PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

A ROBOTICS SIMULATOR FOR PYTHON

GERMÁN LARRAÍN MUÑOZ

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
MIGUEL TORRES TORRITI

Santiago de Chile, March 2014

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

A ROBOTICS SIMULATOR FOR PYTHON

GERMÁN LARRAÍN MUÑOZ

Members of the Committee:
MIGUEL TORRES TORRITI
LUCIANO CHIANG SÁNCHEZ
TOMÁS ARREDONDO VIDAL
MIGUEL NUSSBAUM VOEHL

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, March 2014

Gratefully to my family

ACKNOWLEDGEMENTS

Firstmost, I would like to thank professor Torres not only for accepting me as a graduate student under his project "Simulation and Optimal Control of Skid Steer Mobile Manipulators", but also for constantly encouraging me to aim for the best while working on it.

I would like to thank my fellow students, many of whom graduated before me, for their invaluable advice, comments and corrections.

Last but not least, I would like to express my gratitude to my parents. The love and support they have given me through all my life have allowed me to pursue my dreams with confidence and humility. Without them, I would not be who I am nor would I have ever thought of a graduate degree.

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Modeling and simulation of robotic systems is essential for robot design and programming purposes, as well as operator training. Of the tools developed throughout the years, some have reached reasonable levels of maturity, but are specific to robot manipulators and do not include appropriate tools for modeling mobile bases, while other focus on mobile bases in 2D planar spaces, but do not consider 3D environments. Other tools may integrate already made models to simulate complex mobile manipulators or even humanoids, but have steep learning curves and require significant programming time and skills.

The situation is further complicated due to the lack of widely accepted simulation standards and languages among robot developers, whom often implement custom simulations specific to the application or other particular aspects, e.g. robot motion planning and navigation, mechanical design and evaluation, training, etc. However, the application of robots to increasingly complex tasks calls for accurate, yet simple to use, modular and standardized simulation tools capable of describing such things as the robot's interaction with the environment (e.g. effector-object, robot-ground, and sensor-world interactions).

This paper describes the design of such a tool for physically accurate robot and multibody systems simulation. The software, implemented in Python on top of well regarded physics and visualization libraries, is open-source, modular, easy to learn and use, and can be a valuable tool in the process of robot design, in the development of control and reasoning algorithms, as well as in teaching and educational activities.

Keywords: simulation of robot dynamics, multibody physics, robot simulation, software tools, public domain software, Python

RESUMEN

La modelación y simulación de sistemas robóticos es esencial para efectos del diseño y programación de robots, así como para el entrenamiento de operadores. De las herramientas desarrolladas a lo largo de los años, algunas han alcanzado niveles razonables de madurez pero son específicas a robots manipuladores y no incluyen herramientas apropiadas para modelar bases móviles, mientras que otras se centran en bases móviles en espacios planos en 2D pero que no consideran entornos 3D. Otras herramientas pueden integrar modelos ya construidos para simular manipuladores móviles complejos o incluso humanoides pero tienen empinadas curvas de aprendizaje y requieren tiempo y habilidades significativas de programación.

La situación es aún más complicada debido a la falta de estándares de simulación y lenguajes ampliamente aceptados entre los desarrolladores de robots, quienes a menudo implementan simulaciones personalizadas específicas a una cierta aplicación u otros aspectos particulares como, por ejemplo, la planificación de movimiento y navegación de robots, diseño y evaluación mecánica, entrenamiento, etc. Sin embargo, la aplicación de robots a tareas cada vez más complejas requiere de herramientas de simulación precisas –pero simples de usar–, modulares y estandarizadas, capaces de describir cosas como la interacción del robot con el entorno (e.g. interacciones efector-objeto, robot-terreno y sensor-entorno).

En este trabajo se describe el diseño de una herramienta de este tipo para la simulación físicamente precisa de robots y sistemas multicuerpo. El software, implementado en Python sobre librerías muy respetadas de física y visualización, es de código abierto, modular, fácil de aprender y usar, y puede ser una herramienta valiosa en el proceso de diseño de robots, en el desarrollo de algoritmos de control y razonamiento, así como en docencia y otras actividades educacionales.

Palabras Claves: simulación de dinámica de robot, física de multicuerpo, simulación de robot, herramientas de software, software de dominio público, Python

1. INTRODUCTION

Since the early days of robotics, simulation tools have played an essential role in the development of newer and better systems because simulation provides an excellent way for testing and experimenting, while avoiding the costs of building unfeasible designs, the consequences of accidents and the difficulty of controlling disturbances present in the physical world (Evans-Pughe, 2006; Chiang, 2010; Wettach, Schmidt, & Berns, 2010; Koenig & Howard, 2004). For example, current industrial robots rely on off-line simulators to validate and evaluate the work cycle of entire production lines (Zlajpah, 2008; Kovačić, Bogdan, Petrinec, Reichenbach, & Punčec, 2001). The design and feasibility testing of sophisticated robotic systems, ranging from underwater mobile manipulators (McMillan, Orin, & McGhee, 1995) to sophisticated humanoids (Reichenbach, 2009) and robot swarms (Pinciroli et al., 2011) would have not been possible without adequate simulation tools. In addition to the research and engineering of robotic systems, simulation has been key to the development of virtual environments for operator training, as well as in teaching the fundamental concepts of robotic systems (Chiang, 2010; Marhefka & Orin, 1996).

1.1. Problem Description

Future autonomous robots will require better self-simulation capabilities to become more self-aware and capable of taking decisions in real-time while operating in complex and dynamic environments of the real world (Diankov, 2010; Johns & Taylor, 2008; Koenig & Howard, 2004). However, the existing tools (see recent reviews in (Kumar & Reel, 2011; Castillo-Pizarro, Arredondo, & Torres-Torriti, 2010)) are often conceived for specific sub-problems, e.g. motion control, motion planning, navigation, and typically consider simplified dynamical models that do not take into account other aspects, such as robot-ground interactions, collisions, energy consumption or advanced sensor models, to name a few. Extending existing tools that were not designed with that in mind is difficult and sometimes even impossible, because their software architecture is not modular enough or

forbids modifications. On the other hand, although some of the existing simulators produce sophisticated and high resolution visualization of the robots' motion, the velocity and force profiles are not accurate due to the underlying simplified motion models (Castillo-Pizarro et al., 2010).

1.1.1. Python

Python is a general purpose, object-oriented programming language that has been steadily gaining popularity among researchers and educators who see it as an alternative to Matlab because of, among other reasons, the existence of libraries like numPy and SciPy (Jones, Oliphant, Peterson, et al., 2001–). Some features of Python and its ecosystem are: (i) is open source unlike Matlab, Java, C#; (ii) has a wide user base that has been growing since its was officially released in 1991; (iii) is easy to learn; (iv) improves productivity because of its small overhead compared to C/C++; (v) is cross-platform and runs on Windows, Mac and Unix-like systems with little or no modification unlike C/C++; and (vi) is multi-domain oriented with applications ranging from basic desktop programs to full-grown programming suites, as the Python Package Index (PyPI) (Python Software Foundation, 2011b) can attest.

Python aims to increase developer productivity and it is one the main reasons users choose it. The Python Software Foundation (PSF) claims that "*Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs*", (Python Software Foundation, 2011a).

1.1.2. Rigid body dynamics simulation

Robots are modeled essentially as bodies connected by joints. Thus a robotics simulator needs a component to simulate the time-changing state of bodies and joints. That component is what is usually called a *physics engine*, a concept original from games development: "a common piece of code that knows about physics in general but isn't programmed with the specifics of each game's scenario" (Millington, 2007). However, a physics engine

does not have any conceptual constraint preventing their use in other applications (e.g. simulators), although the particular preferences for the existing trade-offs may be different. It is important to note that it is not the physics engine's concern if and how the simulation will be rendered graphically

The problem of creating a 3D robotics simulator that supports mobile manipulators requires many features that only a handful of open-source physics engines provide because of the great complexity of dealing with multiple objects connected in many different ways plus collision and time-varying external forces and torques. If a restriction of modeling only kinematic chains was imposed then a much simpler physics processing would suffice (a correct translation of coordinate frames as depicted in fig. 1.1 is still fundamental); that is the case of robotics arms connected to a static entity (e.g. "world", "ground"). Nonetheless, we do have a limitation: bodies will be considered rigid (i.e. assume zero deformation) because soft bodies involve a much more complex topic and first it is necessary that the rigid case is solved properly with a very high level of accuracy, which has not happened in the current generation of physics engines.

For our simulator we care about *forward dynamics*: "the calculation of the acceleration response of a given rigid-body system to a given applied force" (Featherstone, 2008). Mathematically, we can express that response as

$$\ddot{\boldsymbol{q}} = \mathrm{FD}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}) \tag{1.1}$$

where $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$ are vectors of position, velocity and acceleration variables, respectively, and $\boldsymbol{\tau}$ is a vector of applied forces (Featherstone, 2008), all of them time-dependent (fig. 1.2 shows the case for a single body). The $model$ is the data that describes the rigid-body system being simulated. It encompasses bodies' geometric (i.e. shape) and dynamic properties (e.g. mass, inertia tensor), and the joints (with the corresponding parameters) that connect them together and constrain their movement by removing some *degrees of freedom* (DOF) of the 6 available (3 linear and 3 angular). Common joints are the ball-and-socket (also known as spheric, 3 angular DOF remaining), revolute (also known as rotary or hinge, 1

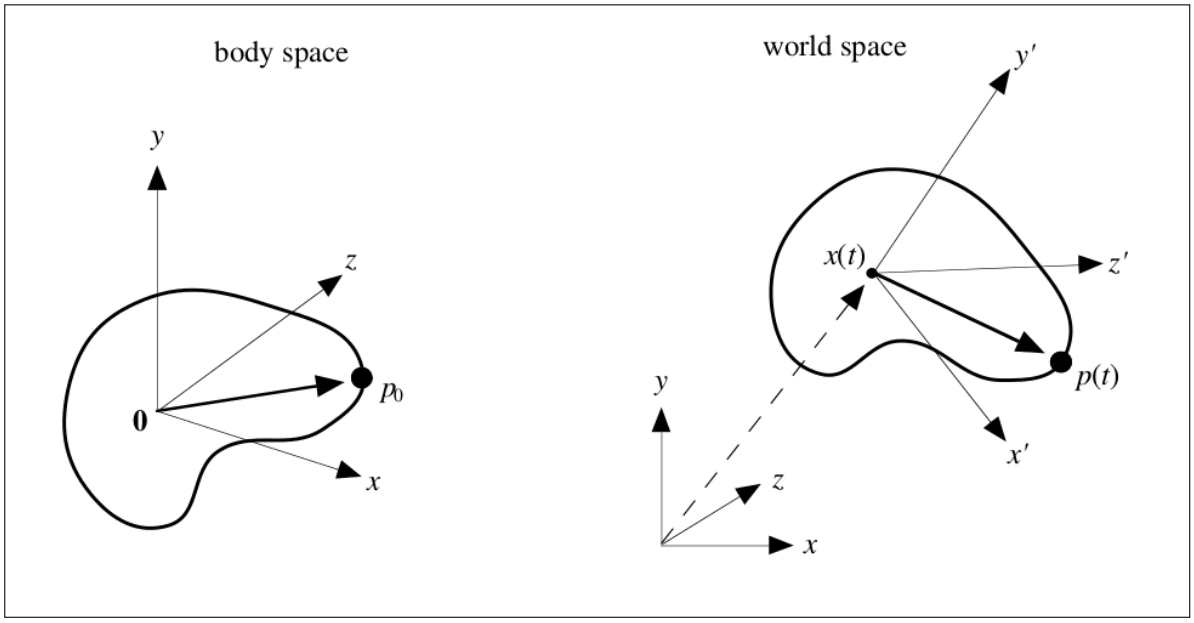


FIGURE 1.1. Body's local coordinate system is translated to world coordinates. The origin (center of mass) and the point p_0 are transformed to points $x(t)$ and $p(t)$ respectively. Source: (Baraff, 2001).

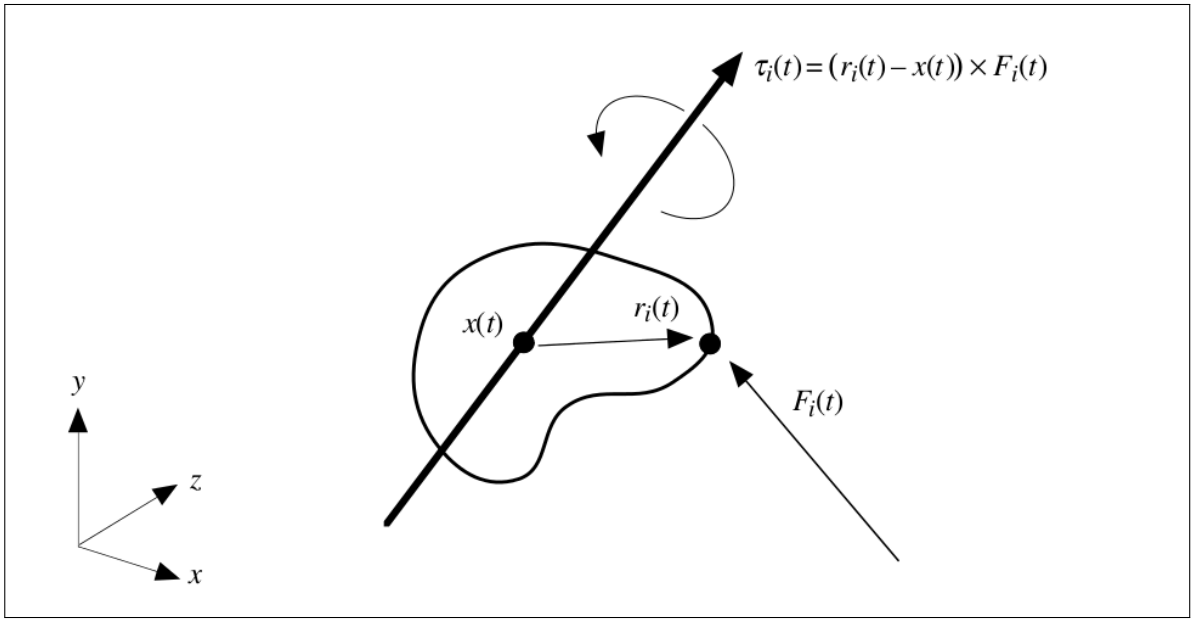


FIGURE 1.2. Torque due to a force applied to a rigid body at an off-center point. Source: (Baraff, 2001).

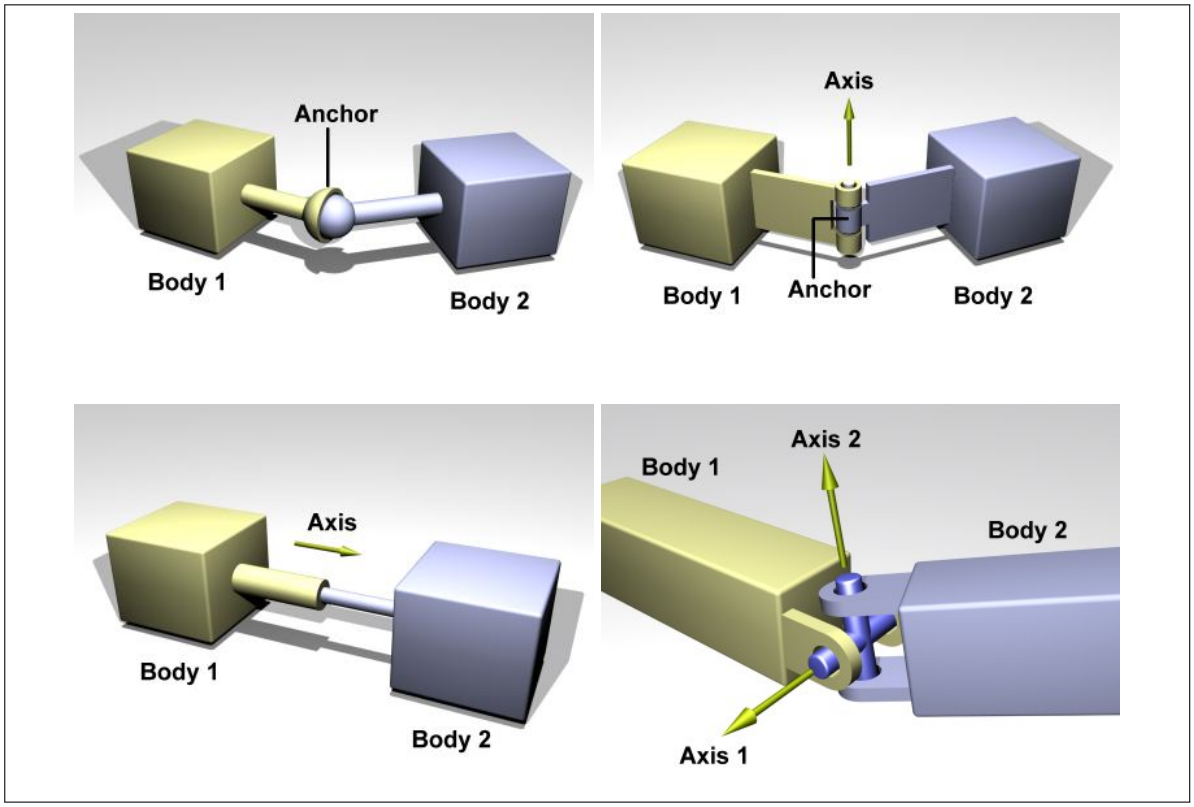


FIGURE 1.3. Basic joints used in simulation: (from left to right, top to bottom) ball and socket, revolute, piston and universal. Source: (Open Dynamics Engine, 2013a).

angular DOF), prismatic (also known as linear or slider, 1 linear DOF), piston (1 linear and 1 angular DOF) and universal (2 angular DOF). All these are depicted in fig. 1.3 except the prismatic joint, which is identical to the piston but without the remaining angular DOF.

For each simulation step, the physics engine has to:

(i) detect collisions between bodies' geometries,

(ii) create motion constraints for those collisions,

(iii) solve the system's equations,

(iv) update the system's state (bodies' linear and angular position and velocity),

(v) advance the simulation time (usually a fixed amount).

Collision detection is a relatively self-contained topic and it will be explained in the next subsection. On the other hand, the way the physics engine solve the system's equations is

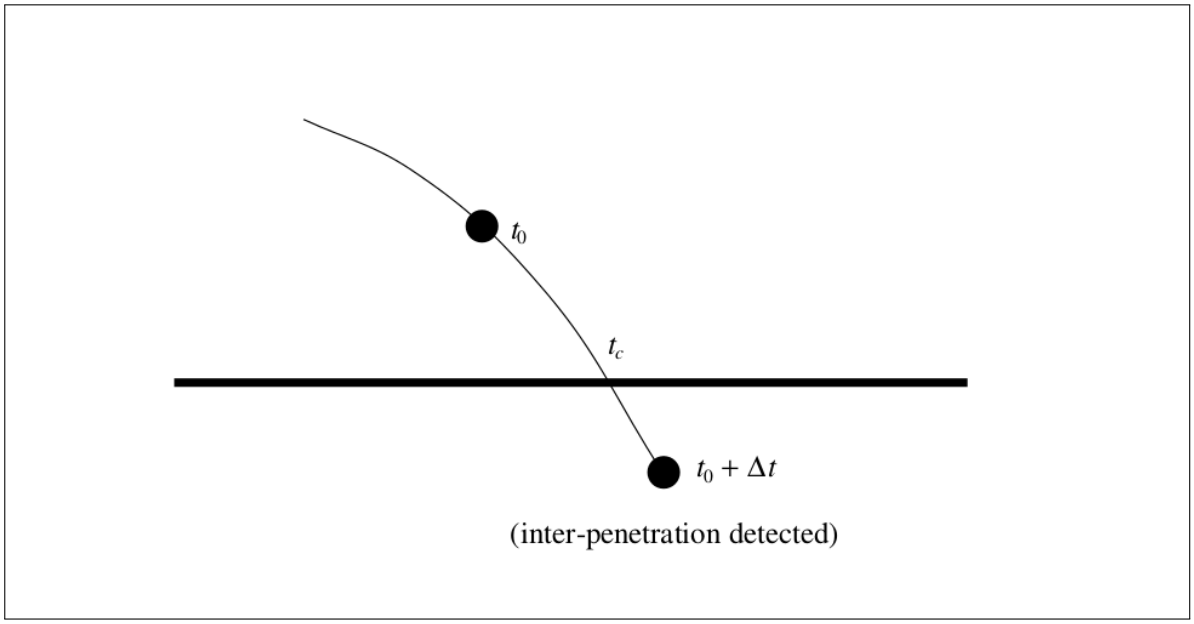


FIGURE 1.4. Particle interpenetration. Source: (Baraff, 2001).

tightly coupled to how motion constraints are translated into equations. For example, Open Dynamics Engine (ODE, not to be confused with Ordinary Differential Equation) solves simultaneously the Newton-Euler dynamic equations and the contact (collision) constraints by posing the equations governing the system of rigid bodies as a *linear complementarity problem* (LCP) in the maximal coordinate system to solve for impulses satisfying the constraints imposed by joints connecting bodies and contact points.

1.1.3. Rigid body collision handling

Collision handling deals with the problem of preventing bodies from penetrating each other. The fundamental task is detecting if a particle has penetrated a given volume. As fig. 1.4 shows, this is not as simple as it seems because the time when the particle starts penetrating is probably between two sequential discrete simulation times thus when the condition is detected the particle is already in an illegal position.

