

Locosim: an Open-Source Cross-Platform Robotics Framework

Michele Focchi^{1,2,*}, Francesco Roscia^{1,3,*}, and Claudio Semini¹

¹ Dynamic Legged Systems (DLS), Istituto Italiano di Tecnologia (IIT), Genoa, Italy

² Department of Information Engineering and Computer Science (DISI), University of Trento, Trento, Italy

³ Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS), University of Genoa, Genova, Italy
{michele.focchi}, {francesco.roschia}, {claudio.semini}@iit.it

* The authors equally contributed to this paper

Abstract The architecture of a robotics software framework tremendously influences the effort and time it takes for end users to test new concepts in a simulation environment and to control real hardware. Many years of activity in the field allowed us to sort out crucial requirements for a framework tailored for robotics: modularity and extensibility, source code reusability, feature richness, and user-friendliness. We implemented these requirements and collected best practices in Locosim, a cross-platform framework for simulation and real hardware. In this paper, we describe the architecture of Locosim and illustrate some use cases that show its potential.

Keywords: Computer Architecture for Robotics; Software Tools for Robot Programming; Software-Hardware Integration for Robot Systems

1 Introduction

Writing software for robotic platforms can be arduous, time-consuming, and error-prone. In recent years, the number of research groups working in robotics has grown exponentially, each group having platforms with peculiar characteristics. The choice of morphology, actuation systems, and sensing technology is virtually unlimited, and code reuse is fundamental to getting new robots up and running in the shortest possible time. In addition, it is pervasive for researchers willing to test new ideas in simulation without wasting time in coding, for instance, using high-level languages for rapid code prototyping. To pursue these goals, in the past years several robotics frameworks have been designed for teaching or for controlling specific platforms, e.g., OpenRAVE [1], Drake [2] and SL [3].

To avoid roboticists *reinventing the wheel* whenever they buy or build a new robot, we present our framework Locosim¹. Locosim is designed with the

¹ Locosim can be downloaded from www.github.com/mfocchi/locosim.

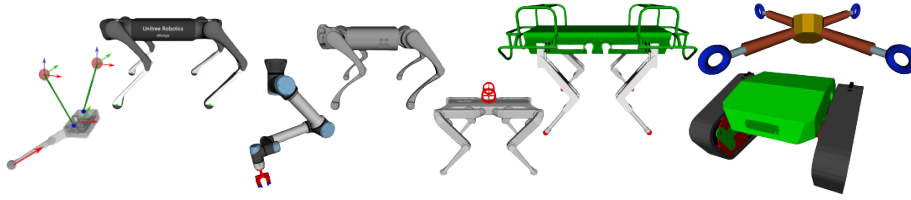


Figure 1. Examples of robots already included in Locosim (from left to right, top to bottom): Aliengo [7], Go1 [8], HyQ [9], Starbot, CLIO [10], UR5 [11] with Gripper, Solo with Flywheels [12], Tractor (images are not in scale).

primary goal of being platform-independent, dramatically simplifying the task of interfacing a robot with planners and controllers. Locosim consists of a ROS control node [4] (the *low-level* controller), written in C++, that interfaces a custom Python ROS node (the *high-level* planner/controller) with either a Gazebo simulator [5] or the real hardware. The planner relies on Pinocchio [6] for computing the robot’s kinematics and dynamics and closes the control loop at a user-defined frequency.

1.1 Advantages of Locosim

The benefits of the proposed framework are multiple.

- Locosim is platform-independent. It supports a list of robots with different morphology (e.g., quadrupeds, arms, hybrid structures, see Fig. 1) and provides features for fast designing and adding new robots.
- Locosim implements functions needed for all robots. Once the robot description is provided, no effort is spent on libraries for kinematics, dynamics, logging, plotting, or visualization. These valuable tools ease the synthesis of a new planner/controller.
- Locosim is easy to learn. The end user invests little time in training and gets an open-source framework with all the benefits of Python and ROS.
- Locosim is modular. Because it heavily uses the inheritance paradigm, classes of increasing complexity can provide different features depending on the nature of the specific robotic application. For instance, the controller for fixed-base robotic arms with a grasping tool can be reused for a four-legged robot with flywheels since it is built upon the same base class.
- Locosim is extensible. Our framework is modifiable and the end-user can add any supplementary functionality.
- Locosim is easy to install. It can be used either inside a virtual machine, a docker container, or natively by manually installing dependencies.

