Studies in Computational Intelligence 895

Anis Koubaa *Editor*

Robot Operating System (ROS)

The Complete Reference (Volume 5)

Springer

Studies in Computational Intelligence

Volume 895

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

The books of this series are submitted to indexing to Web of Science, EI-Compendex, DBLP, SCOPUS, Google Scholar and Springerlink.

More information about this series at http://www.springer.com/series/7092

Anis Koubaa
Editor

Robot Operating System (ROS)

The Complete Reference (Volume 5)

Springer

Editor
Anis Koubaa
College of Computer Science
and Information Systems
Prince Sultan University
Riyadh, Saudi Arabia

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-030-45955-0 ISBN 978-3-030-45956-7 (eBook)
https://doi.org/10.1007/978-3-030-45956-7

© Springer Nature Switzerland AG 2021
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

About This Book

This book is the fifth volume of the successful book series on *Robot Operating System: The Complete Reference.*

The objective of the book is to provide the reader with comprehensive coverage on the Robot Operating Systems (ROS) and the latest trends and contributed systems. ROS is currently considered as the primary development framework for robotics applications.

There are twelve chapters organized into six parts. Part I presents for the first time two chapters on the emerging ROS 2.0 framework. In Part II, two chapters deal with multi-robot systems, namely on SLAM and Swarm coordination. Part III provides two chapters on autonomous systems, namely self-driving cars, and unmanned aerial systems. Part IV deals with the contribution of simulation frameworks for ROS. In Part V, two chapters present contributions to the robotic manipulator and legged robots. Finally, Part VI presents emerging topics on monocular SLAM and a chapter in fault tolerance system for ROS. I believe that this book will be a valuable companion for ROS users and developers to learn more ROS capabilities and features.

Contents

ROS 2.0

Development of an Industry 4.0 Demonstrator Using Sequence Planner and ROS2 3
Endre Erős, Martin Dahl, Atieh Hanna, Per-Lage Götvall, Petter Falkman, and Kristofer Bengtsson

ROS2 For ROS1 Users 31
George Stavrinos

Multi-Robot Systems

mrgs: A Multi-Robot SLAM Framework for ROS with Efficient Information Sharing 45
Gonçalo S. Martins, David Portugal, and Rui P. Rocha

Agile Experimentation of Robot Swarms in Large Scale 77
Vivian Cremer Kalempa, Marco Antonio Simões Teixeira, André Schneider de Oliveira, and João Alberto Fabro

Autonomous Systems

Lessons Learned Building a Self Driving Car on ROS 127
Nicolò Valigi

Landing an Autonomous UAV on a Moving Platform Using only a Front Facing Camera 157
George Stavrinos

Simulation Frameworks

Integrating the Functional Mock-Up Interface with ROS and Gazebo 187
Ralph Lange, Silvio Traversaro, Oliver Lenord, and Christian Bertsch

An ARVA Sensor Simulator 233
Jonathan Cacace, Nicola Mimmo, and Lorenzo Marconi

Robotic Manipulation and Control

ROS Implementation for Untethered Microrobot Manipulation 269
Aytaç Kahveci, Nail Akçura, Levent Çetin, Abdulkareem Alasli, and Özgür Tamer

ClegS: A Meta-Package to Develop C-Legged Robots 295
Jorge De León, Raúl Cebolla, Jaime del Cerro, and Antonio Barrientos

Emerging Topics

Video Frames Selection Method for 3D Reconstruction Depending on ROS-Based Monocular SLAM 351
Yasin Maan Yousif and Iyad Hatem

ROS Rescue: Fault Tolerance System for Robot Operating System 381
Pushyami Kaveti and Hanumant Singh

ROS 2.0

Development of an Industry 4.0 Demonstrator Using Sequence Planner and ROS2

Endre Erős, Martin Dahl, Atieh Hanna, Per-Lage Götvall, Petter Falkman, and Kristofer Bengtsson

Abstract In many modern automation solutions, manual off-line programming is being replaced by online algorithms that dynamically perform tasks based on the state of the environment. Complexities of such systems are pushed even further with collaboration among robots and humans, where intelligent machines and learning algorithms are replacing more traditional automation solutions. This chapter describes the development of an industrial demonstrator using a control infrastructure called Sequence Planner (SP), and presents some lessons learned during development. SP is based on ROS2 and it is designed to aid in handling the increased complexity of these new systems using formal models and online planning algorithms to coordinate the actions of robots and other devices. During development, SP can auto generate ROS nodes and message types as well as support continuous validation and testing. SP is also designed with the aim to handle traditional challenges of automation software development such as safety, reliability and efficiency. In this chapter, it is argued that ROS2 together with SP could be an enabler of intelligent automation for the next industrial revolution.

E. Erős · M. Dahl · P. Falkman · K. Bengtsson (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: kristofer.bengtsson@chalmers.se

E. Erős
e-mail: endree@chalmers.se

M. Dahl
e-mail: martin.dahl@chalmers.se

P. Falkman
e-mail: petter.falkman@chalmers.se

A. Hanna · P.-L. Götvall
Research & Technology Development, Volvo Group Trucks Operations, Gothenburg, Sweden
e-mail: atieh.hanna@volvo.com

P.-L. Götvall
e-mail: Per-Lage.Gotvall@volvo.com

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_1

1 Introduction

As anyone with experience with real automation development knows, developing and implementing a flexible and robust automation system is not a trivial task. There are currently many automation trends in industry and academia, like Industry 4.0, cyber-physical production systems, internet of things, multi-agent systems and artificial intelligence. These trends try to describe how to implement and reason about flexible and robust automation, but are often quite vague on the specifics. When it comes down to the details, there is no silver bullet [1].

Volvo Group Trucks Operations has defined the following vision to better guide the research and development of next generation automation systems: Future Volvo factories [2], will be *re-configurable* and *self-balancing* to better handle rapid changes, production disturbances, and diversity of the product. Collaborative robots and other machines can support operators at workstations, where they use the same smart tools and are interchangeable wit operators when it comes to performing routine tasks. Operators are supported with mixed reality technology, and are provided with digital work instructions and 3D geometry while interacting intuitively with robots and systems. Workstations are equipped with smart sensors and cameras that feed the system with real-time status of products, humans, and other resources in the environment. Moreover, they are supported by advanced yet intuitive control, dynamic safety, online optimization and learning algorithms.

Many research initiatives in academia and industry have tried to introduce collaborative robots ("cobots") in the final assembly [3–5]. Despite their relative advantages, namely that they are sometimes cheaper and easier to program and teach [5] compared to conventional industrial robots, they are mostly deployed as robots "without fences" for co-active tasks [6]. Current cobot installations are in most cases not as flexible, robust or scalable as required by many tasks in manual assembly. Combined with the lack of industrial preparation processes for these types of systems, new methods and technologies must be developed to better support the imminent industrial challenges [7].

1.1 The Robot Operating System and Sequence Planner

This chapter discusses some of the challenges of developing and implementing an industrial demonstrator that includes robots, machines, smart tools, human-machine interfaces, online path and task planners, vision systems, safety sensors, etc. Coordination and integration of the aforementioned functionality requires a well-defined communication interface with good monitoring properties.

During the past decade, various platforms have emerged as middle-ware solutions trying to provide a common ground for integration and communication. One of them is the Robot Operating System (ROS), which stands out with a large and enthusiastic community. ROS enables all users to leverage an ever-increasing num-

ber of algorithms and simulation packages, as well as providing a common ground for testing and virtual commissioning. ROS2 takes this concept one step further, integrating the scalable, robust and well-proven Data Distribution Service (DDS) as its communications layer, eliminating many of the shortcomings of ROS1.

ROS2 systems are composed of a set of nodes that communicate by sending typed messages over named topics using a publish/subscribe mechanism. This enables a quick and semi-standardized way to introduce new drivers and algorithms into a system. However, having reliable communication, monitoring tools, as well as a plethora of drivers and algorithms ready to be deployed is not enough to be able to perform general automation. When composing a system of heterogeneous ROS2 nodes, care needs to be taken to understand the behavior of each node. While the interfaces are clearly separated into message types on named topics, knowledge about the workings of each node is not readily available. This is especially true when nodes contain an internal state that is not visible to the outside world. In order to be able to coordinate different ROS2 nodes, a control system needs to know both *how to control* the nodes making up the system, as well as how these nodes *behave*.

To control the behavior of nodes, an open-source control infrastructure called Sequence Planner (SP) has been developed in the last years. SP is used for controlling and monitoring complex and intelligent automation systems by keeping track of the complete system state, automatically planning and re-planning all actions as well as handling failures or re-configurations.

This chapter presents the development of a flexible automation control system using SP and ROS2 in a transformed truck engine final assembly station inspired by the Volvo vision. The chapter contributes with practical development guidelines based on the **lessons learned** during development, as well as detailed design rationales from the final demonstrator, using SP built on top of ROS2.

The next section introduces the industrial demonstrator that will be used as an example throughout the chapter. Section 3 discusses robust discrete control, why distributed control states should be avoided and the good practice of using state-based commands. The open-source control infrastructure SP is introduced in Sect. 4 and the generation of ROS2 code for logic and tests is presented in Sect. 5.

2 An Industrial Demonstrator

The demonstrator presented in this paper is the result of a transformation of an existing manual assembly station from a truck engine final assembly line, shown in Fig. 1, into an intelligent and collaborative robot assembly station, shown in Fig. 2.

In the demonstrator, diesel engines are transported from station to station in a predetermined time slot on Automated Guided Vehicles (AGVs). Material to be mounted on a specific engine is loaded by an operator from kitting facades located adjacent to the line. An autonomous mobile platform (MiR100) carries the kitted material to be mounted on the engine, to the collaborative robot assembly station.

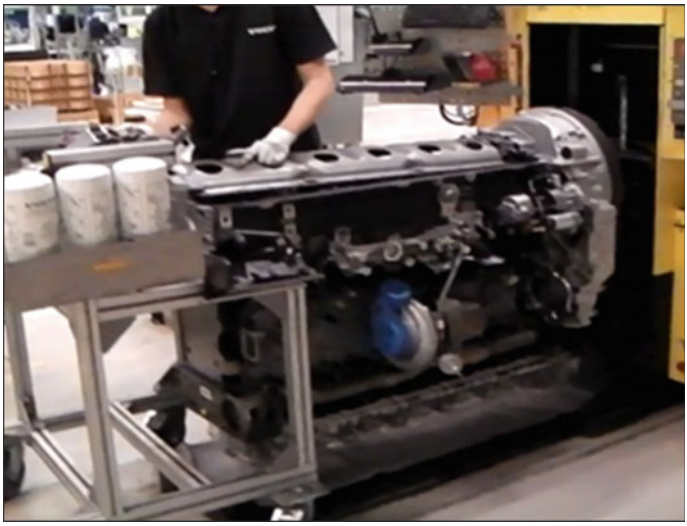


Fig. 1 The original manual assembly station

Fig. 2 Collaborative robot assembly station controlled by a network of ROS2 nodes. A video clip from the demonstrator: https://youtu.be/TK1Mb38xiQ8

In the station, a robot and an operator work together to mount parts on the engine by using different tools suspended from the ceiling. A dedicated camera system keeps track of operators, ensuring safe coexistence with machines. The camera system can also be used for gesture recognition.

Before the collaborative mode of the system starts, an authorized operator has to be verified by a RFID tag. After verification, the operator is greeted by the station

and operator instructions are shown on a screen. If no operator is verified, some operations can still be executed independently by the robot, however, violation of safety zones triggers a safeguard stop.

After the AGV and the kitting material have arrived, a Universal Robots (UR10) robot and an operator lift a heavy ladder frame on to the engine. After placing the ladder frame on the engine, the operator informs the control system with a button press on a smartwatch or with a gesture, after which the UR10 leaves the current end-effector and attaches to the nutrunner used for tightening bolts. During this tool change, the operator starts to place 24 bolts, which the UR10 will tighten with the nutrunner.

During the tightening of the bolts, the operator can mount three oil filters. If the robot finishes the tightening operation first, it leaves the nutrunner in a floating position above the engine and waits for the operator. When the operator is done, the robot attaches a third end-effector and starts performing the oil filter tightening operations. During the same time, the operator attaches two oil transport pipes on the engine, and uses the same nutrunner to tighten plates that hold the pipes to the engine. After executing these operations, the AGV with the assembled engine and the empty MiR100 leave the collaborative robot assembly station. All of this is controlled by the hierarchical control infrastructure Sequence Planner that uses ROS2.

2.1 ROS2 Structure

ROS2 has a much improved transport layer compared to ROS1, however, ROS1 is still ahead of ROS2 when it comes to the number of packages and active developers. In order to embrace the strengths of both ROS1 and ROS2, i.e. to have an extensive set of developed robotics software and a robust way to communicate, all sub-systems in the demonstrator communicate over ROS2 where a number of nodes have their own dedicated ROS1 master behind a bridge [8].

A set of related nodes located on the same computer are called a hub [8], where one or more hubs can be present on the same computer. ROS hubs are considered to have an accompanying set of bridging nodes that pass messages between ROS and ROS2.

In the demonstrator, ROS nodes are distributed on several computers shown in Table 1, including standard PCs as well as multiple Raspberry Pies and a Lattepanda Alpha (LP Alpha). Most of the nodes are only running ROS2.

All computers in the system are communicating using ROS2 and they can easily be added or removed. All computers connect to the same VPN network as well, to simplify the communication over cellular 4G.

One important **lesson learned** was that ROS1 nodes should only communicate with a ROS1 master on the same computer, else the overall system becomes hard to setup and maintain. During the demonstrator development, we tried to use ROS2 nodes whenever possible, and we only used ROS2 for communication between computers.

Table 1 Overview of the computers in the demonstrator. The ROS versions used were Kinetic for ROS1 and Dashing for ROS2. Some nodes, like number 5 and 7, need both

No.	Name	ROS v.	Computer	OS	Arch.	Explanation
1	Tool ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	Smart tool and lifting system control
2	RSP ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	Pneumatic conn. control and tool state
3	Dock ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	State of docked end-effectors
4	MiRCOM	Dashing	LP Alpha	Ubuntu 18	amd64	ROS2 to/from REST
5	MiR	Kinetic+Dashing	Intel NUC	Ubuntu 16	amd64	Out-of-the-box MiR100 ROS suite
6	RFIDCAM	Dashing	Desktop	Win 10	amd64	Published RFID and camera data
7	UR10	Kinetic+Dashing	Desktop	Ubuntu 16	amd64	UR10 ROS suite
8	DECS	Dashing	Laptop	Ubuntu 18	amd64	Sequence planner

There is a number of available implementations of DDS, and since DDS is standardized, ROS2 users can choose an implementation from a vendor that suits their needs. This is done through a ROS Middleware Interface (RMW), which also exposes Quality of Service (QoS) policies. QoS policies allow ROS2 users to adjust data transfer in order to meet desired communication requirements. Some of these QoS policies include setting message history and reliability parameters. This QoS feature is crucial, especially in distributed and heterogeneous systems, since setups are usually unique.

However, in the demonstrator described in this chapter, default settings were sufficient for all nodes since there were no real-time requirements. Even though standard QoS settings were used, how the messaging and communication was setup highly influenced the robustness of the system. In the next section, we will study this in more details. The communication between the hubs and the control system is in some cases auto-generated by Sequence Planner and will be introduced in more details in Sect. 5.

The main focus when developing this demonstrator was to perform the engine assembly using both humans and intelligent machines in a robust way. Other important aspects are non-functional requirements like safety, real-time constraints and performance. However, these challenges were not the main focus when developing the demonstrator. This will be important in future work, especially to handle the safety of operators.

3 Robust Discrete Control

The presented demonstrator consists of multiple sub-systems that need to be coordinated and controlled in a robust way. A robust discrete control system must handle unplanned situations, restart or failures of sub-systems, as well as instructing and monitoring human behavior.

During development, we also **learned** that one of the most complex challenges for a robust discrete control system, is how to handle asynchronous and non-consistent control states. In this section, we will therefore take a look at this and why it is important to avoid having a distributed control state and why state-based commands should be used instead of event-based. In many implementations, this challenge is often neglected and handled in an ad-hoc fashion. To better understand this challenge and how to handle it, let us start with some definitions.

3.1 States and State Variables

There are many different ways to define the state of a system. For example, if we would like to model the state of a door, we can say that it can be: $\{opened, closed\}$, or maybe even: $\{opening, opened, closing, closed\}$. We could also use the door's angle, or maybe just a boolean variable. Depending on what we would like to do with the model and what can we actually measure, we will end up with very different definitions. In this chapter, a model of a state of a system is defined using state variables:

Definition 1 A state variable v has a domain $V = \{x_1, \ldots, x_n\}$, where each element in V is a possible value of v. There are three types of variables: v^m: *measured* state variables, v^e: *estimated* state variables, and v^c: *command* state variables. For simplicity, we will use the following notation when defining a variable: $v = \{x_1, \ldots, x_n\}$.

State variables are used when describing the state of a system, where a state is defined as follows:

Definition 2 A state S is a set of tuples $S = \{\langle v_i, x_i \rangle, \ldots, \langle v_j, x_j \rangle\}$, where v_i is a state variable and $x_i \in V_i$ is the current value of the state variable. Each state variable can only have one current value at the time.

Let us go back to the door example to better understand states and state variables. The door can for example have the following state variables:

$$pos^e = \{opened, closed\}$$
$$locked^m = \{true, false\}$$

In this example, the position of the door is either *opened* or *closed* and is defined by the *estimated* state variable pos^e. It is called estimated since we can not measure the position of this example door. The pos^e variable must therefore be updated by the control system based on what actions have been executed (e.g. after we have opened the door, we can assume that it is opened).

The door also has a lock, that can either be locked or not, and is defined by the *measured* state variable $locked^m$. It is called measured since this door has a sensor that tells us if the door is locked or not.

The door can be in multiple states, for example, the state $\{\langle pos^e, closed \rangle, \langle locked^m, true \rangle\}$ defines that the door is closed and locked. To be able to change the state of the door, a control system needs to perform actions that unlocks and opens the door.

3.2 Event-Based Commands and Actions

When controlling different resources in an automation system, a common control approach is for the control system to send commands to resources, telling them what to do or react on. Also, the control system reacts on various events from the resources. When communicating in this fashion, which we can call event-based communication, all nodes will wait for and react on events. This type of communication and control often leads to a highly reactive system but is often challenging to make robust.

In ROS2, nodes communicate via topics, services or actions. It is possible to get some communication guarantees from the QoS settings in DDS, however, even if the communication is robust, we **learned** the hard way that it is not easy to create a robust discrete control system using only event-based communication. To better explain this, let us study how one of the nodes from the demonstrator could have been implemented, if we have had used event-based communication:

Example 1: The nutrunner: event-based communication

In the demonstrator, either the robot or the human can use the nutrunner. When started, it runs until a specific torque or a timeout has been reached. In the example, the nutrunner is controlled by the following commands: *start* or *reset*, and responds with the events *done* or *failed*. The communication could be implemented with ROS actions but for simplicity, let us use two publish-subscribe topics and the following ROS messages:

```
# ROS2 topic:  /nutrunner/command
string cmd     # 'start', or 'reset'

# ROS2 topic:  /nutrunner/response
string event   # 'done', or 'failed'
```

To start the nutrunner, the *start* command message is sent to the nutrunner node, which will start running. When a specific torque is reached, the node sends a *done* event back. If for some reason the torque is not reached after a timeout, or if the nutrunner fails in other ways, it will respond with a *failed* event. To be able to start the nutrunner again, the *reset* command must be sent to initialize the nutrunner.

If we can guarantee that each message is only received once and everything behaves as expected, this may work quite fine. However, if for some reason either the controller or the node fails and restarts, we will have a problem. If the controller executes a sequence of actions, it will assume that it should first send the *start* command and then wait for *done* or *failure*. However, if the nutrunner node is in the wrong state, for example if it is waiting for a *reset* command, nothing will happen. Then the control sequence is not really working, since the states of the two nodes are out of sync (i.e. the control node thinks that the nutrunner node is in a state where the control can send start).

A common problem in many industrial automation systems is that the state of the control system gets out of sync with the actual states of some resources. The reason for this is that many implementations are based on sequential control with event-based communication to execute operations, with the assumption that the sequence always starts in a well defined global state. However, this is often not the case, e.g. when a system is restarted after a break or a failure.

During development, we **learned** that in order to achieve flexibility and to handle failures and restart, a system should never be controlled using strict sequences or if-then-else programming. When the control system gets out of sync, the common industrial practice is to remove all products and material from the system, restart all resources and put them in well defined initial “home” states. Since we need a robust control that can handle resource failures, flexible and changing work orders and other unplanned events, a better control strategy is needed.

So, if the problem is that some nodes need to know the states of other nodes, maybe we can solve this by sharing internal state of the node and keep the event-based communication?

3.3 Distributed State

If each node shares its state and is responsible for updating it, we can say that the global state and the control is distributed on all nodes. This is a popular control strategy in many new automation trends, since it is simple to implement and hide some of the node details from other nodes. Let us take a look at how a distributed control strategy could have been implemented in the nutrunner example:

Example 2: The nutrunner: Sharing state

In this example, the node communicates with the hardware via 1 digital output and 3 digital inputs. These can be defined as state variables:

$run^c = \{false, true\}$
$run^m = \{false, true\}$
$tqr^m = \{false, true\}$
$fail^m = \{false, true\}$

To run the nutrunner, the command state variable run^c is set to true, which is a digital output, and then the nutrunner responds back by setting the measured state variable run^m to true when it has started. When the torque has been reached, tqr^m becomes true, or if it fails or timeouts, $fail^m$ becomes true. These inputs are reset to false by setting run^c to false again, which can also be used during execution to stop the nutrunner.

Since the nutrunner node is controlled via events (like in the example before), it does not need to expose these details, but instead implements this as an internal node control loop. In our example it communicates its state using the following message:

```
# ROS2 topic:  /nutrunner/state
string state   # 'idle', 'running', 'torque', or 'failed'
```

Now, when the nutrunner node is sharing its state, it is easier for an external control system to know the correct state and act accordingly. However, it is quite challenging to implement a robust state machine that actually works as expected.

When hiding implementation details by using a distributed control state, like in the nutrunner example, the control will initially appear to be simple to implement and may work at first. However, based on our struggling during the demonstrator development, we found out that it is still quite hard to implement fully correct logic in each node and keep the states of all nodes in sync with each other. The main reason for this is that the control system anyway needs to know the details of each node to actually figure out how to restart a system when something happens. So, the **lesson learned:** restart of most nodes almost always depends on the state of other nodes.

It is also challenging to handle safety concerns where the system must guarantee some global specifications when the detailed control is distributed. We have learned

the hard way that a local control loop and internal state in each node almost always becomes tangled with multiple cross-cutting concerns making the complete system hard to maintain and troubleshoot. Another approach is to centralize the control and to use state-based commands.