Assuming that problem is handled correctly (for example, by using an adaptive time step or a fixed one small enough), when detecting a contact between two different bodies it

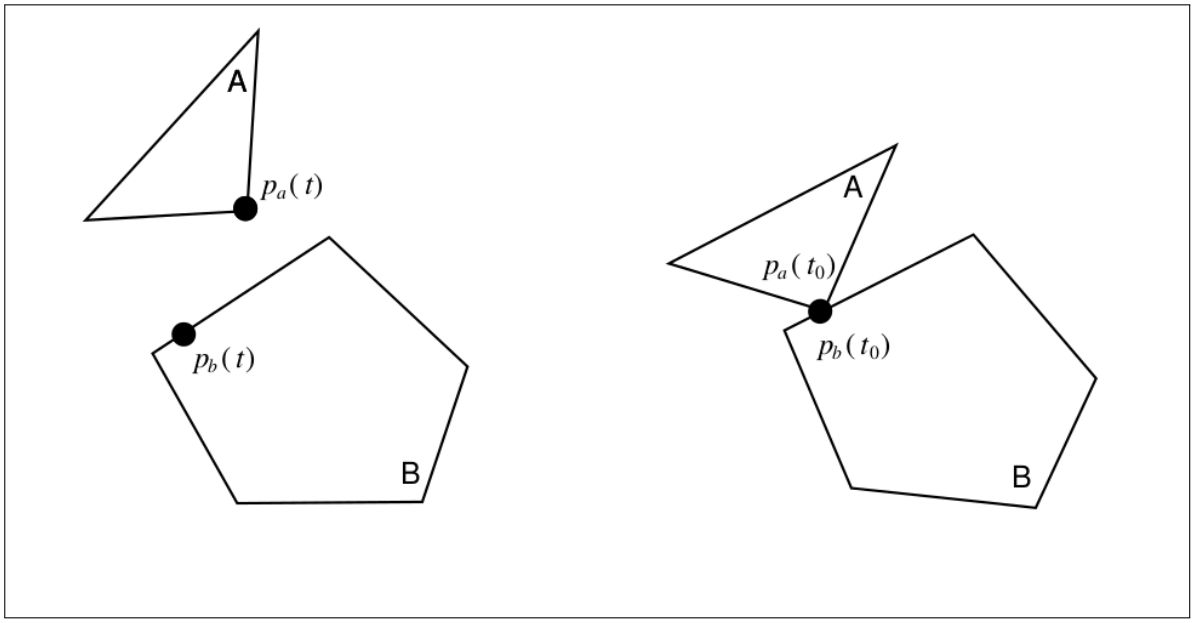


FIGURE 1.5. Corresponding points in bodies A and B come into contact. Source: (Baraff, 2001).

must be determined the type of contact and what to do. For that reason the difference of the velocity of the corresponding points coming into contact (fig. 1.5) is checked (the points are at the same position at time t_0 but their velocities can be very different). If that vector points in the same direction that $\hat{n}(t_0)$ (unit vector orthogonal to the surface of body B) then the bodies are separating and nothing needs to be done whereas if the vector points in the opposite direction (fig. 1.6) penetration will happen right after t_0 i.e. it is a collision contact, and a force needs to be exerted to prevent that penetration. In the particular case that the relative velocity is perpendicular to $\hat{n}(t_0)$ (fig. 1.7), it is a resting contact and may or may not need a force application to prevent the bodies from accelerating towards each other. Accurately solving motion constraints can be very complicated when there are multiple contacts simultaneously even if they all are of the resting type (fig. 1.8).

Because a physics engine is a higly complex and specialized software some do not even have their own collision detection code but outsource this task to even more specific third-party packages. Or, like ODE, provide a decoupled API so the developer is able to use a different collision detection engine.

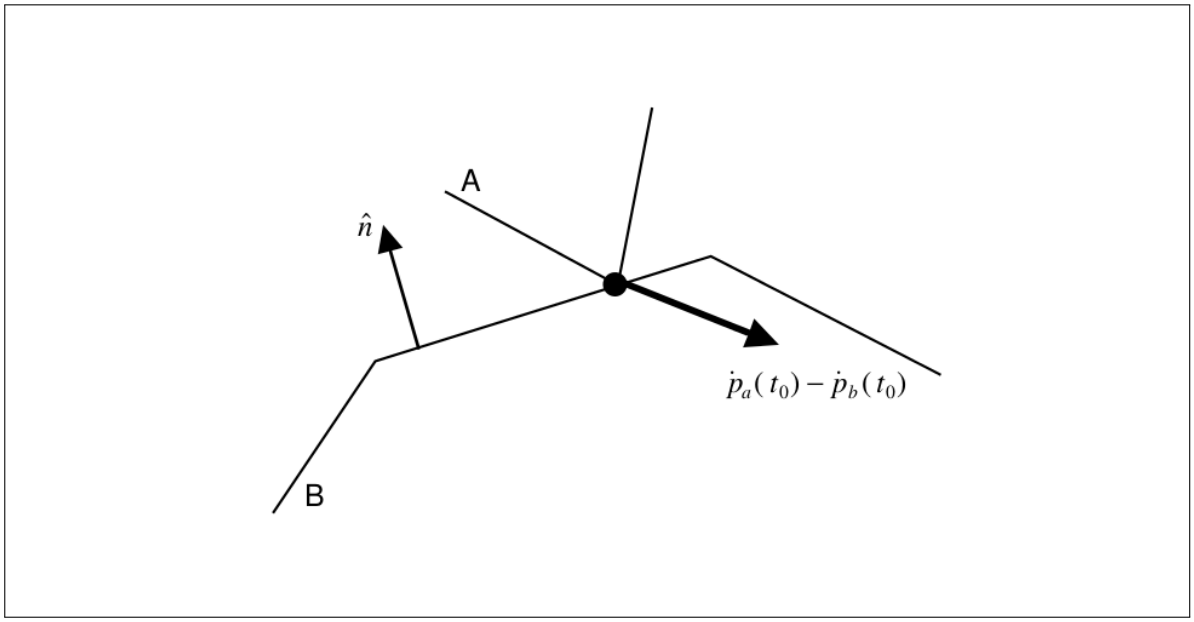


FIGURE 1.6. Colliding contact. Source: (Baraff, 2001).

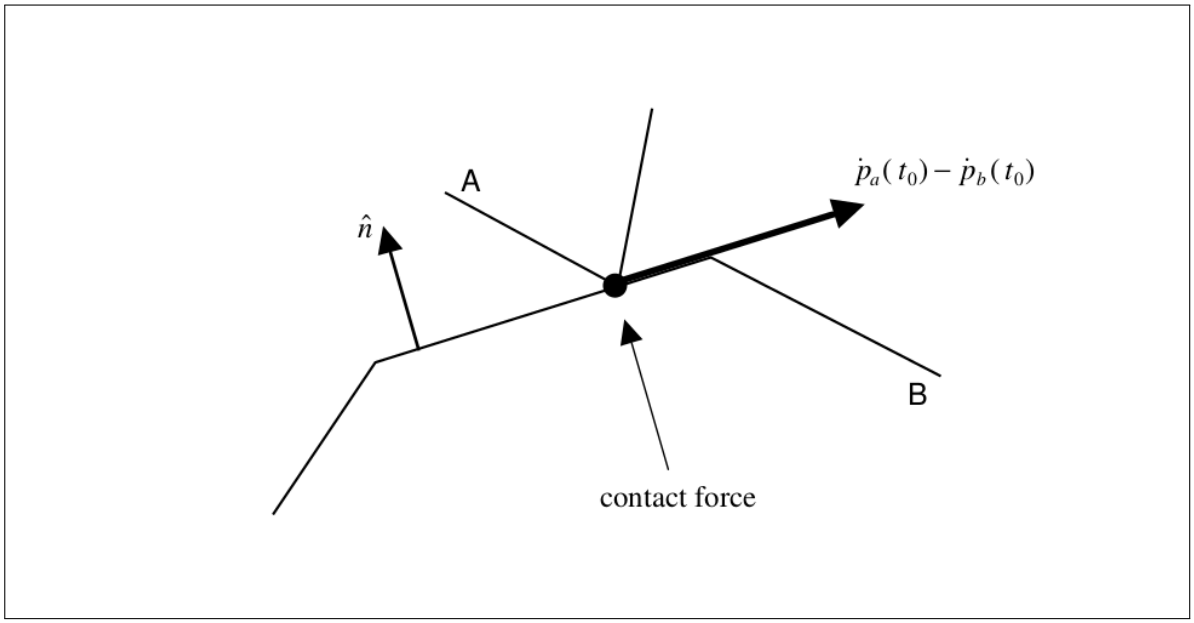


FIGURE 1.7. Resting contact. Source: (Baraff, 2001).

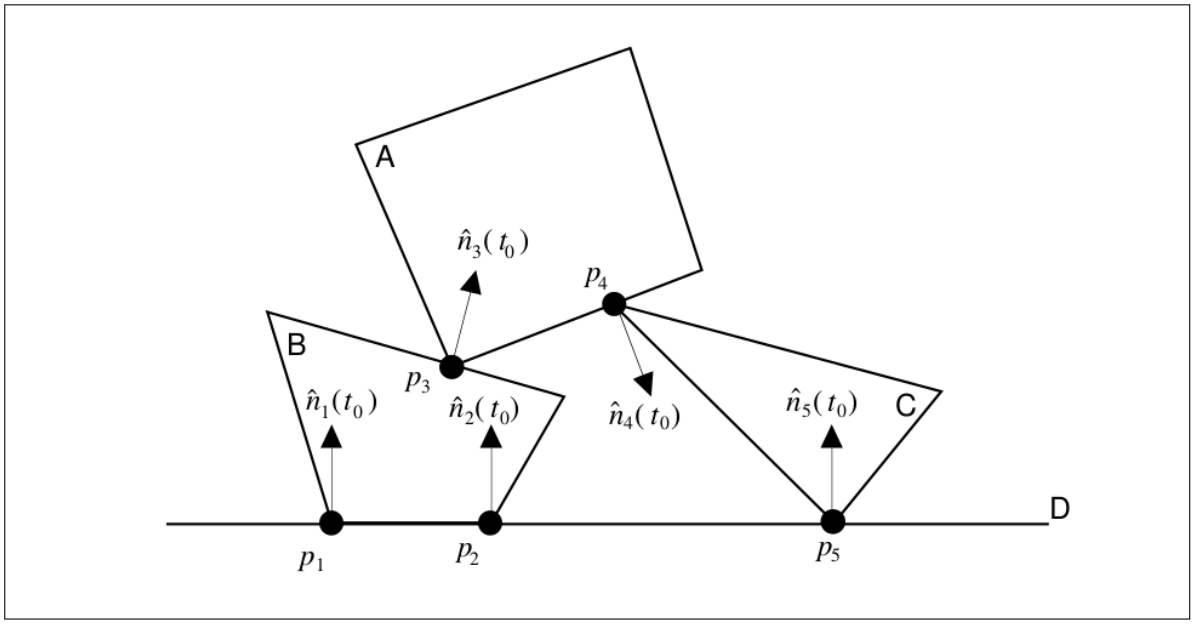


FIGURE 1.8. Multiple resting contacts. Source: (Baraff, 2001).

1.1.4. A robotics simulator for the Python community

We believe the Python community needs an open-source (distributed under an OSI-approved license (Open Source Initiative, 2013)) robotics simulator that is written in pure Python (although it is alright if it uses C/C++ extensions, which is common in scientific sofware (Jones et al., 2001–)), has 3D physics (dynamics and collision), and supports mobile manipulators. Soft requirements are that it should be extendable, have an open development process (public and forkable code repository, public bug tracker, public documentation, mailing list), and have 3D visualization.

1.2. Motivation

Each simulation program run by ARS will create a 3D simulation window like the one in fig. 1.9 where a basic mobile manipulator is displayed.

Simulations are defined with straightforward Python code. A Hello World example is:

```
from ars.app import Program
```

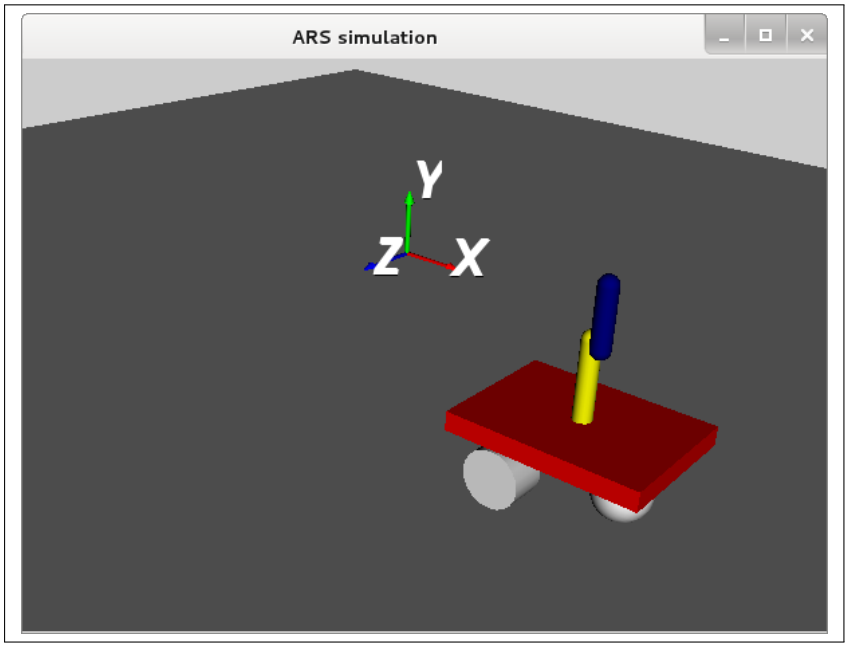


FIGURE 1.9. An ARS simulation window.

```
class FallingBall(Program):


    def create_sim_objects(self):
        self.sim.add_sphere(0.5, (1, 1, 1), mass=1)
```

where a sphere of radius 0.5 m and 1.0 kg is created at position $(1, 1, 1)$ meters. More interesting examples are available in the official videos posted online (ARS, 2013a).

1.2.1. An example for physical validation

Here we present an example that illustrates the physical accuracy of ARS's implementation using Open Dynamics Engine (ODE). It is a robot with two degrees of freedom (DOF), involving a waist and a shoulder joint, which is actuated as a conical pendulum to show that the numerical results match those that can be computed analytically applying the laws of physics.

The robot is shown in fig. 1.10 and its geometry is expressed using the standard Denavit-Hartenberg (DH) convention with parameters summarized in table 1.1. The inertial parameters are presented in table 1.2. The robot has two rotary joints, whose positions

TABLE 1.1. Denavit-Hartenberg parameters for the robot of the example.

Joint	θ_i [rad]	d_i [m]	a_i [m]	α_i [rad]
1	θ_1	1	0.1	$\pi/2$
2	θ_2	0	1	0

TABLE 1.2. Inertial and electro-mechanical parameters for the robot of the example.

	Link/Joint 1	Link/Joint 2
Mass m_i (kg)	10	10
Inertia tensor I_i (kg m^2) with $r = \sqrt{2}/10$ m	$\begin{bmatrix} \frac{m_1(3r^2+d_1^2)}{12} & 0 & 0 \\ 0 & mr^2 & 0 \\ 0 & 0 & \frac{m_1(3r^2+d_1^2)}{12} \end{bmatrix}$	$\begin{bmatrix} mr^2 & 0 & 0 \\ 0 & \frac{m_2(3r^2+a_2^2)}{12} & 0 \\ 0 & 0 & \frac{m_2(3r^2+a_2^2)}{12} \end{bmatrix}$
Motor Inertia J_{mi} (kg m^2)	10^{-4}	10^{-4}
Gear ratio	100:1	100:1
Viscous friction B_i (N s)	$50 \cdot 10^{-3}$	$50 \cdot 10^{-3}$

are described by the DH parameters θ_1 (waist) and θ_2 (shoulder). The robot is actuated applying a torque to the waist motor according to

$$\tau_1 = \begin{cases} 20t, & 0 \leq t < 1 \\ 20, & 1 \leq t \end{cases} \text{ Nm},$$

while the shoulder joint is left to coast freely with no torque applied to it, i.e. $\tau_2 = 0$ Nm, $t \geq 0$. The robot actuated in this manner behaves like a *conical pendulum*, a device which was classical in clockwork timing mechanisms and key element in centrifugal governors during the 1800s.

To validate the accuracy of the results the same robot was implemented using the Robotics Toolbox for Matlab (Corke, 1996). Figure 1.11 shows the results obtained from the simulation with ARS and the ideal curve obtained using the classical recursive Newton-Euler algorithm implemented in the Robotics Toolbox. The curves in blue correspond to the angular velocity $\omega_1 = \dot{\theta}_1$ of the robot's waist joint, while the dark-green curves correspond to the angle of the robot's shoulder joint. Figure 1.11 shows that as torque in the waist joint is ramped to 20 Nm in one second, the waist angular velocity reaches almost 4 rad/s. The centrifugal forces acting on the shoulder link cause it to raise from $\pi/2$ rad to about 0.87 rad.

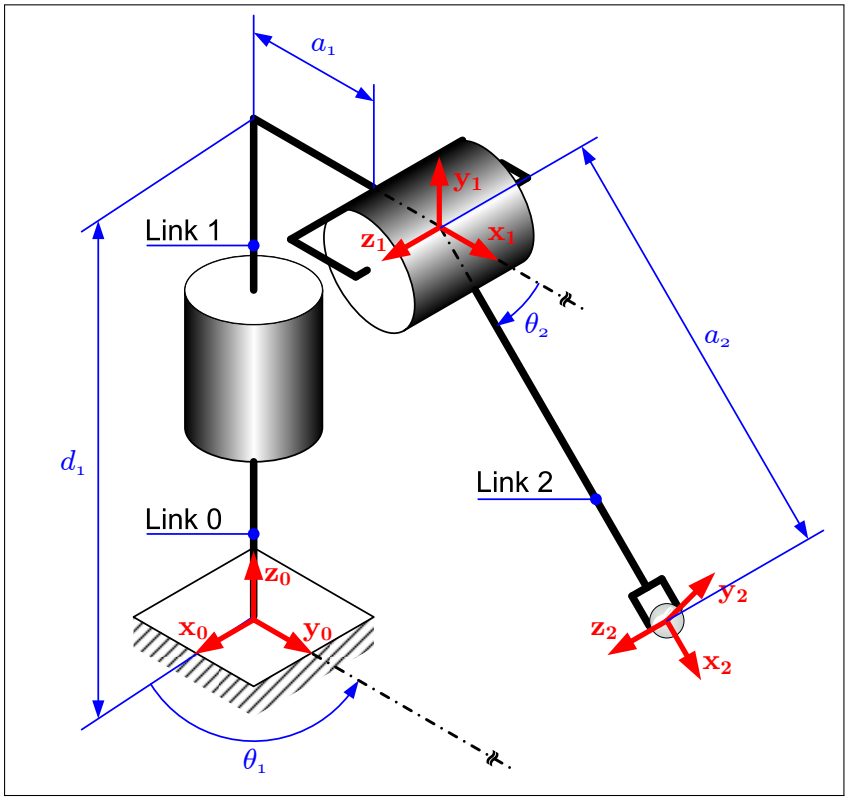


FIGURE 1.10. 2-DOF robot arm.

It is possible to appreciate that the curves in figure 1.11 obtained in ARS and with the Robotics Toolbox match well and reach the same steady-state values with a similar time-response. The small difference in the transient is due to ODE's different internal damping and constraint handling parameters, such as the constraint force mixing (CFM) and the error reduction parameter (ERP), that are needed by ODE because it employs a maximal coordinate formulation of the dynamics together with a formulation for the constraints in terms of a linear complementarity problem (LCP) instead of the reduced coordinate approach of the recursive Newton-Euler algorithm (Drumwright, Hsu, Koenig, & Shell, 2010). It is to be noted that we computed the results using ODE's default internal parameters and we did not do any kind of special adjustment to find a better approximation to the expected curve. This should demonstrate that the ARS simulator and the underlying dynamics engine should be fairly reliable and should not demand the user to spend time adjusting parameters unless some extremely high level of accuracy is required by the application. Moreover, despite

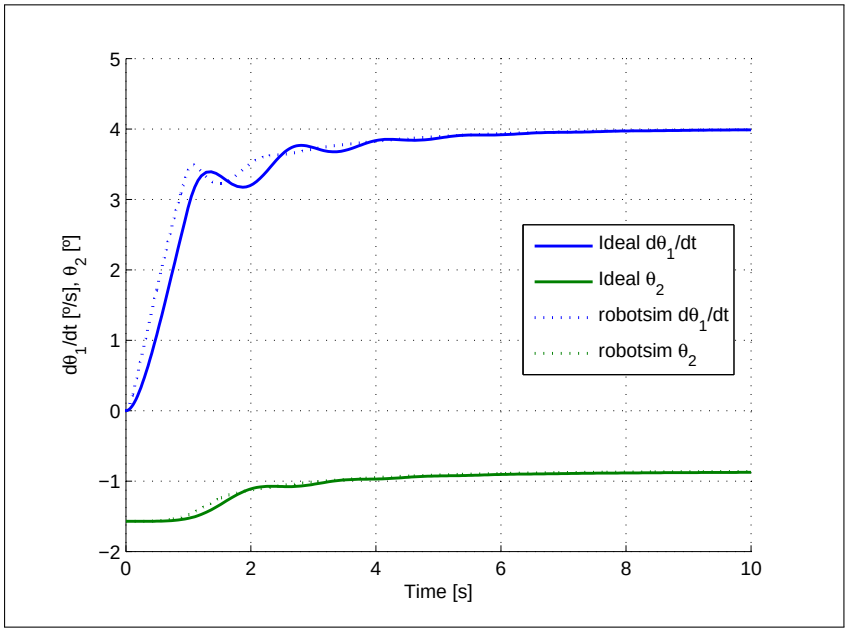


FIGURE 1.11. Trajectory for the freely coasting shoulder as the waist speed of the 2DOF robot arm increases.

the small differences in the transient responses, the response obtained with ARS reasonably preserves the characteristic oscillations and response time. This should allow mechanical engineers and control designers to develop robots and controllers knowing that the model dynamics is consistent with that of the real system, and thus that commonly relevant aspects such as actuator forces, energy consumption or controllers' stabilizing properties exhibited by the simulated solutions will be preserved by the actual physical implementations. Certainly, time could be spent to adjust ODE's parameters so that its response perfectly matches that of the ideal dynamical model. However, seeking absolute accuracy may be beyond reasonable because in practice even when ideal models are available, some time must be spent on the calibration and validation of the real sub-systems, such as actuators, sensors and controllers, due to other uncertainties and disturbances that can be costly or simply very difficult to model.