1.2 Outline

The remainder of this paper is organized as follows. In Section 2 we highlight the critical requirements of a cross-platform robotics framework. In Section 3

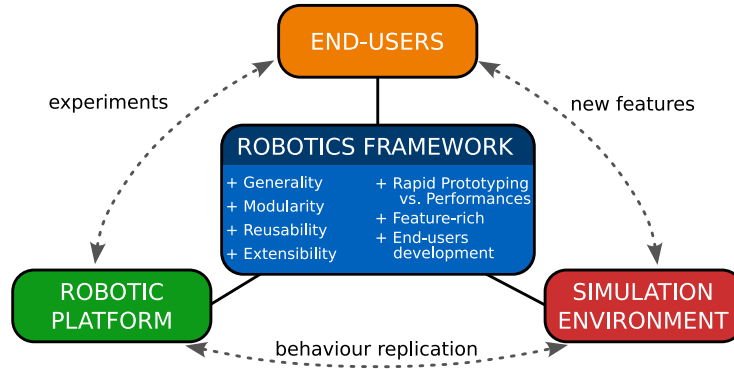


Figure 2. End-users, robotic platform and simulation environment make a triad only if an effective robotics framework can join them.

we detail structure and features of Locosim. In Section 4 we discuss use-case examples of our framework, either with the real robot or with its simulated digital twin. Eventually, we condense the results and present future works in Section 5.

2 Key aspects of a robotics framework

In the most general sense, a *robotics framework* is a software architecture of programs and data that adhere to well-defined rules for operating robots. *End-users* are people who will ultimately use the robotics framework and potentially bring some modifications. The robotics framework is the center of a triangle having at its vertices the end-user, the robotic platform and the simulation environment (see Fig. 2). The simulation must replicate the behaviour of the robot with a sufficient degree of accuracy. The end-user must be able to test new features and ideas in the simulation environment before running them on the robot platform. A robotics framework should provide the link among these three. In this context, we identify a list of crucial requirements that a robotics framework must possess: generality, modularity, reusability, extensibility, rapid prototyping vs. performances, feature-rich, and end-users development.

Generality. It is essential to release software free from unnatural restrictions and limitations. An end-user may require to design software for controlling a single robot, for multi-robot cooperation, for swarm coordination, or for reinforcement learning, which requires an abundant number of robots in the training phase. It should be possible to model any kinematic structure (floating base/fixed base robot, kinematic loops, etc.).

Modularity. A robotics framework should provide separate building blocks of functionalities according to the specific needs of the robot or the application.

These building blocks must be replaceable and separated by clear interfaces. The end-user may want to only visualize a specific robot in a particular joint configuration, move the joints interactively, or plan motions and understand the effects of external forces. Each of these functions must be equipped with tools for debugging and testing, e.g., unit tests. Replacing each module with improved or updated versions with additional functionalities should be easy.

Reusability. Pieces of source code of a program should be reused by reassembling them in more complex programs to provide a desired set of functionalities, with minor or no modifications. From this perspective, parametrization is a simple but effective method, e.g., the end-user should be able to use the same framework with different robots changing only a single parameter. In the same way, *digital twins* [13] can be realized by varying the status of a flag that selects between the real hardware and the simulation environment. This avoids writing different codes for the simulator and the real robot.

Extensibility. A robotics framework must be designed with the foresight to support the addition and evolution of hardware and software that may not exist at implementation time. This property can be achieved by providing a general set of application programming interfaces (APIs). Concepts typical of Object-Oriented Programming, such as inheritance and polymorphism, play a crucial role in extensibility.