3.4 State-Based Commands

The benefit of centralized control is that the control system knows about everything and can guarantee global requirements. It can figure out the optimal operation sequence and can restart the complete system in an efficient way.

In the door control earlier, the controller sent the *open* event and then it was waiting for the *done* event. When using state-based commands, the controller instead sends a reference position, $refpos^c$, telling the door what state it wants it to be in while the door is sending back its real state. These messages are sent when something changes as well as at a specific frequency. If one of the commands is lost, another one will soon be sent.

Let us look at how a state-based command could be implemented for the nutrunner example.

Example 3: The nutrunner: State-based communication

Instead of using events to control the nutrunner, the control system and the node instead communicate using the already defined state variables with the following messages:

```
# ROS2 topic:  /nutrunner/command
bool run_c

# ROS2 topic:  /nutrunner/state
bool run_m
bool tqr_m
bool fail_m
bool run_c
```

The control system is sending a reference state while the nutrunner is returning its current state as well as mirroring back the reference. The centralized control system now knows everything about the nutrunner since in practice, the detailed control loop is moved from the nutrunner node to the control node. The mirror is monitored by an external ROS2 node which checks that all nodes are receiving their commands.

During the development of the demonstrator, we discovered that by using state-based commands and avoiding local states in each node, it is possible to implement control strategies for efficient restart, error handling, monitoring and node implementation. It is much easier to recover the system by just restarting it, since the nodes do

not contain any local state. Stateless nodes also makes it possible to add, remove or change nodes while running, without complex logic. This has probably been one of the most important **lessons learned** during the development.

The control of the demonstrator is using centralized control, a global state, and state-based communication. While state-based communication requires a higher number of messages to support control compared to event-based communication, and centralized control with a global state is harder to scale compared to decentralized approaches, we believe that these trade-offs are worth it when creating a robust and efficient control architecture for these types of automation systems. Perhaps the bigger challenge with this approach though, is that the control system needs to handle all the complexity to make it robust. To aid in handling this complexity, we have developed a new control infrastructure called Sequence Planner for robust automation of many sub-systems.

4 Sequence Planner

The use of state-based commands introduced in Sect. 3 implies that a number of devices should continuously get a reference (or goal) state. An overall control scheme is then needed in order to choose what these goal states should be at all time. At the same time, the control system should react to external inputs like state changes or events from machines, operators, sensors, or cameras. Developing such systems quickly becomes difficult due to all unforeseen situations one may end up in. Manual programming becomes too complex and executing control sequences calculated off-line become very difficult to recover from when something goes wrong.

Several frameworks in the ROS community are helping the user with composing and executing robot tasks (or algorithms). For example, the framework ROS-Plan [9] that uses PDDL-based models for automated task planning and dispatching, SkiROS [10] that simplifies the planning with the use of a skill-based ontology, eTaSL/eTC [11] that defines a constraint-based task specification language for both discrete and continuous control tasks or CoSTAR [12] that uses Behavior Trees for defining the tasks. However, these frameworks are mainly focused on robotics rather than automation.

4.1 A New Control Infrastructure

Based on what we **learned**, we have employed a combination of on-line automated planning and formal verification, in order to ease both modeling and control. Formal verification can help us when creating a model of the correct behavior of device nodes while automated planning lets us avoid hard-coded sequences and "if-then-else" programming and allowing us to be resource-agnostic on a higher level. We end

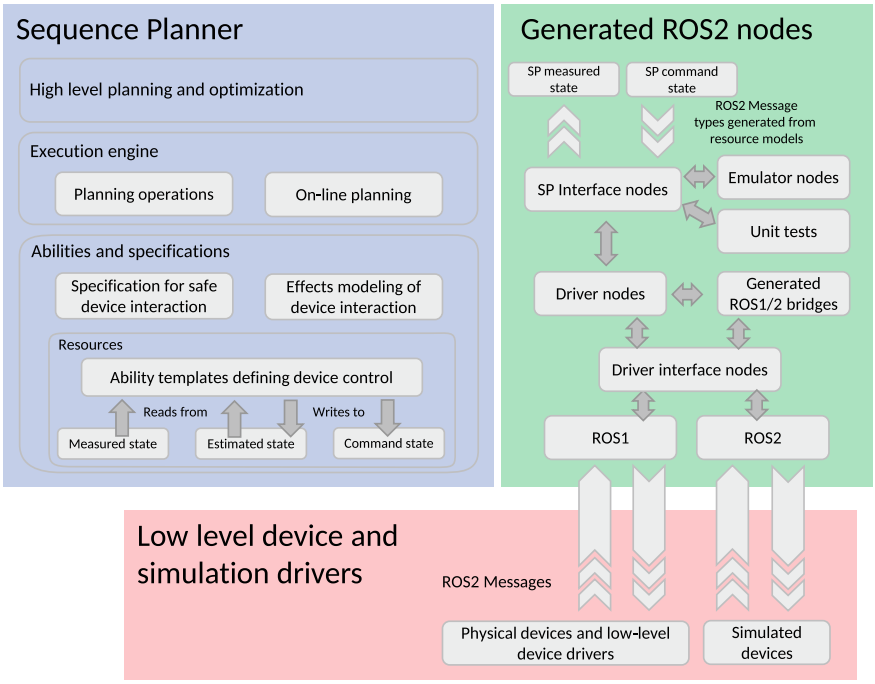


Fig. 3 Sequence Planner control infrastructure overview

up with a control scheme that continuously deliberates the best goals to distribute to the devices. This is implemented in our research software Sequence Planner (SP).

SP is a tool for modeling and analyzing automation systems. Initially [13], the focus was on supporting engineers in developing control code for programmable logical controllers (PLCs). During the first years, algorithms to handle product and automation system interaction [14], and to visualize complex operation sequences using multiple projections [15] were developed. Over the years, other aspects have been integrated, like formal verification and synthesis using *Supremica* [16], restart support [17], cycle time optimization [18], energy optimization and hybrid systems [19], online monitoring and control [20], as well as emergency department online planning support [21]. Recently, effort has been spent to use the developed algorithms and visualization techniques for control and coordination of ROS- and ROS2-based systems [22]. This section will give an overview of how SP can be used to develop ROS2-based automation systems.

The remainder of this section will describe how these SP control models look like, starting from the bottom up of the left part of Fig. 3. In Sect. 5 it is described how the model is translated into ROS2 nodes with corresponding messages (top right of Fig. 3. Because model-based control is used, it is essential that the models accurately represent the underlying systems (bottom of Fig. 3). Therefore Sect. 5.2 describes how randomized testing is used to ease the development of ROS2 nodes.

4.2 Resources

Devices in the system are modeled as *resources*, which groups the device's local state and discrete descriptions of the tasks the device can perform. Recall Definition 1, which divides system state into variables of three kinds: *measured state*, *command state*, and *estimated state*.

Definition 3 A resource i is defined as $r_i = \langle V_i^m, V_i^c, V_i^e, O_i \rangle, r_i \in R$ where V_i^m is a set of *measured* state variables, V_i^c is a set of *command* state variables, V_i^e is a set of *estimated* state variables, and O_i is a set of *generalized operations* defining the resource's abilities. R is the set of all resources in the system.

The resources defined here are eventually used to generate ROS2 nodes and message definition files corresponding to state variables.

4.3 Generalized Operations

Control of an automation system can be abstracted into performing *operations*. By constructing a transition system modeling how operations modify states of the resources in a system, formal techniques for verification and planning can be applied. To do this in a manner suitable to express both low-level ability operations and high-level planning operations, we define a *generalized operation*.

Definition 4 A generalized operation j operating on the state of a resource i is defined as $o_j = \langle P_j, G_j, T_j^d, T_j^a, T_j^E \rangle, o_j \in O_i$. P_j is a set of named predicates over the state of resource variables V_i. G_j is a set of un-named guard predicates over the state of V_i. Sets T^d and T^a define *control* transitions that update V_i, where T^d defines transitions that require (external from the ability) deliberation and T^a define transitions that are automatically applied whenever possible. T_j^E is a set of *effect* transitions describing the possible consequences to V_i^M of being in certain states. A transition $t_i \in \{T^d \cup T^a \cup T^E\}$ has a guard predicate which can contain elements from P_j and G_j and a set of actions that update the current state if and only if the corresponding guard predicate evaluates to true.

T_j^d, T_j^a, and T_j^E have the same formal semantics, but are separated due to their different uses. The effect transitions T_j^E define how the *measured state* is updated, and as such they are not used during actual low-level control like the control transitions T_j^d and T_j^a. They are important to keep track of since they are needed for on-line planning and formal verification algorithms, as well as for simulation based validation.

It is natural to define when to take certain actions in terms of what state the resource is currently in. To ease both modeling, planning algorithms and later on online monitoring, the guard predicates of the generalized operations are separated into one set of named (P_j) and one set of un-named (G_j) predicates. The named

predicates can be used to define in what state the operation currently is in, in terms of the set of local resource states defined by this predicate. The un-named predicates are used later in Sect. 4.6 where the name of the state does not matter.

4.4 Ability Operations

The behavior of resources in the system is modeled by *ability operations* (abilities for short). While it is possible to define transitions using only un-named guard predicates in G (from Definition 4), it is useful to define a number of "standard" predicate names for an ability to ease modeling, reuse and support for online monitoring. In this work, common meaning is introduced for the following predicates: *enabled* (ability can start), *starting* (ability is starting, e.g. a handshaking state), *executing* (ability is executing, e.g. waiting to be finished), *finished* (ability has finished), and *resetting* (transitioning from finished back to enabled). In the general case, the transition between *enabled* and *starting* has an action in T^d, while the transition from *finished* has an action in T^a. In other types of systems, other "standard" names could be used (e.g. *request* and *receive*).

Example 4: The nutrunner: resource and ability template

The resource nr containing state variables of the nutrunner can be defined as $r_{nr} = \langle \{run^m, tqr^m, fail^m\}, \{run^c\}, \emptyset, O_{nr} \rangle$. Notice that $V^e_{nr} = \emptyset$. This is the ideal case, because it means all local state of this resource can be measured.

The table below shows the transitions of a "run nut" ability, where each line makes up one possible transition of the ability. The ability models the task of running a nut, by starting to run the motors forward until a pre-programmed torque (tqr^m) has been reached. Notice that for this ability, $G = \emptyset$. Control actions in T^d are marked by ★.

pred. name	predicate	control actions	effects
$enabled$	$\neg run^c \wedge \neg run^m$	$run^c = T$★	$-$
$starting$	$run^c \wedge \neg run^m$	$-$	$run^m = T$
$executing$	$run^c \wedge run^m \wedge \neg tqr^m$	$-$	$tqr^m = T \vee fail^m = T$
$finished$	$run^c \wedge run^m \wedge tqr^m$	$run^c = F$	$-$
$failed$	$run^c \wedge run^m \wedge fail^m$	$run^c = F$	$-$
$resetting$	$\neg run^c \wedge run^m$	$-$	$run^m = F, tqr^m = F, fail^m = F$

To implement this logic, we have **learned** that it is as complex as implementing it in a distributed state machine, which we talked about in Sect. 3. However, since this model is directly verified and tested together with the complete system behavior, we directly find the errors and bugs. This is much harder in a distributed and asynchronous setup.

While abilities make for well isolated and reusable components from which larger systems can be assembled, they can mainly be used for manual or open loop con-

trol. The next step is therefore to model *interaction* between resources, for example between the robot and the nutrunner, or between the state of the bolts and the nutrunner.

4.5 Modeling Resource Interaction

Figure 3 illustrates two types of interaction between resources: "specification for safe device interaction" and "effects modeling of device interaction". The latter means modeling what can happen when two or more devices interact. As it might not be possible to *measure* these effects, many of them will be modeled as control actions updating estimated state variables.

Given a set of resources and generalized operations as defined by Definition 4, a complete control model is created by instantiating the needed operations into a *global* resource r_g ($r_g \notin R$). Additional estimated state added if needed in order to express the result of different resources interacting with each other. See Example 5.

Safety specifications are created to ensure that the resources can never do something "bad". The instantiated ability operations can be used together with a set of global specifications to formulate a supervisor synthesis problem from the variables and transitions in r_g. Using the method described in [23], the solution to this synthesis problem can be obtained as additional guard predicates on the deliberate transitions in T^d. Examples of this modeling technique can be found in [24, 25]. By keeping specifications as a part of the model, we **learned** that there are fewer points of change which makes for faster and less error-prone development compared to changing the guard expressions manually.

4.6 Planning Operations

While ability operations define low-level tasks that different devices can perform, planning operations model how to make the system do something *useful*. As the name suggests, planning operations define intent in the form goal states.

A planning operation has a preconditions which define when it makes sense to try to reach the goal, and a postcondition specifying a target, or goal, state. Planning operations also introduce an estimated state variable o^e to keep track of its own state, $o^e = \{initial, executing, finished\}$. When in the initial state, if the precondition is satisfied, the planning operation updates its estimated state variable to *executing* which will trigger this planning operation to be considered by the on-line planner. When the goal is reached, the operation's state variable transitions to *finished*, and the goal is removed from the on-line planner.

Example 5: The nutrunner: Specifying device interaction and planning operations

We would like to create a planning operation called `TightenBolt1`. When a bolt is in position, which is a precondition of `TightenBolt1`, the operation instructs the system to reach the goal where bolt1 has been tightened. This will happen when the nutrunner has reached the correct torque and the robot is in the correct position at the bolt.

The robot has a number of state variables and in this example, we are interested in its position, where the *measured* state variable $ur.pos^m = \{pos_1, \ldots, pos_n\}$ defines what named pose the robot can be in. In our system we would also like to know the state of bolt1, which is modeled with the *estimated* state variable $bp_1^e = \{empty, placed, tightened\}$.

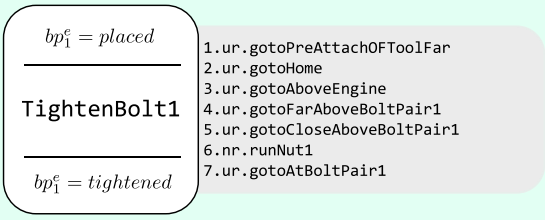
This state variable is not only used by one ability, but in multiple abilities and planning operations. If a variable is only used by planning operations, it is updated by their actions when starting or when reaching the goal, but in this case also other abilities need to know the state of the bolts.

For the planner to be able to find a sequence of abilities that can reach the goal, an ability needs to take the action $bp_1^e =$ tightened. The nut running ability or the robot can be extended with this extra action, but then special variants of abilities need to be made. If, for example, we extend the nut running ability, it needs to be duplicated for each bolt with just minor differences.

A better approach is to generate new special abilities to track these results. In this case, an ability is created only with the following transition:

pred. name	predicate	control actions	effects
–	$tqr^m \wedge ur.pos^m = bp_1$	$bp_1^e = tightened$	–

A natural way to model `TightenBolt1`, is to start in the pre-state defined by its precondition $bp_1^e = placed$ and end in the post-state defined by the postcondition $bp_1^e = tightened$. The figure below shows a planning operation, with a sequence of dynamically matched ability operations (to the right) to reach the goal state $bp_1^e = tightened$.



Modeling operations in this way does two things. First, it makes it possible to add and remove resources from the system more easily - as long as the desired goals can be reached, the upper layers of the control system do not need to be changed. Second, it makes it easier to model on a high level, eliminating the need to care about specific

sequences of abilities. As shown in Example 5, the operation `TightenBolt1` involves sequencing several abilities controlling the robot and the tool to achieve the goal state of the operation.

4.7 Execution Engine

The execution engine of the control system keeps track of the current state of all variables, planning operations, ability operations and two deliberation plans. The execution engine consists of two stages, the first evaluates and triggers planning operation transitions and the second evaluates and triggers ability operation transitions. When a transition is evaluated to true in the current state and is enabled, the transition is triggered and the state is updated. After the two steps have been executed, updated resource state is sent out as ROS2 messages.

Each stage includes both deliberation transitions and automatic transitions. The automatic transitions will always trigger when their guard predicate is true in the current state, but the deliberation transitions must also be enabled by the deliberation plan. The deliberation plan is a sequence of transitions defining the execution order of deliberation transitions. The plan for the planning operation stage is defined either by a manual sequence or by a planner or optimizer on a higher level. For the ability stage, the deliberation plan is continuously refined by an automated planner that finds the best sequence of abilities to reach a goal defined by the planning operations.

In this work, planning is done by finding counter-examples using bounded model checking (BMC) [26]. Today's SAT-based model checkers are very efficient in finding counter-examples, even for systems with hundreds of variables. By starting in the current state, the model of the system is unrolled to a SAT problem of increasing bound, essentially becoming an efficient breadth-first search. As such, a counter-example (or deliberation plan) is minimal in the number of transitions taken from the current state. Additionally, well-known and powerful specification languages like *Linear Temporal Logic* (LTL) [27] can be used to formulate constraints.

When decisions are taken by the SP execution engine instead of by the resources themselves, nodes essentially become shallow wrappers around the actual devices in the system. The fact that the core device behavior is modeled in SP allows for generation of a lot boilerplate code (and tests!) in the nodes, which is described in the next section.

5 Auto Generated ROS Nodes

The architecture of the system is based on the ROS hubs described in Sect. 2.1 and in [8], which allow us to separate the system into functional entities. Each ROS hub represents one SP resource which is exchanging messages based on the SP model of

the resource. Based on these models of resources, SP's component generator module generates a set of ROS2 nodes with accompanying messages during compile-time.

5.1 Auto Generated Code for Resource

Five nodes per resource hub are generated: *interfacer*, *emulator*, *simulator*, *driver* and *test*. The *interfacer* node serves as a standardized interface node between SP and nodes of the resource. *Emulator* nodes are generated based on abilities defined in SP and are used when testing the control system since they emulate the expected behavior of the ability. *Simulator* and *driver* nodes implement the actual device control and also an initial template to be used when implementing the connection with the real drivers in ROS. *Test* nodes implement automatic testing of *simulator* and *driver* nodes based on the SP model.

To use the abilities in formal planning algorithms and to support testing of the SP model without being connected to the real subsystem, *measured* variables must be updated by *someone*. We have chosen to place this updating as individual *emulation* nodes [28] rather than to let SP perform it internally. This allows us to always maintain the structure of the network which allows an easy transition to simulated or actual nodes at any time, while at the same time keeping the core execution engine uncluttered. The internal structure of the emulator node is quite simple and can be fully generated based on the SP model. This is because the internal state of an emulation does not have to reflect the internal state of the target which it is emulating, it only needs to mimic the observable behavior to match an existing target.

After the SP model has been tested with generated emulator nodes, the next step in the workflow is to use a variety of simulators instead of emulators connected in the ROS2 network via *simulator* nodes. These nodes are partially generated based on the model of the resource and partially manually assembled. The manually assembled part is decoupled from the main node and is imported into the *simulator* so that the node itself can be independently re-generated when the SP model changes.

These imports contain interface definitions for specific hardware and software. In the example of the nutrunner, the *driver* node is run on a Raspberry Pi and its manually written part contains mapping between variables and physical inputs and outputs.

Per resource, the *Simulator* node talks to SP over the *interfacer* node using same message types generated for all three nodes, *emulator*, *simulator* and *driver*. This way, the *interfacer* node does not care if the equipment is real, simulated or just emulated.

The driver nodes are in most cases exactly the same as simulator nodes, however we distinguish them by trying to be general. Both the simulator and the driver nodes use same imported manually assembled interfacing components.

Example 6: The nutrunner: model-based component generation

Given the nutrunner's "run nut" ability defined in Example 4, the component generator module generates one ROS2 package containing the different node types for that resource and another package containing the necessary message types:

```
├── nutrunner
│   ├── launch
│   ├── package.xml
│   ├── resource
│   │   └── nutrunner
│   ├── setup.cfg
│   ├── setup.py
│   ├── src
│   │   ├── __init__.py
│   │   ├── nutrunner_sp_driver.py
│   │   ├── nutrunner_sp_emulator.py
│   │   nutrunner_sp_interfacer.py
│   │   ├── nutrunner_sp_simulator.py
│   │   └── nutrunner_sp_test.py
│   └── test
│       ├── test_copyright.py
│       ├── test_flake8.py
│       └── test_pep257.py
├── nutrunner_msgs
│   ├── bridges
│   ├── CMakeLists.txt
│   ├── launch
│   ├── mapping_rules
│   │   ├── nutrunner_12.yaml
│   │   └── nutrunner_21.yaml
│   ├── msg
│   │   ├── NutrunnerDriverToInterfacer.msg
│   │   ├── NutrunnerInterfacerToDriver.msg
│   │   ├── NutrunnerInterfacerToSP.msg
│   │   └── NutrunnerSPToInterfacer.msg
│   └── package.xml
```

5.2 *Model Based Testing*

Unit-testing is a natural part of modern software development. In addition to the nodes generated from SP models, *tests* are also generated. Because the SP model clearly specifies the intended behavior of the nodes via *effects*, it is also possible to generate tests which can be used to determine if the model and the underlying driver implementation do in fact share the same behavior. The generated tests are run over the ROS2 network, using the same interfaces as the control system implementation. This enables *model based testing* using a seamless mixture of emulated, simulated, and real device drivers, which can be configured depending on what is tested. Based on what we **learned**, the generated unit tests can:

- ensure that the device drivers and simulated devices adhere to the behavior specified in the SP model.
- help eliminate "simple" programming errors that waste development resources, for example making sure that the correct topics and message types are used for user-written code.
- provide means to validate if the specifications in the SP model make sense. While formal methods can guarantee correctness w.r.t. some specification, writing the specification is still difficult and needs to be supported by continuous testing.

The tests generated by SP employ property-based testing using the Hypothesis library [29], which builds on ideas from the original QuickCheck [30] implementation. These tools generate random test vectors to see if they can break user-specified properties. If so, they automatically try to find a minimal length example that violates the given properties. The properties that need to hold in our case are that effects specified in the SP model always need to be fulfilled by the nodes implementing the resource drivers. Of course, when bombarded by arbitrary messages, it is not surprising to also find other errors (e.g. crashes).