Finally, it is also to be noted that for the robot rotating as a conical pendulum, and with the simplifying assumption that the links are described by point-masses, the simple application of Euler's second law for the torque balance about the shoulder's joint axis

allows to establish that in steady-state, the arm reaches a position $\theta_2^\star = \beta^\star - \pi/2$, where $\beta^\star$ is a solution of the equation:

$$\frac{\tan(\beta)}{a_1 + a_2 \sin(\beta)} = \frac{\dot{\theta}_1^2}{g},$$

with DH parameters a_1, a_2 and β defining the angular position of the shoulder joint about axis $\mathbf{z}_1$ measured starting from the negative portion of the vertical axis $\mathbf{y}_1$ till reaching axis $\mathbf{x}_2$ aligned with the robot's second link, i.e. $\beta = \theta_2 - \pi/2$. Solving the above equation allows to obtain the same steady-state results as those presented in fig. 1.11 further confirming the consistency of the model with respect to what is physically expected.

1.2.2. Some Features

With ARS it is possible to, among others things,

- have multiple robots in the same simulation
- control a robot in real-time with a keyboard, as if driving a real one
- change the environment e.g. set a trimesh as ground instead of a plane
- alter the visualization e.g. move camera, change colors, add objects
- add new objects to the simulation (or remove some) at runtime
- have multiple, independent instances of the same simulation program

1.3. Existing robotics simulators for Python

When considering the publicly available software for robotics simulation using Python, it is very important to exclude those that seem to be written in this languague but are actually implemented in C/C++. In many cases they provide wrappers (i.e. bindings) for Python and other languages, which are usually generated automatically with tools like SWIG (Simplified Wrapper and Interface Generator) (SWIG, 2013). A special case is *breve* (Klein, 2013, 2003), which allows for simulations to be written in Python (or in a software-specific language named *steve*), but it is implemented in C++ thus it is not possible to modify or extend it in a straight-forward way using Python, its plugin architecture also

requires C/C++, and unfortunately it is no longer under active delopment (its latest release was in February 2008).

To the best of our knowledge, the only existing software that complies with the minimum requirements aforementioned is *MORSE* (Modular OpenRobots Simulation Engine) (Echeverria, Lassabe, Degroote, & Lemaignan, 2011; Echeverria et al., 2012), presented in 2011, about the same time ARS's development began. It is a "generic simulator for academic robotics" (LAAS/CNRS, 2013) that has had good reception by the scientific community (Echeverria et al., 2012; Lemaignan et al., 2012), showing there is a need for tools like this. ARS and MORSE share some goals and features, but there is a clear divergence in their focus. While ARS's objective is to provide robotics simulations with physically-accurate results, MORSE outsources the physics and collision processing to Bullet (physics library meant for "Real-Time Physics Simulation" (Advanced Micro Devices, 2013)) through Blender's (very sophisticated open-source graphics software (Blender Foundation, 2013); see the interface for editing in fig. 1.12) Game Engine. In the field of games physical accuracy is not the primary objective: the concern is to display simulations that look visually realistic and run fast and fluid enough, i.e. real-time with large frame rates. For ARS, execution speed is a secondary goal, and visualization should be just a precise graphic representation of what is actually happening in the simulation.

MORSE is to be "used in different contexts for the testing and verification of robotics systems as a whole, at a medium to high level of abstraction" (Echeverria et al., 2011). For example, the robots section of the components library includes submarine, helicopter, quadrotor, Hummer, among others, all of them with greatly simplified dynamics and collision –or even without considering them at all–, but are nonetheless useful for some kind of simulations. The model depicted in fig. 1.13 is an example of this.

Being tightly coupled to Blender in many different areas restricts in a great deal how MORSE can be customized in sensible aspects such as physics and collision (some libraries provide only one or the other, unlike ODE and Bullet). For example, customizing the use of Bullet (e.g. adjusting parameters, solvers, etc.) is not currently possible, while replacing

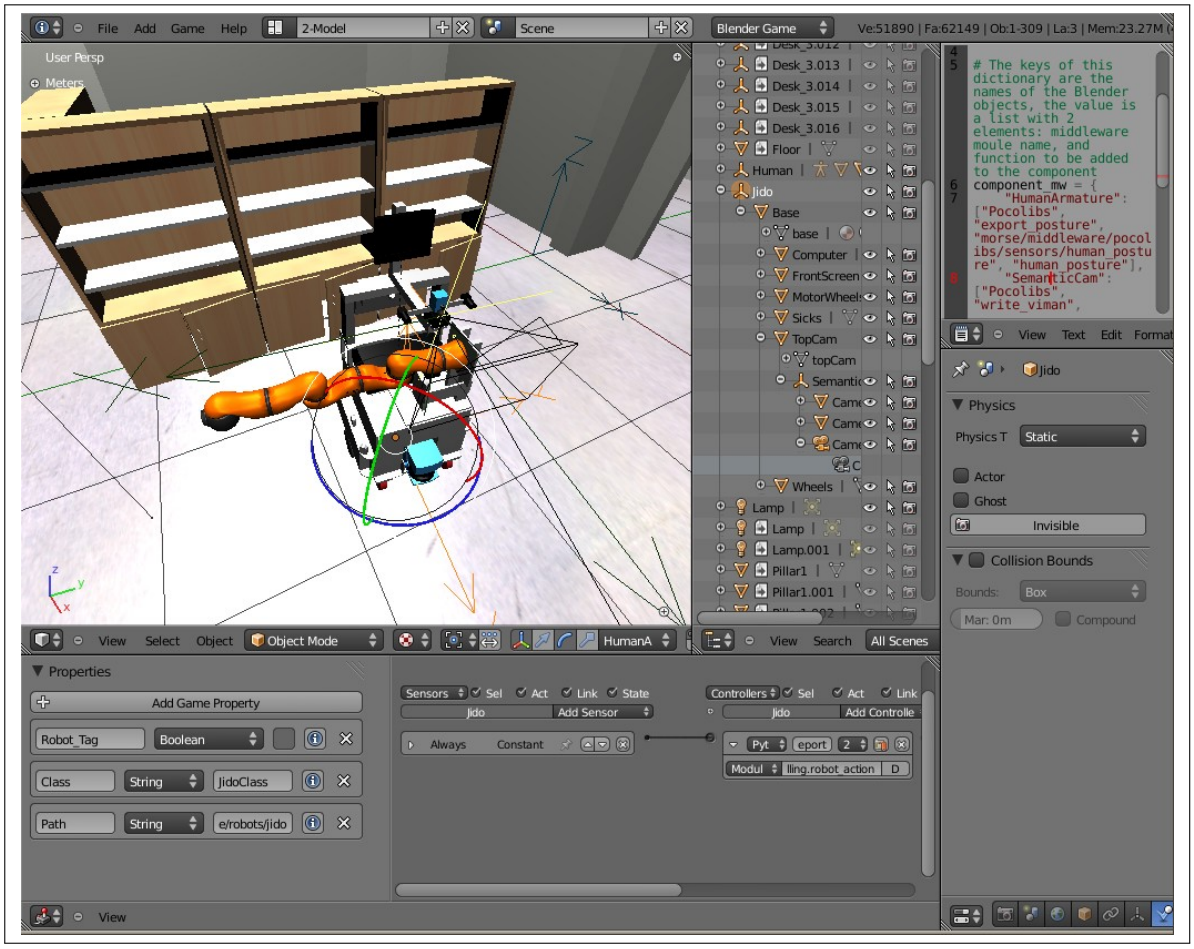


FIGURE 1.12. MORSE interface for editing i.e. Blender. Source: (LAAS/CNRS, 2013).

it would require redesigning the software from scratch. Also related to the mentioned coupling, it is not possible to import MORSE simulation code from the Python interpreter or plain modules (as it is done with normal Python software) to do things such as run simulations, customize entities, manage simulations or process data. This constrains the ability of users and developers to take advantage of the simulator.

Related specifically to Python, there are some issues with MORSE. The most notable is that it does not run on Python 2.x even though this is far more popular than Python 3.x at this moment, and the default version in most of the latest releases of operating systems

FIGURE 1.13. A MORSE simulation view. Source: (LAAS/CNRS, 2013).

where it comes pre-installed such as Mac OS X, Ubuntu, Debian and Fedora. With regard to packaging and distribution, it has a non-standard installation process (can not use `easy_install` (PyPA, 2013), `pip` (PyPA, 2013) nor last resort `python setup.py install` (Python Software Foundation, 2013)), requires non-standard compilation tools (Kitware's CMake (Kitware, 2013a)) and its installed packages are not in the Python system path without manual modifications. It is not available at PyPI (Python Package Index, the official repository of third-party Python packages (Python Software Foundation, 2011b)).

1.4. Summary of Contributions

Following the requirements indicated in subsection 1.1.4 we have developed a robotics simulator for the Python community. This tool, which we have named *ARS* (for autonomous robot simulator), should help to reduce the overall costs of designing, testing and programming complex robot systems, such as mobile manipulators or humanoids. The main contributions of ARS are that it is simple to use, yet general enough to allow modeling any type of rigid multi-body system, e.g. open-loop kinematic manipulators, closed-loop kinematic chains, mobile manipulators, or humanoid robots. As of December 2013 it has been downloaded over 20,000 times.

Some preliminary results that included basic actuators and no sensors functionality were presented in October 2012 at the international conference *XV Congreso Latinoamericano de Control Automático* (CLCA), organized in cooperation with the *International Federation of Automatic Control* (IFAC), appended to this document in appendix A. The complete work was submitted in December 2013 as a paper to the ISI journal *Simulation Modeling Practice and Theory*, included in appendix B.

1.5. Thesis Outline

In the introduction we described the problem tackled in this thesis including some related concepts and issues; showed some examples of simulations using ARS; and mentioned the relevant existing software and compared them with our simulator. The rest of the thesis is organized as follows: chapter 2 discusses the design and implementation details of the proposed simulation tool, followed by chapter 3, which shows applications examples and how easy it is to create and run a robot simulation. The conclusions and discussion on future enhancements to ARS are presented in chapter 4.

2. SIMULATOR DESIGN

In this section we describe what we need from the external libraries that support key aspects of the software, which alternatives were considered and the reasons that lead us to select ODE and VTK. Then we describe the design and implementation of ARS, particularly how the code is organized and what is the concern of each one of the most relevant packages and modules.

2.1. Enabling Technologies

The simulator needs a library for physics and collision (it could be different libraries but it is better a single one that provides both functionalities), and another for visualization. Both need to have 3D support, be cross-platform and have an open source license.

For physics/collision, the alternatives were Bullet (Advanced Micro Devices, 2013), Newton and Open Dynamics Engine (ODE) (Open Dynamics Engine, 2013b), of which we chose ODE because it is a popular option for robotics software (Drumwright et al., 2010; Harris & Conrad, 2011), has relatively high accuracy (Castillo-Pizarro et al., 2010), its code is mature (first released in 2001 (Open Dynamics Engine, 2013c)) and has official, well developed Python bindings. However, as most physics engines, it was designed to be fast enough to be used by games, which require real-time simulation. This performance was achieved with many simplifications, at the expense of accuracy, some of which affect the simulation of common robotics scenarios. Nonetheless, Drumwright et al. recently identified the relevant shortcomings and suggested solutions in (Drumwright et al., 2010).

With respect to visualization we considered OGRE, OpenSceneGraph, Panda3D, Visualization Library and Visualization Toolkit (VTK) (VTK, 2013b). We chose VTK, an "open-source software system for 3D computer graphics, image processing and visualization" (VTK, 2013b), because it is a very powerful library, has a mature codebase (presented in (1996) (Schroeder et al., 1996)), it is oriented to scientific software (created by GE Corporate R&D researchers (VTK, 2013a)), includes official Python bindings, supports a broad collection of graphics cards and is developed by a company with many popular open source

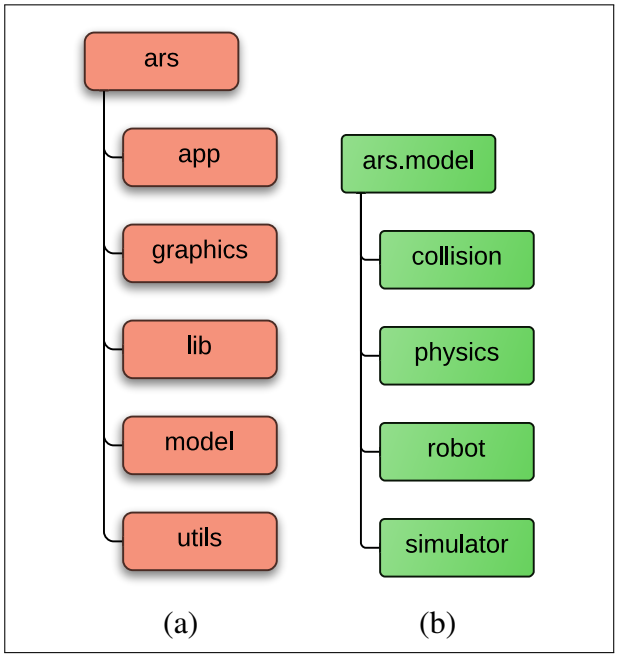


FIGURE 2.1. Packages hierarchy.

products, Kitware (Kitware, 2013b). On the other hand, using VTK seems a little excessive because the features we need are a small fraction of those provided. By being such a big library it has sizable impact on computer resources and its distribution files are somewhat large.

2.2. Implementation and Architecture

The code is organized hierarchically in five root-level packages: `app`, `graphics`, `lib`, `model` and `utils` (see fig. 2.1 (a)), and some of those contain more packages or modules as well. The core one is `model`, which in turn includes `collision`, `physics`, `robot` and `simulator` (fig. 2.1 (b)).

`model.collision` has two main modules: `base` and `ode_adapter`. The former defines the basic functionality for collision, as well as the base classes that compose an abstract interface to the library developers choose to use, e.g. `Space` and `Geom` –*geom* is a geometry object– (parent class of `Ray`, `Trimesh` –*trimesh* is a triangular mesh–, `Box`,

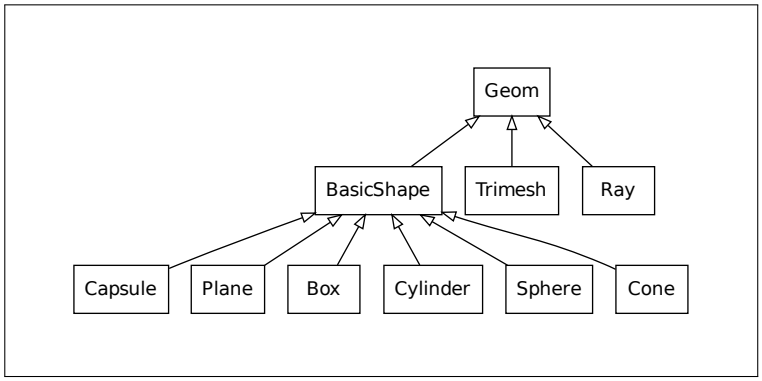


FIGURE 2.2. Subclasses of `collision.base.Geom`.

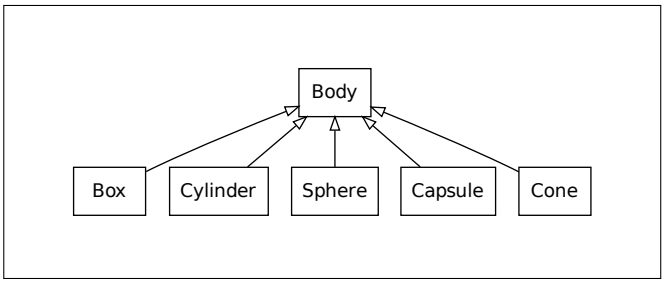


FIGURE 2.3. Subclasses of `physics.base.Body`.

`Sphere`, `Plane`, etc), which wrap the corresponding "native" object that the adapted library uses, assigned to a private attribute. On the other hand, `collision.ode_adapter` has the classes and functions to connect with the collision library included in ODE that implement the interface defined in `base`. The class hierarchy of `Geom` is depicted in fig. 2.2.

Subpackage `physics`, as `collision`, also has modules `base` and `ode_adapter`. Instead of `Space` and `Geom` there is `World` and `Body` (parent class of `Box`, `Sphere`, `Capsule`, `Cylinder` and `Cone`). The subclasses of `Body` can be appreciated in fig. 2.3.

Subpackage `simulator` contains all the classes directly related to a simulation like the "environment" (`Simulation`) and the objects to simulate (fig. 2.4).

In the module `robot.joints` there are both actuated and freely-moving joints. Because they share many properties, they are abstracted with a common parent class named `Joint` (fig. 2.5). In the latest release of ARS the only actuators included are rotary (used for hinges and wheels) and slider, both with one degree of freedom.

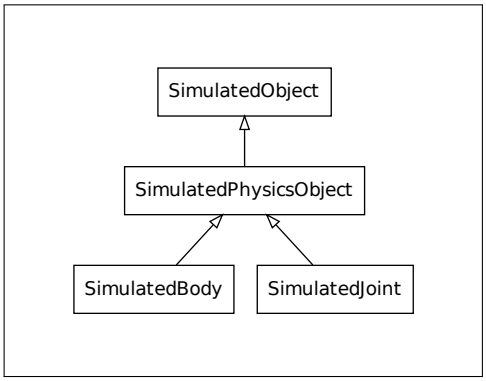


FIGURE 2.4. Subclasses of `simulator.SimulatedObject`.

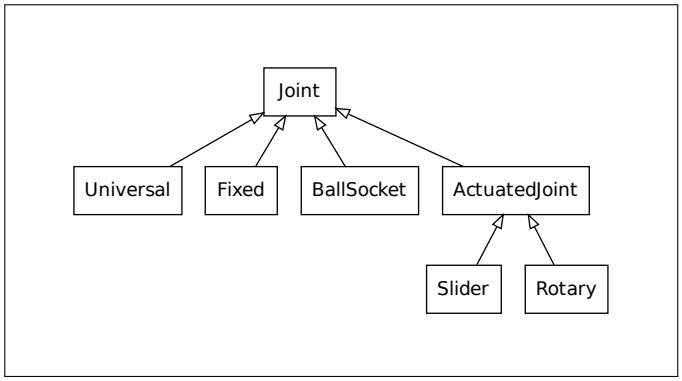


FIGURE 2.5. Subclasses of `robot.joints.Joint`.

Module `robot.sensors` has two top-level classes: `BaseSourceSensor` (fig. 2.6 (a) and 2.6 (b)) and `BaseSignalSensor` (fig. 2.6 (c)). The former has an associated object that is the source of information. The program must call the `on_change` method every time it wants to save a measurement. On the other hand, sensors based on signals work by subscribing to them upon creation (and optionally specifying a given "sender/emitter"), so each time they are fired a measurement will be recorded.

The package `app` handles the program flow through its class `Program`. This is the main class to understand in ARS, since simulation programs are defined by subclassing it. Each simulation run is an instance of a derived class. It wraps the corresponding `model.simulator.Simulation` instance and the associated visualization window. It also provides the methods to allocate and release the resources necessary to run the simulations. The class `ActionMap` is a dictionary-like structure used to map keys to

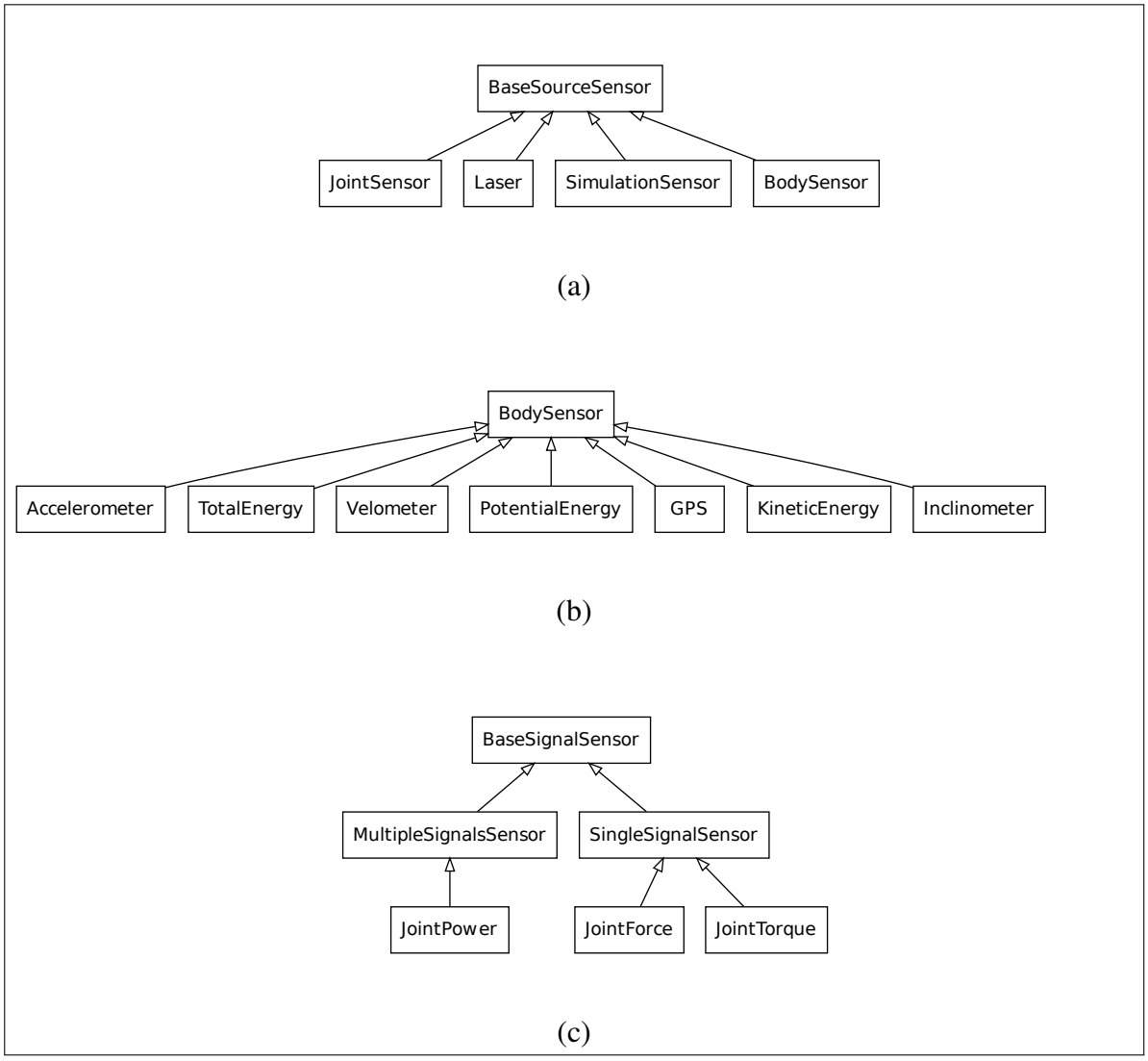


FIGURE 2.6. In package `robot.sensors`, subclasses of: (a) `BaseSourceSensor` (one level deep), (b) `BodySensor`, (c) `BaseSignalSensor`.

pre-defined actions (e.g. apply torque to a wheel when the user presses a certain key) by associating a callable object (e.g. function, method) to each key.