Rapid Prototyping vs. Performances. A framework should allow for fast code prototyping of researchers' ideas. More specialized controllers/planners are built from simpler ones in the form of recursive inheritance (*matryoshka principle*). In this way end-users have unit tests at different levels of complexity. With fast code prototyping, end-users can quickly write software without syntax and logic errors. However, they do not have any assurance about the performance: such code is just good enough for the case of study in a simulation environment. Stringent requirements appear when executing codes on real robots, e.g., short computation time and limited memory usage. Thus, the framework must expose functionalities that can deal with performance.

Feature-rich. Most of the end-users need a sequence of functionalities when working with robots. These include but are not limited to the computation of kinematics and dynamics, logging, plotting, and visualization. A robotics framework should provide them, and they must be easily accessible.

End-users Development. Besides its implementation details, a robotics framework should provide methods, techniques and tools that allow end-users to create, modify, or extend the software [14] in an easy way. It should run on widely used Operating Systems and employ renowned high-level programming

languages that facilitate software integration. Clear documentation for installation and usage must be provided, and modules should have self-explanatory names and attributes.

3 Locosim Description

Locosim was born as a didactic framework to simulate both fixed- and floating-base robots. Quickly it evolved to be a framework for researchers that want to program newly purchased robots in a short time. Locosim runs on machines with Ubuntu as Operating System, and it employs ROS as middleware. Within Locosim, end-users can write robot controllers/planners in Python.

3.1 Architecture

Locosim consists of four components: Robot Descriptions, Robot Hardware Interfaces, ROS Impedance Controller, and Robot Control. We illustrate each component in the following⁴.

Robot Descriptions. The Robot Descriptions component contains dedicated packages for the characterization of each robot. For instance, the package that contains files to describe the robot `myrobot`, a generic mobile robot platform, is named `myrobot_description`. With the main focus on fast prototyping and human readability, the robot description is written in Xacro (XML-based scripting language) that avoids code replication through macros, conditional statements and parameters. It can import descriptions of some parts of the robot from `urdfs` sub-folder or meshes describing the geometry of rigid bodies from the `meshes` sub-folder. At run time, the Xacro file is converted into URDF, allowing the end-user to change some parameters. The `gazebo.urdf.xacro` launches `ros_control` package and the `gazebo_ros_p3d` plugin, which publishes the pose of the robot trunk in the topic `/ground_truth` (needed only for floating-base robots). The robot description directory must contain the files `upload.launch` and `rviz.launch`. The former processes the Xacro generating the URDF and loads into the parameter server, and the latter allows to visualize the robot and interact with it by providing the `conf.rviz` file. This is the configuration for the ROS visualizer RViz, which can be different for every robot. Additionally, the file `ros_impedance_controller.yaml` must be provided for each robot: it contains the sequence of joint names, joint PID gains and the home configuration. The Python script `go0` will be used during the simulation startup to drive the robot to the home configuration.

⁴ Some of the functions in Locosim components, which are quite established for the robotics community, are named after [3].

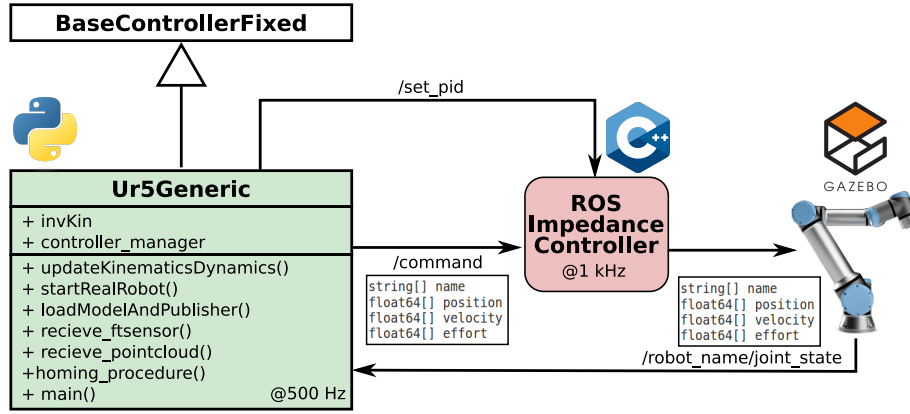


Figure 3. Schematic representation of a typical use-case of Locosim. The end-user wants to simulate the UR5 robot arm. An instance of **Ur5Generic**, which is a derived class of **BaseControllerFixed**, sends the command to the robot through **ros.impedance.controller** and it receives back the actual state. **Ur5Generic** implements features to manage the real robot and the gripper, perform a homing procedure at startup and a class for inverse kinematics.