Example 7: The nutrunner: automated testing during node development

Let's exemplify with the nutrunner resource again. The figure below shows the invocation of *pytest* for executing a generated unit test. The test will send arbitrary commands to the nutrunner node, via the same interface that SP uses for control, and check that the responses from the system match what is expected in the SP model.

```
$ pytest nutrunner_sp_tests.py
========================================= test session starts =========================================
platform linux -- Python 3.6.8, pytest-4.6.3, py-1.8.0, pluggy-0.12.0
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/martin/rosbook/dashing_ws/sr
c/nutrunner/src/.hypothesis/examples')
rootdir: /home/martin/rosbook/dashing_ws/src/nutrunner
plugins: hypothesis-4.24.3, repeat-0.8.0, rerunfailures-7.0, cov-2.5.1
collected 1 item

nutrunner_sp_tests.py F                                                                          [100%]

============================================== FAILURES ===============================================
_____________________________________ test_nutrunner_random _____________________________________

    @given(cmdMsg)
>   @settings(max_examples=50, deadline=None)
    def test_nutrunner_random(data):

nutrunner_sp_tests.py:26:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

data = {'seq': 0, 'set_run': False}

    @given(cmdMsg)
    @settings(max_examples=50, deadline=None)
    def test_nutrunner_random(data):
        [ ... code removed ... ]
E           AssertionError: Node did not produce desired effect within time limit
E           assert False

nutrunner_sp_tests.py:195: AssertionError
---------------------------------------------- Hypothesis ----------------------------------------------
Falsifying example: test_nutrunner_random(data={'seq': 0, 'set_run': False})
expecting effect: ['tqr_m = False']
Last message seen: nutrunner_msgs.msg.NutrunnerInterfacerToSP(got_set_run=False, run_m=False, tqr_m=True,
fail_m=False)
===================================== 1 failed in 14.36 seconds ======================================
$
```

The test fails, showing the generated test vector for which the effect failed to emerge, as well the message last observed on the ROS2 network. This makes it easier to spot problems which often arise due to the model and the actual device behavior differing. In this case, it turns out that the driver node does not properly reset tqr^m when run^c resets. Investigating how the driver node works yields:

```
msg.tqr_m = GPIO.input(self.gpi5) == 1
```

It can be seen from the snippet above that the driver node simply forwards what the lower-level device emits to some input pin. It turns out that for this particular nutrunner, the torque reached flag is reset only when the output used to set start running forward goes high.

Instead of introducing more state, and with that, complexity to the driver node, the proper fix for this is to go back and change the SP model to reflect the actual behavior of the node. By moving the torque reset effect from the *resetting* state to the *starting* state of the ability template definition and re-generating our package, the generated test now succeeds:

```
$ pytest nutrunner_sp_tests.py
========================================== test session starts ==========================================
platform linux -- Python 3.6.8, pytest-4.6.3, py-1.8.0, pluggy-0.12.0
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/martin/rosbook/dashing_ws/sr
c/nutrunner/src/.hypothesis/examples')
rootdir: /home/martin/rosbook/dashing_ws/src/nutrunner
plugins: hypothesis-4.24.3, repeat-0.8.0, rerunfailures-7.0, cov-2.5.1
collected 1 item

nutrunner_sp_tests.py .                                                                          [100%]

========================================== 1 passed in 35.07 seconds ==========================================
$
```

The nutrunner node should now be safe to include in the automation system.

The unit-test described in Example 7 deals only with testing one resource individually: it is not the instantiated abilities but the templates that are tested. This means that additional preconditions (e.g. handling zone booking) are not tested in this step. As such, while the real driver for the nutrunner could, and probably should be subjected to these kinds of tests, it is not safe to do with, for instance, a robot. Triggering random target states (e.g. robot movements) would be dangerous and such tests need to be run in a more controlled manner. Luckily ROS2 makes it simple to run the tests on a simulated robot by simply swapping which node runs.

It can also be interesting to generate tests from the complete SP model. Consider an ability `ur.moveTo` for moving the UR robot between predefined poses, that is instantiated for a number of different target poses, each with a different precondition describing in which configuration of the system the move is valid. If a simulated node triggers a simulated protective stop when collisions are detected (e.g. between collision objects in the MoveIt! framework), it becomes possible to test whether the SP model is correct w.r.t allowed moves (it would not reach the target if protective stop is triggered). In fact, such a test can be used to *find* all the moves that satisfy the stated property, which eases modeling.

5.3 Node Management and Integrated Testing

The first step in the engineering workflow would be testing the model with emulator nodes, iterating the model until the desired behavior is reached. Afterwards, a simulation or real hardware can be interfaced for one of the resources, while the others remain emulated. This way, targeted model properties can be tested.

To extend the usability of code generation, a node manager is generated to launch node groups for different testing and commissioning purposes, supporting testing of targeted model properties.

Testing the model using emulator nodes can be referred to as virtual commissioning (VC) [31]. The purpose of VC is to enable the control software, which controls

and coordinates different devices in a production station, to be tested and validated before physical commissioning.

The concept of integrated virtual preparation and commissioning (IVPC) [32] aims to integrate VC into the standard engineering workflow as a continuous support for automation engineers. Exactly this is gained with the described workflow of continuous model improvements, autogeneration of components and testing using ROS2 as a common platform that supports integration of different smart software and hardware components.

So, during the development, we have **learned**, that model-based code generation speeds up development and eliminates a lot of coding related errors since components are generated as a one-to-one match to the model. This enables for continuous model improvements during development based on behavior of the generated emulator node.

6 Conclusions

This chapter has presented the development of a industrial demonstrator and the control infrastructure Sequence Planner, together with some practical development guidelines and lessons learned. The demonstrator includes robots, machines, smart tools, human-machine interfaces, online path and task planners, vision systems, safety sensors, optimization algorithms, etc. Coordination and integration of all these functionality has required the well-defined communication interface provided by ROS2 as well as the robust task planning and control of Sequence Planner.

From a technical perspective, ROS2, is an excellent fit for industry 4.0 systems as shown in this chapter. However, being a well-structured and reliable communication architecture is just one part of the challenge. As we have shown, various design decisions will greatly influence the end result, for example in choosing how to communicate between nodes or in how the system behavior is modeled.

Since the presented control approach uses state based communication, the nodes are continuously sending out messages. This implies that care needs to be taken to avoid flooding the network. However, during work on the demonstrator, the bottleneck was more related to planning time rather than network capacity. Future work should include studying how many resources can be handled by SP, at what publishing frequency messages can be reliably received and how this is influenced by different QoS settings.

During the development and implementation of the industrial demonstrator, we learned that:

- To be able to restart the control system, it was important to avoid distributed control state, as the restart of a node almost always depends on the states of other nodes.
- To make the intelligent automation robust and functional, it was much easier to have a centralized control approach.

- It was easier to handle failures, troubleshoot, and maintain the system when using state-based commands.
- For the system to be flexible and robust, we had to stay away from hard-coded control sequences and if-then-else programming.
- Online automated planning and optimization is necessary for any type of intelligent automation.
- It is impossible to develop these types of systems without continuous testing and verification.

The major challenges during development were always related to implementation details and we learned the hard way that the devil is in the details. Developing any type of automation systems will never be a simple task, but it is possible to support the creative design process using algorithms. The control infrastructure Sequence Planner is an open source project that can be found here: http://github.com/sequenceplanner/sp.

Acknowledgements This work was funded by Volvo Groups Trucks Operations and the Swedish Innovation Agency Vinnova through the Production 2030 project Unification and the FFI project Unicorn.

References

1. F.P. Brooks Jr., No silver bullet: Essence and accidents of software engineering. Computer **20**, 10–19 (1987)
2. Volvo GTO Vision. https://www.engineering.com/PLMERP/ArticleID/18868/Vision-and-Practice-at-Volvo-Group-GTO-Industry-40-and-PLM-in-Global-Truck-Manufacturing.aspx. Accessed 14 Apr 2019
3. P. Tsarouchi, A.-S. Matthaiakis, S. Makris, G. Chryssolouris, On a human-robot collaboration in an assembly cell. Int. J. Comput. Integr. Manufact. **30**(6), 580–589 (2017)
4. Å. Fast-Berglund, F. Palmkvist, P. Nyqvist, S. Ekered, M. Åkerman, Evaluating cobots for final assembly, in *6th CIRP Conference on Assembly Technologies and Systems (CATS), Procedia CIRP*, vol. 44 (2016), pp. 175–180
5. V. Villani, F. Pini, F. Leali, C. Secchi, Survey on human-robot collaboration in industrial settings: safety, intuitive interfaces and applications. Mechatronics **55**, 248–266 (2018)
6. W. He, Z. Li, C.L.P. Chen, A survey of human-centered intelligent robots: issues and challenges. IEEE/CAA J. Autom. Sin. **4**(4), 602–609 (2017)
7. A. Hanna, K. Bengtsson, M. Dahl, E. Erős, P. Götvall, and M. Ekström, Industrial challenges when planning and preparing collaborative and intelligent automation systems for final assembly stations, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2019), pp. 400–406
8. E. Erős, M. Dahl, H. Atieh, K. Bengtsson, A ROS2 based communication architecture for control in collaborative and intelligent automation systems, in *Proceedings of 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2019)* (2019)
9. M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carreraa, N. Palomeras, N. Hurtós, M. Carrerasa, Rosplan: planning in the robot operating system, in *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'15* (AAAI Press, 2015), pp. 333–341

10. F. Rovida, M. Crosby, D. Holz, A.S. Polydoros, B. Großmann, R.P.A. Petrick, V. Krüger, *SkiROS—A Skill-Based Robot Control Platform on Top of ROS* (Springer International Publishing, Cham, 2017), pp. 121–160
11. E. Aertbeliën, J. De Schutter, ETASL/ETC: a constraint-based task specification language and robot controller using expression graphs, in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2014), pp. 1540–1546
12. C. Paxton, A. Hundt, F. Jonathan, K. Guerin, G.D. Hager, Costar: Instructing collaborative robots with behavior trees and vision, in *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017), pp. 564–571
13. P. Falkman, B. Lennartson, K. Andersson, Specification of production systems using PPN and sequential operation charts, in *2007 IEEE International Conference on Automation Science and Engineering* (2007), pp. 20–25
14. K. Bengtsson, B. Lennartson, C. Yuan, The origin of operations: interactions between the product and the manufacturing automation control system, in *13th IFAC Symposium on Information Control Problems in Manufacturing, IFAC Proceedings Volumes*, vol. 42, no. 4, (2009), pp. 40–45
15. K. Bengtsson, P. Bergagard, C. Thorstensson, B. Lennartson, K. Akesson, C. Yuan, S. Miremadi, P. Falkman, Sequence planning using multiple and coordinated sequences of operations. IEEE Trans. Autom. Sci. Eng. **9**, 308–319 (2012)
16. P. Bergagård, M. Fabian, Deadlock avoidance for multi-product manufacturing systems modeled as sequences of operations," in *2012 IEEE International Conference on Automation Science and Engineering: Green Automation Toward a Sustainable Society, CASE 2012, Seoul, 20–24 August 2012* (2012), pp. 515–520
17. P. Bergagård, *On restart of automated manufacturing systems*. Ph.D. Thesis at Chalmers University of Technology (2015)
18. N. Sundström, O. Wigström, P. Falkman, B. Lennartson, Optimization of operation sequences using constraint programming. IFAC Proc. Vol. **45**(6), 1580–1585 (2012)
19. S. Riazi, K. Bengtsson, R. Bischoff, A. Aurnhammer, O. Wigström, B. Lennartson, Energy and peak-power optimization of existing time-optimal robot trajectories, in *2016 IEEE International Conference on Automation Science and Engineering (CASE)* (2016)
20. A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, B. Lennartson, An event-driven manufacturing information system architecture for industry 4.0. Int. J. Product. Res. 1–15 (2016)
21. K. Bengtsson, E. Blomgren, O. Henriksson, L. Johansson, E. Lindelöf, M. Pettersson, Å. Söderlund, Emergency department overview - improving the dynamic capabilities using an event-driven information architecture, in *IEEE International Conference on Emerging technologies and factory automation (ETFA)* (2016)
22. M. Dahl, E. Erős, A. Hanna, K. Bengtsson, M. Fabian, P. Falkman, Control components for collaborative and intelligent automation systems, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 378–384 (2019)
23. S. Miremadi, B. Lennartson, K. Åkesson, A BDD-based approach for modeling plant and supervisor by extended finite automata. IEEE Trans. Control Syst. Technol. **20**(6), 1421–1435 (2012)
24. P. Bergagård, P. Falkman, M. Fabian, Modeling and automatic calculation of restart states for an industrial windscreen mounting station. IFAC-PapersOnLine **48**(3), 1030–1036 (2015)
25. M. Dahl, K. Bengtsson, M. Fabian, P. Falkman, Automatic modeling and simulation of robot program behavior in integrated virtual preparation and commissioning. Proc. Manufact. **11**, 284–291 (2017)
26. A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDS, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, 1999), pp. 193–207
27. A. Pnueli, The temporal logic of programs, in *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)* (IEEE, 1977), pp. 46–57

28. E. Erős, M. Dahl, A. Hanna, A. Albo, P. Falkman, K. Bengtsson, Integrated virtual commissioning of a ROS2-based collaborative and intelligent automation system, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2019), pp. 407–413
29. D.R. MacIver, Hypothesis 4.24 (2018), https://github.com/HypothesisWorks/hypothesis
30. K. Claessen, J. Hughes, Quickcheck: a lightweight tool for random testing of haskell programs, in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00* (ACM, New York, 2000), pp. 268–279
31. C.G. Lee, S.C. Park, Survey on the virtual commissioning of manufacturing systems. J. Comput. Des. Eng. **1**(3), 213–222 (2014)
32. M. Dahl, K. Bengtsson, P. Bergagård, M. Fabian, P. Falkman, Integrated virtual preparation and commissioning: supporting formal methods during automation systems development. IFAC-PapersOnLine **49**(12), 1939–1944 (2016). 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016

Endre Erős received his M.Sc. degree in Control Engineering in 2018 at the Faculty of Mechanical Engineering, University of Belgrade, Serbia. He is currently working as a Ph.D. student in the automation research group at Chalmers University of Technology, researching discrete control of complex systems. Endre is involved in developing, testing and integrating discrete control methods in two projects that aim to bring humans and robots closer together in working, as well as in living environments.

Martin Dahl received the M.Sc.Eng. degree in Computer Science and Engineering in 2015 from Chalmers University of Technology in Gothenburg, Sweden. He is currently working towards a Ph.D. degree in Electrical Engineering, also at Chalmers University of Technology. His research is currently focused on discrete control of complex systems.

Atieh Hanna received the M.Sc.Eng. degree in Electrical Engineering in 2001 from Chalmers University of Technology in Gothenburg, Sweden. From 2001 to 2011, she was working at Volvo Group Trucks Technology as a Development Engineer and from 2012 as a Research and Development Engineer at Volvo Group Trucks Operations within Digital and Flexible Manufacturing. She is currently working towards a Ph.D. degree and her research is focused on planning and preparation process of collaborative production systems.

Per-Lage Götvall received his B.Sc. from Chalmers in 1996 after an eight-year career in the Royal Swedish navy as a marine engineer. He joined the Volvo Group the same year and has worked in various roles at the company since then. In 2015 started a work aimed at understanding what was required for humans and robotic machines to work together. This work, as a manager for Volvo Group Trucks area of Transport system, robotics, was followed by a position as a Senior research engineer at Volvo Group Trucks Operations within the area of Flexible manufacturing, which is also the current position at the Volvo Group.

Petter Falkman received the Ph.D. degree in Electrical Engineering in 2005 from Chalmers University of Technology, Göteborg, Sweden. Dr. Falkman is currently Associate Professor at the Department of Signals and Systems, Chalmers University of Technology. He received his Master of Science in Automation and Mechatronics at Chalmers in 1999. Dr. Falkman is vice head of utilization in the electrical engineering department. Dr. Falkman is program director for the Automation and Mechatronics program at Chalmers. He is profile leader in the Chalmers production area of advance. He is also part of the Wingquist Laboratory, a research center for virtual product and production development at Chalmers. His main research topic concerns information integration, virtual preparation, machine learning, control, and optimization of automation systems.

Kristofer Bengtsson received the Ph.D. degree in Automation in 2012 from Chalmers University of Technology, Gothenburg, Sweden. From 2001 to 2005 he was with Advanced Flow Control AB developing automation control systems and user interfaces, and from 2005 to 2011 with Teamster AB, an automation firm in Gothenburg. From 2012 he is a researcher in the Automation research group at Chalmers and from 2011 he is also with Sekvensa AB, a research consulting firm. His current research interest includes control architectures based on ROS2 and new AI-based algorithms for complex automation systems as well as supporting healthcare providers with online prediction, planning and optimization.

ROS2 For ROS1 Users

George Stavrinos

Abstract This tutorial chapter takes a look at changes and features introduced in ROS2, from the ROS1 user perspective. For this chapter, the Melodic and Dashing ROS distributions are used, as the latest Long-Term-Support (*LTS*) releases for ROS1 and ROS1 respectively. Starting with how to configure a ROS2 environment and install new packages, ROS1 users should understand that the main rationale behind the newest version of the Robot Operating System stays the same. Continuing with topics, services and actions, terms widely known to ROS1 users, the reader is introduced to the newly added communication protocol in ROS2. The following sections highlight differences in command line tools, source code and compilation procedure, as well as workspace and package structures. The basic procedure on using ROS1 nodes on a ROS2 setup is presented. Additionally, all known simulation and visualisation tools are discussed. Finally, a quick reference chapter summarises some important information in a table, and links to the complementary git repository [6] of this chapter.

Keywords ROS2 · Tutorial · ROS1 Users · ROS Dashing · ROS Melodic

1 Introduction

ROS2 is being actively developed to introduce new features to the Robot Operating System community and provide a more robust robot control and planning framework. ROS2 has been in the foreground of the Robot Operating System community in the last couple of years. In 2020 the last official version of ROS1 will be released, which will be used as the last bridge to ROS2 migration [7].

With this time schedule in mind, it is clear that ROS2 is the future of ROS. This does not mean that ROS2 will completely replace ROS1. At least not for a minimum of 4 or 5 years. Knowing how to use ROS1 nodes combined with ROS2 nodes and

G. Stavrinos (✉)
National Centre of Scientific Research "Demokritos", Athens, Greece
e-mail: gstavrinos@iit.demokritos.gr
URL: http://gstavrinos.github.com

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_2

also understand the basic differences and advantages of ROS2 compared to ROS1, are vital skills for all ROS1 users that want to keep up with the interesting changes in the beloved Robot Operating System.

In the following sections we are going to discuss all the main differences between the two major versions of the Robot Operating System. For the comparison, we are going to use ROS *Melodic*, the latest LTS version of ROS1, and ROS *Dashing*, the latest LTS version of ROS2. In addition to the differences, examples are going to be presented for both C++ and Python nodes, as well as additional software used for simulation and visualisation.

2 ROS Environment Configuration

As stated earlier, the ROS versions used for this tutorial are ROS Melodic and ROS Dashing. Installing ROS2 follows exactly the same procedure as for ROS1, so this is not covered in this tutorial. The exact instructions including the Personal Package Archives (*PPA*s) can be found in the ROS2 Dashing wiki page [2]. ROS2 does not conflict with ROS1, so both versions can be installed and used on the same computer concurrently. It should be noted though that it is strongly recommended to have only one of the two environments (ROS1 or ROS2) enabled per shell.

Enabling both environments can be achieved by running the following commands:

- For ROS Melodic: *echo "source /opt/ros/melodic/setup.bash" >> /.bashrc*
- For ROS Dashing: *echo "source /opt/ros/dashing/setup.bash" >> /.bashrc*

3 Installing New Packages

ROS2 offers binary files for its packages, exactly like ROS1. Efforts have been made to maintain consistency in package installation procedures. For example, in order to install the standard messages (std_msgs) package, the equivalent commands for Melodic and Dashing are:

- For ROS Melodic: *sudo apt install ros-melodic-std-msgs*
- For ROS Dashing: *sudo apt install ros-dashing-std-msgs*

4 Topics, Services and Actions

There are three basic ways to transfer information between nodes in ROS1: topics, services and actions. All three of these types have been ported to ROS2, maintaining the same rationale ROS1 users are accustomed to. Although practically almost

nothing changes in the way they are used, fundamental changes have been made that ROS1 users need to know when using ROS2 communication mechanisms.

The typical ROS1 communication protocols are *User Datagram Protocol (UDP)* and *Transmission Control Protocol (TCP)*. ROS2 introduced the *Data Distribution Service (DDS)* protocol, that follows the publish-subscribe rationale. ROS2 as of ROS Dashing, supports various implementations of the DDS protocol, including eProsima Fast RTPS, RTI Connext and ADLINK Opensplice. It should be noted that as of ROS Dashing, the message and service definitions are exactly the same as in ROS1, with the *char* type mapping again to *uint8*.

The DDS integration in ROS2 offers a variety of important features, but the most important and most requested is **true real-time** data transmission support. Among other features, DDS enables nodes to:

- Run in **real-time**, as stated earlier. Since ROS is running on Linux based systems (kernels) real-time execution is not a trivial task to be achieved. By integrating the DDS connection protocol into the ROS2 Middleware API, users are allowed to prioritise their threads and nodes. This feature enables processes to be treated in a real-time manner, far from the kernel level of the Operating System. In more detail, the *Executor*, a special kind of process embedded into ROS2, takes care of all data transmissions and when specified by the user, prioritises certain processes based on time contraints.
- Have **life cycles**. Many ROS1 users would, at some point of their ROS experience, have to create a state machine that manages or reports the status of their nodes. With life cycles, each node can have a series of different states, with *"Active"* being the one that is allowed to have a priority for real-time capabilities. The rest of the states are mainly states that do not interfere with the rest of the system and are non real-time.
- Have **exclusive** access to topics. Topics can be encrypted to allow only for authorized nodes to access them.
- Comply to application-specific **Quality of Service (QoS)** constraints. Users apart from real time constraints, can also specify other constraints, like maximum waiting times for delays of data transmission, or callback execution. QoS policies can be applied to various entities of a ROS2 ecosystem separately.
- Easily **discover** publishers and subscribers, with a minimum resource overhead. There is no need for a ROS Master as in ROS1, eliminating its extra overhead.
- Be **tolerant to transmission delays**. As stated earlier, the QoS capabilities of the DDS protocol allow for additional system stability, fault tolerance and disaster confrontation.
- Have **backup/secondary nodes** in case of failure, that automatically interchange on demand. Modularity is one of the most important features of the DDS protocol. ROS2's capabilities are enhanced by DDS's node hot-swapping feature, from which even real time applications can be interchanged without tampering the rest of the system.

It is clear that ROS2 has a more robust and feature-rich communication protocol compared to ROS1. It enables various new implementations through its rich adaptability, enhances application fault-tolerance, node performance and consistency and most importantly, offers real-time capabilities.

5 Typically Used Commands and Tools

ROS1 offers complementary command line tools to, among other utilities, monitor, test and quickly communicate with nodes. In ROS1, the majority of the command line commands, have the following pattern:

ros[command] [parameter1] [parameter2].

ROS2 maintains this structure with a slight variation. ROS2 commands have the following structure:

ros2 [command] [parameter1] [parameter2].

It should be noted that there is a space character between *ros2* and the specific *[command]*, something that is not true for ROS1. This new structure not only ensures that the users knows which version of ROS is currently being used, but also avoids conflicts between ROS1 and ROS2 distributions, allowing them to be installed concurrently on the same machine.