Package `graphics` contains modules `base` and `vtk_adapter`. The former defines the interface to implement by the latter, composed by classes `Engine`, `Entity` and its subclasses (fig. 2.7), and `ScreenshotRecorder`.

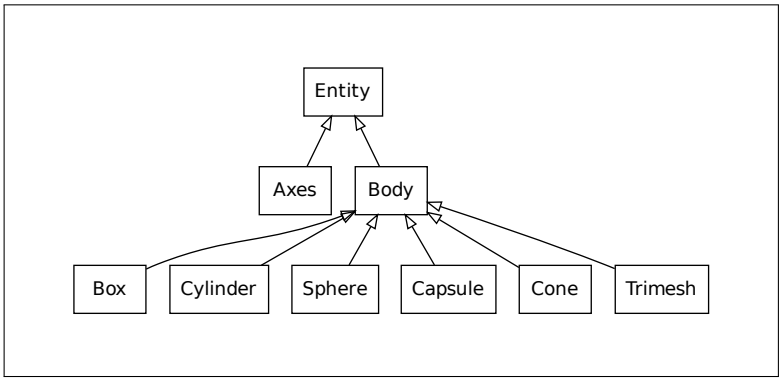


FIGURE 2.7. Subclasses of `graphics.base.Entity`.

3. APPLICATIONS EXAMPLES

In this section we present some examples that illustrate the physical accuracy of the simulator, its simplicity and some features. Due to space limitations only the code for the first examples is included but all the source code is stored in the project's repository (ARS, 2013b), and there are demonstration videos available (ARS, 2013a) too.

The first example is a "Hello World" type of program that shows the smallest piece of code necessary to define a simulation and run it. Next there is a group of simulations that show, in detail, how to use different sensors to record data along the execution (the results are plotted individually in fig. 3.2). Then we make a mobile manipulator follow a trapezoidal velocity profile with a proportional-derivative controller, analyzing the torque applied to the wheels as the time-dependent set point changes. Finally, we mention other features available in ARS as well as possible applications. Some snapshots of the examples are shown in fig. 3.1.

The parameters values for all the examples are ARS's defaults: time step of 0.4 ms and gravity acceleration of 9.81 m/s. ODE-specific parameters are defined in the corresponding `ode_adapter` module in the physics and collision packages. They are all adimensional quantities: global *ERP* (Error Reduction Parameter) of 0.2, global *CFM* (Constraint Force Mixing) of 10^{-10}, contact joint bounce of 0.2, and contact joint mu of 500.

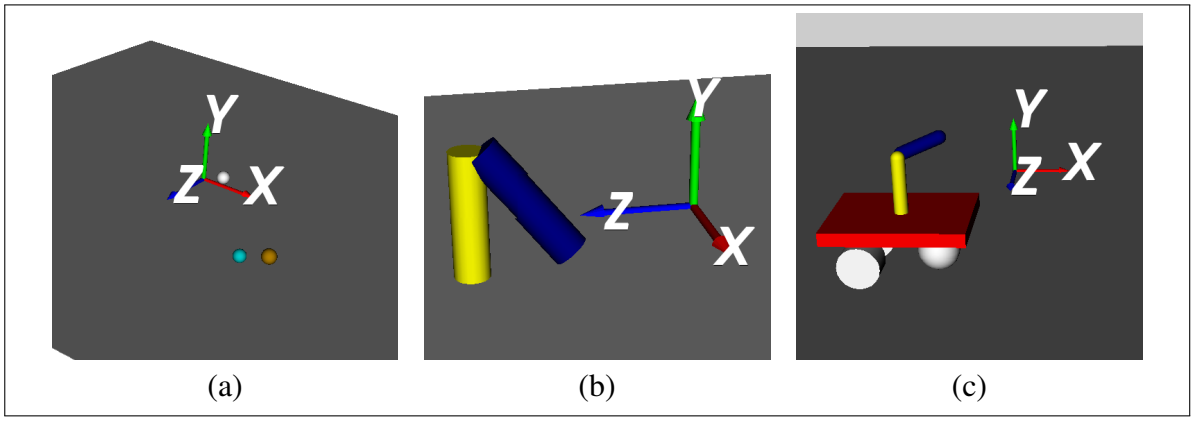


FIGURE 3.1. Basic simulations implemented using ARS: (a) falling balls, (b) simple 2-DOF arm with a waist and shoulder joint, (c) mobile manipulator with differential-drive base and 2-DOF arm.

3.1. Hello World

The following code shows what is necessary to define a simple simulation program. In this case, a sphere of radius 0.5 m and 1.0 kg is created at position $(1, 1, 1)$ meters.

```
from ars.app import Program

class FallingBall(Program):

    def create_sim_objects(self):
        self.sim.add_sphere(0.5, (1, 1, 1), mass=1)
```

The procedure is very straight-forward:

(i) From package `ars.app` import class `Program`.

(ii) Define a subclass of `Program` (named `FallingBall` in this case).

(iii) Implement in it the required method `create_sim_objects`.

(iv) To create objects and add them to the environment (a sphere in this case) add calls to methods of class `Simulation` (of which attribute `sim` is an instance).

With these steps, a custom simulation has been defined. To run it, just two lines of code are necessary: one to create an instance of `FallingBall` and another to begin the simulation. It will run until the just opened visualization window is closed by the user.

```
sim_program = FallingBall()
sim_program.start()
```

The simplicity of the procedure (just five lines of code plus those regarding the bodies to be created) necessary to define and execute a 3D multibody physics simulation saves the user both time and effort, and makes the process less error prone.

3.2. Sensors

Body, joint and environment sensors are available to record simulation data as well as to trigger user-defined behavior. The examples in this section show how to use some

specific sensors but the general procedure is the same for those not covered here. Because some basic code is common to all the examples, we refactor it into a class named `SensorExampleBase`, which is also useful to show how ARS's design allows and encourages code reuse.

The following is the standard way to initialize a sensor, subscribe it to signals (i.e. events) and define what should the program do each time those are fired. After the simulation has ended, the collected data is still available, perhaps to be printed or saved for offline processing.

```
from ars.app import Program, dispatcher
from ars.model.simulator import signals

class SensorExampleBase(Program):

    def __init__(self):
        Program.__init__(self)
        dispatcher.connect(self.on_post_step, signals.SIM_POST_STEP)

    def create_sim_objects(self):
        obj_name = self.sim.add_sphere(0.5, (1, 1, 1), mass=1)
        self.sphere = self.sim.get_object(obj_name)
        self.create_sensor()

    def create_sensor(self):
        # self.sensor = ...
        raise NotImplementedError("Implement this method in subclasses")

    def on_post_step(self):
        time = self.sim.sim_time
        self.sensor.on_change(time)

    def print_sensor_data(self):
        collected_data = self.sensor.data_queue
```

```
        print(collected_data)
```

The differences with *Hello World* are:

- override constructor to subscribe a method (`on_post_step`) to a signal (`SIM_POST_STEP`)
- in the creation of simulation objects, store in attribute `sphere` a reference to the object just added to the simulation, and call method `create_sensor`
- new method `create_sensor`, which must be implemented by subclasses, to instante a sensor and store it in new attribute `sensor`
- new method `on_post_step` to tell the sensor to record a new data instance
- new method `print_sensor_data` to print the collected data to console (changing it to write to a text file is trivial)

To run any of the following examples, it is the same as in *Hello World*:

```
sim_program = MySensorExample()
sim_program.start()
```

where `MySensorExample` is a subclass of `SensorExampleBase`. To print the data collected by the sensor, it is just an additional call:

```
sim_program.print_sensor_data()
```

With the `GPS` sensor attached to a body it is possible to record its position in every simulation step. In the corresponding plot (fig. 3.2 (a)) it can be seen how the ball begins at a height of 1 m and then falls until it reaches the floor and bounces a few times. The lowest position is 0.5 m because that is the object's radius.

```
class GPSExample(SensorExampleBase):


    def create_sensor(self):
        self.sensor = GPS(body=self.sphere)
```

Analogously, to record body's velocity, acceleration, or energy (potential or kinetic), the last sentence should be replaced, respectively, by one of the following:

```
self.sensor = Velometer(body=self.sphere)


self.sensor = Accelerometer(body=self.sphere, self.sim.time_step)
```

```
self.sensor = PotentialEnergy(body=self.sphere, self.sim.gravity)
```

```
self.sensor = KineticEnergy(body=self.sphere)
```

while for system's total energy it should be:

```
self.sensor = SystemTotalEnergy(self.sim, disaggregate=True)
```

Note that some sensors require different arguments: for the accelerometer, the simulation time step; for the potential energy, the gravity vector; and for the system's total energy, the simulation object as well as whether the potential and kinetic components should be stored separately or combined.

In the plot of fig. 3.2 (b) it can be appreciated that the ball hits the floor significatively three times: 0.3194 seconds (at a speed of 3.13135 m/s), 0.4470 seconds (at a speed of 0.62156 m/s) and 0.4722 seconds (at a speed of 0.11898 m/s). The coefficient of restitution is 0.200000 every time, which is coherent with ODE's contact joint bounce parameter value of 0.2, whereas the acceleration is 9394 m/s^2, 1865 m/s^2 and 357 m/s^2 respectively (fig. 3.2 (c)).

In the given example there is only one body thus the system energy sensor data (fig. 3.2 (d)) corresponds to the kinetic and potential energy of the single sphere. A more interesting plot is created by using two spheres. For that we define a different parent class for the simulations (based in `SensorExampleBase`) that creates two spheres, both with the same properties (radius of 1.0 m and mass of 1.0 kg) but at different heights and separated enough so they do not collide. The result is in fig. 3.3 (a).

```
class SensorExampleBase2(SensorExampleBase):


    def create_sim_objects(self):
        self.sim.add_sphere(1.0, (3, 4, 1), mass=1)
        self.sim.add_sphere(1.0, (5, 3, 1), mass=1)
        self.create_sensor()
```

This new simulation allows us to demonstrate how to use the `Laser` sensor. The following code sets it in space at position $(1.5, 1.5, 1)$ meters and orientation $(1, 0, 0)$, the X axis (the

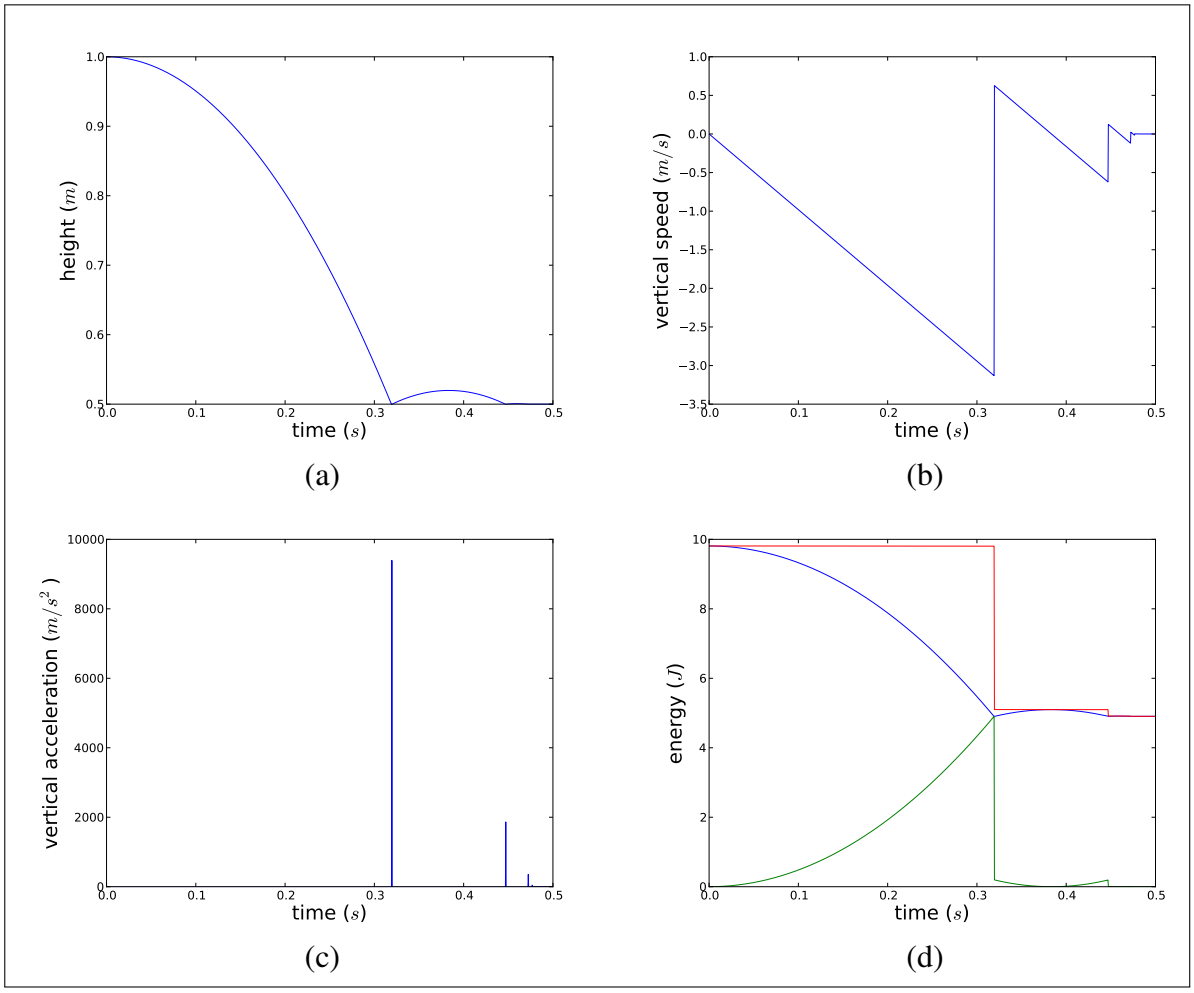


FIGURE 3.2. Data collected using sensors in the single falling ball example: (a) body vertical position, (b) body vertical speed, (c) body vertical acceleration, (d) system energy (from top to bottom: total, potential, kinetic).

default is $(0, 0, 1)$, the Z axis, but with a helper function it is rotated $90°$ around $(0, 1, 0)$, the Y axis). The sensor will measure the distance to the nearest object intersecting its projection line, up to a maximum distance.

```
class LaserExample(SensorExampleBase2):


    def create_sensor(self):
        space = self.sim.collision_space
        rot_matrix = calc_rotation_matrix((0, 1, 0), pi / 2)
```

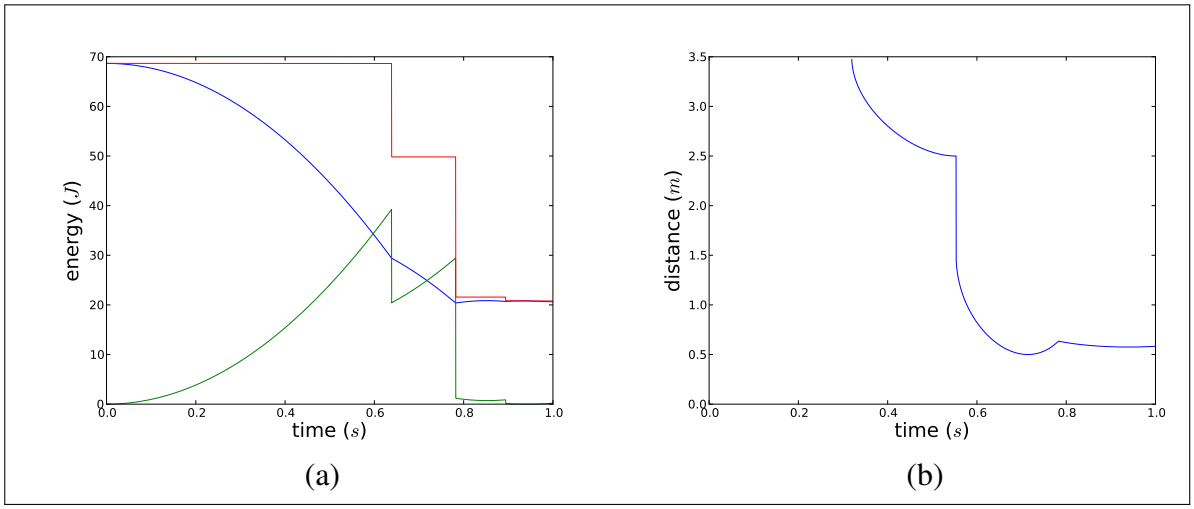


FIGURE 3.3. Data collected using sensors in the two falling balls example: (a) system energy (from top to bottom: total, potential, kinetic), (b) laser-measured distance.

```
self.sensor = Laser(space, max_distance=1000.0)
self.sensor.set_position((1.5, 1.5, 1))
self.sensor.set_rotation(rot_matrix)
```

The plot of fig. 3.3 (b) shows that the intersection begins (and corresponds to the maximum distance, 3.5 m) 0.32 seconds into the simulation, when the bottom of the furthest sphere is at $(5, 1.5, 1)$ m. When the system comes to rest, the distance measured by the sensor is 0.63397 m, approximately equal to $3.0 - 1.5 - \cos(\arcsin(0.5)) = 1.5 - \cos 30°$, the theoretical value obtained by simple geometry.

3.3. Mobile Manipulator with Freely-Coasting 2-DOF Arm

This example considers a mobile manipulator composed of a 2-wheeled differential drive mobile base and robot arm mounted on top of the base. The arm has only two degrees of freedom (waist and shoulder) because such a design allows to easily visualize and understand the influence of the motion of the robot's base on its arm joints. More complex arms can be easily implemented, but the higher complexity due to the multiple interacting robot links makes it more difficult to understand how motion propagates across the arm and its joints. The mobile manipulator is shown in fig. 3.1 (c).

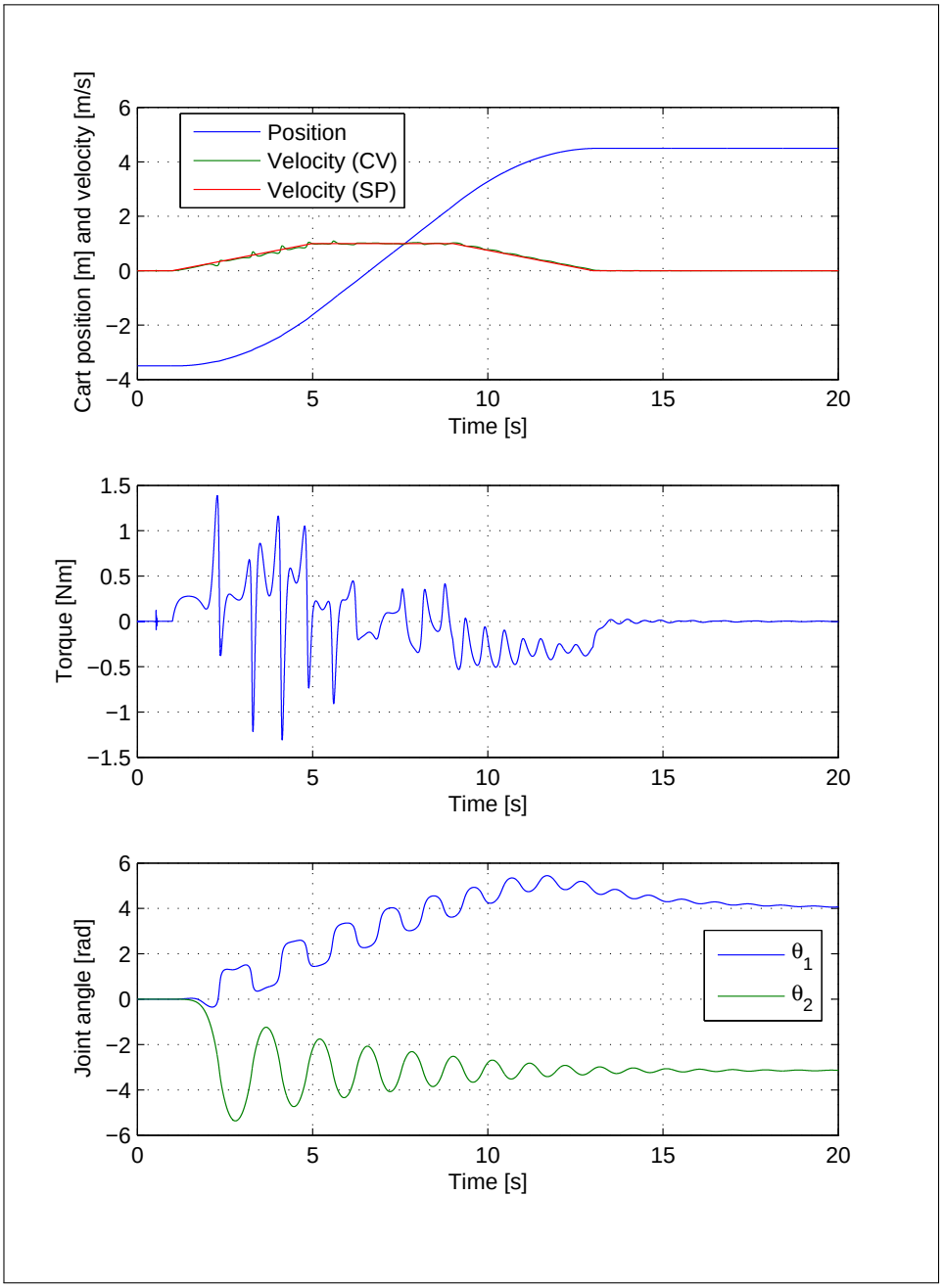


FIGURE 3.4. Mobile manipulator example: base position, velocity and velocity profile (top), total wheels motor torque (middle), arm joint angles (bottom).