Robot Hardware Interfaces. This folder contains drivers for the real hardware platforms supported by Locosim. They implement the interface that bridges the communication between the controller and the real robot, abstracting the specificity of each robot and exposing the same interface (e.g., **EffortInterface**). For instance, the UR5 robot through its driver provides three possible ROS hardware interfaces: Effort, Position and Velocity, hiding the details of the respective underlying low-level controllers.

ROS Impedance Controller. ROS Impedance Controller is a ROS package written in C++ that implements the *low-level* joint controller, in the form of PID with feedforward effort (force or torque). The `/joint_state_publisher` publishes the actual position, velocity and effort for each robot joint. By default, the loop frequency is set to 1 kHz. This and other parameters can be regulated in the launch file called **ros.impedance.controller.launch**. Robots with specific needs can be dealt with by specifying a custom launch file. This is the case of the CLIO climbing robot that requires the model of the mountain to which it is attached. The `robot_name` parameter is used to load the correct robot description. If the `real_robot` flag is set to true, the robot hardware interface is initialized; otherwise, the Gazebo simulation starts running first the `go0` script from the robot description. In any case, Rviz will be opened with the robot's specific configuration file `conf.rviz`. The end-user can manually change the location where the robot is spawned in Rviz with the `spawn` parameters. Physics parameters for the simulator are stored in the sub-folder **worlds**.

Table 1. Main attributes and methods of the **BaseControllerFixed** (BCF) and of the **BaseController** (BC) classes. All vectors (unless specified) are expressed in world frame. For methods with the same name, the derived class loads the method of the parent class and adds additional elements specific to that class.

	Name	Meaning	Class
Attributes	q, q_des	actual / desired joint positions	BCF, BC
	qd, qd_des	actual / desired joint velocities	BCF, BC
	tau, tau_ffwd	actual / feed-forward joint torques	BCF, BC
	x_ee	position of the end-effector expressed in base frame	BCF
	contactForceW	contact force at the end-effector	BCF
	contactMomentW	contact moment at the end-effector	BCF
	basePoseW	base position and orientation in Euler angles	BC
	baseTwistW	base linear and angular velocity	BC
	b_R_w	orientation of the base link	BC
	contactsW	position of the contacts	BC
	grForcesW	ground reaction forces on contacts	BC
Methods	loadModelAndPublishers()	creates the object robot (Pinocchio wrapper) and loads publishers for visual features (ros_pub), joint commands and declares subscriber to /ground_truth and /joint_states	BCF, BC
	startFramework()	launch ros_impedance_controller	BCF, BC
	send_des_jstate()	publishes /command topic with set-points for joint positions, velocities and feed-forward torques	BCF, BC
	startupProcedure()	initializes PID gains	BCF, BC
	initVars()	initializes class attributes	BCF, BC
	logData()	fill in the variables x_log with the content of x for plotting purposes, it needs to be called at every loop	BCF, BC
	receive_jstate()	callback associated to the subscriber /joint_states , fills in actual joint positions, velocities and torques	BCF, BC
	receive_pose()	callback associated to the subscriber /ground_truth , fills in the actual base pose and twist, and publishes the fixed transform between world and base	BC
	updateKinematics()	from q, qd, basePoseW, baseTwistW , computes position and velocity of the robot's center of mass, the contacts position, the associated Jacobians, the ground reaction forces, the centroidal inertia, the mass matrix and the non-linear effects	BC