The table below shows some examples of the most common commands used in ROS1 and their ROS2 equivalent:

ROS Melodic	**ROS Dashing**
rostopic list	ros2 topic list
rosservice list	ros2 service list
rosbag play [bag file]	ros2 bag [bag file]
rostopic echo [topic name]	ros2 topic echo [topic name]
rosrun [package name] [executable]	ros2 run [package name] [executable]
roslaunch [package name] [launch file]	ros2 launch [package name] [launch file]

For simplicity reasons, many common examples have been omitted, but it should be clear from the list above that all commands follow the same rationale, so translating a ROS1 command to ROS2 should be straight forward.

Apart from the built-in tools, ROS1 users often have to pass command line arguments to the rosrun command. In ROS2 the equivalent parameter passing is simplified with the introduction of a new YAML file parsing feature. With this feature, instead of specifying a value for each parameter, users can use the *__params:=[filename].yaml* extension to specify a configuration file. This leads us to the fact that ROS2 maintained the use of the YAML format for configuration files, which ROS1 should be very familiar with.

6 C++ Example Code

The default C++ version for ROS2, is C++14. The code below creates a simple "Hello world" publisher node on topic "ros2_check_string". The major difference between this and a ROS1 node is the transition from *roscpp* to *rclcpp*, which is the DDS version of ROS1 *roscpp*. The rest of the code follows the same rationale as with a ROS1 node, so it is not discussed in detail. To close this loop, we will create a subscriber node using Python in the next section.

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class CppPublisherExample : public rclcpp::Node{
public:
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr pub;


  CppPublisherExample(): Node("ros2_example_publisher"){
      pub = create_publisher<std_msgs::msg::String>
      ("ros2_check_string", 10);
  };
};

int main(int argc, char * argv[]){

  rclcpp::init(argc, argv);

  auto node = std::make_shared<CppPublisherExample>();

  while(rclcpp::ok()){
    std_msgs::msg::String msg = std_msgs::msg::String();
    msg.data = "Hello World!";
    node->pub->publish(msg);
  }

  rclcpp::shutdown();
  return 0;
}
```

7 Python Example Code

The default Python version for ROS2 is Python 3. The code below creates a simple subscriber node on topic "ros2_check_string". The publisher node was created using C++ in the previous section. There are quite a few differences from the equivalent ROS1 node, which are listed below:

– Transition from *rospy* to *rclpy*: This transition is Python's equivalent to the *rclcpp* (C++) implementation of the DDS communication protocol in ROS2.

– *spin*: Python's spin in ROS1 was the equivalent of *spinOnce* in C++. ROS2's version of Python's spin works exactly like ROS1's C++ equivalent (looping forever instead of once).
– Quality of Services (QoS) specifications: Subscribers in ROS2, are required to specify a Quality of Service profile. This profile can be created by the user based on their needs, or use one of the built-in profiles, like the one used in the example names *qos_profile_sensor_data*. Quality Of Service configurations and profiles are omitted from this chapter for the sake of reader's ease of use, since they would require their own separate chapter.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from rclpy.qos import qos_profile_sensor_data

class PythonSubscriber(Node):

    def __init__(self):
        super().__init__(''ros2_example_subscriber'')
        self.sub = self.create_subscription(String,
        ''ros2_check_string'',
        self.callback,
        qos_profile=qos_profile_sensor_data)

    def callback(self, msg):
        print(msg.data)


def main():
    rclpy.init()
    node = PythonSubscriber()
    rclpy.spin(node)

    node.destroy_node()
    rclpy.shutdown()


if __name__ == ''__main__'':
    main()
```

8 Launch Example File

A quite drastic (and needed) change in ROS2 is the launch file format. Launch files have switched from an XML format to a Python format. This has a variety of advantages, including controllable routines and conditional execution as well as monitoring of nodes. A simple example of a launch file is presented below. This example launch file highlights a variety of useful changes which are listed below:

– The *generate_launch_description* method: This method is a had requirement for all launch files and include the needed information for node execution.
– The *Node* class: Simple nodes, the ones known from ROS1 are of type *None*. These nodes are introduced inside the *generate_launch_description* method to specify which nodes we need to run. Other types of nodes are the *LifeCycleNodes* that apart from running like a normal node, can been programmed to work based on various states, as discussed in Sect. 4.
– Remappings: Remapping is a powerful tool in ROS1 and could not be missing from ROS2. In launch files remapping is specified during *Node* construction using an array of tuples. It should be noted that remapping also works in the command line using the := combination, thus exactly the same as in ROS1.
– Node execution order: One of the major problems in ROS1 is the non-deterministic order of the nodes specified for execution in launch files. This is solved in ROS2, since the order with which nodes are added to the *LaunchDescription*, is the order of execution.

```
import launch
import launch_ros.actions

def generate_launch_description():
    sub_node = launch_ros.actions.Node(
        package='ros2_example_package_py',
        node_executable='ros2_subscriber',
        output='screen',
        parameters=[''config.yaml''],
        remappings=[
        ('ros2_check_string', 'ros2_check_string_remapped')
        ])
    pub_node = launch_ros.actions.Node(
        package='ros2_example_package_cpp',
        node_executable='ros2_publisher',
        output='screen',
        remappings=[
        ('ros2_check_string', 'ros2_check_string_remapped')
        ])
    ld = launch.LaunchDescription()
    ld.add_action(sub_node)
    ld.add_action(pub_node)
    return ld
```

9 Creating and Compiling ROS2 Packages

ROS1 uses the *catkin* tool to create and compile ROS1 packages. ROS2 utilises the *colcon* tool, which is based on the *ament* toolset. The main difference is that in contrast with catkin, colcon does not include a command to create a new package. In ROS2, this can be achieved by running the

ros2 pkg create [package_name]

command. This command generates all the required template files (package.xml, CMakeLists.txt, etc) just like the

catkin_create_package [package_name]

command would. Keep in mind, that unlike ROS1, ROS2 requires C++ and Python source files to be included in separate packages, thus no hybrid packages are supported yet. In the case of a Python package, a setup.py file replaces the CMakeLists.txt file. A CMakeLists.txt file and a setup.py file are presented below, that are required for the source files presented earlier.

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.5)
project(ros2_example_package_cpp)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
    ''Clang'')
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rclcpp REQUIRED)

add_executable(ros2_publisher src/publisher.cpp)
ament_target_dependencies(ros2_publisher
  ''rclcpp''
  ''rcutils''
  ''std_msgs'')
install(TARGETS ros2_publisher
DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

From this CMakeLists.txt example, it is clear that not many changes have been introduced in the compilation of C++ packages. The only apparent difference is the introduction of pure C (non-C++) cmake flags.

setup.py:

```
from setuptools import setup

package_name = "ros2_example_package_py"

setup(
    name=package_name,
    version="0.0.1",
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=["setuptools"],
    zip_safe=True,
    author="George Stavrinos",
    author_email="stavrinosgeo@gmail.com",
    maintainer="George Stavrinos",
    maintainer_email="stavrinosgeo@gmail.com",
    keywords=["ROS", "ROS2"],
    classifiers=[
        "Intended Audience :: Developers",
        "License :: OSI Approved :: Apache Software License",
        "Programming Language :: Python",
        "Topic :: Software Development",
    ],
    description=(
        "Python node as an example for a minimal subscriber
        to a C++ publisher node."
    ),
    license="Apache License, Version 2.0",
    tests_require=["pytest"],
    entry_points={
        "console_scripts": [
            "ros2_subscriber =
            ros2_example_package_py.subscriber:main"
        ],
    },
)
```

The setup.py file is a simple way to specify information for the Python nodes of a package. Most of the information are very straight forward and include the name of the package, licenses, author and maintener information etc. What should be noted though, is the *entry_points* field. Here the names of the executables are specified, along with the driver (main) method method of the equivalent Python source file.

ROS2 and ROS1 workspaces follow the same structure and rationale. ROS1 users are accustomed to building their packages from the root directory of their workspace, and this remains true for ROS2. The equivalent of the ***catkin_make*** or ***catkin build*** commands for ROS2, is ***colcon build −−symlink-install***. By using the *−−symlink-install* flag, colcon ensures that the newly compiled package(s) will be discoverable, given that the ROS2 workspace has been sourced in the user's .bashrc file, with the same procedure as in ROS1. It should be noted, that the building method in ROS2 does not include the devel environment of ROS1, and thus installs all built packages in the same location. This caused the deprecation of the ROS1 *roscd* command, since users do not have to remember or look for the location of their built packages.

10 Combining ROS1 and ROS2 Nodes

Migrating whole sets and pipelines of nodes from ROS1 to ROS2 can be a quite unpleasant and time-consuming task. The ros1_bridge package [3], was developed to allow for communication between ROS1 and ROS2 nodes, without the need of source code migration on either side.

The seamless communication is achieved by creating a network bridge between the different networking protocols of the two ROS versions. Thus, the user only has to run a ROS2 node that is included in the ros1_bridge package, and all ROS topics will be visible to ROS2 nodes and vice versa. The command is the expected one for all ROS2 nodes:

ros2 run ros1_bridge dynamic_bridge

The installation of the package is thoroughly described in the documentation of the ros1_bridge repository [3], and is not covered in this section.

11 Complementary Software: Simulation and Visualisation

ROS2 utilises the same simulation and visualisation software as ROS1, but both of those tools have been ported to ROS2 to take advantage of all the newer version's features and capabilities. As of ROS Dashing, both Gazebo 9 [5] and rViz [4] are work-in-progress with some functionality already ported.

Additionally, MoveIt 2 [1], the MoveIt implementation for ROS2 is under development, with the first fully functional planning and visualisation capabilities already available.

12 Quick Reference

This section provides a reference table with some of the discussed differences between ROS1 and ROS2. For more examples, the complementaty github repository offers updated versions of the code presented in this chapter [6].

Category	ROS1	ROS2
C++ Version	C++03	C++14
Python Version	Python2.7	Python3
Launch file format	XML	Python
package.xml format	1 and 2	2
Compilation tool	catkin	colcon/ament
ROS C++ library	roscpp	rclcpp
ROS Python library	rospy	rclpy
Topic monitoring	rostopic	ros2 topic
Node monitoring	rosnode	ros2 node
Parameter monitoring	rosparam	ros2 param
Package monitoring	rospkg	ros2 pkg
Launching files	roslaunch	ros2 launch
Executing files	rosrun	ros2 run

13 Conclusion

In this chapter we presented all the basic differences and features of ROS2 compared to ROS1. A ROS1 perspective was used, in order to allow experienced ROS1 users to easily understand and migrate to ROS2.

Among the information provided, was minimal C++14 and Python3 example ROS2 nodes, launch files, ROS1 commands and their ROS2 equivalents and more. It has to be noted that a lot of information has been omitted from this chapter to allow for easier and quicker reading.

This chapter and especially the Quick Reference section is intended to be used as a cheatsheet for ROS1 users on their journey to learning ROS2.

References

1. R. Planning, MoveIt2 Dashing github repository (2019). https://github.com/ros-planning/moveit2
2. O. Robotics, ROS2 Dashing installation wiki (2019). https://index.ros.org/doc/ros2/Installation/Dashing/
3. ROS2. ros1_bridge repository (2019). https://github.com/ros2/ros1_bridge
4. ROS2. rVizGazebo github repository (2019). https://github.com/ros2/rviz
5. R. Simulation, Gazebo github repository (2019). https://github.com/ros-simulation/gazebo_ros_pkgs

6. G. Stavrinos, Tutorial Chapter github repository (2019). https://github.com/gstavrinos/ros2-for-ros1-users
7. D. Thomas, ROS1 Roadmap ROS Discourse (2019). https://discourse.ros.org/t/planning-future-ros-1-distribution-s/6538

George Stavrinos received a BSc degree in Computer Science from Athens University of Economics and Business in 2014 and a MSc in Intelligent Systems and Robotics from De Montfort University in 2018. He is currently pursuing his second MSc in Space Science, Technologies and Applications in the National Observatory of Athens. He is currently working as a researcher in RoboSKEL Lab in the National Centre of Scientific Research "Demokritos" in Athens, Greece. His main research interests are robots under zero-gravity (microgravity) conditions (rovers or humanoids), autonomous spacecrafts, flying robots (autonomous UAVs), navigation autonomy, dynamics and kinematics analysis and motion planning.

Multi-Robot Systems

mrgs: A Multi-Robot SLAM Framework for ROS with Efficient Information Sharing

Gonçalo S. Martins, David Portugal, and Rui P. Rocha

Abstract The exploration of unknown environments is a particularly and intuitively detachable problem, allowing the division of robotic teams into smaller groups, or even into individuals, which explore different areas in the environment. While exploring, the team can discretely reassemble and build a joint representation of the region. However, this approach gives rise to a new set of problems, such as communication, synchronization and information fusion. This work presents `mrgs`, an open source framework for Multi-Robot SLAM. We propose a solution that seeks to provide any system capable of performing single-robot SLAM with the ability to efficiently communicate with its teammates and to build a global representation of the environment based on the information it exchanges with its peers. The solution is validated through experiments conducted over real-world data and we analyze its performance in terms of scalability and communication efficiency.

Keywords SLAM · Multi-Robot · ROS · Open source software · Efficient information sharing

1 Introduction

Mapping can be a dangerous task: it may require the mapper to spend long periods of time in hazardous conditions, e.g. underwater, exposed to extreme temperatures, radiation or other deadly environments. Unlike humans, robots can be designed to resist harsh conditions, can be extremely precise in their measurements, can be made

G. S. Martins (✉)
Ingeniarius, Ltd., R. Coronel Veiga Simão, Edifício B CTCV, 3025-307 Coimbra, Portugal
e-mail: gondsm@ingeniarius.pt

D. Portugal · R. P. Rocha
Institute of Systems and Robotics, University of Coimbra, Pólo II, Coimbra, Portugal
e-mail: davidbsp@isr.uc.pt

R. P. Rocha
e-mail: rprocha@isr.uc.pt

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_3

to be very small and, in dangerous situations, are less valuable than a human life; robots are expendable and replaceable, whereas humans are most certainly not. Thus, robots appear to be a perfect choice for mapmakers.

The robotic mapping problem has been an active field of research for the last few decades, and several appropriate approaches exist already. When mapping indoor locations, in the absence of global references such as GPS (Global Positioning System), the mapping robot has no way of knowing where it is when it starts recording data. This fact constitutes a chicken-and-egg problem: we need a map to locate ourselves, and we need to locate ourselves in order to be able to create a map. These operations must be performed simultaneously, a problem usually known as SLAM: Simultaneous Localization And Mapping.

In the presence of a large, intricate environment, such as an abandoned factory, or a cave, humans tend to break up into small groups to try and maximize the gain of new information per unit of time on the structure and accessibility of this new environment. Various groups can later meet, or *rendezvous*, communicate to each other what they have learned from their experience and, together, create an approximate representation of the location. This behavior is extremely beneficial to the group, and is an efficient way of gathering information, as opposed to having a single person exploring, or the entire group together. Intuitively, then, a team of robots could have tremendous advantages over a single robot at mapping certain locations, particularly vast ones. This also constitutes an open field of research, and this problem is usually known as Multi-Robot SLAM. As with SLAM, this is a deeper challenge than meets the eye. For example, unlike single robot SLAM, Multi-Robot SLAM requires that the robots are able to communicate amongst themselves, so that they can coordinate their exploratory efforts. While exchanging information, efficiency is the key to scalability. In order to build a scalable multi-robot system, the agents need to be able to communicate in an efficient way, so that the addition of new members to the team does not cause failures in the network connecting them, i.e. that no information is lost or corrupted in transit.

Given our knowledge of previous work, our prime objective is to develop a 2D Multi-Robot SLAM approach. Our approach should be distributed, i.e. must not depend on a centralized processing unit. In this sense, we will make use of preexisting software tools to enable independent robots to communicate through a wireless network. Our technique must also be robust to failures in communication, i.e. the mission should not be compromised by the corruption or loss of a message, for instance due to a link failure or robot withdrawal. Our approach should be scalable, in the sense that adding robots to the mission should not compromise their performance, within the limits of reason. This should be achieved both by making robots communicate as efficiently as possible, and by carefully planning the execution of our software. Finally, our approach should be SLAM-technique-agnostic, i.e. able to work regardless of the SLAM technique employed. Hence, the main contribution of this work is a 2D Multi-Robot SLAM framework for ROS, referred to as `mrgs`, which provides an easy-to-use collection of packages, and an open source implementation that allows the inclusion, swapping and testing of different modules for Multi-Robot SLAM.

This chapter is organized as follows: we start by overviewing the state of the art on Multi-Robot SLAM and Information sharing, in Sect. 2; in Sect. 3, we present and describe the system developed in ROS for performing Multi-Robot SLAM, and provide instruction to run the system on ROS, in Sect. 4. We then validate the system experimentally in Sect. 5. Finally, in Sect. 6, we reflect upon the advantages and handicaps of our system, as well as on the completion of our objectives and on possible future work.

2 Background

2.1 Single Robot SLAM

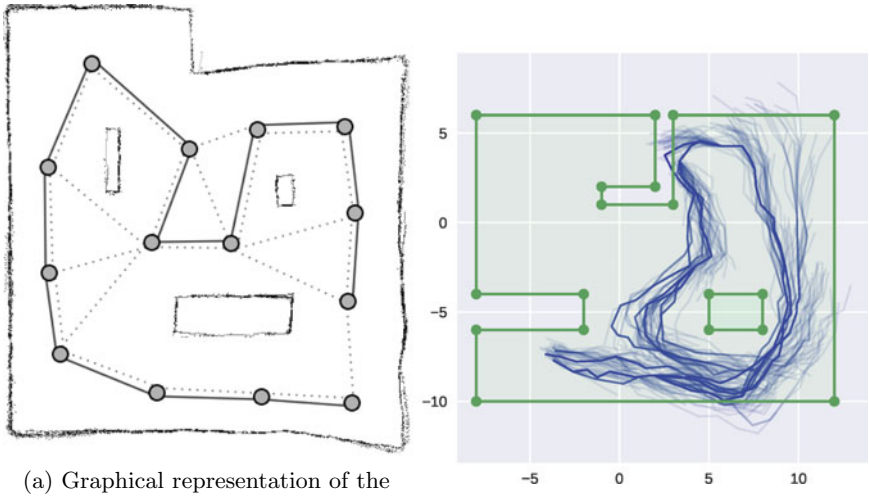
SLAM stands for Simultaneous Localization and Mapping. Essentially, this means that a robot performing SLAM is tasked with both creating a map of the environment and localizing itself in it.

Using its sensors, which usually include range and odometric sensors, the robot gathers data as it explores the world. Since all data gathered through sensors is plagued by noise and uncertainty, probabilistic solutions to the SLAM problem prevail over mathematically simpler approaches [1]. Odometry, in particular, produces errors that accumulate over time. In fact, with no assistance from other sensors, odometric errors can grow arbitrarily large [2].

As data is received, the robot must be able to relate it to the previous data it has been gathering, i.e., the data must be *aligned*. This is usually known as the *correspondence problem*, and solving it is usually accomplished by means of *feature matching*: the robot extracts a number of features from every scan and tries to match them with features extracted from previous scans. This process is of the utmost importance during *loop closure*, which consists of the algorithm's ability to recognize a loop in the environment, i.e. to recognize the fact that the robot has already visited a certain location, and to take that fact into account in its calculations. The inability to recognize a loop in the environment may lead to an erroneous, unintelligible map.

Classical solutions to the SLAM problem include Filtering techniques based on Bayes Filters [1], such as the Kalman Filter (KF), presented for the first time in [3], which uses Gaussian distributions for the posterior probability distribution; the Extended Kalman Filter, which is based on the linearization of the inherent non-linearities of both the motion and the sensor models [1], and Particle Filters (PFs), which are recursive Bayes Filters that estimate the posterior of poses conditioned by gathered data representing it by a set of weighted samples, also known as particles, instead of a function or a set of parameters [4].

Recently, Graph-based SLAM techniques became very popular [5]. Unlike filtering techniques, in which data is discarded after being processed and incorporated into the filter's state, Graph-Based SLAM techniques save all gathered data, and a full notion of uncertainty, in the form of a graph, keeping a complete history of past poses



(a) Graphical representation of the constraint network used by Graph-SLAM. There can be various links from each node, and multiple links connecting any two nodes.

(b) Trajectories for all particles involved in a RBPF SLAM approach [11].

Fig. 1 Graph-based SLAM and Particle Filter SLAM concept

(see Fig. 1a). Graph-based SLAM performs a global optimization over all poses, thus growth in computation power, and techniques such as variable elimination [6] are key to allow the use of the technique in real time.

Graph Optimizers receive as input all the nodes and edges of the graph [7, 8], and return the optimized poses as output. Implementations of graph optimizers include the earlier TORO [7, 9]; SPA [10], an improved version of the algorithm originally proposed by Lu and Milios; and g2o [8], an open-source C++ framework for general graph optimization.

2.2 SLAM with Multiple Robots

Multi-Robot Systems (MRS) have a number of advantages over single-robot solutions, such as parallelism, distribution in space and time, problem decomposition, reliability, robustness and redundancy. Thus, they are very useful for monotonous, repetitive, complex, dangerous, large-scale and dividable problems [12]. Multi-Robot SLAM is a natural extension to the original SLAM problem: if we are able to map a 2D environment using a single mobile robot, why not apply this powerful concept and use multiple robots, so as to achieve our goal in a faster, more reliable and robust fashion?

Despite their many advantages, the usage of multi-robot systems gives way to the rise of multiple new problems. First and foremost, coordination is fundamental and, thus, inter-robot communication becomes a capital factor in the team's performance. Furthermore, in regards to SLAM, one of the biggest challenges is that of combining the information gathered by multiple robots.

2.2.1 Communicating Data

Communication is an essential part of cooperation, as robots have the need to share information in order to perform their cooperative tasks. Robots can either communicate using an existing communication infrastructure, e.g. a WiFi network, or a mobile, self-configuring, infrastructure-less dynamic network, namely Mobile Ad Hoc Networks (MANETs).

Keeping track of which robot has which pieces of information can be a very important step in reducing data traffic between robots. For instance, if two robots meet a significant time after deployment, transferring the entire data they have gathered so far would probably cause a considerable delay before they could resume their tasks. However, if the robots can inform each other of how much data they need from each other, and if they have met a few times before, the total amount of data they need to transfer can be dramatically reduced. This issue is discussed in [13], where the concept of rendezvous is used. Briefly, there is a successful rendezvous for time T if all robots have all the relevant data up to time T. This is an important concept, not only in cooperative robotic mapping, but in multi-robot systems in general.

2.2.2 Information Fusion

Information fusing is one of the greatest challenges related to Multi-Robot SLAM. After a successful exchange of data, robots need to combine their local information with the received one into a single, consistent representation of the environment. There are various solutions to this problem. In this section, we review and discuss some of them.