Torque is applied to the wheels of the robot's base using a proportional-derivative (PD) controller so that the base moves in a straight line following a trapezoidal velocity profile. Figure 3.4 shows the reference velocity profile (red curve: Velocity set-point (SP)), the

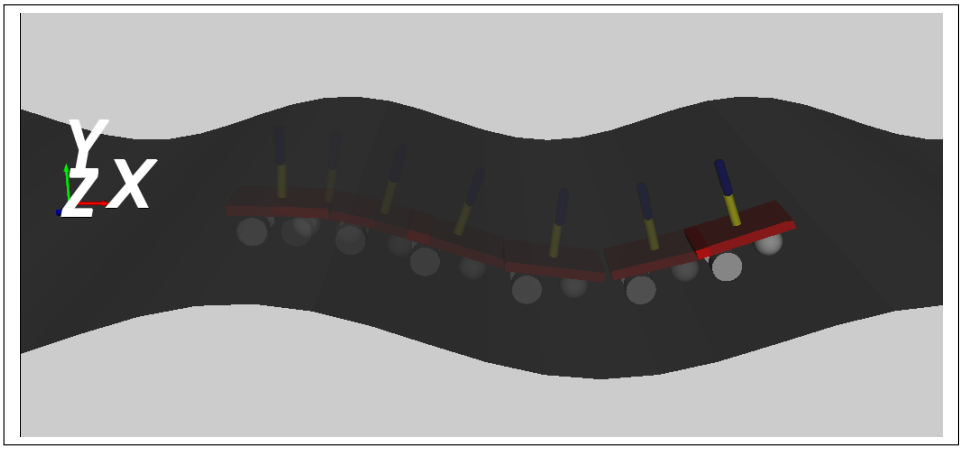


FIGURE 3.5. Mobile manipulator with a PD-controlled arm driven along a straight line on a sinusoidal terrain (modeled as trimesh).

actual velocity (green curve: Velocity controlled variable (CV)) and the longitudinal position of the robot. The torque applied to the wheels (manipulated variable) is shown in the middle graph of fig. 3.4. The bottom graph of fig. 3.4 shows the arm's joint trajectories. It is possible to see that as the robot accelerates, the shoulder link falls from the unstable equilibrium (vertical position) and swings, thus driving the waist joint to oscillate. The shoulder joint reaches its natural stable equilibrium point (arm down), however the waist joint stabilizes to a non-zero position thanks to the joints friction, but does not return to its original location as the waist joint motor is turned-off. It is interesting to note that the wheels must apply non-constant torques in order to comply with the requested longitudinal velocity profile. The reason the torque is not constant despite the trapezoidal velocity profile would require a constant acceleration/deceleration is that the swinging arm is propagating back disturbances to the base.

3.4. Complex simulations

With ARS it is possible to create complex simulations and interact and modify them in different ways. It allows to, among other things, have multiple robots in the same simulation; control a robot in real-time with a keyboard, as if driving a real one; change the

environment e.g. set a trimesh as ground instead of a plane (fig. 3.5); alter the visualization e.g. move camera, change colors, add objects; add new objects to the simulation (or remove some) at runtime; and have multiple, independent instances of the same simulation program. Some of these aspects can be appreciated in the posted videos (ARS, 2013a).

4. CONCLUSIONS AND FUTURE WORK

This paper presented ARS, a free and open-source robotics simulator for research and education, aimed at providing a tool that the Python scientific software ecosystem lacked. It focuses in physical accuracy and simplicity, enabling users to realistically simulate the dynamics of multibody mechanical systems that include linear and rotary actuated joints, and implement proprioceptive sensors (e.g. encoders, IMUs, system energy) and exteroceptive sensors (e.g. laser range scanners, GPS). Thanks to these features, ARS can be used to simulate custom environments with multiple robots or evaluate new robot designs, as well as for controller synthesis for complex robots such as mobile manipulators. With examples we show how simple it is to define a simulation, and how the user can reuse and extend those definitions by subclassing them.

Ongoing work includes creating models for real-life robots and hardware as well as to build a GUI to further simplify the software usage, with interactive simulation creation, editing and save; multi-view visualization window; and logged and real-time data plotting. Also, to take advantage of the software architecture, new adapters for the physics, collision and visualization abstraction layers are being developed.

The project goals are to increase its popularity (as of December 2013 it has been downloaded over 20,000 times) to make it the standard open source Python robotics simulator –like the projects in SciPy (Jones et al., 2001–) are in their own fields– and expand its developer community, incorporating external code contributions.

References

Advanced Micro Devices. (2013, November). *Bullet collision detection and physics library.* `http://bulletphysics.org`.

ARS. (2013a, November). *Project videos at Vimeo.* `https://vimeo.com/arsproject`.

ARS. (2013b, November). *Source code repository.* `https://bitbucket.org/glarrain/ars`.

Baraff, D. (2001). Physically based modeling: Rigid body simulation..

Blender Foundation. (2013, November). *Blender home page.* `http://www.blender.org`.

Castillo-Pizarro, P., Arredondo, T., & Torres-Torriti, M. (2010). Introductory survey to open-source mobile robot simulation software. In *Robotics symposium and intelligent robotic meeting (lars), 2010 latin american* (p. 150-155). doi: 10.1109/LARS.2010.19

Chiang, L. E. (2010). Teaching robotics with a reconfigurable 3d multibody dynamics simulator. *Computer Applications in Engineering Education*, *18*(1), 108–116. doi: 10.1002/cae.20202

Corke, P. (1996, March). A robotics toolbox for matlab. *IEEE Robotics and Automation Magazine*, *3*(1), 24-32. doi: 10.1109/100.486658

Diankov, R. (2010). *Automated construction of robotic manipulation programs*. Unpublished doctoral dissertation, The Robotics Institute, Carnegie Mellon University.

Drumwright, E., Hsu, J., Koenig, N., & Shell, D. (2010). Extending open dynamics engine for robotics simulation. In N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, & O. von Stryk (Eds.), *Simulation, modeling, and programming for autonomous robots* (Vol. 6472, p. 38-50). Springer Berlin / Heidelberg. doi: 10.1007/978-3-642-17319-6_7

Echeverria, G., Lassabe, N., Degroote, A., & Lemaignan, S. (2011). Modular open robots simulation engine: MORSE. In *Robotics and automation (icra), 2011 ieee international conference on* (p. 46-51). doi: 10.1109/ICRA.2011.5980252

Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., ... Stinckwich, S. (2012). Simulating complex robotic scenarios with MORSE. In I. Noda, N. Ando, D. Brugali, & J. Kuffner (Eds.), *Simulation, modeling, and programming for autonomous robots* (Vol. 7628, p. 197-208). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-34327-8_20

Evans-Pughe, C. (2006, dec.). Design - getting it together - it's slow and expensive to build physical prototypes. *Engineering Technology*, *1*(9), 38 -41.

Featherstone, R. (2008). *Rigid body dynamics algorithms*. Springer.

Harris, A., & Conrad, J. (2011). Survey of popular robotics simulators, frameworks, and toolkits. In *Southeastcon, 2011 proceedings of ieee* (p. 243-249). doi: 10.1109/SECON.2011.5752942

Johns, K., & Taylor, T. (2008). *Professional microsoft robotics developers studio*. Wiley Publishing.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). *SciPy: Open source scientific tools for Python.* `http://www.scipy.org`.

Kitware. (2013a, November). *CMake home page.* `http://www.cmake.org`.

Kitware. (2013b, June). *Open source / VTK.* http://www.kitware.com/opensource/vtk.html.

Klein, J. (2003). breve: a 3d environment for the simulation of decentralized systems and artificial life. In *Proceedings of the eighth international conference on artificial life* (pp. 329–334). Cambridge, MA, USA: MIT Press.

Klein, J. (2013, November). *The breve simulation environment.* http://www.spiderland.org.

Koenig, N., & Howard, A. (2004, sept.-2 oct.). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent robots and systems, 2004. (iros 2004). proceedings. 2004 ieee/rsj international conference on* (Vol. 3, p. 2149 - 2154 vol.3). doi: 10.1109/IROS.2004.1389727

Kovačić, Z., Bogdan, S., Petrinec, K., Reichenbach, T., & Punčec, M. (2001, June). Leonardo – the off-line programming tool for robotized plants. In *Proceedings of the 9^{th} mediterranean conference on control and automation, june 27-29.* Dubrovnik, Croatia.

Kumar, K., & Reel, P. (2011, march). Analysis of contemporary robotics simulators. In *Emerging trends in electrical and computer technology (icetect), 2011 international conference on* (p. 661 -665). doi: 10.1109/ICETECT.2011.5760200

LAAS/CNRS. (2013, October). *OpenRobots - MORSE.* http://www.openrobots.org/wiki/morse.

Lemaignan, S., Echeverria, G., Karg, M., Mainprice, J., Kirsch, A., & Alami, R. (2012). Human-robot interaction in the MORSE simulator. In *Human-robot interaction (hri), 2012 7th acm/ieee international conference on* (p. 181-182).

Marhefka, D., & Orin, D. (1996, jun). Xanimate: an educational tool for robot graphical simulation. *Robotics Automation Magazine, IEEE*, *3*(2), 6 -14. doi: 10.1109/100.511779

McMillan, S., Orin, D., & McGhee, R. (1995, may). Object-oriented design of a dynamic simulation for underwater robotic vehicles. In *Robotics and automation, 1995. proceedings., 1995 ieee international conference on* (Vol. 2, p. 1886 -1893 vol.2). doi: 10.1109/ROBOT.1995.525541

Millington, I. (2007). *Game physics engine development*. Elsevier.

Open Dynamics Engine. (2013a, May). *Community wiki.* `http://ode-wiki.org/wiki/`.

Open Dynamics Engine. (2013b, May). *Home.* `http://www.ode.org`.

Open Dynamics Engine. (2013c, May). *Project at SourceForge.* `http://sourceforge.net/projects/opende/`.

Open Source Initiative. (2013, November). *Open source licenses.* `http://opensource.org/licenses`.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., ... Dorigo, M. (2011, sept.). ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics. In *Intelligent robots and systems (iros), 2011 ieee/rsj international conference on* (p. 5027 -5034). doi: 10.1109/IROS.2011.6094829

PyPA. (2013, November). *setuptools documentation - easy_install.* `http://pythonhosted.org/setuptools/easy_install.html`.

Python Software Foundation. (2011a, December). *Python home.* `http://python.org`.

Python Software Foundation. (2011b, December). *Python Package Index (PyPI).* `https://pypi.python.org/pypi`.

Python Software Foundation. (2013, November). *Python documentation - installing python modules / standard build and install.* `http://docs.python.org/2/install/#standard-build-and-install`.

Reichenbach, T. (2009). A dynamic simulator for humanoid robots. *Artificial Life and Robotics*, *13*, 561-565. doi: 10.1007/s10015-008-0508-6

Schroeder, W. J., Martin, K. M., & Lorensen, W. E. (1996). *The design and implementation of an object-oriented toolkit for 3d graphics and visualization.* doi: 10.1.1.24.5958

SWIG. (2013, November). *Home.* `http://www.swig.org`.

VTK. (2013a, June). *About.* `http://www.vtk.org/VTK/project/about.html`.

VTK. (2013b, June). *Home.* `http://www.vtk.org/`.

Wettach, J., Schmidt, D., & Berns, K. (2010, November). Simulating vehicle kinematics with SimVis3D and Newton. In *Simulation, modeling, and programming for autonomous robots* (Vol. 6472, p. 156-167). Berlin Heidelberg: Springer-Verlag.

Zlajpah, L. (2008). Simulation in robotics. *Mathematics and Computers in Simulation*, *79*(4), 879 - 897. doi: 10.1016/j.matcom.2008.02.017

APPENDIX A. IFAC-CLCA 2012 PAPER - A PYTHON PACKAGE FOR ROBOT SIMULATION

A Python Package for Robot Simulation ⋆

Germán Larraín-Muñoz, Miguel Torres-Torriti *

* *Escuela de Ingeniería, Pontificia Universidad Católica de Chile, Vicuña Mackenna 4860, Santiago, Chile (e-mail: germanlarrainm@gmail.com, mtorrest@ing.puc.cl)*

Abstract: Robotics engineering requires modeling and simulation tools for mechanical design, controller synthesis, robot programming and operator training. Often each task employs its own simplifications and assumptions, however with the growth the advancements of physics engines, it is becoming possible to develop simulators that are both visually realistic and physically accurate, and that can integrate not only the usual systems dynamics and collisions, but also sensor and actuator models, as well as robot-environment interactions, everything in a single package. This paper describes the design of a tool for physically accurate robot and multi-body systems simulation. The software is implemented in Python integrating the Open Dynamics Engine (ODE) and the Visualization Toolkit (VTK). The simulator is open-source, modular, easy to learn and use, and can be a valuable tool to a wide audience encompassing engineers, researchers and students involved in robot mechanical design, as well as robot controllers and intelligence development.

Keywords: mobile manipulator robots; dynamics; physical simulation; public domain software; open dynamics engine; Python; VTK.

1. INTRODUCTION

Future autonomous robots will require better self-simulation capabilities to become more self-aware and capable of taking decisions in real-time while operating in complex and dynamic environments of the real world (Diankov, 2010; Johns and Taylor, 2008; Koenig and Howard, 2004). However, the existing tools (see recent reviews in (Kumar and Reel, 2011; Castillo-Pizarro et al., 2010)) are often conceived for a specific sub-problems, e.g. motion control, motion planning, navigation, and typically consider simplified dynamical models that do not allow to take into account in a straight-forward way other aspects, such as robot-ground interactions, collisions, energy consumption or advanced sensor models, to name a few. Extending the existing tools is difficult and sometimes even impossible, because their software architecture is not modular enough or forbids modifications. On the other hand, although some of the existing simulators produce visually realistic representations of the robots' motion, the velocity and force profiles are not accurate due to the underlying simplified motion models (Castillo-Pizarro et al., 2010). These reasons have motivated the development of an open-source tool for modeling and simulating multi-robot systems, which we have named *ARS* (for autonomous robot simulator). The main contributions of ARS are that it is simple to use, yet general enough to allow modeling any type of rigid multi-body system, e.g. open-loop kinematic manipulators, closed-loop kinematic chains, mobile manipulators, or humanoid robots. The source code and examples for ARS are publicly available at the project's SourceForge site (ARS Project, 2011). Despite previous efforts to achieve similar goals (Pinciroli et al., 2011; Bonaventura and Jablokow, 2005; Koenig and Howard, 2004), to the best of our knowledge, there does not exist thus far an established open source simulator that could define the simulation standard as in the case of the electronics discipline, where there are several well-adopted simulation tools and languages (Wikipedia, 2011).

This paper is organized as follows. Section 2 discusses the design and implementation details of the proposed simulation tool, followed by section 3, which shows by means of examples the physical accuracy and capabilities of the simulator. The conclusions and discussion on future enhancements to the simulator7 is presented in section 4.

2. SIMULATOR DESIGN

2.1 Programming Language, Libraries and Development Tools

Python (PSF, 2011a) (version 2.6.5) was chosen as the main programming language among other alternatives because it has some characteristics that were deemed essential, and which can be summarized in that Python: (i) is open source unlike Matlab, Java, C#; (ii) has a wide user base that has been growing since its was officially released in 1991; (iii) has been steadily gaining popularity among researchers and educators who see it as an alternative to Matlab because of its libraries like numPy and SciPy (PSF, 2011b); (iv) is easy to learn; (v) improves productivity because of its small overhead compared to C/C++; (vi) is cross-platform and runs on Windows, Mac and Unix-like systems with little or no modification unlike C/C++; and (vii) is multi-domain oriented as the Python Package Index (PyPI) (PYPI, 2011) can attest with applications ranging from basic desktop programs to full-grown programming suites. Python aims to increase developer productivity and it is one the main reasons users choose it. The Python Software Foundation (PSF) officially claims that "*Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs*", (PSF, 2011a).

⋆ This project has been supported by the National Commission for Science and Technology Research of Chile (Conicyt) under Fondecyt Grant 1110343.

The current ARS implementation comprises the following open-source libraries: (i) Open Dynamics Engine (ODE), rev. 1858; (ii) the ODE Python wrapper called PyODE, snapshot 2010.10.13; and (iii) the Visualization Toolkit (VTK), version 5.2.1, for 3D rendering and visualization. ODE was developed by Russell Smith and released in 2001. Nowadays, ODE is one of the most popular tools for the implementation of robot simulation applications (Drumwright et al., 2010). ODE solves simultaneously the Newton-Euler dynamic equations and the contact (collision) constraints by posing the equations governing the systems of rigid bodies as a linear complementarity problem (LCP) in the maximal coordinate system to solve for impulses satisfying the constraints imposed by joints connecting bodies and contact points. 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. (2010). Similarly, VTK is a very popular tool for data visualization and 3D rendering (Schroeder et al., 2000). It has been continuously improved since its release back in 1993.

In addition to the libraries integrated as part of ARS, its development has relied on development tools, such as the Eclipse 3.7.1 Integrated Development Environment (IDE), the Mercurial (Hg) source control management, the PyDev 2.2.4 Python plug-in for Eclipse, and epydoc 3.0.1 for generating the documentation, and graphviz 2..20.2 for drawings and graphs.

2.2 ARS Implementation and Architecture

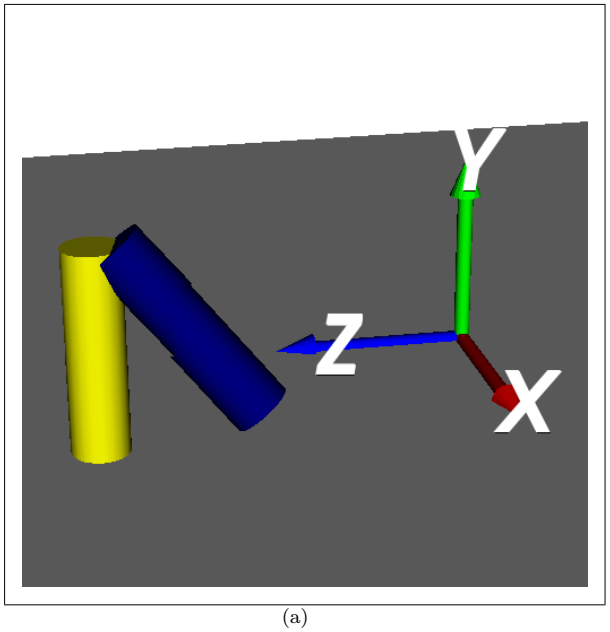
Python programs are composed of files containing definitions and statements called *modules*. Several modules can be grouped together to into higher level units called *packages*. From a practical standpoint, packages are just folders that in addition to the modules contain also a file named `__init__.py`, which is usually empty, but may contain initialization code to be executed when the package is imported into some other module. ARS is divided into several packages and subpackages that group modules and functions with similar scopes and purposes. The packages that make up ARS and their methods is summarized in table 1.

2.3 ARS Installation and Usage

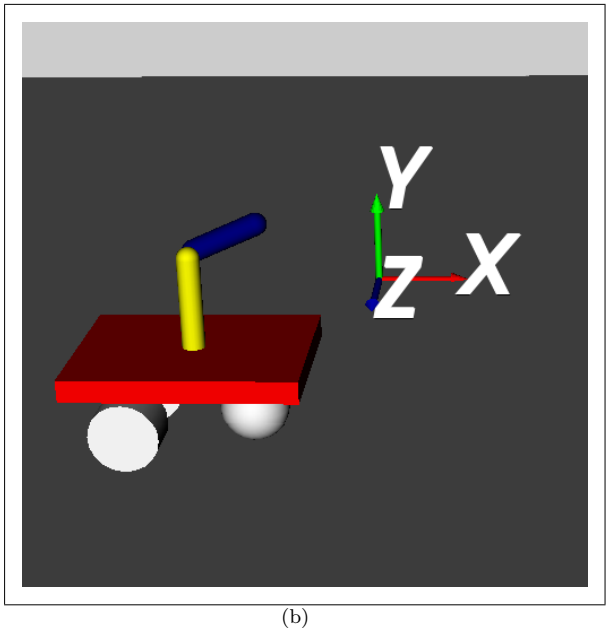
In order to setup ARS, the Python programming language must be installed first, followed by ODE, the PyODE package and VTK. Using ARS involves the following simple steps: (i) open a command window, with the current directory set to the project main directory, (ii) execute the ARS setup script by typing `python setup.py install` at the command prompt, (iii) in the Python command interpreter execute `import ars`. It will be then possible to explore the contents by entering `help(ars)`. To exit the interpreter type `quit()`. The installation process of ARS is fairly automated, however for further details see the `README` and `INSTALL` files contained in the current release available at the project's SourceForge site (ARS Project, 2011).