Robot Control. From the end-user perspective, this is the most crucial component. It embraces classes and methods for computation of the robot’s kinematics and dynamics, logging and plotting time series variables, and real-time visualization on Rviz. Within this component, *high-level* planning/control strategies are implemented. The codes of the component are entirely written in Python and have a few dependencies: above all, NumPy [15] and Pinocchio [6]. The former offers tools for manipulating multidimensional arrays; the latter implements functions for the robot’s kinematics and dynamics. Pinocchio can be an essential instrument for researchers because of its efficient computations. Nevertheless, it can be time-consuming and cumbersome to understand for newcomers. To facilitate the employment, we developed a `custom_robot_wrapper` for building a `robot` object, computing robot mass, center of mass, Jacobians, centroidal inertia, and so on with easy-to-understand interfaces.

For the end-user, the starting point for building up a robot planner is the class `BaseControllerFixed`, suitable for fixed-base robots, and the derived class `BaseController`, which handles floating-base ones. In Table 1, we report a list of the main attributes and methods of `BaseController` and of its parent `BaseControllerFixed`. For having a more complex and specific controller, the end-user can create its own class, inheriting from one of the previous two and adding additional functionalities. E.g., `QuadrupedController` inherits from `BaseController`, and it is specific for quadruped robots. `Ur5Generic` adds to `BaseControllerFixed` the features of controlling the real robot, a gripper and a camera attached (or not) to the robotic arm UR5 (see Fig. 3). The controller class is initialized with the string `robot_name`, e.g., we write `BaseController(myrobot)` for the controller of `myrobot`. The end-user must pay particular attention to this string because it creates the link with the robot description and the robot hardware interface if needed. The `robot_name` is used for accessing the dictionary `robot_param` too. Among the other parameters of the dictionary, the flag `real_robot` permits using the same code for both the real (if set to true) and simulated (false) robot, resulting in the digital twin concept. We specify the mandatory parameters in Table ???. The `BaseControllerFixed` class contains a `ControllerManager` to seamlessly swap between the control modes and controller types if the real hardware supports more than one. For instance, UR5 has two control modes (point, trajectory) and two controller types (position, torque), whereas Go1 supports a single low-level torque controller. Additionally, the `GripperManager` class manages the gripper in different ways for the simulation (i.e., adding additional finger joints) and for the real robot (on-off opening/closing service call to the UR5 driver as specified by the manufacturer), hiding this complexity to the end-user. The method `startFramework()` permits to launch the simulation or the driver, executing `ros_impedance_controller.launch`. It takes as input the list `additional_args` to propagate supplementary arguments, dealing with robot and task specificity. In the `components` folder there are additional classes for inverse kinematics, whole-body control, leg odometry, IMU utility, filters and more. Finally, the

folder `lab_exercises` contains an ample list of scripts, with didactic exercises of incremental complexity, to learn Locosim and the main robotics concepts.

3.2 Analysis of Design Choices

To fulfill the requirements stated in Section 2, we made a number of choices. We want to focus on why we selected ROS as middleware, Python as (preferred) programming language, and Pinocchio as computational tool for the robot’s kinematics and dynamics.

Why ROS. The ROS community is spread worldwide. Over the last decade, the community produced hundreds or thousands of open-source tools and software: from device drivers and interface packages of various sensors and actuators to tools for debugging, visualization, and off-the-shelf algorithms for multiple purposes. With the notations of nodes, publishers, and subscribers, ROS quickly solves the arduous problem of resource handling between many processes. ROS is reliable: the system can still work if one node crashes. On the other hand, learning ROS is not an effortless task for newcomers. Moreover, modeling robots with URDF can take lots of time, as well as starting simulations [16]. Locosim relieves end-users from these complications by adopting a *common skeleton* infrastructure for the robot description and for the high-level planner/controller.

Why Python. Among the general-purpose programming languages, Python is one of the most used. It is object-oriented and relatively straightforward to learn and use compared to other languages. The availability of open-source libraries and packages is virtually endless. These reasons make Python perfect for fast prototyping software. Being an interpreted language, Python may suffer in terms of computation time and memory allocation, colliding with the real-time requirements of the real hardware. In these cases, when testing the code in simulation, the end-user may consider using profiling tools to look for the most demanding parts of the code. Before executing the software on the real robot, these parts can be optimized within Python or translated into C++ code, providing Python bindings. For the same performance reasons, the most critical part of the framework, the low-level controller, is directly implemented in C++.