Map fusion, i.e. stitching together the various contributions into a combined representation, can take place on one of two levels, either by merging data such as poses, landmarks, graphs, etc. [14–17], or by merging the rendered occupancy grids themselves [18].

In [14], map fusion is assisted by what is called *rendezvous measurements*. While exploring, robots often pass by each other, going in the same or different directions. When this occurs, it is possible to measure the relative pose between them using robot-to-robot methods, such as the visual detection system described in [19], in which virtual landmarks mounted on the robots are detected by cameras also mounted on the robots. These measurements are then used to estimate the rotation and translation needed to transform one robot's local coordinates into the other's. Once this is determined, merging local maps is a trivial matter of matching features

present in both maps and building the combined map. This method suffers from the need of precision measurements of relative poses, which is not always possible, making this technique unfit for our particular case.

The method introduced in [15] approaches the problem in a different manner: robots carry a camera, which creates a stream of images as the robot explores. From these images, features can be extracted such that each location on the map can be identified in an unambiguous way. Thus, each time a robot needs to merge two maps, it can search for locations that exist in both maps, and thus extract translational information that relates them. This technique, however, requires that model images exist previously, taken in ideal conditions, which will be used by robots in location matching.

Fox et al. proposed, in [16], a complete method for Graph-Based SLAM with multiple robots, which merges maps on the graph-level. As robots explore the environment, Graph-Based SLAM dictates that constraints be created between each pose at which a range scan is taken. The authors introduce a new type of constraint, which is derived by matching poses between various local maps. It is also important to note that, unlike the classical graph-based approach, this technique uses a local representation for pose relations. This way, pose relations are independent of the world coordinate system and, thus, invariant with rigid transformations (such as rotation and translation). This technique, in its proposed form, does not linearize the constraints, which makes it impossible to solve the optimization problem through classical methods. However, the authors propose alternative methods which, given a "close enough" estimate, can reportedly solve relatively large problems in under a second. This technique is able to achieve remarkable precision, even surpassing manual methods of measurement in this field.

In [18], an image-based method is proposed as the solution to the map alignment and merging problem. This approach operates on occupancy grids, commonly produced by SLAM implementations. The Hough Transform is an established method for detecting lines and other parametric-natured forms, such as circles or ellipses. The algorithm detailed in [18] uses the Hough Spectrum, as described in [20] for use in scan matching, to conduct its spectra-based determination of rotations. The output of this method is a set of candidate transformations. Since multiple solutions are to be expected, a metric named *acceptance index* is proposed to provide the solution which better fits the available data.

2.2.3 Multi-Robot SLAM

In [21], one of the first attempts at a Multi-Robot SLAM system is described. This technique is a generalization of the Rao-Blackwellized Particle Filter (RBPF) technique to the Multi-Robot SLAM problem. Although not clearly stated, this approach assumes that only one instance of the filter is running at any given time, and that data from all robots is processed in a centralized fashion. Initially, it is assumed that initial poses are known, either by deploying all robots from the same place or by externally monitoring the mapping process. The algorithm is, then, generalized for

the case where initial poses are unknown. This algorithm does not explicitly use a map alignment/merging technique, rather merging data at the landmark and pose level. This merging is executed when robots "bump into" each other, i.e. when they find each other and measure their relative pose. This issue is discussed in [19], where the mutual detection and relative pose estimation problems are further explored.

The authors of [22] describe an interesting technique, also based on Rao-Blackwellized Particle Filters, which assumes each robot as an isolated entity, i.e. unlike in [21], robots take the steps of performing SLAM in an isolated manner, and occasionally and discretely exchange and merge information. As before, merging occurs at the data level (as opposed to merging rendered maps), and relative poses are found using pan-tilt cameras. When two robots rendezvous, they exchange all data, transform it according to the measured relative pose, and integrate it into their respective filters.

In [23], the authors present an approach to Multi-Robot Graph-Based SLAM. This technique is based on condensed graphs and on a novel efficient communication model. The authors assume that all communication is conducted peer to peer, robot to robot. Local maps are transmitted under the form of a single range scan, the latest acquired, and a set of up-to-date, adjusted poses. Robots perform this operation at each step, i.e. at every new node. The transmission of up-to-date poses at each step enables each robot to provide its peers with the best estimate it can generate of its past poses. Additionally, by transmitting all range scans, one at a time, the authors ensure that all communicating robots have knowledge of each other's local maps, and that communication is as reliable as possible. Map fusion is based on keeping, on each robot, a list of candidate edges between the robot's graph and each teammate's graph. These edges are found by employing a correlative scan matching technique, which consists of matching scans originating from different robots, in a way similar to the process mentioned in [16]. Each of these new edges suggest a transformation between graphs. A RANSAC (RANdom SAmple Consensus) technique is used to determine a consensus among these transformations. The edges that are considered inliers by the RANSAC method are then communicated to the robot with which the map is currently being aligned, which then replies with a series of nodes, a condensed graph containing only the nodes affected by these edges. These new nodes are, then, added to the first robot's graph, resulting in a higher consistency of the map in the overlapping area and in the knowledge of relative poses in that area, which allows for the merging of the two maps.

Recently, cloud-based approaches have also become popular for scalable and real-time Multi-Robot SLAM. In [24], the authors distribute the SLAM process to multiple computing hosts in a cluster, which enables 3D map building in parallel. They promote switchable messaging patterns to meet different transmission scenarios and eliminate the bottleneck from data sharing. Another cloud-based strategy is described in [25], where an efficient architecture is proposed to parallelize complex steps in Multi-Robot SLAM, via cloud computing nodes to free the robots from all the computational efforts. The system also decides between two alignment methods (data association and inter-robot observations), which is more appropriate for computing the coordinate transformation between the robot reference frames.

In this work we propose a technique-agnostic multi-robot SLAM package, `mrgs`. Our technique differs from other multi-robot SLAM packages by providing a framework that does not depend on the underlying SLAM technique, thus outperforming the state of the art in versatility. Unlike state-of-the-art techniques, `mrgs` relies on image-based techniques [18] to fuse maps, thus being able to seamlessly operate above single-robot techniques.

2.3 ROS: Robot Operating System

Algorithms and approaches must exist as compilable code so that they can be experimentally validated. In order for an approach to be effortlessly and effectively testable by the robotics community, a common software framework must be established, with the goal of reducing to a minimum the time it takes to prepare a solution to be tested. ROS, standing for Robot Operating System [26], achieves just that. It is a software framework that aims to ease the development of modular code targeting robots, and a very widespread tool in the field of robotics.

ROS establishes a communication layer running on top of an ordinary host operating system, which allows for the exchange of messages between multiple ROS nodes. ROS nodes are programs which use the facilities provided by ROS to communicate and perform their tasks. ROS nodes operate independently and concurrently, and need not even be running on the same computer.

Communication is achieved through two main mechanisms: topics and services, which both carry messages. Topics are a means of asynchronous communication: a node publishes messages in a topic, to which other nodes may or may not have subscribed. Once a node publishes a message on a topic, all nodes that have subscribed to that topic receive that message. Services, on the other hand, provide synchronous communication between two nodes, and require that both a request and a response message be transmitted between the communicating nodes in order to be successful.

ROS nodes can be implementations of all kinds of functions: data management, mathematical functions, or anything else that can be implemented in any of the languages supported by ROS. Hardware drivers are a prime example of just how powerful ROS's modularity is: for a given robot, it is possible to develop ROS nodes which subscribe to a set of standard topics to receive commands, and that implement the low-level code needed to relay those commands to the robot. Thus, it is possible to directly use code developed for other robots to, for example, implement a given exploration algorithm on our robot. ROS allows us to abstract away the hardware intricacies of many robots and to develop as if we were writing code that targeted a standardized robot. ROS promotes code reutilization and has become a de facto standard in Robotics.

2.4 ROS Packages for SLAM

ROS software is distributed in packages, which generally contain code, messages types and other support files. We can find several ROS packages that implement SLAM algorithms.

SLAM packages in ROS usually deal with two kinds of data: range scans and odometry. In other words, the ROS node in charge of the mapping process subscribes to topics which are published by the node(s) in charge of relaying information from the robot's sensors. Given this data, the SLAM node then creates a map, usually in occupancy grid form, and publishes it into a dedicated topic, thus transmitting it to any subscribing nodes. The SLAM node is also responsible for determining the robot's position in the map it created, thus constituting a solution to the complete SLAM problem. Popular SLAM packages include GMapping[1] [27], Karto[2] and Hector[3] [28], which are compared in [29]; as well as the recent Cartographer graph-based SLAM approach implemented by a Google research team[4] [30].

The roscore,[5] a collection of nodes and programs that are pre-requisites to any ROS-based system, including the ROS Master, is a critical part of the ROS framework, and is responsible of handling the communication between nodes. Each ROS system uses a single *roscore* to overview all operation, and it is of the utmost importance that no node ever lose the ability to communicate with it. However, we envisage a system in which robots are able to cooperate but also capable of working alone if communication becomes impossible. We need, then, multiple ROS systems, one for each robot, each with its own *roscore*, in order to have a scalable, fault-tolerant system.

Communication between multiple roscores is not supported by ROS out-of-the-box: ROS systems always assume the existence of a single core that manages all communication between nodes. There is, however, interest in multimaster systems[6] [31].

In the context of this work, a workaround based on the *wifi_comm*[7] package is used. This package propagates messages between various independent ROS systems, by mirroring a topic on a remote machine: any messages published to that topic on the sending robot will be broadcast, under the same topic, on the receiving robot(s). This allows for the exchange of arbitrary data between different ROS networks and, thus, the implementation of Multi-Robot SLAM systems in conditions in which communication is not reliable: by isolating a ROS system in each robot, lack of communication with the rest of the network does not implicate a lack of communication with the *roscore*. The *wifi_comm* package can be used with a multiplicity

[1] http://wiki.ros.org/gmapping.
[2] http://wiki.ros.org/karto.
[3] http://wiki.ros.org/hector_mapping.
[4] http://wiki.ros.org/cartographer.
[5] http://wiki.ros.org/roscore.
[6] http://wiki.ros.org/sig/Multimaster.
[7] http://wiki.ros.org/wifi_comm.

of routing algorithms. By default, it uses the Optimized Link State Routing (OLSR) protocol [32].

2.5 Efficient Communication in Multi-Robot Systems

Cooperation among mobile robots usually involves the use of a wireless network. Commonly, this network is taken for granted and little care is taken in minimizing the amount of data that flows through it, namely to assist the robot's exploration through the environment.

However, in several specific scenarios, such as search and rescue operations, constrained connectivity can become an issue, and caution must be taken to avoid overloading the network. Additionally, in real-world applications, the exploration effort can be but a small part of the tasks that must be dealt with by a complete robotic system [33]. An efficient model of communication is also a key element of a scalable implementation: as the number of robots sharing the network increases, the amount of data that needs to be communicated does as well. Thus, greater care in preparing data for transmission is needed, so as to avoid burdening the network by transmitting redundant or unnecessary data.

Previous works, such as [23, 34, 35], have worked on a solution for efficient inter-robot communication by creating new models of communication for robotic teams, i.e. by developing new ways of representing the data needed to accomplish the mission. Other research efforts focused on developing information utility metrics, e.g. by using information theory [36], which the robot can use to avoid transmitting information with a utility measure below a certain threshold. These techniques, while successful in their intended purpose, rely on modifications to the inner workings of their respective approaches. In our case, we intend to create a generalized optimization solution, i.e. that does not depend on modifications to the intricacies of the underlying techniques.

General-purpose data compression techniques, on the other hand, are able to deal with any kind of data, with varying degree of success. Compression methods are widely used in the transmission and storage of bulky data, such as large numbers of small files, logs, sound and video, and their main objective is to represent data in as few bytes as possible, regardless of its contents. These offer us the solution to our problem: a way of achieving efficient communication without having to rework the SLAM technique's basic functionality, so that we can build our approach to be as general as possible.

In our particular case, we intend to use a fast technique that efficiently utilizes resources, as long as it demonstrates an acceptable compression ratio. In a previous study, we have conducted a comparison of general-purpose FOSS compression techniques for the optimization of inter-robot communication [37], and we concluded that the best technique in terms of temporal efficiency is LZ4,[8] making it very suitable

[8] Available at https://github.com/lz4/lz4.

for real-time missions. Additionally, the compression ratio it achieves is appropriate for our purposes; as it leads to a minimum of 10x reduction in necessary bandwidth for map transmissions. Thus, we selected LZ4 as the most suitable technique for use in our Multi-Robot SLAM approach.

In the next section, we describe in detail the proposed `mrgs` system for Multi-Robot 2D SLAM.

3 `mrgs`: A ROS-Based Multi-Robot SLAM Solution

The Multi-Robot SLAM system implemented, `mrgs`,[9] is completely modular and its main functionality is divided into five components, distributed into four ROS packages, forming the main contribution of our work, as summarized in Fig. 2.

The system can be run on top of any other ROS system that performs SLAM, as long as it conforms to ROS's standards, as illustrated in Fig. 3. In other words, this system can quickly enable any system running SLAM to run Multi-Robot SLAM.

The underlying individual SLAM technique benefits from laser scan readings acquired by robots, and possibly odometry estimation and/or other multimodal sources to assist the robot's local positioning (see for instance [29] for a list of

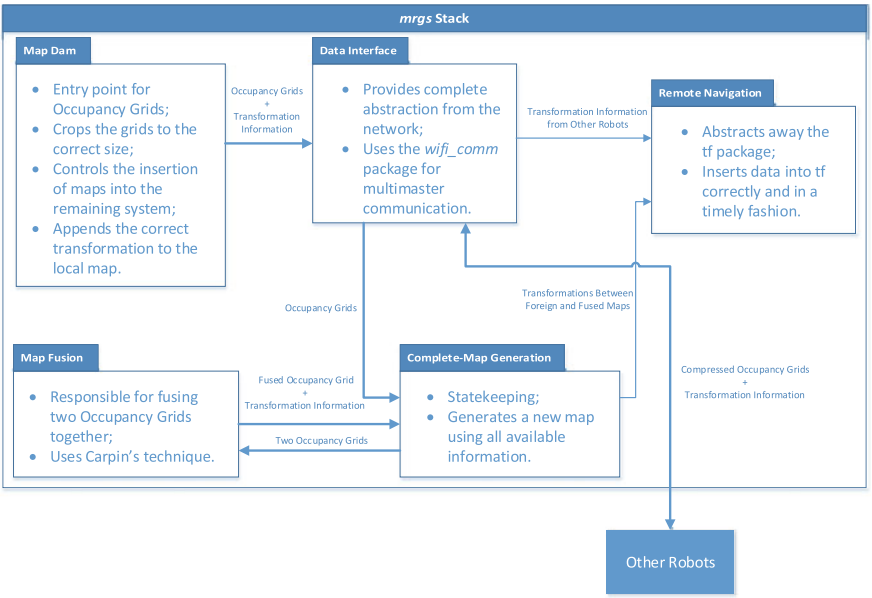


Fig. 2 An overview of our system's design and data flow. This system is replicated in each of the team members

[9]The `mrgs` framework is openly available at https://github.com/gondsm/mrgs.

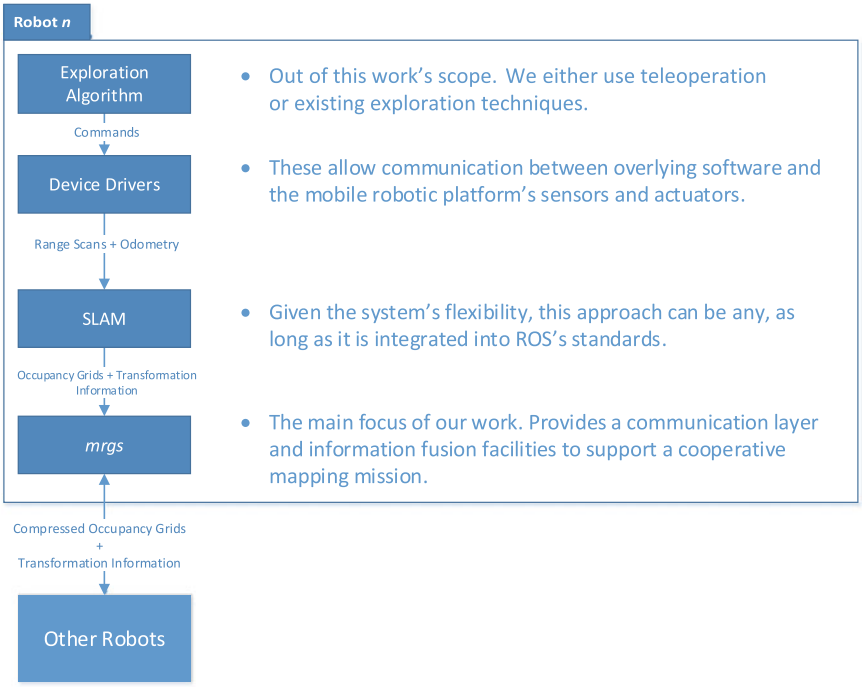


Fig. 3 A general overview of the complete system that runs on each robot. The focus of our work is on the `mrgs` block, a ROS metapackage containing our framework for Multi-Robot SLAM. `mrgs` enables robots to communicate with its peers and to build a global representation of the environment

popular SLAM techniques available in ROS), and provides as output an estimation of the robot's pose, as well as a local occupancy grid.

The local occupancy grids enter the system one by one via the `Map Dam` node. This node crops out any excess empty space, attaches to the maps the local robot's local pose within the map, generated by the SLAM technique, under the form of a transformation between reference frames, and sends the newly-received map to the `Data Interface` node. These are then prepared to be sent to the team members, first by being compressed and then by attaching to them the local robot's MAC address. Finally, they are sent into the network.

As maps are being sent, the `Data Interface` node is also receiving grids sent from other robots, which are running the same process. These are all, both the local and the foreign grids, packed into a vector and sent to the `Complete-Map Generation` node. They are then iteratively fused into a single representation of the environment at the `Complete-Map Generation` node, which stores the fused occupancy grids into a tree-like data structure. The fusion itself takes place at the Map Fusion node, which provides a service for fusing occupancy grids in pairs.

The system does not impose a limit to the team size, nor does it depend on a priori knowledge of the team's composition. Both the `Data Interface` and

the `Complete-Map Generation` nodes are able to deal with any number of teammates and maps, respectively.

This framework also includes a fifth, support package, whose purpose is to hold various scripts and configuration files that, while an important part of the system in the practical sense, are not vital to its operation, and are not the result of a significant research effort; they are solutions for minor problems related to the solution's implementation.

3.1 Modes of Operation

While the system's requirements only state that it should be able to operate in a distributed fashion, this implementation supports a total of three modes of operation by dividing the team members into three different classes: distributed robots, central robots and mapping robots. All classes are able to collaborate with one another, which makes this approach extremely flexible.

The distributed mode of operation initially required is achieved by populating the team solely with robots running in the corresponding mode. These execute the full framework, as they gather, propagate and fuse data. This is the most common use case: we use a homogeneous team of robots to explore an unknown environment.

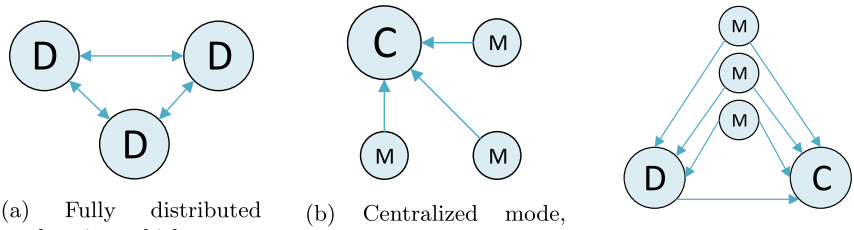
Mapping robots are the simplest of the three, they run a SLAM technique and use the Map Dam and Data Interface nodes to propagate the maps they have build. Central nodes, on the other hand, are assumed to be computationally powerful; they run the full framework, except for the `Map Dam` node, do not produce their own maps, and are solely tasked with building a global map based on information gathered by the mapping robots. Combining these two classes of robots produces the centralized mode of operation, which can be useful if the network we are using is reliable, fast and widespread. In this case, the central node may even run on a base station, e.g. a powerful desktop computer.

Finally, a third mode of operation can be achieved by combining the use of robots running in distributed mode with robots running in mapping mode. This mode combines the simplicity of the mapping robots with the versatility of the distributed ones, so that all robots in the field produce their own local map. Optionally, this mode of operation can employ central nodes, for added redundancy in the data fusion process. All three modes of operation are illustrated in Fig. 4.

3.2 Design and Implementation Principles

This framework was designed and built with efficiency, scalability and maintainability in mind. As such, all software strictly follows ROS's C++ guidelines,[10] and

[10]The guidelines are available at http://wiki.ros.org/CppStyleGuide.



(a) Fully distributed mode, in which every robot gathers and fuses data.

(b) Centralized mode, where mapping robots gather data and central robots fuse it.

(c) Mixed mode, which can employ up to three classes of robots.

Fig. 4 An illustration of the three modes of operation supported. "D" nodes are distributed nodes, "C" nodes are central nodes and "M" nodes are mapper nodes. The blue arrows represent the flow of local maps

Software Engineering good practices. The system was purposefully segmented into concurrently-executing modules so as to explore the flexibility of multimaster systems, as well as, for example, guaranteeing that the robot remains able to communicate during the map fusion process. Additionally, having a well-segmented system, i.e. not divided into too little or too many modules, maximizes the utility of the code by making it reusable.

Aside from inter-robot communication, all inter-module communication relies as much as possible solely on standard ROS data structures, in order to maximize compatibility with other systems and code reusability. Even custom communication structures, such as the messages exchanged between the `Data Interface` and the `Complete-Map Generation` nodes, have their roots in ROS's standard data types, ensuring that reusing specific nodes from this system within other ROS systems is a fairly straightforward process. All ROS topics used solely by our system are packed into their own *namespace*, to enable the user to quickly identify which topics belong to our system, as well as minimize possible interference with other systems running on the machine. In our implementation, code reuse is restricted to the usage of the reference implementation of Stefano Carpin's `mapmerge` [18], all remaining code is completely original.

3.3 Multi-Robot SLAM System Components

In this section, we provide details about the system components developed in this work, namely the `Data Interface`, `Map Fusion`, `Complete-Map Generation` and other auxiliary ROS nodes.