3. APPLICATIONS EXAMPLES

In this section we present some examples that illustrate the physical accuracy of the simulator as well as its simplicity. The code for all examples is publicly available at the (ARS Project, 2011).



(a)



(b)

Fig. 1. Basic simulations implemented using ARS: (a) simple 2-DOF arm with a waist and shoulder joint, (b) mobile manipulator with differential-drive base and 2-DOF arm.

The first example is consider a robot with two degrees of freedom (DOF), involving a waist and a shoulder joint, which is actuated as a conical pendulum to show that the numerical results match those that can be computed analytically applying the laws of physics. The second and third examples show how to setup a mobile manipulator and demonstrate that ARS can be employed to analyze forces arising from the interaction between the base and the arm, analyze disturbance forces produced by varying terrains, design and analyze controllers, as well as allow the user to supply inputs to the simulation in real-time as if driving a real robot. Snapshots of the five examples are shown in fig. 1.

Table 1. ARS architecture.

ARS subpackage	Modules in the subpackage
app	Program, ActionMap
graphics	Adapter, Axes, Body, Box, Sphere, ScreenshotRecorder
adapters	VtkAdapter, VtkBody
model	
geometry	shapes (Shape, Trimesh, Primitive, Box, Sphere)
robot	actuators (Actuator, Servo, Motor), joints (Joint, Fixed, Rotary, Universal, BallSocket), sensors
simulator	Simulation, SimulatedObject, SimulatedJoint collision (Space, near_callback) physics (Body, Box, Sphere)
utilities	generic (write_var_to_file, insert_in_tuple, write_settings) mathematical (radians_to_degrees, matrix_as_tuple, rot_matrix_to_hom_transform, calc_rotation_matrix)

Table 2. Denavit-Hartenberg parameters for the robot of example 2.

Joint	θ_i [rad]	d_i [m]	a_i [m]	α_i [rad]
1	θ_1	1	0.1	$\pi/2$
2	θ_2	0	1	0

3.1 Example 1: Actuating a 2-DOF Robot Arm as a Conical Pendulum

A simple 2-DOF robot arm with waist and shoulder joints is implemented as a benchmark case to further demonstrate the physical accuracy of the simulator. The robot is shown in fig. 2 and its geometry is expressed using the standard Denavit-Hartenberg (DH) convention with parameters summarized in table 2. The inertial parameters are presented in table 3. The robot has two rotary joints, whose positions are described by the DH parameters θ_1 (waist) and θ_2 (shoulder). The robot is actuated applying a torque to the waist motor according to

$$\tau_1 = \begin{cases} 20t, & 0 \leq t < 1 \\ 20, & 1 \leq t \end{cases} \text{ Nm},$$

while the shoulder joint is left to coast freely with no torque applied to it, i.e. $\tau_2 = 0$ Nm, $t \geq 0$. The robot actuated in this manner behaves like a *conical pendulum*, a device which was classical in clockwork timing mechanisms and key element in centrifugal governors during the 1800s.

The implementation of the simulation can be found in the sample files of the ARS project (ARS Project, 2011). To validate the accuracy of the results the same robot was implemented using the Robotics Toolbox for Matlab (Corke, 1996). Figure 3 shows the results obtained from the simulation with ARS and the ideal curve obtained using the classical recursive Newton-Euler algorithm implemented in the Robotics Toolbox. The curves in blue correspond to the angular velocity $\omega_1 = \dot{\theta}_1$ of the robot's waist joint, while the dark-green curves correspond to the angle of the robot's shoulder joint. Figure 3 shows that as torque in the waist joint is ramped to 20 Nm in one second, the waist angular velocity reaches almost 4 rad/s. The centrifugal forces acting on the shoulder link cause it to raise from $\pi/2$ rad to about 0.87 rad.

It is possible to appreciate that the curves in figure 3 obtained in ARS and with the Robotics Toolbox match well and reach the same steady-state values with a similar time-response. The small difference in the transient is due to ODE's different internal damping and constraint handling parameters, such as the constraint force mixing (CFM) and the error reduction parameter (ERP), that are needed by ODE because it employs a maximal coordinate formulation of the dynamics together with a formulation for the constraints in terms of a linear complementarity problem (LCP) instead of the reduced coordinate approach of the recursive Newton-Euler algorithm (Drumwright et al., 2010). It is to be noted that we computed the results using ODE's default internal parameters and we did not do any kind of special adjustment to find a better approximation to the expected curve. This should demonstrate that the ARS simulator and the underlying dynamics engine should be fairly reliable and should not demand the user to spend time adjusting parameters unless some extremely high level of accuracy is required by the application. Moreover, despite the small differences in the transient responses, the response obtained with ARS reasonably preserves the characteristic oscillations and response time. This should allow mechanical engineers and control designers to develop robots and controllers knowing that the model dynamics is consistent with that of the real system, and thus that commonly relevant aspects such as actuator forces, energy consumption or controllers' stabilizing properties exhibited by the simulated solutions will be preserved by the actual physical implementations. Certainly, time could be spent to adjust ODE's parameters so that its response perfectly matches that of the ideal dynamical model. However, seeking absolute accuracy may be beyond reasonable because in practice even when ideal models are available, some time must be spent on the calibration and validation of the real sub-systems, such as actuators, sensors and

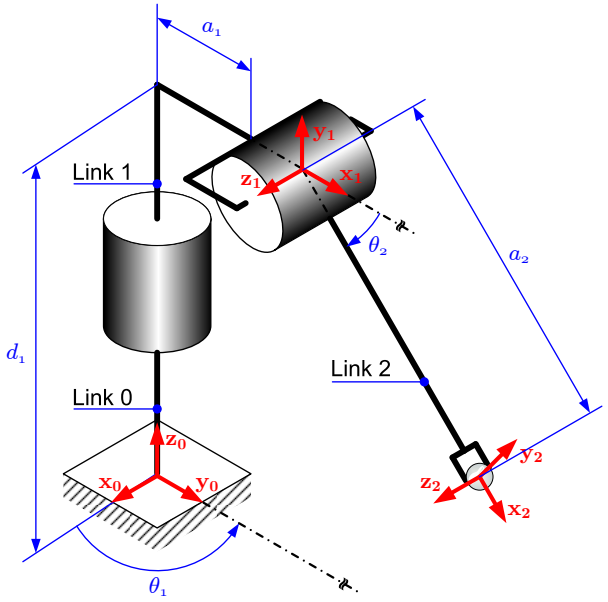


Fig. 2. 2-DOF robot arm.

Table 3. Inertial and electro-mechanical parameters for the robot of example 2.

	Link/Joint 1	Link/Joint 2
Mass m_i (kg)	10	10
Inertia tensor I_i ($\mathrm{kg\,m^2}$) with $r = \sqrt{2}/10$ m	$\begin{bmatrix} m_1(3r^2 + d_1^2)/12 & 0 & 0 \\ 0 & mr^2 & 0 \\ 0 & 0 & m_1(3r^2 + d_1^2)/12 \end{bmatrix}$	$\begin{bmatrix} mr^2 & 0 & 0 \\ 0 & m_2(3r^2 + a_2^2)/12 & 0 \\ 0 & 0 & m_2(3r^2 + a_2^2)/12 \end{bmatrix}$
Motor Inertia J_{mi} ($\mathrm{kg\,m^2}$)	10^{-4}	10^{-4}
Gear ratio	100:1	100:1
Viscous friction B_i (N s)	$50 \cdot 10^{-3}$	$50 \cdot 10^{-3}$

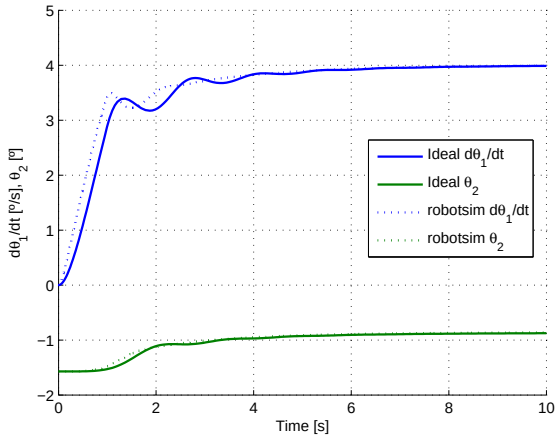


Fig. 3. Trajectory for the freely coasting shoulder as the waist speed of the 2DOF robot arm increases.

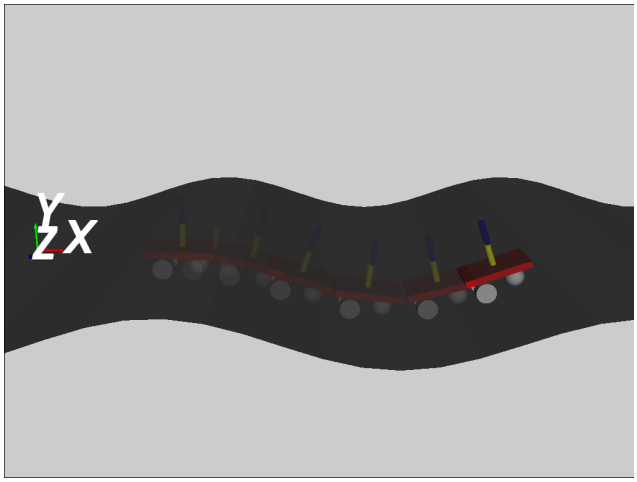


Fig. 4. Mobile manipulator on sinusoidal terrain.

controllers, due to other uncertainties and disturbances that can be costly or simply very difficult to model.

Finally, it is also to be noted that for the robot rotating as a conical pendulum, and with the simplifying assumption that the links are described by point-masses, the simple application of Euler's second law for the torque balance about the shoulder's joint axis allows to establish that in steady-state, the arm reaches a position $\theta_2^\star = \beta^\star - \pi/2$, where $\beta^\star$ is a solution of the equation:

$$\frac{\tan(\beta)}{a_1 + a_2 \sin(\beta)} = \frac{\dot{\theta}_1^2}{g},$$

with DH parameters a_1, a_2 and β defining the angular position of the shoulder joint about axis $\mathbf{z}_1$ measured starting from the negative portion of the vertical axis $\mathbf{y}_1$ till reaching axis $\mathbf{x}_2$ aligned with the robot's second link, i.e. $\beta = \theta_2 - \pi/2$. Solving the above equation allows to obtain the same steady-state results as those presented in fig. 3 further confirming the consistency of the model with respect to what is physically expected.

3.2 Example 2: Mobile Manipulator with Controlled Arm on Sinusoidal Terrain

This example is similar to the previous one, but now the robot is driven along a straight line on a sinusoidal terrain and its arm is controlled using a PD controller to keep the arm straight in the upright position regardless of the disturbance to its equilibrium produced by the sinusoidally changing slope of the terrain. Figure 4 shows snapshots as phantoms of the robot traversing the sinusoidal terrain. The numerical results obtained for this example are presented in fig. 5 and show that the arm angles deviate less than 3° as the robot's vertical position oscillates until it falls off the terrain (modeled as a trimesh of finite extension). In order to keep the arm vertical, the arm motors must apply torques that are visibly in counter-phase the joint angle error, which in this case is equal to the joint angle because the reference angles were defined to be zero when the arm is in the upright position.

3.3 Example 3: Mobile Manipulator with Controlled Arm with User Input

In this example the mobile manipulator's shoulder joint is controlled to a set-point using a PD controller, while the waist joint is let to rotate freely and the base is commanded in open-loop by the user employing keyboard inputs to adjust the wheel torques. This example's purpose is demonstrating that ARS includes functionality useful for creating interactive simulations.

Figure 6 shows the attempts by the user to drive the robot along a circular trajectory in the XZ plane using the computer keys, and because of this the trajectory is not a perfect circle. It is also to be noted that the error of the waist joint θ_1 grows almost in a linear fashion, as shown in the top graph of fig. 7. This is explained by the fact that the waist joint has been allowed to rotate free and by inertia it tends to keep its orientation as the robot turns on the plane. The PD-controlled shoulder joint θ_2 has a permanent error smaller than 1.5° as shown by the plot in the middle of fig. 7. This permanent error could be driven to zero adding integral action to the controller. Finally, the user-applied torques to the wheels are shown in the bottom graph of fig. 7.

4. CONCLUSIONS AND FUTURE WORK

This paper presented an open-source tool for robot simulation in Python built upon the integration of the Open Dynamics Engine (ODE) and the Visualization Toolkit

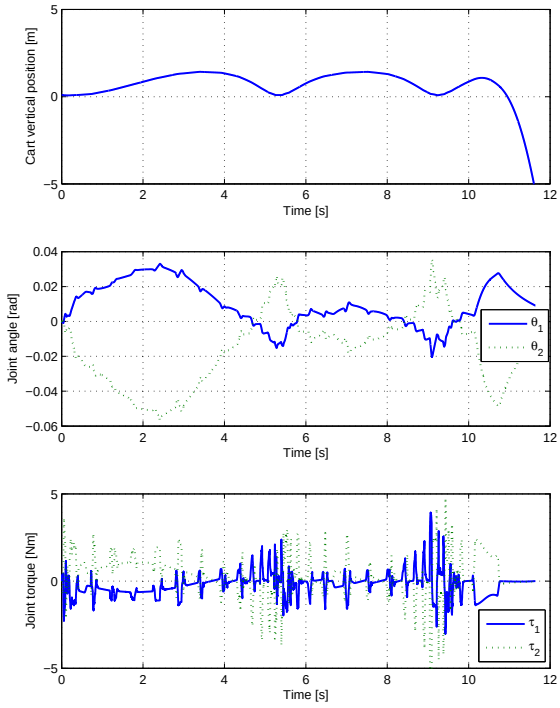


Fig. 5. Mobile robot with PD-controlled arm on sinusoidal terrain: cart vertical position (top), arm angles (middle), arm torques (bottom).

(VTK). The main contribution of this tool is that it should allow users to focus on the mechanical design of the robot rather than on the programming details, thus helping users to quickly obtain physically accurate results that can be visualized in its graphical interface. The tool provides classes of objects and functions that should be easy to learn and employ to model complex mechanical systems. Unlike other tools for robot and mechanical multi-body systems simulations, the models are implemented with code that is easier to understand because it is designed to be modular and expandable, in terms of objects that represent standard mechanisms and processes associated to the actuators or sensors that can be applied to the mechanisms. Another important aspect is that the simulations take into account collisions and motion constraints through ODE. Taking into account these aspects when implementing simulations using the traditional recursive Newton-Euler or Lagrange-Euler approach for deriving the dynamical equations is often difficult and time consuming. While ignoring collisions and motion constraints can be valid for some situations, such as for industrial manipulators in unconstrained workspaces, the current robotics challenges involving humanoid robots and mobile manipulators require simulators capable of taking into account the interaction between the robot and the environment (Wettach et al., 2010). Ongoing work involves the development of models with parameters that can be quickly adjusted by the user to represent new or commercially available actuators and sensors, e.g. dc motors, visual sensors, and laser range scanners (Saso et al., 2009). Additional research is concerned the development of novel algorithms for multibody dynamics and

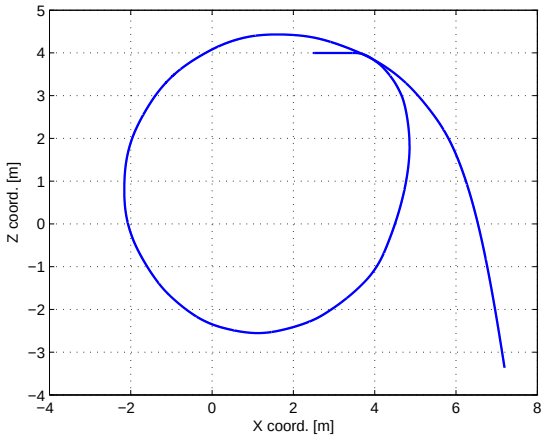


Fig. 6. Trajectory in the XZ plane for the user-driven mobile manipulator.

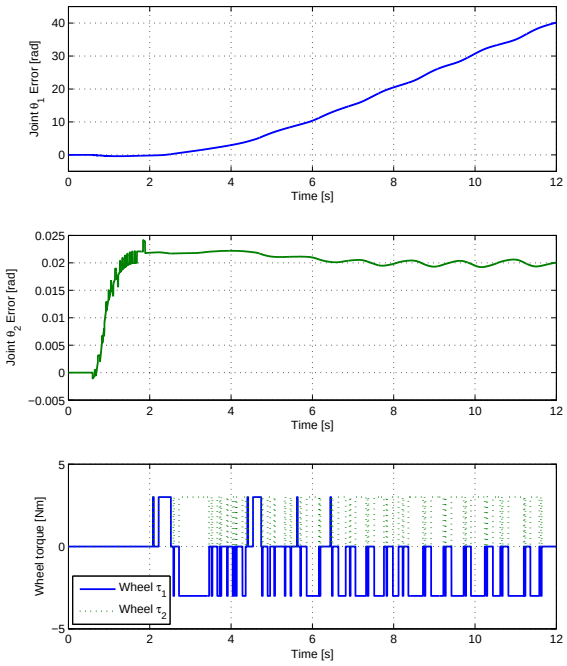


Fig. 7. Mobile robot with PD-controlled arm and user-driven base: joint θ_1 error (top), joint θ_2 error (middle), wheel torques (bottom).

collision handling in order to improve the accuracy and speed of the simulations.

A mobile manipulator model implemented with the proposed simulation tool was employed to show that the results are physically consistent with the theory. The tool should be valuable to a broad audience of engineers, researchers, and students alike. Ongoing work is concerned with incorporating sensor models, motion planners and controllers, as well as adding other components for interactive graphical editing of the models and simulations.

ACKNOWLEDGEMENTS

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

REFERENCES

ARS Project (2011). Autonomous robot simulator for python. http://sourceforge.net/projects/arsproject/.

Bonaventura, C. and Jablokow, K. (2005). A modular approach to the dynamics of complex multirobot systems. *Robotics, IEEE Transactions on*, 21(1), 26 – 37. doi: 10.1109/TRO.2004.833809.

Castillo-Pizarro, P., Arredondo-Vidal, T., and Torres-Torriti, M. (2010). Introductory survey to open-source mobile robot simulation software. In *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, 150 –155. doi:10.1109/LARS.2010.19.

Corke, P. (1996). A robotics toolbox for matlab. *IEEE Robotics and Automation Magazine*, 3(1), 24–32. doi: 10.1109/100.486658.

Diankov, R. (2010). *Automated Construction of Robotic Manipulation Programs*. Ph.D. thesis, The Robotics Institute, Carnegie Mellon University.

Drumwright, E., Hsu, J., Koenig, N., and Shell, D. (2010). Extending open dynamics engine for robotics simulation. In N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk (eds.), *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, 38–50. Springer Berlin / Heidelberg.

Johns, K. and Taylor, T. (2008). *Professional Microsoft Robotics Developers Studio*. Wiley Publishing.

Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, 2149 – 2154 vol.3. doi:10.1109/IROS.2004.1389727.

Kumar, K. and Reel, P. (2011). Analysis of contemporary robotics simulators. In *Emerging Trends in Electrical and Computer Technology (ICETECT), 2011 International Conference on*, 661 –665. doi:10.1109/ICETECT.2011.5760200.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Stirling, T., Gutierrez, A., Gambardella, L.M., and Dorigo, M. (2011). Argos: A modular, multi-engine simulator for heterogeneous swarm robotics. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, 5027 – 5034. doi:10.1109/IROS.2011.6094829.

PSF (2011a). Python. http://python.org/.

PSF (2011b). SciPy. http://www.scipy.org/.

PYPI (2011). Python Package Index. http://pypi.python.org/pypi.

Saso, K., Natasa, K., Zobel, P.B., and Durante, F. (2009). *Modeling and simulation of custom developed 3D laser range scanner*, 183–188. IASTED - Acta Press.

Schroeder, W., Avila, L., and Hoffman, W. (2000). Visualizing with vtk: a tutorial. *Computer Graphics and Applications, IEEE*, 20(5), 20 –27. doi:10.1109/38.865875.

Wettach, J., Schmidt, D., and Berns, K. (2010). Simulating vehicle kinematics with simvis3d and newton. In *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Artificial Intelligence*, 156–167. Springer-Verlag, Berlin Heidelberg.

Wikipedia (2011). Wikipedia: Simulation programming languages. http://en.wikipedia.org/wiki/Category:Simulation_programming_languages.

APPENDIX B. SMTP PAPER - A ROBOTICS SIMULATOR FOR PYTHON

A Robotics Simulator for Python

Germán Larraín-Muñoz, Miguel Torres-Torriti[a]

[a]*Dept. of Electrical Engineering, Pontificia Universidad Catolica de Chile, Av. Vicuña Mackenna 4860, Santiago, Chile*

Abstract

Modeling and simulation of robotic systems is essential for robot design and programming purposes, as well as operator training. Of the tools developed throughout the years, some have reached reasonable levels of maturity, but are specific to robot manipulators and do not include appropriate tools for modeling mobile bases, while other focus on mobile bases in 2D planar spaces, but do not consider 3D environments. Other tools may integrate already made models to simulate complex mobile manipulators or even humanoids, but have steep learning curves and require significant programming time and skills.

The situation is further complicated due to the lack of widely accepted simulation standards and languages among robot developers, whom often implement custom simulations specific to the application or other particular aspects, e.g. robot motion planning and navigation, mechanical design and evaluation, training, etc. However, the application of robots to increasingly complex tasks calls for accurate, yet simple to use, modular and standardized simulation tools capable of describing such things as the robot's interaction with the environment (e.g. effector-object, robot-ground, and sensor-world interactions).

This paper describes the design of such a tool for physically accurate robot and multi-body systems simulation. The software, implemented in Python on top of well regarded physics and visualization libraries, is open-source, modular, easy to learn and use, and can be a valuable tool in the process of robot design, in the development of control and reasoning algorithms, as well as in teaching and educational activities.

