $H(q) : \mathbb{R}^{n_q} \to \mathbb{R}^{n_v}$ defines a purely kinematic mapping that relates the rate of change of the pose vector q to the velocity vector v according to $v = H(q)\dot{q}$. The matrix $H(q)^{\dagger} : \mathbb{R}^{n_v} \to \mathbb{R}^{n_q}$ denotes the left inverse of $H(q)$. Usually $H(q)$ has full row rank and therefore can be calculated as the Moore–Penrose left-pseudoinverse $H^{\dagger} = (H^T H)^{-1} H^T$. The matrix $M(q) \in \mathbb{R}^{n_v \times n_v}$ is a symmetric positive-definite matrix often referred to as generalized *mass matrix*[2,10,14,22] or *inertia matrix*.[14,18,36] The function $f : \mathbb{R}^{n_q} \times \mathbb{R}^{n_v} \times \mathbb{R} \to \mathbb{R}^{n_v}$ represents the contribution of centrifugal, Coriolis, and external forcing terms. The equation $g : \mathbb{R}^{n_q} \times \mathbb{R} \to \mathbb{R}^{n_g}$ defines an $n_q - n_g$ dimensional time-dependent submanifold of $\mathbb{R}^{n_q}$ resulting from motion constraints. It is often assumed that $g \in \mathcal{C}^2$, i.e. g is twice continuously differentiable, and that it gives rise to a velocity constraint Jacobian $G = \frac{\partial g}{\partial q}$ that has full row rank. The dependence of constraint equation on time allows to model different motion drivers. Finally, $\lambda \in \mathbb{R}^{n_g}$ is the vector of Lagrange multipliers associated with constraint forces.

The reduced coordinate formulation has been the classic modeling approach for articulated bodies, such as robot manipulators.[18,49] In this context $q = [\mathbf{x}^T \ \theta^T]^T$ is the vector of joint coordinates expressing the position of each link relative to its preceding link, and $v = [\mathbf{v}^T \ \omega^T]^T$ are the velocity variables of the corresponding links. In the relative coordinates formulation, $H(q)$ is a block diagonal matrix composed of each link's velocity Jacobian matrices $\mathcal{H}_i(q)$ relating the link i's velocity v_i to the configuration variables in the vector q, according to $v_i = \mathcal{H}_i(q)q$.[36] The inertia matrix can be expressed as $M(q) = \sum_i \mathcal{H}_i^T(q)\mathcal{M}_i\mathcal{H}_i(q)$, where $\mathcal{M}_i = \left[\begin{smallmatrix} m\mathbb{I}_{3\times3} & \mathbb{O} \\ \mathbb{O} & J_b \end{smallmatrix}\right]$ is the generalized inertia matrix for body i, comprising body's mass m and inertia tensor $J_b \in \mathbb{R}^{3\times3}$ in body coordinates.[36] The force vector $f(q, v, t) = \tau - C(q, v)v - N(q, v)$ is the combination of joint forces or torques represented by τ, the Coriolis and centrifugal forces $C(q, v)v$, and the gravitational and other forces represented by $N(q, v)$. Since the parameterization of motion often takes into account motion constraints, an explicit statement of constraint equations represented by $g(q, t) = 0$ is not necessary in the reduced coordinate formulation.

On the other hand, multibody system simulators and physics engines[10,14,22] have typically preferred the maximal coordinates formulation in which $q = [\mathbf{x}^T \ \Psi^T]^T$ is the vector expressing the position and orientation of different bodies relative to a global inertial reference frame, and $v = [\mathbf{p}^T \ \mathbf{L}^T]^T$ is the vector of linear and angular momentum of corresponding bodies.[14] The term Ψ is typically the orientation quaternion $\Psi = (\psi_s, \psi_x, \psi_y, \psi_z)$. In this formulation, $H(q) = \left[\begin{smallmatrix} m^{-1}\mathbb{I} & \mathbb{O} \\ \mathbb{O} & \mathbf{Q}J^{-1} \end{smallmatrix}\right]$, where m represents the mass of bodies, and $J^{-1}(t) = R(t)J_b^{-1}R(t)^T$ represents the matrix inverse of bodies' inertia tensor in global coordinates computed by applying the rotation transformations $R(t)$ to bodies'