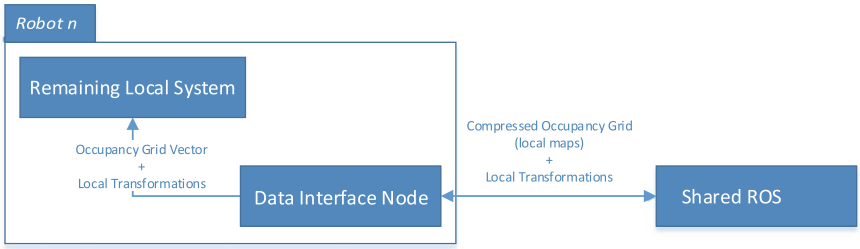


Fig. 5 The Data Interface node publishes into and subscribes to a topic that is shared among all robots. All information published into this topic is broadcast to all known robots in range in order to propagate the latest local map and transformation information

3.3.1 The Data Interface Node

The `Data Interface` node for ROS, existing in its own package, is responsible for dealing with one of the biggest challenges we have proposed to tackle: it is responsible for communicating all the relevant data between robots, and for doing so in an efficient manner. This node's operation is summarized in Fig. 5.

In the proposed approach, we have decided not to rely on explicit *rendezvous* events, since actively seeking a *rendezvous* with another robot is a complex and costly operation [13]. Instead, data is published into the shared topic, and received by all robots within range. This approach greatly simplifies the transmission of information: robots do not need to trade control messages, instead relying on the capabilities of the multimaster communication approach of *wifi_comm* to deliver all data.

One of our goals related to communication was that it be relatively robust to network failure. To partially solve this issue, this node uses MAC addresses to identify particular robots. MAC addresses are a better identification key than IP address, since they are not repeatable, not only in the same network but across all networks of physical devices; MAC addresses are tied to the hardware itself and are not subject to change during the network device's lifespan. This way, we guarantee that a robot is always able to clearly and indubitably identify the sender of a given message, regardless of the possible changes the sender's IP may suffer due to network failures. Additionally, ROS's transport layer protocol, TCPROS, makes use of TCP in the transmission of messages. Thus, we expect that messages are either delivered correctly or completely lost, since TCP applies error-checking measures that should prevent any messages from getting corrupted. Furthermore, a lost message does not compromise the mission; the system does not depend on a reliable connectivity with its peers, instead processing data as it arrives. If the connection to one of the peers is lost, the latest information it sent is used in the map fusion process. As such, the system is expected to be robust to single node failure, as well as link failure.

Furthermore, this node is able to deal with a team of unknown size. Thanks to the combination of OLSR, *wifi_comm* and the usage of MAC addresses as identifiers, we are able to dynamically create a list of teammates that is able to grow as long

as robots are added to the team. The usage of OLSR [32] further strengthens our solution's robustness to network failure. Being a mesh networking routing algorithm, it is designed to recover from failures in links between nodes, and also to be able to establish new links between nodes as needed.

It is also required that communication be as efficient as possible. That goal is achieved by employing the LZ4 compression algorithm to all occupancy grids meant to travel through the network, following the results presented in [37].

The insertion of a new local map into the network is triggered by the insertion of a new map into the system, which means that all new local maps that pass through the *Map Dam* node are sent into the network. This methodology is meant to facilitate the generation of an interpretable map as soon as possible, by quickly relaying new information to all robotic agents. Furthermore, maps are transmitted regardless of any measured link quality; we believe the usage of the LZ4 data compression technique makes the maps small enough to go through low-quality links.

3.3.2 The Map Fusion Node

The `Map Fusion` node, also inhabiting its own package, is responsible for fusing (i.e. aligning and merging) a pair of maps and for calculating the geometric transformations between the original occupancy grids' reference frames and the fused reference frame.

This module features the reference implementation of the alignment algorithm presented in [18] at its core. As referred in Sect. 2.2.2, this algorithm fuses occupancy grids using an image processing method based on the Hough Spectrum to determine the candidate transformation that better fits the existing occupancy data. The aforementioned implementation is only capable of dealing with its own data representations, and only able to determine the transformations of the maps, but not to actually merge them. This node extends that implementation to be able to perform the remaining operations needed, while simultaneously wrapping it in ROS's standards, i.e. embedding it into a ROS node. This node also features a self-tuning ability, i.e. it gauges the computer's performance on startup, and adjust the amount of CPU dedicated to the alignment effort accordingly.

Since there is always the possibility of building a better, more robust algorithm for map fusion, special care has been taken to ensure that this node is as decoupled from the rest of the system as possible, so as to ensure that it can be easily swapped for another that communicates in the same way.

3.3.3 The Complete-Map Generation Node

This node is responsible for building a representation of the environment using all the available information. It does so by controlling the `Map Fusion` node, which only fuses grids two at a time. This node receives a vector of occupancy grids and

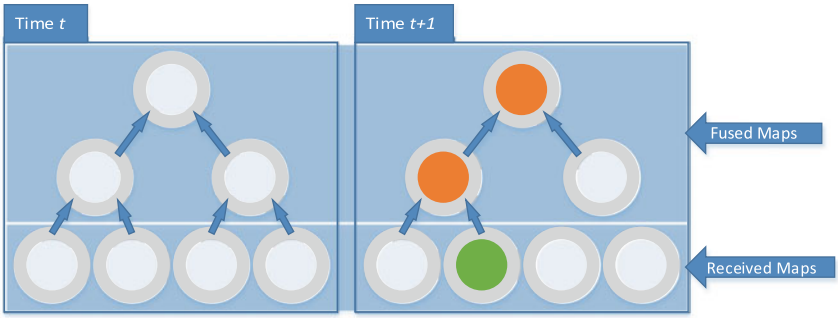


Fig. 6 Illustration of the tree-like structure for storing and updating maps. At a given instant *t*, the system receives a vector of maps, and iteratively merges them until it obtains a single representation. When a new map is received (green), only the maps that depend on it must be re-merged (orange). Each pair of arrows represents a merging operation

iteratively builds a tree-like data structure for storing occupancy grids, as represented in Fig. 6.

When a new occupancy grid is received, be it from the local or from a remote robot, this node starts building a new representation of the environment, incorporating into it the new information. Doing this in an incremental way, i.e. by attempting to integrate the new map in the previously merged one, could lead to disastrous results: once a merging step fails, which is likely at the beginning of the mission due to the fact that little information is available, all future complete maps will include that error.

The tree-like representation of map storage allows us to reuse this information when new maps are received. When a new occupancy grid is received, we only re-merge the intermediate grids that depend on the newly-received grid, thus avoiding costly operation of completely rebuilding the final occupancy grid. This behavior is illustrated in Fig. 6. This method allows us to recover from an erroneous merge in a natural way: when new information is received, all grids that depend on it are rebuilt, thus giving the algorithm the opportunity to correct its previous error.

This structure is initialized with the first two maps received from different sources, i.e. the iterative map fusion process only starts when there are at least two different maps available. The structure then grows laterally as new teammates join the mission (or as their maps are received for the first time) to accommodate more information, as illustrated in Fig. 7.

This node also keeps a similar structure for calculating the transformations between the various occupancy grids, so that the relative poses between robots can be determined. These are also updated as needed when new information arrives, and are fed into the `Remote Navigation` node every time a new Complete-Map is computed.

Note, however, that the fused map, also known as global map, is not shared between the robots. It is only accessible to the robots that run the `Complete-Map`

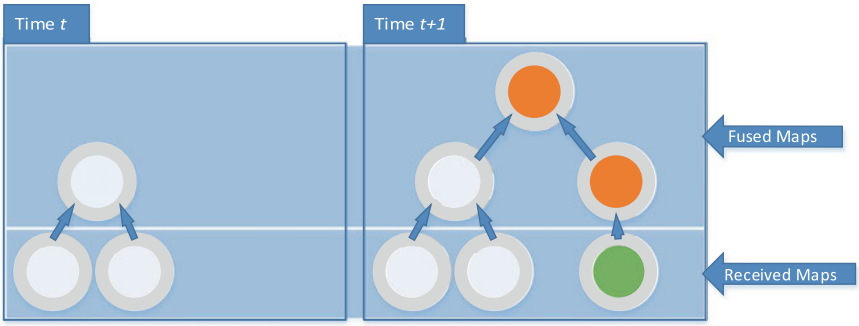


Fig. 7 When a new robot joins the mission, the tree-like structure is grown sideways to accommodate the new data. Iterative merging then proceeds as usual, regardless of whether the current number of maps is even or odd

`Generation` Node, i.e. robots that have interest on the data. Sharing the global map can easily be achieved by transmission of the map to other robots using the exact same process of sharing the individual partial maps. Yet, this is out of the scope of this work.

3.3.4 Auxiliary Nodes

A fourth package which contains two additional ROS nodes is included. While not difficult to implement or intricate in their operation, they are vitally important for the performance of the system.

The `Map Dam` node, as the name implies, intends to add control and intelligence to the way occupancy grids flow through our system, as well as further decouple our system's functionality from the behavior of the SLAM approach. This node intercepts all the occupancy grids output by the SLAM technique, and introduces them into our system. Before the introduction into the system, the grids are trimmed, i.e. any excess free space is removed from the grid.

The `Remote Navigation` node was designed to cope with the requirements of the *tf* ROS package. *tf* is responsible for managing the geometrical transformations between the various frames that compose the robot, and is an extremely useful and widespread tool. However, it requires that the information it receives be very carefully formatted and timed. It was also designed to work within a single robot, which creates a need to re-tag and re-format much of the information that is passed to it.

This last node receives transformation information from the `Complete-Map Generation` node and from the `Data Interface` node, and periodically propagates it into the *tf* topic, correctly tagged and formatted.

4 Installation and Usage

4.1 Installation

The `mrgs` system is meant to be installed as any cutting-edge ROS package: by cloning its repository into the current source space, installing its dependencies and building the workspace. Thus, we assume that the user has a working installation of ROS Melodic.[11] Earlier distributions should also be compatible, provided they are post-Fuerte, i.e. provided they use `catkin` and support the necessary dependencies.

Clone the package into your workspace by running:

```
git clone https://github.com/gondsm/mrgs
```

in your catkin source space (the `src` folder in your workspace).

The `mrgs` metapackage, included in the repository, lists all of the necessary dependencies that must be fulfilled for the system to work. Thus, installing the system's dependencies should be achieved by simply running:

```
rosdep install mrgs
```

which should be followed by ROS installing a number of packages through `apt`, as necessary. Once that procedure ends, you'll need to compile your workspace by changing the directory into its root and running:

```
catkin_make
```

4.2 Running the System

The `mrgs` package contains the necessary launch files to run the system.[12] As seen in Sect. 3, the system can be run in two main modes: centralized mode, which runs the full stack, and distributed mode, in which not all robots fuse information. These modes are achieved by mixing and matching two kinds of nodes: the central nodes, which receive information from all of the robots in the team and fuse it to obtain a global map; and mapper nodes, which run only the bare minimum system necessary to input information into the remainder of the team.

To run the system in centralized mode, only one robot in the team should run a central node. To achieve this, this main robot should use the launch file:

```
roslaunch mrgs central_node.launch
```

[11] http://wiki.ros.org/melodic/Installation/Ubuntu.

[12] It is important to note that, since `mrgs` is still under heavy development, it is possible that the general operating guidelines for the system change over time. As such, this text includes the most generic instructions necessary, with all detail being included in the project's repository. In case of conflict, the instructions on the repository should take precedence, as they will be significantly more up to date.

and, thus, run the full system. The remaining robots need to be launched using the launch file:

```
roslaunch mrgs mapper_node.launch
```

Which will ensure that they do not run the relatively CPU-heavy map merging operations. To run the system in distributed mode, all robots should run as central nodes. Thus, all robots will share and receive information from all other nodes, which results in a highly redundant system, better suited to hazardous situations.

5 Experiments

5.1 Experimental Method

In order to evaluate and validate the performance of our system, experimental tests were performed, both in simulated and real-world environments. Since `mrgs` is a Multi-Robot SLAM solution, the most appropriate way of validating its performance is using several robots in an appropriate environment. In order to test our system, we have segmented the experimental process into two steps: the data-gathering step, and the processing step. The data-gathering step consisted of collecting several runs of data from a single location, emulating an exploration performed by a team. We took advantage of ROS's facilities for data recording,[13] ensuring that this data would later be played back as if it had just been acquired. This has enabled us to test the system with real-world data from the very beginning, by replacing the first blocks illustrated in Fig. 3 with a single ROS node tasked with playing back the recorded data. We designate the tests with data-gathering via recording without the `mrgs` system running in real time, as *offline* tests.

The processing step took place in a controlled environment. Unit tests were performed on modules as they were constructed, and system tests were then run using a mixture of virtual machines and real, physical computers, as illustrated in Fig. 8, each acting as a robot. This has allowed us to, while using real-world data, repeatedly test our system as it was improved.

This approach had the very significant advantage of providing a lifelike environment for testing the solution, down to the transmission of data through a network interface, that was not only repeatable but also moderately easy to set up, thanks to the ability of cloning and creating snapshots of virtual machines. The only significant drawback of this approach is the large amount of resources needed for running these virtual machines on a single host computer. However, this testing solution is lean enough to be run using three simultaneous machines on a consumer-grade laptop with only 8 GB of RAM and a dual-core Intel i7-620M processor. Without the use of virtual machines, at least three computers would be needed for each test, which would

[13] Namely the `rosbag` tool, described at http://wiki.ros.org/rosbag.

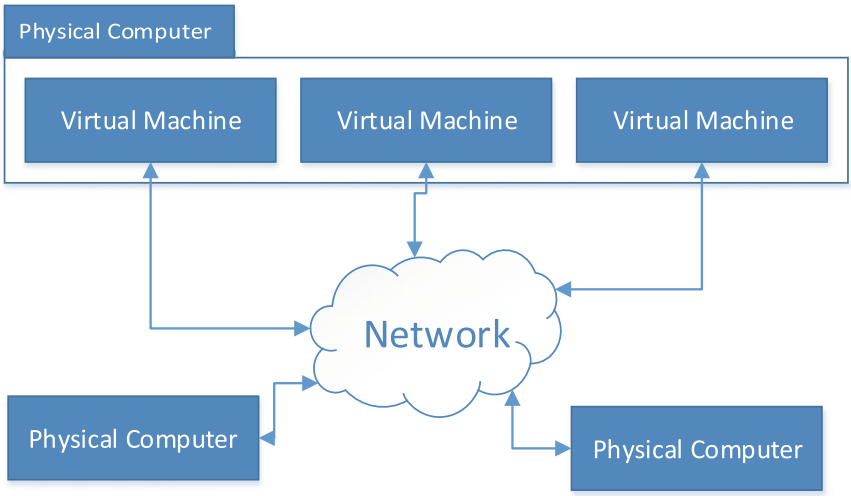


Fig. 8 An illustration of the setup used to process data. Virtual machines and real computers can be used as equals, in any combination, in a network routed by OLSR

have to be correctly configured and simultaneously operated, severely harming the repeatability of the experiment.

With the purpose of illustrating the system's performance, two main offline tests took place: one in the MRL Arena, illustrated in Fig. 9; and on ISR's corridors, which is illustrated in Fig. 10. For these experiments, two sets of data were gathered at the MRL Arena, and three were gathered at the corridors of the Institute, simulating a mission composed of two and three robots, respectively.

Tests with actual multiple robots also took place, in a setup similar to Fig. 9. These online tests were used to determine if the results we were obtaining in the offline setup were correct, and to demonstrate the system's ability to operate in real-time. These tests also took place in the MRL arena and the corridors of the Institute, and achieved results that are identical to those of the offline methodology. Thus, we discuss only the results of the offline tests.

To further mimic the real-world usage of the package, the communication between the machines was compressed by the `data_interface` node.

5.1.1 Hardware

Real-world data was gathered using one or several Pioneer 3-DX mobile platforms, shown in Fig. 10, running the software framework described in Fig. 3. These are equipped with a Hokuyo URG-04LX-UG01 Laser Range Finder, and the Pioneer's build-in encoders were used for odometry. The platforms were teleoperated using a Wiimote or a remote machine. For teleoperation and recording purposes, each plat-

Fig. 9 A small experiment taking place in the MRL Arena. In this experiment, a Traxbot and a Stingbot platforms were used. In this experiment, the centralized mode was used, to cope with the low processing power of the smaller laptops

Fig. 10 An illustration of the Institute's arrow-straight corridors, where experiments took place. In this instance, the distributed mode can be used, since the Pioneer 3-DX is able to carry larger laptops that are able to run the full system

form had an Asus EEE 1005 mini-laptop mounted on it. The simulation and virtual machine-related operations were performed on a more powerful Toshiba Qosmio F60 laptop.

5.2 Results and Discussion

5.2.1 System Performance and Immunity to SLAM Technique Variation

In order to test the system's immunity to the variation of SLAM technique, i.e. its ability to perform its functionality regardless of the SLAM technique used in the mission, data was gathered in the MRL Arena, which was then processed using the three different SLAM techniques mentioned in Sect. 2.4: *slam_gmapping*, *slam_karto* and *hector_slam*, using their default parameters, namely regarding the rate at which they output maps, and their size. We have configured them all to use a resolution of 5 cm/cell.

These three techniques, while equivalent in their results, have different operational requirements. For instance, *Hector* does not use odometric data. Their results are also different, both in format and quality. *GMapping* tends to output maps that have significant empty areas surrounding a region of interest, as does *Hector*; *Karto* outputs maps that are very closely trimmed to the region of interest. Our system was designed to deal with this issue by cropping all the input maps to include only the region that has useful information. The `Map Fusion` node then applies padding around the maps to guarantee that no information is lost during rotation.

Figures 11, 12, 13 show the results of the MRL Arena experiment for each of the SLAM techniques, which were run over the same recorded data. These results show our system's ability to handle and adapt to the SLAM technique in use, being able to successfully attain a global representation of the environment. We can see that while every SLAM technique builds a different slightly representation of the environment,

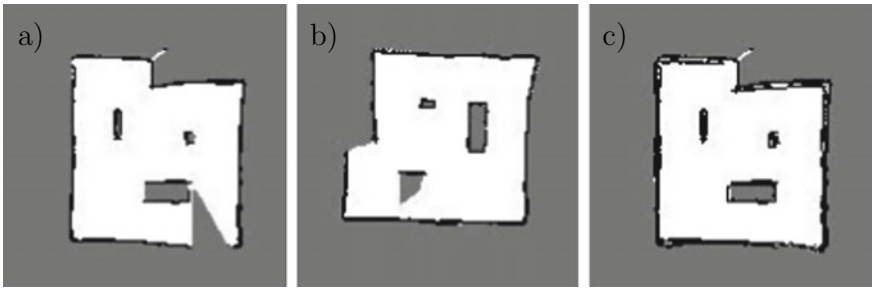


Fig. 11 An example of the results obtained using *GMapping* in the MRL Arena. **a** and **b** are the local maps of each of the robots, **c** is the result of the map fusion process

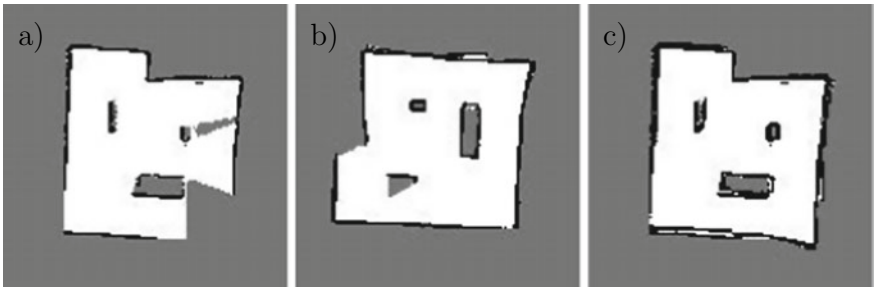


Fig. 12 An example of the results obtained using *Karto* in the MRL Arena. **a** and **b** are the local maps of each of the robots, **c** is the result of the map fusion process

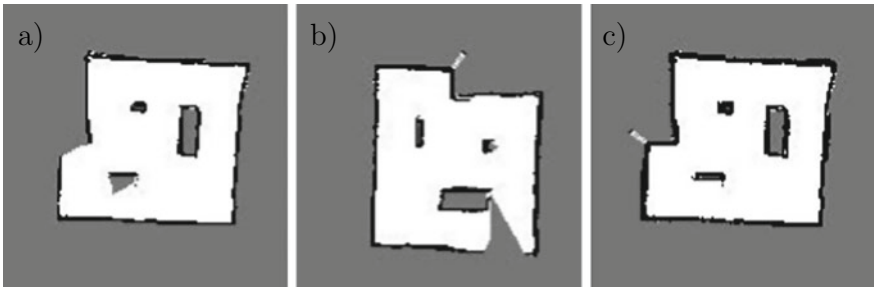


Fig. 13 An example of the results obtained using *Hector* in the MRL Arena. **a** and **b** are the local maps of each of the robots, **c** is the result of the map fusion process

our system is able to process the data regardless. It is also important to note that the map fusion process fuses the *second* map *into* the *first*, which is why there is a disparity in the orientation of the final representations.

The experiment that took place in the corridors involved a team of three robots. In this scenario, the `Map Fusion` node was expected to be able to deal with maps it had already fused before, as described and illustrated in Sect. 3.3.2. However, its performance revealed room for improvement, as illustrated in Fig. 14.

An unsuccessful fusion step can be catastrophic to the remaining effort, utterly invalidating the final result. This unsuccessful fusion can be attributed to several factors, such as the fact that the corridors are almost featureless, which hinders the efforts of both the SLAM technique and the map fusion process; the presence of glass panes, visible in the lower part of the image, that interfere with the laser range finder; disparities in the way different maps represent the same real-world area, and other factors.

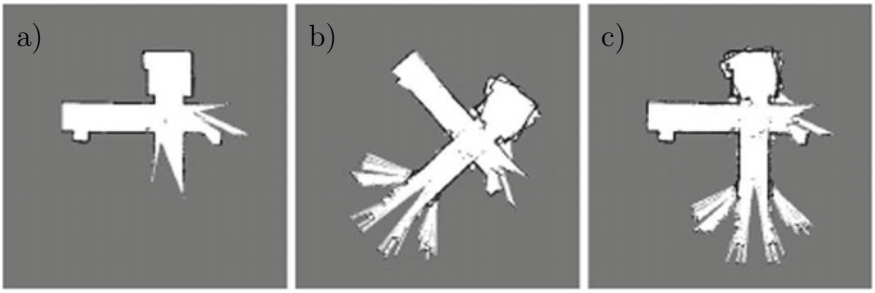


Fig. 14 An example of a failed three-way map fusion in the Institute's corridors. **a** is one of the robot's local map, and **b** is the result of the fusion of two other local maps. We can see that, while the maps were roughly rotated and translated correctly, there is a noticeable misalignment on one of the maps

5.2.2 Communication Efficiency

Table 1 displays the results obtained during the MRL Arena mission. N represents the number of processed maps (output by the SLAM technique into our system), $\bar{R}$ is the average compression ratio achieved during the mission, L_t is the total size of the maps received by our system (before compression, after cropping), L_s is the total amount of data sent into the network by this robot and, D_s is the amount of data we have saved, i.e. the difference between the total size of the maps and the data actually transmitted, and, finally, T_p is the total time spent processing maps, in milliseconds. Essentially, these show that the technique adopted to ensure efficiency in communication, compression using the LZ4 technique, is a viable option.