Email addresses: gelarrai@uc.cl (Germán Larraín-Muñoz), mtorrest@ing.puc.cl (Miguel Torres-Torriti)

Keywords:
simulation of robot dynamics, multibody physics, robot simulation, software tools, public domain software, Python

1. Introduction

Since the early days of robotics, simulation tools have played an essential role in the development of newer and better systems because simulation provides an excellent way for testing and experimenting, while avoiding the costs of building unfeasible designs, the consequences of accidents and the difficulty of controlling disturbances present in the physical world [6, 11, 19, 39]. For example, current industrial robots rely on off-line simulators to validate and evaluate the work cycle of entire production lines [20, 40]. The design and feasibility testing of sophisticated robotic systems, ranging from underwater mobile manipulators [26] to sophisticated humanoids [34] and robot swarms [30] would have not been possible without adequate simulation tools. In addition to the research and engineering of robotic systems, simulation has been key to the development of virtual environments for operator training, as well as in teaching the fundamental concepts of robotic systems [6, 25].

Future autonomous robots will require better self-simulation capabilities to become more self-aware and capable of taking decisions in real-time while operating in complex and dynamic environments of the real world [7, 13, 19]. However, the existing tools (see recent reviews in [5, 21]) are often conceived for specific sub-problems, e.g. motion control, motion planning, navigation, and typically consider simplified dynamical models that do not take into account other aspects, such as robot-ground interactions, collisions, energy consumption or advanced sensor models, to name a few. Extending existing tools that were not designed with that in mind is difficult and sometimes even impossible, because their software architecture is not modular enough or forbids modifications. On the other hand, although some of the existing simulators produce sophisticated and high resolution visualization of the robots' motion, the velocity and force profiles are not accurate due to the underlying simplified motion models [5].

The aforementioned reasons have motivated us to develop an open-source tool for modeling and simulating multi-robot systems. This tool, which we have named *ARS* (for autonomous robot simulator), should help to reduce the overall costs of designing, testing and programming complex robot systems, such as mobile manipulators or humanoids. The main contributions

of ARS are that it is simple to use, yet general enough to allow modeling any type of rigid multi-body system, e.g. open-loop kinematic manipulators, closed-loop kinematic chains, mobile manipulators, or humanoid robots. Some preliminary results that included basic actuators and no sensors functionality were presented in [23].

We need an open-source robotics simulator for the Python community. It is absolutely necessary for it to be written in Python (although it is alright if it uses C/C++ extensions, which is common in scientific sofware [14]), be distributed under an OSI-approved license [29] and have 3D physics (dynamics and collision). Soft requirements are that it should be extendable, have an open development process (public and forkable code repository, public bug tracker, public documentation, mailing list), and have 3D visualization.

This paper is organized as follows: section 2 mentions the relevant existing software and compares them with ARS. Section 3 discusses the design and implementation details of the proposed simulation tool, followed by section 4, which shows applications examples and how easy it is to create and run a robot simulation. The conclusions and discussion on future enhancements to ARS are presented in section 5.

2. Existing robotics simulators for Python

When considering the publicly available software for robotics simulation using Python, it is very important to exclude those that seem to be written in this languague but are actually implemented in C/C++. In many cases they provide wrappers (i.e. bindings) for Python and other languages, which are usually generated automatically with tools like SWIG (Simplified Wrapper and Interface Generator) [36]. A special case is *breve* [17, 18], which allows for simulations to be written in Python (or in a software-specific language named *steve*), but it is implemented in C++ thus it is not possible to modify or extend it in a straight-forward way using Python, its plugin architecture also requires C/C++, and unfortunately it is no longer under active delopment (its latest release was in February 2008).

To the best of our knowledge, the only existing software that complies with the minimum requirements aforementioned is *MORSE* (Modular OpenRobots Simulation Engine) [9, 10], presented in 2011, about the same time ARS's development began. It is a "generic simulator for academic robotics" [22] that has had good reception by the scientific community [10, 24], showing there is a need for tools like this. ARS and MORSE share some goals and

features, but there is a clear divergence in their focus. While ARS's objective is to provide robotics simulations with physically-accurate results, MORSE outsources the physics and collision processing to Bullet (physics library meant for "Real-Time Physics Simulation" [1]) through Blender's (very sophisticated open-source graphics software [4]) Game Engine. In the field of games physical accuracy is not the primary objective: the concern is to display simulations that look visually realistic and run fast and fluid enough, i.e. real-time with large frame rates. For ARS, execution speed is a secondary goal, and visualization should be just a precise graphic representation of what is actually happening in the simulation.

MORSE is to be "used in different contexts for the testing and verification of robotics systems as a whole, at a medium to high level of abstraction" [9]. For example, the robots section of the components library includes submarine, helicopter, quadrotor, Hummer, among others, all of them with greatly simplified dynamics and collision –or even without considering them at all–, but are nonetheless useful for some kind of simulations.

Being tightly coupled to Blender in many different areas restricts in a great deal how MORSE can be customized in sensible aspects such as physics and collision (some libraries provide only one or the other, unlike ODE and Bullet). For example, customizing the use of Bullet (e.g. adjusting parameters, solvers, etc.) is not currently possible, while replacing it would require redesigning the software from scratch.

Also related to the mentioned coupling, it is not possible to import MORSE simulation code from the Python interpreter or plain modules (as it is done with normal Python software) to do things such as run simulations, customize entities, manage simulations or process data. This constrains the ability of users and developers to take advantage of the simulator.

Related specifically to Python, there are some issues with MORSE. The most notable is that it does not run on Python 2.x even though this is far more popular than Python 3.x at this moment, and the default version in most of the latest releases of operating systems where it comes pre-installed such as Mac OS X, Ubuntu, Debian and Fedora. With regard to packaging and distribution, it has a non-standard installation process (can not use `easy_install` [31], `pip` [31] nor last resort `python setup.py install` [33]), requires non-standard compilation tools (Kitware's CMake [15]) and its installed packages are not in the Python system path without manual modifications. It is not available at PyPI (Python Package Index, the official repository of third-party Python packages [32]).

3. Simulator Design

In this section we describe what we need from the external libraries that support key aspects of the software, which alternatives were considered and the reasons that lead us to select ODE and VTK. Then we describe the design and implementation of ARS, particularly how the code is organized and what is the concern of each one of the most relevant packages and modules.

3.1. Enabling Technologies

The simulator needs a library for physics and collision (it could be different libraries but it is better a single one that provides both functionalities), and another for visualization. Both need to have 3D support, be cross-platform and have an open source license.

For physics/collision, the alternatives were Bullet [1], Newton and Open Dynamics Engine (ODE) [27], of which we chose ODE because it is a popular option for robotics software [8, 12], has relatively high accuracy [5], its code is mature (first released in 2001 [28]) and has official, well developed Python bindings. However, as most physics engines, it was designed to be fast enough to be used by games, which require real-time simulation. This performance was achieved with many simplifications, at the expense of accuracy, some of which affect the simulation of common robotics scenarios. Nonetheless, Drumwright et al. recently identified the relevant shortcomings and suggested solutions in [8].

With respect to visualization we considered OGRE, OpenSceneGraph, Panda3D, Visualization Library and Visualization Toolkit (VTK) [38]. We chose VTK, an "open-source software system for 3D computer graphics, image processing and visualization" [38], because it is a very powerful library, has a mature codebase (presented in 1996 [35]), it is oriented to scientific software (created by GE Corporate R&D researchers [37]), includes official Python bindings, supports a broad collection of graphics cards and is developed by a company with many popular open source products, Kitware [16]. On the other hand, using VTK seems a little excessive because the features we need are a small fraction of those provided. By being such a big library it has sizable impact on computer resources and its distribution files are somewhat large.

3.2. Implementation and Architecture

The code is organized hierarchically in five root-level packages: `app`, `graphics`, `lib`, `model` and `utils` (see fig. 1 (a)), and some of those con-

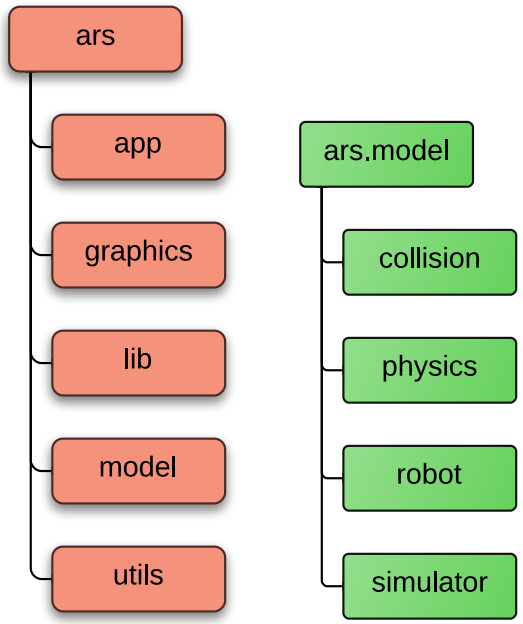


Figure 1: Packages hierarchy: (a) root packages, (b) model packages.

tain more packages or modules as well. The core one is `model`, which in turn includes `collision`, `physics`, `robot` and `simulator` (fig. 1 (b)).

`model.collision` has two main modules: `base` and `ode_adapter`. The former defines the basic functionality for collision, as well as the base classes that compose an abstract interface to the library developers choose to use, e.g. `Space` and `Geom` –*geom* is a geometry object– (parent class of `Ray`, `Trimesh` –*trimesh* is a triangular mesh–, `Box`, `Sphere`, `Plane`, etc), which wrap the corresponding "native" object that the adapted library uses, assigned to a private attribute. On the other hand, `collision.ode_adapter` has the classes and functions to connect with the collision library included in ODE that implement the interface defined in `base`. The class hierarchy of `Geom` is depicted in fig. 2.

Subpackage `physics`, as `collision`, also has modules `base` and `ode_adapter`. Instead of `Space` and `Geom` there is `World` and `Body` (parent class of `Box`, `Sphere`, `Capsule`, `Cylinder` and `Cone`). The subclasses of `Body` can be appreciated in fig. 3.

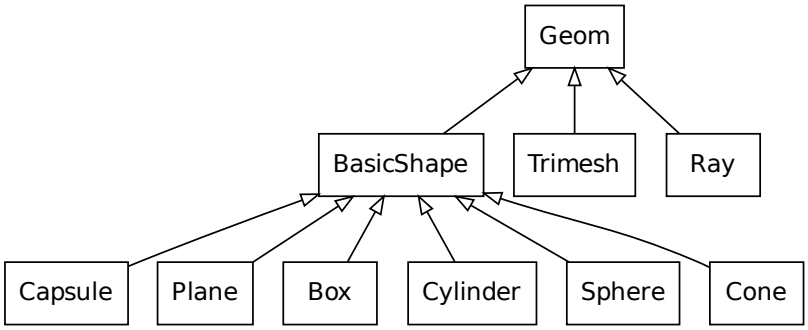


Figure 2: Subclasses of `collision.base.Geom`.

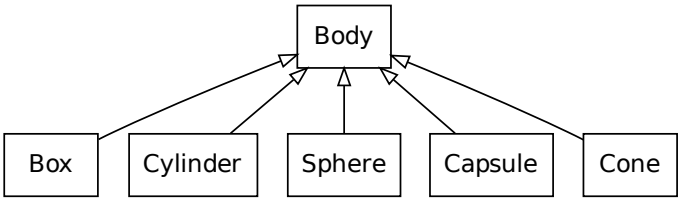


Figure 3: Subclasses of `physics.base.Body`.

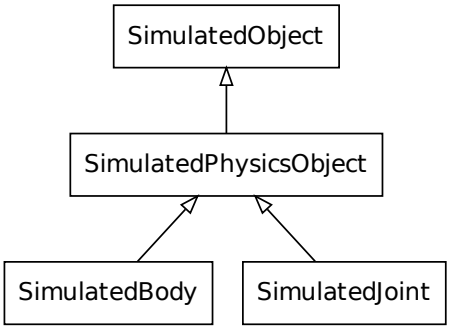


Figure 4: Subclasses of `simulator.SimulatedObject`.

Subpackage `simulator` contains all the classes directly related to a simulation like the "environment" (`Simulation`) and the objects to simulate (fig. 4).

In the module `robot.joints` there are both actuated and freely-moving joints. Because they share many properties, they are abstracted with a common parent class named `Joint` (fig. 5). In the latest release of ARS the only actuators included are rotary (used for hinges and wheels) and slider, both with one degree of freedom. s Module `robot.sensors` has two top-level classes: `BaseSourceSensor` (fig. 6 (a) and 6 (b)) and `BaseSignalSensor` (fig. 6 (c)). The former has an associated object that is the source of information. The program must call the `on_change` method every time it wants to save a measurement. On the other hand, sensors based on signals work by subscribing to them upon creation (and optionally specifying a given "sender/emitter"), so each time they are fired a measurement will be recorded.

The package `app` handles the program flow through its class `Program`. This is the main class to understand in ARS, since simulation programs are defined by subclassing it. Each simulation run is an instance of a derived class. It wraps the corresponding `model.simulator.Simulation` instance and the associated visualization window. It also provides the methods to allocate and release the resources necessary to run the simulations. The class `ActionMap` is a dictionary-like structure used to map keys to pre-defined

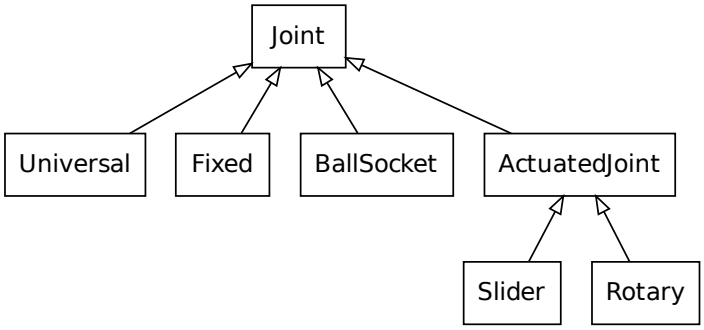


Figure 5: Subclasses of `robot.joints.Joint`.

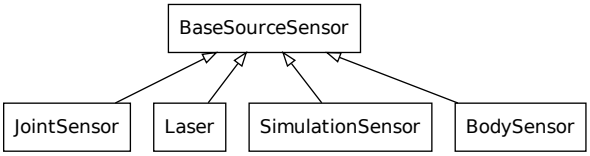
actions (e.g. apply torque to a wheel when the user presses a certain key) by associating a callable object (e.g. function, method) to each key.

Package `graphics` contains modules `base` and `vtk_adapter`. The former defines the interface to implement by the latter, composed by classes `Engine`, `Entity` and its subclasses (fig. 7), and `ScreenshotRecorder`.

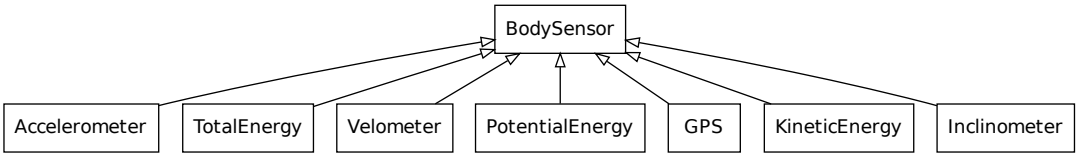
4. Applications Examples

In this section we present some examples that illustrate the physical accuracy of the simulator, its simplicity and some features. Due to space limitations only the code for the first examples is included but all the source code is stored in the project's repository [3], and there are demonstration videos available [2] too.

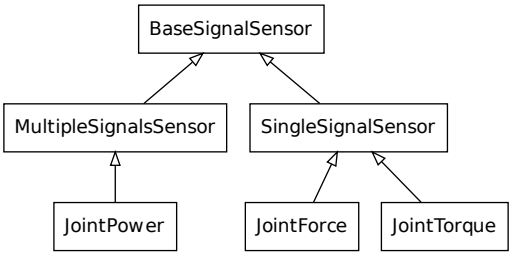
The first example is a "Hello World" type of program that shows the smallest piece of code necessary to define a simulation and run it. Next there is a group of simulations that show, in detail, how to use different sensors to record data along the execution (the results are plotted individually in fig. 9). Then we make a mobile manipulator follow a trapezoidal velocity profile with a proportional-derivative controller, analyzing the torque applied to the wheels as the time-dependent set point changes. Finally, we mention other features available in ARS as well as possible applications. Some snapshots of the examples are shown in fig. 8.



(a)



(b)



(c)

Figure 6: In package `robot.sensors`, subclasses of: (a) `BaseSourceSensor` (one level deep), (b) `BodySensor`, (c) `BaseSignalSensor`.

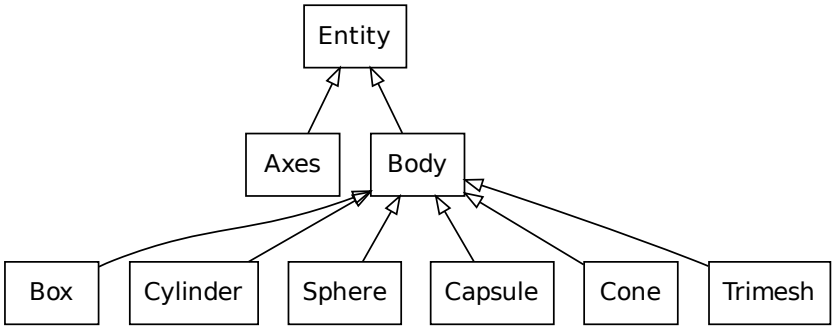


Figure 7: Subclasses of `graphics.base.Entity`.

The parameters values for all the examples are ARS's defaults: time step of 0.4 ms and gravity acceleration of 9.81 m/s. ODE-specific parameters are defined in the corresponding `ode_adapter` module in the physics and collision packages. They are all adimensional quantities: global *ERP* (Error Reduction Parameter) of 0.2, global *CFM* (Constraint Force Mixing) of 10^{-10}, contact joint bounce of 0.2, and contact joint mu of 500.

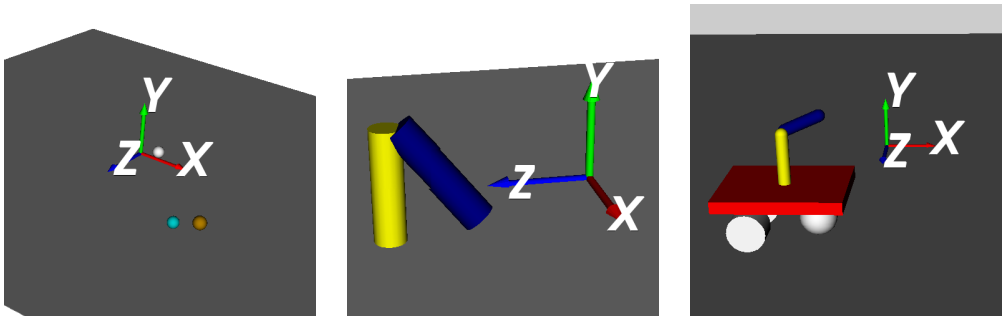
4.1. Hello World

The following code shows what is necessary to define a simple simulation program. In this case, a sphere of radius 0.5 m and 1.0 kg is created at position $(1, 1, 1)$ meters.

```
from ars.app import Program

class FallingBall(Program):

    def create_sim_objects(self):
        self.sim.add_sphere(0.5, (1, 1, 1), mass=1)
```

The procedure is very straight-forward:

1. From package `ars.app` import class `Program`.
2. Define a subclass of `Program` (named `FallingBall` in this case).
3. Implement in it the required method `create_sim_objects`.



(a) (b) (c)

Figure 8: Basic simulations implemented using ARS: (a) falling balls, (b) simple 2-DOF arm with a waist and shoulder joint, (c) mobile manipulator with differential-drive base and 2-DOF arm.

4. To create objects and add them to the environment (a sphere in this case) add calls to methods of class `Simulation` (of which attribute `sim` is an instance).

With these steps, a custom simulation has been defined. To run it, just two lines of code are necessary: one to create an instance of `FallingBall` and another to begin the simulation. It will run until the just opened visualization window is closed by the user.

```
sim_program = FallingBall()
sim_program.start()
```

The simplicity of the procedure (just five lines of code plus those regarding the bodies to be created) necessary to define and execute a 3D multibody physics simulation saves the user both time and effort, and makes the process less error prone.

4.2. Sensors

Body, joint and environment sensors are available to record simulation data as well as to trigger user-defined behavior. The examples in this section show how to use some specific sensors but the general procedure is the same for those not covered here. Because some basic code is common to all the examples, we refactor it into a class named `SensorExampleBase`, which is also useful to show how ARS's design allows and encourages code reuse.

The following is the standard way to initialize a sensor, subscribe it to signals (i.e. events) and define what should the program do each time those

are fired. After the simulation has ended, the collected data is still available, perhaps to be printed or saved for offline processing.

```
from ars.app import Program, dispatcher
from ars.model.simulator import signals

class SensorExampleBase(Program):

    def __init__(self):
        Program.__init__(self)
        dispatcher.connect(self.on_post_step, signals.SIM_POST_STEP)

    def create_sim_objects(self):
        obj_name = self.sim.add_sphere(0.5, (1, 1, 1), mass=1)
        self.sphere = self.sim.get_object(obj_name)
        self.create_sensor()

    def create_sensor(self):
        # self.sensor = ...
        raise NotImplementedError("Implement this method in subclasses")

    def on_post_step(self):
        time = self.sim.sim_time
        self.sensor.on_change(time)

    def print_sensor_data(self):
        collected_data = self.sensor.data_queue
        print(collected_data)
```

The differences with *Hello World* are:

- override constructor to subscribe a method (`on_post_step`) to a signal (`SIM_POST_STEP`)
- in the creation of simulation objects, store in attribute `sphere` a reference to the object just added to the simulation, and call method `create_sensor`
- new method `create_sensor`, which must be implemented by subclasses, to instante a sensor and store it in new attribute `sensor`
- new method `on_post_step` to tell the sensor to record a new data instance

- new method `print_sensor_data` to print the collected data to console (changing it to write to a text file is trivial)

To run any of the following examples, it is the same as in *Hello World*:

```
sim_program = MySensorExample()
sim_program.start()
```

where `MySensorExample` is a subclass of `SensorExampleBase`. To print the data collected by the sensor, it is just an additional call:

```
sim_program.print_sensor_data()
```

With the `GPS` sensor attached to a body it is possible to record its position in every simulation step. In the corresponding plot (fig. 9 (a)) it can be seen how the ball begins at a height of 1 m and then falls until it reaches the floor and bounces a few times. The lowest position is 0.5 m because that is the object's radius.

```
class GPSExample(SensorExampleBase):

    def create_sensor(self):
        self.sensor = GPS(body=self.sphere)
```

Analogously, to record body's velocity, acceleration, or energy (potential or kinetic), the last sentence should be replaced, respectively, by one of the following:

```
self.sensor = Velometer(body=self.sphere)

self.sensor = Accelerometer(body=self.sphere, self.sim.time_step)

self.sensor = PotentialEnergy(body=self.sphere, self.sim.gravity)

self.sensor = KineticEnergy(body=self.sphere)
```

while for system's total energy it should be:

```
self.sensor = SystemTotalEnergy(self.sim, disaggregate=True)
```

Note that some sensors require different arguments: for the accelerometer, the simulation time step; for the potential energy, the gravity vector; and for the system's total energy, the simulation object as well as whether the potential and kinetic components should be stored separately or combined.