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

inertia tensors expressed in body coordinates J_b. The matrix

$$\mathbf{Q} = \begin{bmatrix} -\psi_x & -\psi_y & -\psi_z \\ \psi_s & \psi_z & -\psi_y \\ -\psi_z & \psi_s & \psi_x \\ \psi_y & -\psi_x & \psi_s \end{bmatrix} \tag{A4}$$

corresponds to the matrix form of the orientation quaternion Ψ. Finally, in the maximal coordinates approach, $M(q) = \mathbb{I}$ and $f(q, v, t) = \tau$, where τ represents all the external forces acting on the bodies expressed in global coordinates.

The advantages of maximal coordinate approaches are that they use a more general and uniform representation of bodies with respect to the same global inertial reference frame and that computing Coriolis and centrifugal forces is not necessary because all motions are expressed with respect to a global inertial frame. A drawback of maximal coordinate approaches is that they often involve sparse matrices and redundant information that require more data transfer bandwidth between computer's memory and math processing unit. Maximal coordinate approaches also require that the simulator keeps track of constraints' state. On the other hand, reduced-coordinate requires less data transfer bandwidth because the constraints are implicit in the equations of dynamic model. However, in reduced-coordinate approaches, it is more difficult to express and include arbitrary constraints. Another problem of reduced-coordinate approaches is that it can be very difficult to parameterize the system's degrees of freedom in terms of n generalized coordinates, especially in redundant or closed-kinematic chain systems. An in-depth discussion of the advantages of each approach and the maximal coordinate formulation can be found in ref. [2]. Due to the advantages and drawbacks of each approach, the best recursive reduced coordinate formulations, such as Walker and Orin's Composite Rigid Body algorithm (CRBA)[49] or Featherstone's Articulated Rigid Body algorithm (ABA),[18] are often preferred in robotics for simulating industrial manipulators and other humanoid-like articulated robots. Both CRBA and ABA are forms of the so-called Recursive Newton–Euler algorithm for inverse dynamics of robotic mechanisms. On the other hand, maximal coordinate formulations are preferred among the computer graphics community and developers of mechanical dynamics engines because the Lagrange multiplier formulation allows some additional modularity and facilitates handling of non-holonomic constraints, e.g. velocity-dependent constraints, which are not a natural part of the reduced coordinate formulations.

Solving contact and joint forces may be done in various ways, which can be grouped into two classes: analytical[1,2,45,46] or penalty-based[7,32,35,52] approaches; see also refs [14, 18] and references therein. Analytical approaches employ a system of constraint equations in analytic form that are typically solved by recasting the equations as an LCP solved by numerical methods. Penalty-based methods apply penalty forces at points of penetration to restore position and velocities of objects in the simulation to values that satisfy the constraints. Since the latter are easier to implement and can be faster to compute because they do not strictly enforce non-penetration, this family of methods is older and has dominated the computer animation field, in which realistic-looking motion is often sufficient.[35] The choice of the constraint solving approach depends on the accuracy and speed requirements of the simulator.

Finally, the collision handling module must apply collision impulses computed by the constraint solver to all bodies in the simulation. Since collision impulses produce a discontinuous change in the motion of colliding bodies, the collision information must be passed back to motion solver so that it can correctly update the state of objects. The impulses can be computed using algebraic laws, incremental laws, or full deformation laws,[5] and be applied simultaneously to all bodies or propagated in a sequential manner.[5,14,35] The location of collision impulses is obtained from the collision detection module. This module is one of the main contributors to the computational complexity of a multibody simulator, and often dedicated collision detection libraries and even hardware are employed to solve this computational geometry problem.[7,47]