As postulated in Sect. 2.5, using compression on occupancy grids yields important data savings. In this case, using real occupancy grids, we have saved at least about 7/8 of all data meant to be sent, which equates to about 88% savings in data sent. These savings come at a very reduced cost, as is visible on the last column of Table 1. At the most, we spent a total of about 15 milliseconds processing maps during a mission, which given that during that mission we saved 11/12, or 91.6%, on transmitted data, is a very positive result. It is important to note that these results are limited to occupancy grids, as mentioned before; it is not expected that the sole usage of LZ4 yield such dramatic improvements in the efficiency of the communication. For our use case, however, compression has been shown to be a very competent solution, which we will look to extend in the future.

In general, results show that the performance of our Multi-Robot SLAM solution is acceptable. The technique is able to successfully map an environment using the information gathered by multiple robots, and does so employing a very efficient communication method, and allowing smooth real-time operation of the system. A video of the system running with three robots in the MRL Arena has been made available.[14]

[14] https://drive.google.com/open?id=18jy5uftf5n-CqTDRJKNVOdtcd13BGPkw.

Table 1 Network statistics for outgoing data obtained during in the MRL Arena and the ISR corridors. These are respective to one of the robots in the mission; the results obtained by its teammate are equivalent. All sizes are in bytes

	N	$\bar{R}$	L_t	L_s	D_s	T_p
(a) Results obtained during the MRL Arena mission						
GMapping	21	8.78	169062	19253	149809	2.77
Karto	6	8.03	48357	6015	42342	0.82
Hector	75	8.61	606667	70472	536195	9.61
(b) Results obtained during the ISR corridors mission						
GMapping	21	13.92	930050	66787	863263	7.68
Karto	6	12.06	209799	17402	192397	2.64
Hector	76	12.03	3198376	265883	2932493	14.99

N Number of Processed Maps
$\bar{R}$ Average Compression Ratio
L_t Total Size of the Maps Received
L_s Total Amount of Data Sent into the Network by the Robot
D_s Amount of saved data
T_p Total map processing time (ms)

We do not assess the accuracy of the obtained maps in detail, as this heavily relies on the specific modules used in the `mrgs` framework, especially the `Map Fusion` node, which has been shown to have room for improvement. The interested reader may refer to [38] for an accuracy analysis conducted by a different research group, that made use of our system. Instead, we have focused on demonstrating that the proposed approach is able to correctly handle all the decoupled functionalities developed, namely: collection, compression, transmission and merging of the individual robots' maps, regardless of the underlying techniques used.

6 Conclusions and Future Work

In Sect. 1, we have defined a set of goals for this work. We shall now reflect upon them and on our system's performance.

First and foremost, we needed our system to be completely distributed. By designing the `Data Interface` node to abstract away all the network-related details, and the `Complete-Map Generation` node to be able to fuse an unknown number of maps, we were able to build a system that, by design, does not limit the number of robots we can use; the bottleneck in our system's performance is the machine it is running on.

However, operating in a fully distributed way creates an overly redundant system, where resources are wasted in calculating the same results on various machines. While desirable in a high-risk scenario, a more modest solution was required for situations in which the risk of losing a robot was not as high. To solve this issue, we

have designed our system to support three different modes of operation, which allow us to fully adapt to and exploit the risk-level of the situation.

Failures in communication were also taken into account, and to deal with them we make use of a combination of TCP with the usage of hardware-bound addresses for robot identification so that a failure in the network does not compromise our ability to correctly identify the team members.

Scalability was also an important topic of this work, and was assured in a twofold way. Firstly, by employing compression in our communication, we have been able to save approximately 90% in required bandwidth, which allows us to greatly enlarge the team. Secondly, by designing the system so as to not depend on the number of robots available a priori, i.e. by making it able to dynamically embrace new team members, we have achieved a solution that does not have a theoretical limitation to the team's size.

The last requirement stated that the system should be able to deal with any 2D SLAM technique that was integrated in ROS's standards. To satisfy this requirement, we have designed our system to depend on those same standards as much as possible, and to retrieve information in a standardized way. This has allowed the use of the framework by other researchers in the community (see for example [38]).

As for future work, it would be useful to include a tool that graphically displays the network's status, the bandwidth used by each connection, the current gains obtained by using compression, etc. A tool like this would greatly simplify the control of the mission, and would make information available "at a glance".

The `Data Interface` node is already efficient enough at transmitting data for our purposes. However, it would be interesting to explore the concept of delta encoding in an attempt to further increase its efficiency. Delta encoding is a broad concept that essentially consists of having a backlog of previous messages, and constructing the following message as a set of differences from on or more of the previous. In this case, delta encoding could be applied in one of two levels: at the map level, where each new map would be sent as a set of differences from the last; or at the buffer level, by sending each compressed buffer as a set of differences from the last. It would be interesting to test both hypotheses.

The `Map Fusion` node could also be improved. Being very well decoupled from the rest of the system, it would be very interesting to see it replaced with a more robust approach in the future, such as [39, 40], which could increase the success of the mapping effort itself. We could also test merging maps for robots running different individual 2D SLAM techniques, which would be possible, assuming that the output maps of the different techniques follow the same standard data principles and have the same map resolution.

The `Complete-Map Generation` node could also be improved to detect and deal with failed fusion attempts, for instance by discarding the newest fusion result and keeping the last successful one. Currently, a failed map fusion attempt has catastrophic results for the mapping mission.

Moreover, this work does not address the issue of inter-robot interference, i.e. the fact that the presence of a robot within another's scanning range violates the static world assumption, generating inconsistencies in the final map. It would be

very interesting to look into the possibility of creating a way of filtering robots out of each others' laser scans without using additional sensors.

The current model of distributed operation is highly redundant: every single node that is operating has to create its own complete-map. Even though this model was decided upon to maximize the tolerance to failure, it can represent a large waste of computational resources in scenarios where the network is trustworthy. An improvement over the current system would be to create a new mode of operation, where it would be capable of performing the processing of maps in a distributed fashion, in a manner inspired by computer clusters (e.g. see [41]).

Robots would have to periodically sync, and trade maps as well as control messages to determine how the complete-map generation process would take place. Each robot would be responsible for building a part of the map tree, effectively distributing the computational load across the robotic network. For instance, for four robots, two of them would fuse two pairs of maps, and then send the fused maps to another robot, who would fuse them to create a complete-map, which would then be propagated across the network.

Finally, a major improvement would be to deal with any kind of data. As we know, a representation of an environment is not limited to an occupancy grid. In this sense, the system could be upgraded to transmit, store and organize raw data. In fact, the only node that explicitly relies on occupancy grids is the `Map Fusion` node. If every other node were converted to deal with arbitrary data, only the `Map Fusion` node would have to be deeply rebuilt in order to cope with the change. For instance, we could perform Multi-Robot SLAM based not on occupancy grids, but on pose graphs, as described in Sect. 2.1 (see also [42]). Given the system's architecture, we could retain the communication's efficiency, as well as the map tree method of computing a global representation, simply by replacing occupancy grids in the data structures with content-agnostic structures.

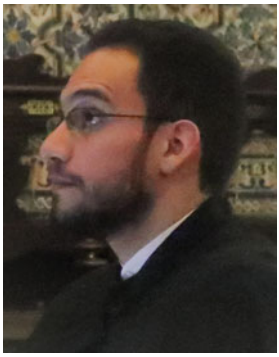
Acknowledgements We are sincerely thankful for the contributions on the free and open source frameworks adopted in this work, particularly: Giorgio Grisetti for his work on *RBPF SLAM*, Brian Gerkey and Vincent Rabaud for porting and maintaining *GMapping* for ROS, Stefan Kohlbrecher for the design and development of *Hector Mapping* for ROS, SRI International for *Karto SLAM*, and its maintainers for ROS over the years (Bhaskara Marthi, Michael Ferguson, Luc Bettaieb and Russell Toris), and to the overall ROS community. This work was supported by the **S**eguranças robó**T**icos co**OP**erativos (STOP) research project (ref. CENTRO-01-0247-FEDER-017562), co-funded by the "Agência Nacional de Inovação" within the Portugal2020 programme.

References

1. S. Thrun, Robotic mapping: a survey, in *Exploring Artificial Intelligence in the New Millennium*, vol. 1(1–35) (2002), p. 1
2. U. Frese, A discussion of simultaneous localization and mapping. Auton. Robots **20**(1), 25–42 (2006)
3. R.E. Kalman, A new approach to linear filtering and prediction problems. J. Basic Eng. **82**(1), 35–45 (1960)

4. S. Thrun, D. Fox, W. Burgard, F. Dellaert, Robust Monte Carlo localization for mobile robots. Artif. Intell. **128**(1–2), 99–141 (2001)
5. F. Lu, E. Milios, Globally consistent range scan alignment for environment mapping. Auton. Robots **4**(4), 333–349 (1997)
6. S. Thrun, M. Montemerlo, The graph SLAM algorithm with applications to large-scale mapping of urban structures. Int. J. Robot. Res. **25**(5–6), 403–429 (2006)
7. R. Kuemmerle, G. Grisetti, H. Strasdat, K. Konolige, W. Burgard, TORO - Tree-Based netwORk Optimizer (2008). http://www.openslam.org/toro.html
8. R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, W. Burgard, g2o: a general framework for graph optimization, in *2011 IEEE International Conference on Robotics and Automation* (IEEE, 2011), pp. 3607–3613. http://www.openslam.org/g2o.html
9. G. Grisetti, C. Stachniss, S. Grzonka, W. Burgard, A tree parameterization for efficiently computing maximum likelihood maps using gradient descent, in *Robotics: Science and Systems*, vol. 3 (2007), p. 9
10. K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, R. Vincent, Efficient sparse pose adjustment for 2D mapping, in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2010), pp. 22–29
11. G. Vallicrosa, P. Ridao, H-slam: Rao-blackwellized particle filter SLAM using Hilbert maps. Sensors **18**(5), 1386 (2018)
12. D. Portugal, R.P. Rocha, Scalable, fault-tolerant and distributed multi-robot patrol in real world environments, in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2013), pp. 4759–4764
13. M. Meghjani, G. Dudek, Multi-robot exploration and rendezvous on graphs, in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2012), pp. 5270–5276
14. L.A. Andersson, J. Nygards, On multi-robot map fusion by inter-robot observations, in *2009 12th International Conference on Information Fusion* (IEEE, 2009), pp. 1712–1721
15. Z. Li, R. Chellali, Visual place recognition for multi-robots maps merging, in *2012 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)* (IEEE, 2012), pp. 1–6
16. D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, B. Stewart, Distributed multirobot exploration and mapping. Proc. IEEE **94**(7), 1325–1339 (2006)
17. R. Natarajan, M.A. Gennert, Efficient factor graph fusion for multi-robot mapping and beyond, in *2018 21st International Conference on Information Fusion (FUSION)* (IEEE, 2018), pp. 1137–1145
18. S. Carpin, Fast and accurate map merging for multi-robot systems. Auton. Robots **25**(3), 305–316 (2008)
19. X.S. Zhou, S.I. Roumeliotis, Multi-robot SLAM with unknown initial correspondence: the robot rendezvous case, in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2006), pp. 1785–1792
20. A. Censi, L. Iocchi, G. Grisetti, Scan matching in the Hough domain, in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation* (IEEE, 2005), pp. 2739–2744
21. A. Howard, Multi-robot simultaneous localization and mapping using particle filters. Int. J. Robot. Res. **25**(12), 1243–1256 (2006)
22. L. Carlone, M.K. Ng, J. Du, B. Bona, M. Indri, Rao-Blackwellized particle filters multi robot SLAM with unknown initial correspondences and limited communication, in *2010 IEEE International Conference on Robotics and Automation* (IEEE, 2010), pp. 243–249
23. M.T. Lazaro, L.M. Paz, P. Pinies, J.A. Castellanos, G. Grisetti, Multi-robot SLAM using condensed measurements, in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2013), pp. 1069–1076
24. P. Zhang, H. Wang, B. Ding, S. Shang, Cloud-based Framework for scalable and real-time multi-robot SLAM, in *2018 IEEE International Conference on Web Services (ICWS)* (IEEE, 2018), pp. 147–154
25. S.S. Ali, A. Hammad, A.S. Tag Eldien, Cloud-based map alignment strategies for multi-robot FastSLAM 2.0. Int. J. Distrib. Sens. Netw. **15**(3), 1550147719829329 (2019)

26. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, ROS: an open-source robot operating system, in *ICRA Workshop on Open Source Software*, vol. 3, No. 3.2 (2009), p. 5
27. G. Grisetti, C. Stachniss, W. Burgard, Improved techniques for grid mapping with Rao-Blackwellized particle filters. IEEE Trans. Robot. **23**(1), 34 (2007)
28. S. Kohlbrecher, O. Von Stryk, J. Meyer, U. Klingauf, A flexible and scalable SLAM system with full 3D motion estimation, in *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics* (IEEE, 2011), pp. 155–160
29. J.M. Santos, D. Portugal, R.P. Rocha, An evaluation of 2D SLAM techniques available in robot operating system, in *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics* (SSRR). Linköping, Sweden, Oct 21–26, (IEEE, 2013), pp. 1–6
30. W. Hess, D. Kohler, H. Rapp, D. Andor, Real-time loop closure in 2D LIDAR SLAM, in *2016 IEEE International Conference on Robotics and Automation (ICRA)* (IEEE, 2016), pp. 1271–1278
31. A. Tiderko, F. Hoeller, T. Röhling, The ROS multimaster extension for simplified deployment of multi-robot systems, in *Robot Operating System (ROS)* (Springer, Cham, 2016), pp. 629–650
32. A. Tonnesen, T. Lopatic, H. Gredler, B. Petrovitsch, A. Kaplan, S.O. Turke, *OLSRD: An Ad Hoc Wireless Mesh Routing Daemon* (2008). http://www.olsr.org/
33. R.P. Rocha, D. Portugal, M. Couceiro, F. Araújo, P. Menezes, J. Lobo, The CHOPIN project: cooperation between Human and rObotic teams in catastroPhic INcidents, in *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)* (IEEE, 2013), pp. 1–4
34. J.C. Bermond, L. Gargano, S. Perennes, A.A. Rescigno, U. Vaccaro, Efficient collective communication in optical networks, in *International Colloquium on Automata, Languages, and Programming* (Springer, Berlin, Heidelberg, 1996), pp. 574–585
35. A. Cunningham, M. Paluri, F. Dellaert, DDF-SAM: fully distributed SLAM using constrained factor graphs, in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE, 2010), pp. 3025–3030
36. R. Rocha, J. Dias, A. Carvalho, Cooperative multi-robot systems: a study of vision-based 3-D mapping using information theory. Robot. Auton. Syst. **53**(3–4), 282–311 (2005)
37. G.S. Martins, D. Portugal, R.P. Rocha, A comparison of general-purpose FOSS compression techniques for efficient communication in cooperative multi-robot tasks, in *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, vol. 2 (IEEE, 2014), pp. 136–147
38. M.A. Abdulgalil, M.M. Nasr, M.H. Elalfy, A. Khamis, F. Karray, Multi-robot SLAM: an overview and quantitative evaluation of MRGS ROS framework for MR-SLAM, in *International Conference on Robot Intelligence Technology and Applications* (Springer, Cham, 2018), pp. 165–183
39. N. Shaik, T. Liebig, C. Kirsch, H. Müller, Dynamic map update of non-static facility logistics environment with a multi-robot system, in *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)* (Springer, Cham, 2017), pp. 249–261
40. J.G. Mangelson, D. Dominic, R.M. Eustice, R. Vasudevan, Pairwise consistent measurement set maximization for robust multi-robot map merging, in *2018 IEEE International Conference on Robotics and Automation (ICRA)* (IEEE, 2018), pp. 2916–2923
41. D. Portugal, B.D. Gouveia, L. Marques, A distributed and multithreaded SLAM architecture for robotic clusters and wireless sensor networks, in *Cooperative Robots and Sensor Networks 2015* (Springer, Cham, 2015), pp. 121–141
42. I. Deutsch, M. Liu, R. Siegwart, A framework for multi-robot pose graph SLAM, in *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)* (IEEE, 2016), pp. 567–572



Gonçalo S. Martins received the M.Sc. degree in Electrical and Computer Engineering from the University of Coimbra (UC) in 2014, and a Ph.D. degree on Automation and Robotics also at the University of Coimbra in 2019, focusing on user-adaptive systems, machine learning and social robotics. Until recently, he was a researcher at the Institute of Systems and Robotics, where he worked on the H2020 GrowMeUp and Interreg EuroAGE projects. He is now working as a Senior Researcher at Ingeniarius, Ltd. His main research interests include Field and Service Robotics, Artificial Perception and Multi-Robot systems.

David Portugal completed his Ph.D. degree on Robotics and Multi-Agent Systems at the University of Coimbra in Portugal, in March 2014. His main areas of expertise are cooperative robotics, multi-agent systems, simultaneous localization and mapping, field robotics, human-robot interaction, sensor fusion, metaheuristics, and graph theory. After his Ph.D., he spent 4 years outside academia, and he is currently working as an Assistant Researcher at the University of Coimbra since 2019. He has been involved in several local and EU-funded research projects in Robotics and Ambient Assisted Living, such as CHOPIN, TIRAMISU, Social Robot, CogniWin, GrowMeUp, STOP, CORE and SEMFIRE. He has co-authored over 65 research articles included in international journals, conferences and scientific books.

Rui P. Rocha is an Assistant Professor (tenure position) in the Department of Electrical and Computer Engineering and a permanent researcher at the Institute of Systems and Robotics, both at the University of Coimbra, Portugal. His research interests include cooperative mobile robotics, cooperative perception, multi-robot systems, human-robot team cooperation, distributed control, ambient assisted living, social robots, and autonomous robots. Rocha received a Ph.D. in electrical and computer engineering from the University of Porto in May 2006. He has been involved in several FP6 and FP7 European funded projects developed in consortium for the past few years, and he is a senior member of IEEE and of the IEEE RAS Technical Committee on Multi-Robot Systems.

Agile Experimentation of Robot Swarms in Large Scale

Vivian Cremer Kalempa, Marco Antonio Simões Teixeira, André Schneider de Oliveira, and João Alberto Fabro

Abstract This chapter aims to present a new ROS package to automate experimentation with multiple mobile robots. A robot swarm is a specific system that requires a complicated setup and has a high cost with regard to experimentation. Virtual environments can be used to expedite testing; however, these also are very laborious. This package is a tool set to easily configure the experimentation environment for swarm tasks with the most popular perception systems and absolute or relative localization references. The user specifies in the tool only the number of robots required, sensors, and functions without having to configure each robot individually for the simulation. This chapter presents two examples of the developed package to show how the new package simplifies working with swarms of robots and focuses on the application rather than the required configuration. An example of a formation maintained through the fuzzy approach is developed to demonstrate the potential of the proposed package. The approach is based on a leader agent that executes autonomous navigation through a LIDAR perception system, and follower agents that are responsible for maintaining the formation based on the leader. A fuzzy intelligent behavior commands the dynamic formation adaptation of the robot swarm as it attempts to overcome obstacles. Finally, the computational cost is evaluated to allow readers to estimate the computational resources necessary to perform practical experimentation.

V. C. Kalempa (✉)
Universidade do Estado de Santa Catarina - UDESC, Luiz Fernando Hastreiter St., 180, São Bento do Sul, Brazil
e-mail: vivian.kalempa@udesc.br

M. A. Simões Teixeira · A. S. de Oliveira · J. A. Fabro
Universidade Tecnológica Federal do Paraná - UTFPR, Sete de Setembro Ave, 3165, Curitiba, Brazil
e-mail: marcoteixeira@alunos.utfpr.edu.br

A. S. de Oliveira
e-mail: andreoliveira@utfpr.edu.br

J. A. Fabro
e-mail: fabro@utfpr.edu.br

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_4

Keywords Swarm · ROS · Package · Experimentation

1 Introduction

A multirobot system consists of two or more robots that act in a coordinated or cooperative way, in the same environment, with or without the same objective. These systems can be composed of robots with identical physical characteristics (homogeneous) [1] or with different (heterogeneous) [2] topologies. In special situations, the number of robots can be substantial to perform simultaneous subtasks to solve a single problem. In this case, the system is called a "swarm".

A robot swarm, according to [3], can also be seen as a study of how to coordinate large groups of relatively simple robots through the use of local rules. Inspiration comes from nature's insects, which can perform tasks that are beyond the capabilities of the individual. Following this analogy, [4] verifies that a group of robots is not just a group. Rather, it has unique characteristics that are found in insect swarms: decentralized control, lack of synchronization, and simple and (almost) identical members. Reinforcing the definition given by [4, 5] defines a swarm of robots as an approach for coordinating several relatively simple, autonomous, noncentrally controlled physical robots capable of establishing local communication and operating by some sense of biological inspiration.

According to [6], the term "swarm intelligence" was originally conceived as a keyword by Beni in the 1980s [7] to denote a class of cellular robotic systems. Later, the term covered a wide range of studies from optimization to insect studies, losing its robotic context over time. In the last decade, the term "swarm robotics" began to be used as an application of swarm intelligence in physically embedded systems. Some properties observed in insects that are desirable in multirobot systems are as follows [6]:

- **Robustness:** the swarm of robots must be able to function even if some of the individuals fail or there are disturbances in the environment;
- **Flexibility:** the swarm must be able to create different solutions for different tasks and be able to change each function of the robot depending on the needs of the moment;
- **Scalability:** the swarm of robots must be able to work in different sizes of groups, from a few individuals to thousands.

More objectively, [8] defines a swarm of robots as the study of how several relatively simple physical agents can be designed in such a way that a desired collective behavior arises from local interactions between the agents and between the agents and the environment.

This definition is complemented by a set of criteria to better understand and differentiate it from other types of multirobot systems:

– Swarm robots should be autonomous robots capable of performing tasks in a real environment;

- The number of robots in the swarm must be large, or at least as large as the control rules allow;
- The robots must be homogeneous. There may be different types of robots in the swarm, but these groups should not be numerous;
- The robots must be incapable or ineffective regarding the main task they need to solve, i.e., they need to collaborate to succeed or improve performance;
- The robots have only local communication and detection capabilities, which ensures distributed coordination, so scalability becomes one of the system properties.