Why Pinocchio. Pinocchio [6] is one of the most efficient tools for computing poly-articulated bodies’ kinematics and dynamics. Differently from other libraries in which a meta-program generates some source code for a single specific robot model given in input, as in the case of RobCoGen [17], Pinocchio is a dynamic library that loads at runtime any robot model. This characteristic makes it suitable for a cross-platform framework. It implements the state-of-the-art Rigid Body Algorithms based on revisited Roy Featherstone’s algorithms [18]. Pinocchio is open-source, mostly written in C++ with Python bindings and several projects rely on it. Pinocchio also provides derivatives of the kinematics and dynamics, which are essential to in gradient-based optimization.

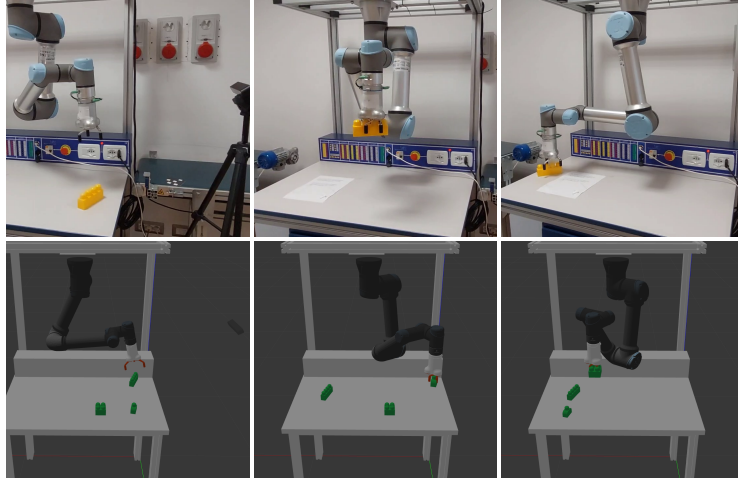


Figure 4. Execution of the pick-and-place task with the anthropomorphic arm UR5. The end-user can drive the real hardware (setting `real_robot` to true) or perform a simulation (`real_robot` to false).

4 Use Cases

We want to emphasize the valuable features of Locosim with practical use cases².

4.1 Visualize a Robot: kinematics check

As first use case, we illustrate the procedure to visualize the quadruped robot Aliengo (see Fig. 1) in RViz and how to manually interact with its joints. This is a debugging tool which is crucial during the design process of a new robot, because it allows to test the kinematics without added complexity. In the Robot Description package, we create a folder named `aliengo_description`. In the `robots` folder we add the XML file for describing the robot’s kinematic and dynamic structure. We make use of the flexibility of the open source Xacro language to simplify writing process: we include files that describe a leg, the transmission model, and the meshes for each of the bodies. We create the `launch` folder containing the files `upload.launch` and `rviz.launch`. Launching `rviz.launch` from command line, the end-user can visualize the robot in RViz and manually move the joints by dragging the sliders of the `joint_state_publisher_gui`. The `conf.rviz` file helps the end-user to restore previous sessions in RViz. With this simple use case we can effectively understand the importance of the key aspects formalized so far. Being extensible, Locosim allows for the introduction of any new robots, reusing parts of codes already present.