In addition to understanding the main components and data flow in a simulator, it is also important to highlight some implementation aspects that are motivated by application requirements and the fact that the operation of mobile robots typically involves concurrent processes, such as processing measurements and feedback signals from multiple sensors, commanding several motors, and carrying out high level planning and decision tasks. Applications involving hardware-in-the-loop for rapid

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press

prototyping of controllers,[42] and robotic applications relying on predictive control techniques that require accurate models of the system whose parameters must be identified online, while using the same self-built models to find adequate future motions,[54] must also consider other requirements such as task concurrency, coordination, and distributed execution of software components possibly over a network of host machines.[55] Hence, simulation tools can be designed to work as stand-alone applications, in which several functions are compiled into one executable linked against the simulator library, or designed to run as multi-process systems. The latter architecture offers several advantages in the simulation of robotic systems. First, it enables to independently manage the simulation of different system components, such as sensors, mobile platform, control logic, output rendering, among others. Second, it allows the distribution of the process load among several computers, much like it would occur in a real robot, in which some computers can be dedicated to navigation tasks, while others to sensing or motion control. However, a disadvantage of the multi-process simulator architectures is the need for ways to pass information between modules and the corresponding latencies, which can vary depending on whether the data exchange is done using file-based memory sharing, RAM-based memory sharing, or employing inter-process communication relying on Unix sockets and TCP/IP message passing mechanisms.[4]

B. Installation and usage aspects of the simulators

This appendix provides a brief discussion concerning installation and usage issues of the main simulators chosen for detailed evaluation.

The installation of Carmen is managed through configure and Makefile scripts. However, due to changes in Linux and its libraries, the installation scripts need some modifications, and some external packages must be installed manually. This fact combined with Carmen's current level of documentation can make the installation process complex for non-expert Linux users.[4] We have provided some hints for installing Carmen 0.7.4 on Linux Fedora in ref. [62].

One of the main disadvantages of Gazebo is that it depends on a large number of third party libraries (e.g. Scons, FLTK, ODE, OGRE, BULLET, OIS, libxml2, wxpython, amongst others) that must be installed using compatible versions[70] in order for it to function properly. Some dependencies are partial or not documented. For example, we found that the latest unnumbered version of Gazebo, as well as the latest released version 0.10.0, would not properly update shared memory when OGRE 1.7.0 was used. Version 1.6.4 of OGRE was required for proper functionality of the system. The list of commands to install Gazebo is very long to be included here; we have provided full instructions in order to successfully install Gazebo 0.10.0 under Ubuntu 9.10 in ref. [62].

Installing MRDS and using the available services is a relatively simple task when compared with other simulators discussed in this paper. In fact, MRDS does not require any major manual configuration procedure. Moreover, since its first release back in 2006, many aspects have been improved and designed considering an audience of non-expert programmers. The number of tutorials and examples has also increased notoriously since its initial release. However, the asynchronous concurrent service-oriented programming model requires one to abandon the classic structured programming mindset and develop skills to understand nontrivial relationships between the concurrent components.[4] In fact, the authors of ref. [25], and members of the MRDS development team, state in their book that understanding and learning to use CCR/DSS is crucial for anyone who wishes to develop new MRDS applications ranging from new robot simulations to the addition of sensors and actuators. Thus, one of the drawbacks of MRDS is having to learn the CCR/DSS framework, which is not as common as other forms of popular interprocess communication schemes mentioned earlier. Another limiting aspect of this framework is the fact that CCR is implemented in C#, thus making it more complex to port MRDS applications or to program in the more classic and common C/C++ languages.

One of the main outstanding aspects of ODE is that it does not have specific software dependencies. The installation package provides different build systems for each platform and includes ready-to-use workspaces for Visual Studio and Code::Blocks. The ODE API is straightforward, flexible, and intuitive. The documentation and community support is extensive and the project is in active development.[65] We have made available a simple list of steps for installing the current version of ODE (0.11.1) on Ubuntu 9.10 in ref. [62].

https://doi.org/10.1017/S0263574714001866 Published online by Cambridge University Press