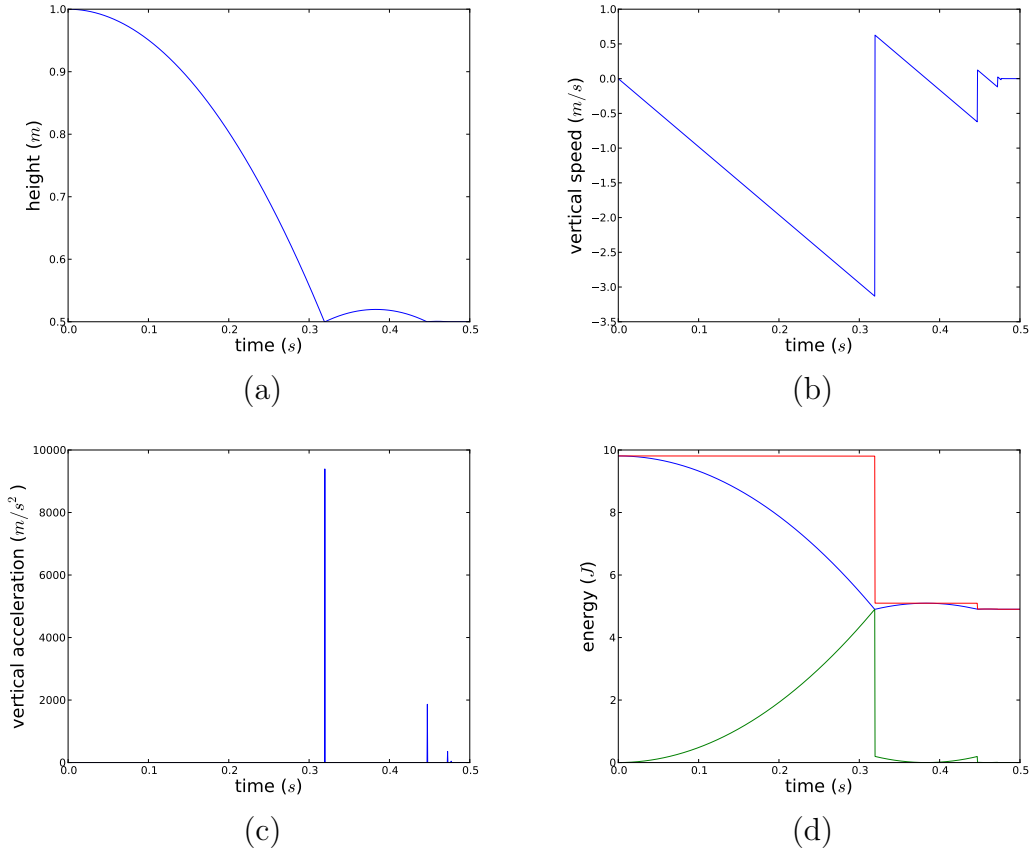


Figure 9: Data collected using sensors in the single falling ball example: (a) body vertical position, (b) body vertical speed, (c) body vertical acceleration, (d) system energy (from top to bottom: total, potential, kinetic).

In the plot of fig. 9 (b) it can be appreciated that the ball hits the floor significatively three times: 0.3194 seconds (at a speed of 3.13135 m/s), 0.4470 seconds (at a speed of 0.62156 m/s) and 0.4722 seconds (at a speed of 0.11898 m/s). The coefficient of restitution is 0.200000 every time, which is coherent with ODE's contact joint bounce parameter value of 0.2, whereas the acceleration is 9394 m/s^2, 1865 m/s^2 and 357 m/s^2 respectively (fig. 9 (c)).

In the given example there is only one body thus the system energy sensor data (fig. 9 (d)) corresponds to the kinetic and potential energy of the single sphere. A more interesting plot is created by using two spheres. For that we define a different parent class for the simulations (based in

`SensorExampleBase`) that creates two spheres, both with the same properties (radius of 1.0 m and mass of 1.0 kg) but at different heights and separated enough so they do not collide. The result is in fig. 10 (a).

```
class SensorExampleBase2(SensorExampleBase):

    def create_sim_objects(self):
        self.sim.add_sphere(1.0, (3, 4, 1), mass=1)
        self.sim.add_sphere(1.0, (5, 3, 1), mass=1)
        self.create_sensor()
```

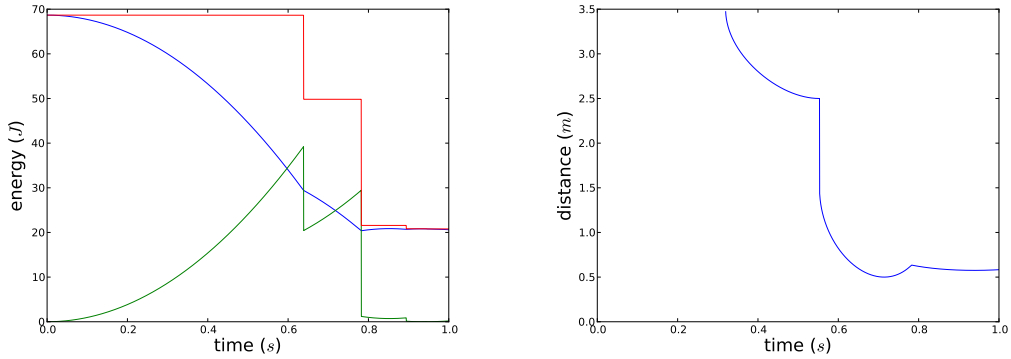
This new simulation allows us to demonstrate how to use the `Laser` sensor. The following code sets it in space at position $(1.5, 1.5, 1)$ meters and orientation $(1, 0, 0)$, the X axis (the default is $(0, 0, 1)$, the Z axis, but with a helper function it is rotated 90° around $(0, 1, 0)$, the Y axis). The sensor will measure the distance to the nearest object intersecting its projection line, up to a maximum distance.

```
class LaserExample(SensorExampleBase2):

    def create_sensor(self):
        space = self.sim.collision_space
        rot_matrix = calc_rotation_matrix((0, 1, 0), pi / 2)

        self.sensor = Laser(space, max_distance=1000.0)
        self.sensor.set_position((1.5, 1.5, 1))
        self.sensor.set_rotation(rot_matrix)
```

The plot of fig. 10 (b) shows that the intersection begins (and corresponds to the maximum distance, 3.5 m) 0.32 seconds into the simulation, when the bottom of the furthest sphere is at $(5, 1.5, 1)$ m. When the system comes to rest, the distance measured by the sensor is 0.63397 m, approximately equal to $3.0 - 1.5 - \cos(\arcsin(0.5)) = 1.5 - \cos 30°$, the theoretical value obtained by simple geometry.

4.3. Mobile Manipulator with Freely-Coasting 2-DOF Arm

This example considers a mobile manipulator composed of a 2-wheeled differential drive mobile base and robot arm mounted on top of the base. The arm has only two degrees of freedom (waist and shoulder) because such a design allows to easily visualize and understand the influence of the motion of the robot's base on its arm joints. More complex arms can be easily implemented, but the higher complexity due to the multiple interacting robot links makes it more difficult to understand how motion propagates across the arm and its joints. The mobile manipulator is shown in fig. 8 (c).



(a) (b)

Figure 10: Data collected using sensors in the two falling balls example: (a) system energy (from top to bottom: total, potential, kinetic), (b) laser-measured distance.

Torque is applied to the wheels of the robot's base using a proportional-derivative (PD) controller so that the base moves in a straight line following a trapezoidal velocity profile. Figure 11 shows the reference velocity profile (red curve: Velocity set-point (SP)), the actual velocity (green curve: Velocity controlled variable (CV)) and the longitudinal position of the robot. The torque applied to the wheels (manipulated variable) is shown in the middle graph of fig. 11. The bottom graph of fig. 11 shows the arm's joint trajectories. It is possible to see that as the robot accelerates, the shoulder link falls from the unstable equilibrium (vertical position) and swings, thus driving the waist joint to oscillate. The shoulder joint reaches its natural stable equilibrium point (arm down), however the waist joint stabilizes to a non-zero position thanks to the joints friction, but does not return to its original location as the waist joint motor is turned-off. It is interesting to note that the wheels must apply non-constant torques in order to comply with the requested longitudinal velocity profile. The reason the torque is not constant despite the trapezoidal velocity profile would require a constant acceleration/deceleration is that the swinging arm is propagating back disturbances to the base.

4.4. Complex simulations

With ARS it is possible to create complex simulations and interact and modify them in different ways. It allows to, among other things, have multiple robots in the same simulation; control a robot in real-time with a keyboard,

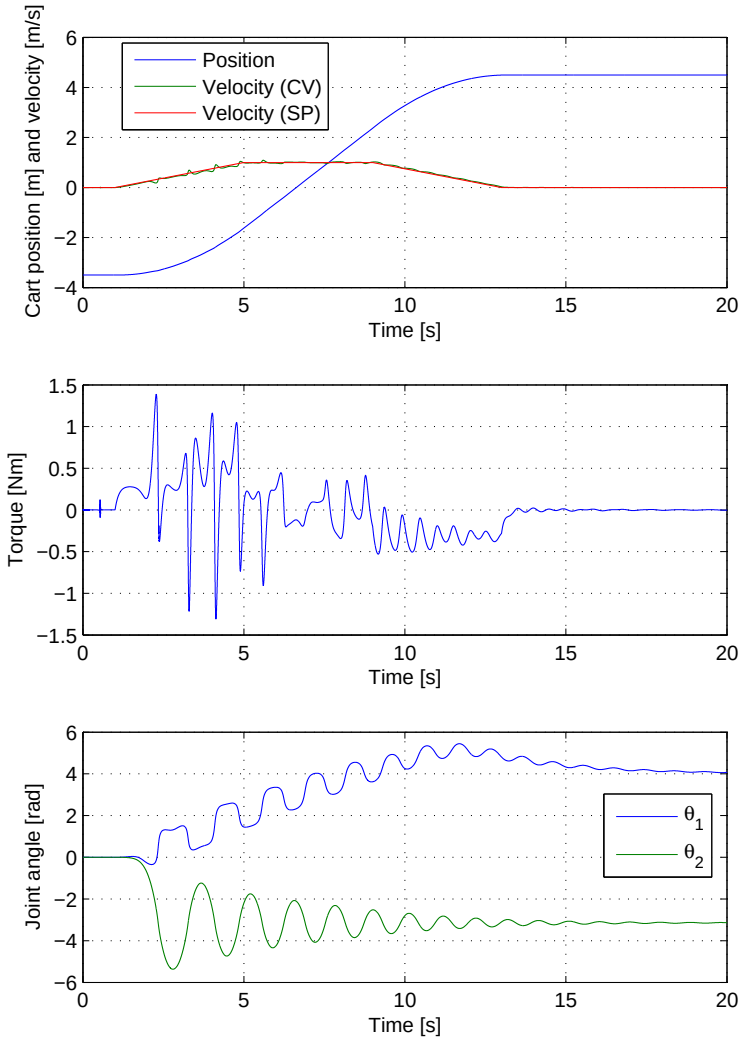


Figure 11: Mobile manipulator example: base position, velocity and velocity profile (top), total wheels motor torque (middle), arm joint angles (bottom).

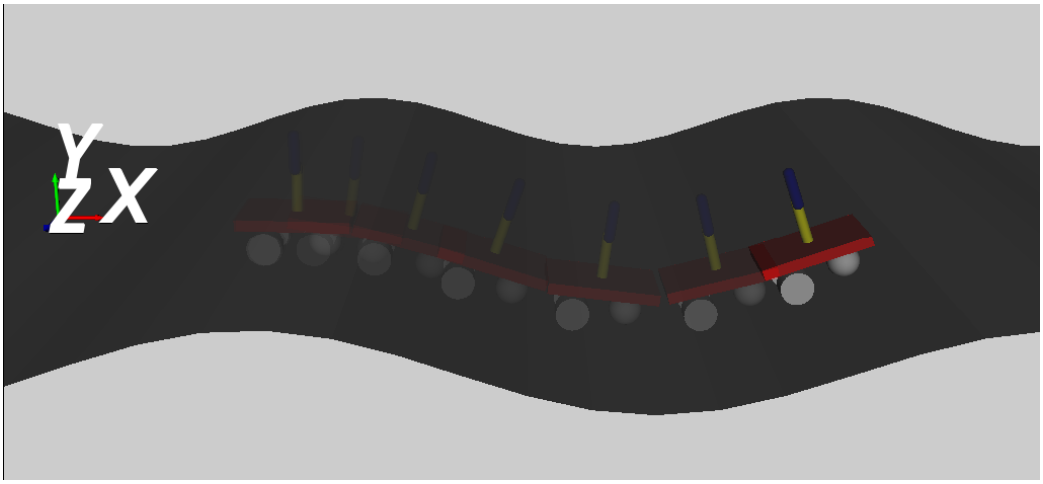


Figure 12: Mobile manipulator with a PD-controlled arm driven along a straight line on a sinusoidal terrain (modeled as trimesh).

as if driving a real one; change the environment e.g. set a trimesh as ground instead of a plane (fig. 12); alter the visualization e.g. move camera, change colors, add objects; add new objects to the simulation (or remove some) at runtime; and have multiple, independent instances of the same simulation program. Some of these aspects can be appreciated in the posted videos [2].

5. Conclusions and Future Work

This paper presented ARS, a free and open-source robotics simulator for research and education, aimed at providing a tool that the Python scientific software ecosystem lacked. It focuses in physical accuracy and simplicity, enabling users to realistically simulate the dynamics of multibody mechanical systems that include linear and rotary actuated joints, and implement proprioceptive sensors (e.g. encoders, IMUs, system energy) and exteroceptive sensors (e.g. laser range scanners, GPS). Thanks to these features, ARS can be used to simulate custom environments with multiple robots or evaluate new robot designs, as well as for controller synthesis for complex robots such as mobile manipulators. With examples we show how simple it is to define a simulation, and how the user can reuse and extend those definitions by subclassing them.

Ongoing work includes creating models for real-life robots and hardware as well as to build a GUI to further simplify the software usage, with interac-

tive simulation creation, editing and save; multi-view visualization window; and logged and real-time data plotting. Also, to take advantage of the software architecture, new adapters for the physics, collision and visualization abstraction layers are being developed.

The project goals are to increase its popularity (as of December 2013 it has been downloaded over 20,000 times) to make it the standard open source Python robotics simulator –like the projects in SciPy [14] are in their own fields– and expand its developer community, incorporating external code contributions.

Acknowledgments

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

References

[1] Advanced Micro Devices, November 2013. Bullet collision detection and physics library. `http://bulletphysics.org`.

[2] ARS, November 2013. Project videos at Vimeo. `https://vimeo.com/arsproject`.

[3] ARS, November 2013. Source code repository. `https://bitbucket.org/glarrain/ars`.

[4] Blender Foundation, November 2013. Blender home page. `http://www.blender.org`.

[5] Castillo-Pizarro, P., Arredondo, T., Torres-Torriti, M., 2010. Introductory survey to open-source mobile robot simulation software. In: Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American. pp. 150–155.

[6] Chiang, L. E., 2010. Teaching robotics with a reconfigurable 3d multibody dynamics simulator. Computer Applications in Engineering Education 18 (1), 108–116.

[7] Diankov, R., 8 2010. Automated construction of robotic manipulation programs. Ph.D. thesis, The Robotics Institute, Carnegie Mellon University.

[8] Drumwright, E., Hsu, J., Koenig, N., Shell, D., 2010. Extending open dynamics engine for robotics simulation. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (Eds.), Simulation, Modeling, and Programming for Autonomous Robots. Vol. 6472 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 38–50.

[9] Echeverria, G., Lassabe, N., Degroote, A., Lemaignan, S., 2011. Modular open robots simulation engine: MORSE. In: Robotics and Automation (ICRA), 2011 IEEE International Conference on. pp. 46–51.

[10] Echeverria, G., Lemaignan, S., Degroote, A., Lacroix, S., Karg, M., Koch, P., Lesire, C., Stinckwich, S., 2012. Simulating complex robotic scenarios with MORSE. In: Noda, I., Ando, N., Brugali, D., Kuffner, J. (Eds.), Simulation, Modeling, and Programming for Autonomous Robots. Vol. 7628 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 197–208.

[11] Evans-Pughe, C., dec. 2006. Design - getting it together - it's slow and expensive to build physical prototypes. Engineering Technology 1 (9), 38 –41.

[12] Harris, A., Conrad, J., 2011. Survey of popular robotics simulators, frameworks, and toolkits. In: Southeastcon, 2011 Proceedings of IEEE. pp. 243–249.

[13] Johns, K., Taylor, T., 2008. Professional Microsoft Robotics Developers Studio. Wiley Publishing.

[14] Jones, E., Oliphant, T., Peterson, P., et al., 2001–. SciPy: Open source scientific tools for Python. `http://www.scipy.org`.

[15] Kitware, November 2013. CMake home page. `http://www.cmake.org`.

[16] Kitware, June 2013. Open source / VTK. `http://www.kitware.com/opensource/vtk.html`.

[17] Klein, J., 2003. breve: a 3d environment for the simulation of decentralized systems and artificial life. In: Proceedings of the eighth international conference on Artificial life. ICAL 2003. MIT Press, Cambridge, MA, USA, pp. 329–334.

[18] Klein, J., November 2013. The breve simulation environment. http://www.spiderland.org.

[19] Koenig, N., Howard, A., sept.-2 oct. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. Vol. 3. pp. 2149 – 2154 vol.3.

[20] Kovačić, Z., Bogdan, S., Petrinec, K., Reichenbach, T., Punčec, M., June 2001. Leonardo – the off-line programming tool for robotized plants. In: Proceedings of the 9^{th} Mediterranean Conference on Control and Automation, June 27-29. Dubrovnik, Croatia, p. .

[21] Kumar, K., Reel, P., march 2011. Analysis of contemporary robotics simulators. In: Emerging Trends in Electrical and Computer Technology (ICETECT), 2011 International Conference on. pp. 661 –665.

[22] LAAS/CNRS, October 2013. OpenRobots - MORSE. http://www.openrobots.org/wiki/morse.

[23] Larraín-Muñoz, G., Torres-Torriti, M., October 2012. A Python package for robot simulation. In: XV Latinamerican Control Conference (CLCA), October 23-26. Lima, Peru, pp. 1–6.

[24] Lemaignan, S., Echeverria, G., Karg, M., Mainprice, J., Kirsch, A., Alami, R., 2012. Human-robot interaction in the MORSE simulator. In: Human-Robot Interaction (HRI), 2012 7th ACM/IEEE International Conference on. pp. 181–182.

[25] Marhefka, D., Orin, D., jun 1996. Xanimate: an educational tool for robot graphical simulation. Robotics Automation Magazine, IEEE 3 (2), 6 –14.

[26] McMillan, S., Orin, D., McGhee, R., may 1995. Object-oriented design of a dynamic simulation for underwater robotic vehicles. In: Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on. Vol. 2. pp. 1886 –1893 vol.2.

[27] Open Dynamics Engine, May 2013. Home. http://www.ode.org.

[28] Open Dynamics Engine, May 2013. Project at SourceForge. http://sourceforge.net/projects/opende/.

[29] Open Source Initiative, November 2013. Open source licenses. http://opensource.org/licenses.

[30] Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Stirling, T., Gutierrez, A., Gambardella, L. M., Dorigo, M., sept. 2011. ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics. In: Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on. pp. 5027 –5034.

[31] PyPA, November 2013. setuptools documentation - easy_install. http://pythonhosted.org/setuptools/easy_install.html.

[32] Python Software Foundation, December 2011. Python Package Index (PyPI). https://pypi.python.org/pypi.

[33] Python Software Foundation, November 2013. Python documentation - installing python modules / standard build and install. http://docs.python.org/2/install/\#standard-build-and-install.

[34] Reichenbach, T., 2009. A dynamic simulator for humanoid robots. Artificial Life and Robotics 13, 561–565.

[35] Schroeder, W. J., Martin, K. M., Lorensen, W. E., 1996. The design and implementation of an object-oriented toolkit for 3d graphics and visualization.

[36] SWIG, November 2013. Home. http://www.swig.org.

[37] VTK, June 2013. About. http://www.vtk.org/VTK/project/about.html.

[38] VTK, June 2013. Home. http://www.vtk.org/.

[39] Wettach, J., Schmidt, D., Berns, K., November 2010. Simulating vehicle kinematics with SimVis3D and Newton. In: Simulation, Modeling, and Programming for Autonomous Robots. Vol. 6472 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin Heidelberg, pp. 156–167.

[40] Zlajpah, L., 2008. Simulation in robotics. Mathematics and Computers in Simulation 79 (4), 879 – 897.