² A video showing the above-mentioned and other use cases can be found here:
<https://youtu.be/ZwV1LEqK-LU>

4.2 Simulation and Real Robot

As a second example, we present a pick-and-place task with the anthropomorphic arm UR5 (6 degrees of freedom, see Fig. 1). The pipeline of planning and control starts with the launch file `ros_impedance_controller.launch`. This is common for all the robots: it loads the `robot_description` parameter calling the `upload.launch` and it starts the Gazebo simulator or the robot driver according to the status of the `real_robot` flag. Additionally, it loads the controller parameters (e.g., PID gains), which are located in the `robot_description` package of the robot. In the simulation case, the launch file spawns the robot at the desired location. In the real robot case, the robot driver is running (in a ROS node) on an onboard computer while the planner runs on an external computer. In both cases, the RViz GUI shows the robot configuration. Another node running on the same computer reads from a fixed frame ZED2 camera and publishes a point cloud message of the environment on a ROS topic. We extract the coordinates of the plastic bricks that are present in the workspace. With `Ur5Generic`, we plan trajectories for the end-effector position and orientation to grasp and relocate the bricks. We set an inverse kinematic problem to find a joint reference trajectory. It is published in the `/ur5/command` topic, together with feed-forward torques for gravity compensation. The low-level `ros_impedance_controller` provides feedback for tracking the joint references, based on the actual state in `/joint.state`. On the real robot there is no torque control and only position set-points are provided to the `ur5/joint_group_pos_controller/command` topic as requested by the robot driver provided by the manufacturer. All this is dealt with by the `controller_manager` class, transparent to the end users. The power of Locosim lies on the fact that it is possible to use the same robot control code both to simulate the task and to execute it on the real robot, as reported in Fig. 4.

5 Conclusions

Locosim is a platform-independent framework for working with robots, either in a simulation environment or with real hardware. Integrating features for computation of robots' kinematics and dynamics, logging, plotting, and visualization, Locosim is a considerable help for roboticists that needs a starting point for rapid code prototyping. If needed, performances can be ensured by implementing the critical parts of the software in C++, providing Python bindings. We proved the usefulness and versatility of our framework with use cases. Future works include the following objectives: Locosim will be able to handle multiple platforms simultaneously to be used in the fields of swarm robotics and collaborative robotics. We want to provide support for ROS2 since ROS lacks relevant qualifications such as real-time, safety, certification, and security. Additionally, other simulator like Coppelia Sim or PyBullet will be added to Locosim.

References

1. "OpenRAVE." [Online]. Available: <https://github.com/rdiankov/openrave>

2. “Drake.” [Online]. Available: <https://github.com/RobotLocomotion/drake>
3. S. Schaal, “The sl simulation and real-time control software package,” Los Angeles, CA, Tech. Rep., 2009, clmc. [Online]. Available: <http://www-clmc.usc.edu/publications/S/schaal-TRSL.pdf>
4. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
5. N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.
6. J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, “The Pinocchio C++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives,” in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 614–619.
7. “Aliengo.” [Online]. Available: <https://www.unitree.com/en/aliengo>
8. “Go1.” [Online]. Available: <https://www.unitree.com/en/go1>
9. C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, “Design of hyq—a hydraulically and electrically actuated quadruped robot,” *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 225, no. 6, pp. 831–849, 2011.
10. M. Focchi, M. Bensaadallah, M. Frego, A. Peer, D. Fontanelli, A. Del Prete, and L. Palopoli, “Clio: a novel robotic solution for exploration and rescue missions in hostile mountain environments,” *arXiv preprint arXiv:2209.09693*, 2022.
11. “UR5.” [Online]. Available: <https://www.universal-robots.com/products/ur5-robot/>
12. F. Roscia, A. Cumerlotti, A. Del Prete, C. Semini, and M. Focchi, “Orientation control system: Enhancing aerial maneuvers for quadruped robots,” *Sensors*, vol. 23, no. 3, p. 1234, 2023.
13. A. K. Ramasubramanian, R. Mathew, M. Kelly, V. Hargaden, and N. Papakostas, “Digital twin for human–robot collaboration in manufacturing: Review and outlook,” *Applied Sciences*, vol. 12, no. 10, p. 4811, 2022.
14. H. Lieberman, F. Paternò, and V. Wulf, *End user development*. Springer, 2006, vol. 9.
15. C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: <https://doi.org/10.1038/s41586-020-2649-2>
16. L. Joseph and J. Cacace, *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.
17. M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, “Robcogen: a code generator for efficient kinematics and dynamics of articulated robots, based on domain specific languages,” *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 36–54, 2016.
18. R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.