The new trends in research about robot swarms examine the development of cooperative or coordinated behaviors in a real robotic swarm (in low scale) or virtual swarms (in high scale). Thus, the main characteristics identified in recent works on robot swarms indicated some tendencies that will be used as setup requirements of the virtual experimentation platform:

- The robots must support wireless communication to exchange information with other robots or with the central node;
- The robots should be programmable in parallel through a wireless communication channel since the control algorithms are the same for all the robots;
- The robots must be able to interact physically with each other and the environment;
- The robots should have a long autonomy (with regard to battery capacity) because the swarm may need to operate for a long period to execute a specific task;
- The robots should be as cheap as possible;
- Interference between multiple sensor systems and the effects of environmental factors should be minimal;
- The robots should be small enough not to substantially increase the environment size, but large enough not to limit the robots' expandability or increase the cost of swarm robots owing to component miniaturization.

Virtual experimentation with multiple robots can be an exhaustive task owing to the setup needed for each robot. In a large-scale swarm (thousands of robots), the virtual configuration can be extremely wearing. The computational effort also needs to be considered because the introduction of each new robot requires new computational resources.

The purpose of this chapter is to present a new ROS package that automatically configures a swarm robot simulator for the simulation of large-scale swarms and minimizes the computational cost. This is done with homogeneous and heterogeneous robots, different perception systems, to the main tasks of robot swarm.

2 Background

Swarms of robots and the related concept of swarm intelligence were inspired by organizations of natural swarms such as ants, bees, birds, fish, wolves, and even humans [9]. Insects provide one of the best-known examples of self-organized biological

Fig. 1 School [12]

Fig. 2 Swarm of birds [12]

behavior. Through local and limited communication, they can perform impressive behavioral feats: maintaining the health of the colony, caring for the young, responding to invasions, and so on [10].

As an example, several species of fish are organized in schools because of the risk of being eaten by other predators in the ocean [11]. These species, when organized in schools, are capable of frightening or confounding predators. Figure 1 shows the swarming behavior observed in fish.

Birds may also exhibit swarming behavior, especially when migrating. According to research studies, the cost of migration is reduced owing to this swarming behavior in birds [13]. Figure 2 shows an example of the swarm of birds.

Another example is ants, which show social behaviors, that is, these insects live in family units and their life cycle is organized around this unit known as an ant colony [12]. The ant colony as a whole is capable of performing complex tasks such as feeding and breeding large nests, although a single ant shows no specific

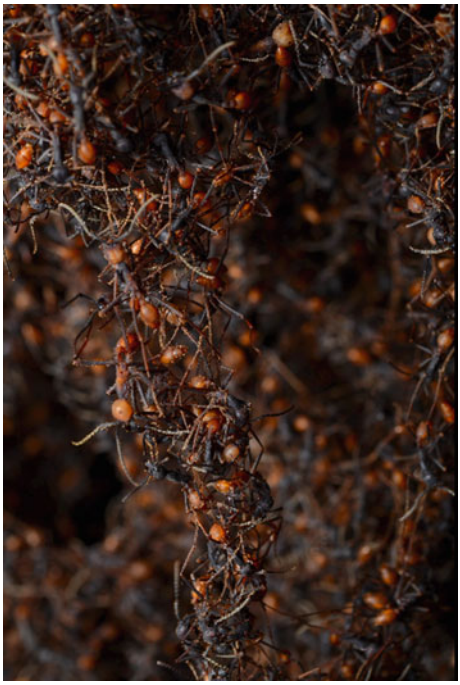


Fig. 3 An ants bridge [12]

intelligence. Figures 3 and 4 show examples of the complex behavior that ants can display in a swarm.

In Fig. 3, it is possible to see that the ants form a bridgelike structure, and Fig. 4 captures some ants on a trail. According to [12], on this trail, some "soldier"-type ants form a tunnel to protect "worker"-type ants, as there is no central leader or control for the swarm. In these behaviors are seen the intelligence and desired standards with simple rules that each swarm agent follows. Thus, to overcome the difficulties in designing a distributed approach, the intelligence of swarms inspired in these biological species can be used.

In this sense, [14] present a pheromone-based solution for efficient mapping of multiple mobile service robots. Experimental results are displayed based on two tasks performed by several robots: cleaning and surveillance. The results show that the proposed scheme allows several service robots to perform cooperative tasks intelligently without any explicit programming.

Reference [15] presents a method for creating a decentralized self-adaptive robot swarm. The task is to define an optimal swarm size in a space exploration function. Communication between robots is restricted to transmission through the environment, and the only way the agents communicate is through artificial pheromone brands.

Reference [16] presented minimalist algorithms inspired by biological morphogenesis for collective swarming behaviors. The proposed algorithms apply to robots

Fig. 4 A trail of ants [12]

without global positioning, direction sensors, motion feedback, or long-range communication devices. First, the author discusses how morphogens can retain the integrity and original form of a swarm of robots without directional sensing, while the swarm moves and interacts with the environment. Thus, a coordinated motion strategy is presented to preserve the connectivity of a real swarm of minimalist robots following a target in their environment. Real experiments were conducted on a swarm of kilobots.

Another example of bio-inspiration for the swarm of robots is the work of [17], where to mimic the operational mechanism of biological neural systems, a map-based self-organizing approach is proposed for assigning tasks to robots from a swarm of robots in three-dimensional dynamic environments. This approach integrates the advantages and characteristics of biological neural systems. It can dynamically plan the paths of a swarm of robots in three-dimensional environments in uncertain situations, such as when some robots are presented or broken or when more than one robot is needed for some particular task locations.

Other examples of robot swarm applications are related to one of the main problems to be solved in multirobot systems: selecting the best robot to perform each task. This is known as a problem of multirobot task allocation (MRTA). The MRTA has been an object of interest to several researchers. In [18], a solution based on noncommunicative assignment of tasks is proposed, since when using wireless communication (WLAN), congestion and interference between the bands are considered

serious. Reference [18] employs game theory to perform task allocation and soccer robot cooperation. Reference [19] also applies game theory to solve the task allocation problem in multirobot missions. The problem considers a swarm with dozens of drones that only know their neighbors, and a mission that consists of visiting a number of locations and performing certain activities. Finally, [20] presents a task allocation method that takes full advantage of the limited power resources of the multirobot system to maximize the number of tasks. Meanwhile, [21] proposes an algorithm based on Monte Carlo trees to solve the problem of allocating multirobot tasks with spatial, temporal, and other constraints.

The next section presents the ROS package developed for a swarm of robots to facilitate the development of applications in this area. In this sense, [22] presented the first description of the use of the Stage simulator for massively multirobot experiments suitable for swarm robotics. However, it did not present a way to automatically configure the simulation environment, as we propose in this chapter. Last, [23] introduces MASON, a discrete event multiagent simulation Java toolkit. MASON is designed to serve as the basis for a wide range of multiagent simulation tasks ranging from swarm robotics to machine learning to environments of social complexity. However, MASON does not work with ROS.

3 Swarm Stage Package

The developed package was named *swarm_stage_ros* and was developed in C++ with the Stage simulator. The Stage simulator was used because of the possibility of total reconfiguration with scripts and by low consumption that allows for a high scale of the swarm.

The package configures some options of environments to work with robots regardless of whether they are in formation. It is possible to determine the number of robots for the scenario as well as the perception systems of these robots. The number of robots can vary from one robot to thousands of robots, depending only on the capacity of the computer that is used to generate the scenario.

The initial research interest in developing this package is to examine the difficulty in configuring a swarm of robots with each new experiment, making it tiresome and purely manual work.

The codes are available in GitHub,[1] and the videos developed with the experiments can be seen on YouTube.[2]

[1] https://github.com/VivianCremerKalempa/swarm_stage_ros.

[2] https://www.youtube.com/channel/UCztalFc6fapGIhQN2Q0jCuw.

3.1 Installation

As a package requirement, you must have installed the Stage simulator. The *swarm_stage_ros* package is available on GitHub and can be installed in your workspace:

```
$ cd /home/user/catkin_ws/src
$ git clone https://github.com/VivianCremerKalempa/swarm_stage_ros
$ cd ..
$ catkin_make
```

To run your package, you need this command:

```
$ roslaunch swarm_stage_ros swarm_stage.launch
```

Note that *roslaunch* starts ROS automatically.

At the end of processing, two files are generated: *custom.world* and *controller.* `launch`. The *custom.world* file has all the necessary specifications for the scenario to be run on the Stage. For example:

```
include "/home/user/catkin_ws/src/swarm_stage_ros/launch/include/map.inc"
include "/home/user/catkin_ws/src/swarm_stage_ros/launch/include/robots.inc"
include "/home/user/catkin_ws/src/swarm_stage_ros/launch/include/sensors.inc"
floorplan (name "cave" size [16.000 16.000 0.800] pose [0 0 0 0] bitmap "/home/user/
      catkin_ws/src/swarm_stage_ros/launch/map/image.png")
turtlebot (name "r0" pose [0.00 0.00 0 0 ]  color "red")
roomba (name "r1" pose [-0.30 0.30 0 0 ]  color "blue")
roomba (name "r2" pose [-0.30 -0.30 0 0 ]  color "blue")
roomba (name "r3" pose [-0.60 0.60 0 0 ]  color "blue")
roomba (name "r4" pose [-0.60 -0.60 0 0 ]  color "blue")
roomba (name "r5" pose [-0.90 0.90 0 0 ]  color "blue")
roomba (name "r6" pose [-0.90 -0.90 0 0 ]  color "blue")
roomba (name "r7" pose [-1.20 1.20 0 0 ]  color "blue")
roomba (name "r8" pose [-1.20 -1.20 0 0 ]  color "blue")
roomba (name "r9" pose [-1.50 1.50 0 0 ]  color "blue")
roomba (name "r10" pose [-1.50 -1.50 0 0 ]  color "blue")
```

The *controller.launch* file is generated to run the environment that has just been created:

```
<?xml version="1.0"?>
<launch>
  <!-- ROSCORE -->
    <master auto="start"/>
  <!--  Stage Simulator -->
    <node pkg="stage_ros" type="stageros" name="stageros" args="$(find swarm_stage_ros)/
          launch/custom.world" respawn="true" output="log"/>
</launch>
```

The *swarm_stage.launch* requires as a parameter the arguments passed by the *swarm.yaml* file:

```
<launch>
  <node name="swarm_stage" pkg="swarm_stage_ros" type="swarm_stage" respawn="false"
        output="screen">
   <rosparam file="$(find swarm_stage_ros)/yaml/swarm.yaml" command="load" />
  </node>
</launch>
```

The *swarm.yaml* file, as well as its parameters, are explained in Sect. 3.2.

3.2 Configuration

The configuration of the desired simulation scenario is accomplished using the *swarm.yaml* parameterization file, as presented in Listing 1.1.

Listing 1.1 Example of a *swarm.yaml* parameter file

```
swarm:
    formation: 'square' #no, wedge, square or diamond
    robots: 0 #used to "no" formation
    random_colors: 'yes' #yes or no
    side: 3 #number of robots per formation side
    completed: 'no' #when used formation
    leader: 'no'  #yes or no
    leader_sonar_sensor: 3 #0, 1  or 3
    leader_laser_sensor: 'no' #yes or no
    other_robots_sonar_sensor: 0 #0, 1  or 3
    other_robots_laser_sensor: 'no' #yes or no
    scenario: 'stadium' #stadium, forest or hospital
    position_controller: 'yes' #yes or no
    reference: 'relative' #relative(odom) or global
    publish_tf: 'yes' #yes or no, when reference = 'relative'

run: 'yes'
```

3.2.1 Parameter: *formation*

The *formation* parameter allows one of four possible values to be chosen: *no*, *wedge*, *square* or *diamond*.

If *no* is chosen, it is understood that no specific formation will be used. However, the desired number of robots in the *robots* parameter will be required.

Consider the following parameterization in Listing 1.2. The scenario generated with the number of robots equal to 10 is presented in Fig. 5.

Listing 1.2 Example of *no* formation and *10* robots

```
swarm:
    formation: 'no' #no, wedge, square or diamond
    robots: 10 #used to "no" formation
```

If the number of robots is 100, then the parameterization should be like that presented in Listing 1.3. The scenario generated is presented in Fig. 6.

Listing 1.3 Example of *no* formation and *100* robots

```
swarm:
    formation: 'no' #no, wedge, square or diamond
    robots: 100 #used to "no" formation
```

Now, consider *formation* = *'wedge'*. In this case, the value for the *side* parameter must be entered. Figure 7 and Fig. 8 present two *wedge* formations with *side* = *6*

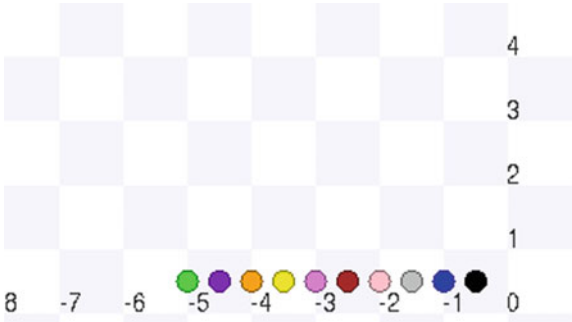


Fig. 5 Scenario generated with the parameter formation= 'no' and robots = 10

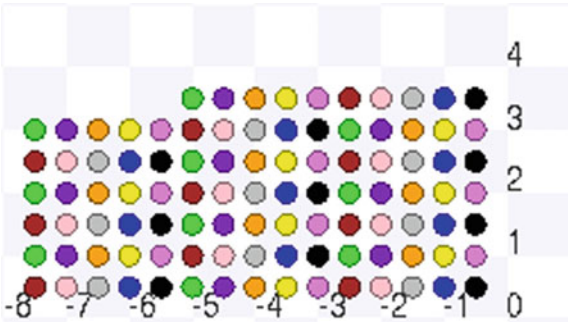


Fig. 6 Scenario generated with the parameter formation = 'no' and robots = 100

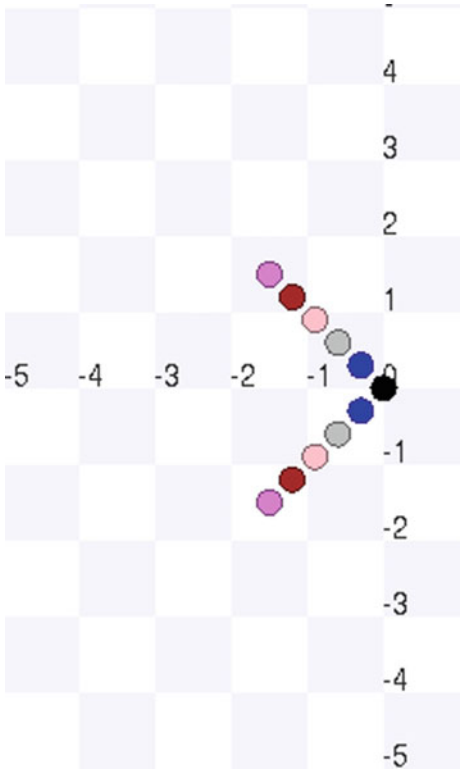


Fig. 7 Scenario generated with the parameter formation = 'wedge' and side = 6

and *side* = *15*, respectively. The parameterization is displayed in Listing 1.4 for *side* = *6*.

Listing 1.4 Example of *wedge* formation and *side* = *6*

```
swarm:
    formation: 'wedge' #no, wedge, square or diamond
    side: 6 # number of robots per formation side
```

For the *square* formation, the *side* parameter also needs to be determined. Figures 9 and 10 present two situations where the *side* parameter is 6 and 15, respectively.

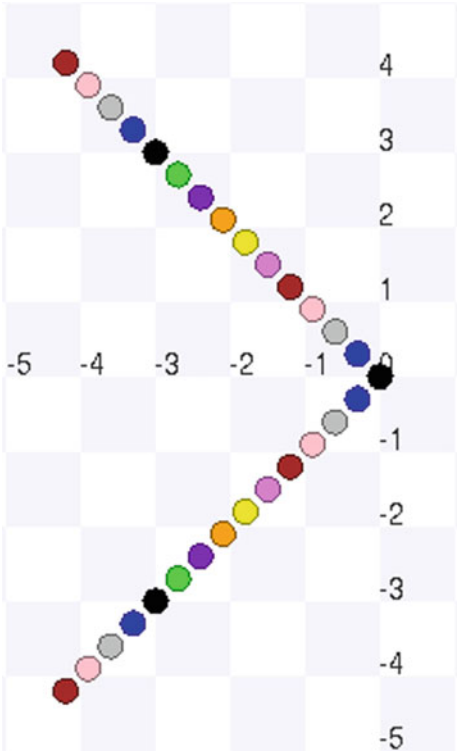


Fig. 8 Scenario generated with the parameter formation = 'wedge' and side = 15

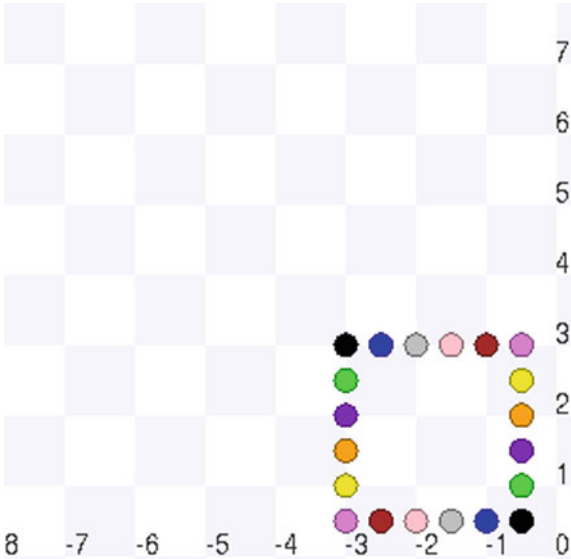


Fig. 9 Scenario generated with the parameter formation = 'square' and side = 6

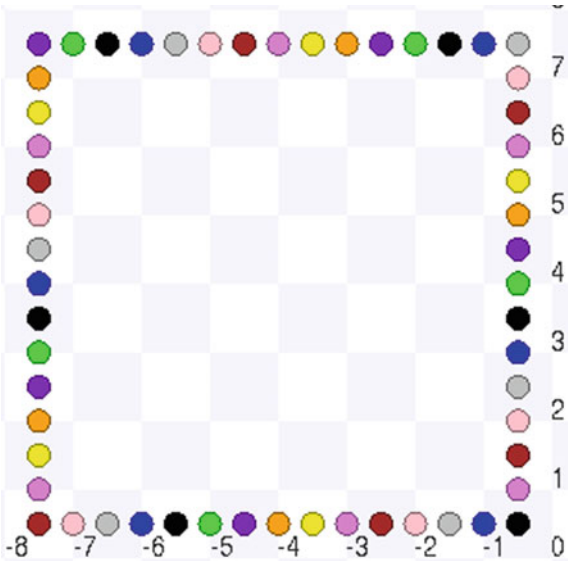


Fig. 10 Scenario generated with the parameter formation = 'square' and side = 15

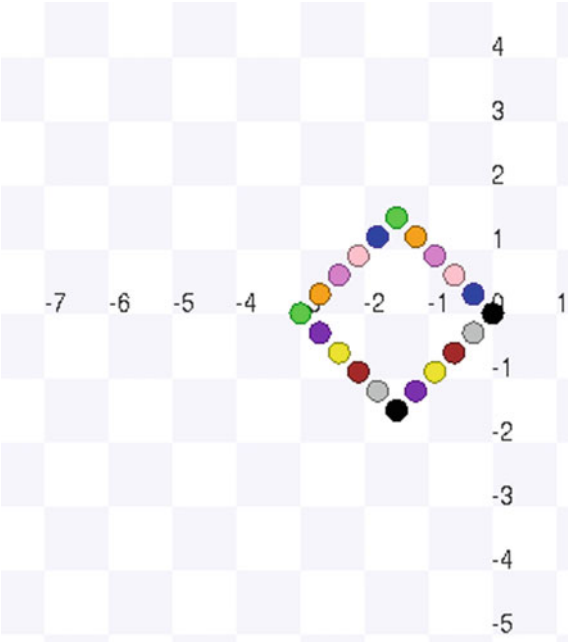


Fig. 11 Scenario generated with the parameter formation = 'diamond' and side = 6

Last, there is the *diamond* formation. For this formation, it is also necessary to determine the number of robots per side. Figures 11 and 12 present the *diamond* formation with *side* = *6* and *side* = *15*, respectively.

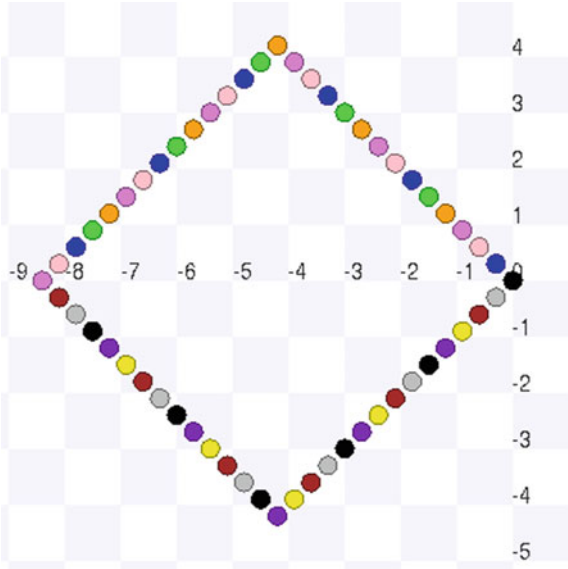


Fig. 12 Scenario generated with the parameter formation = 'diamond' and side = 15

3.2.2 Parameter: *robots*

The *robots* parameter, as shown in Figs. 5 and 6, is used when the *formation* parameter is *no*. This indicates the number of robots to be used in the scenario.

3.2.3 Parameter: *random_colors*

The *random_colors* parameter is used to indicate whether the colors of the robots are to be diversified. The accepted values for this parameter are *yes* or *no*. Figures 13 and 14 present, respectively, two examples: one with *random_colors* = *'yes'* and one with *random_colors* = *'no'*.

The parameterization used for the scenario of Fig. 14 is displayed in Listing 1.5.

Listing 1.5 Example of *random_colors* parameter

```
swarm:
   random_colors: 'no' #yes or no
```

3.2.4 Parameter: *side*

The *side* parameter, as shown in Figs. 7, 8, 9, 10, 11 and 12, specifies the number of robots per side of the formations *wedge*, *square*, and *diamond*. For these formations, the *robots* parameter is ignored.

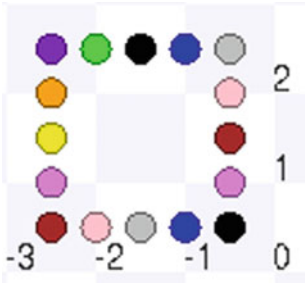


Fig. 13 Scenario generated with the parameter random_colors = 'yes'

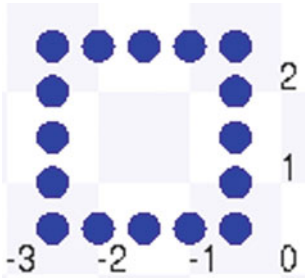


Fig. 14 Scenario generated with the parameter random_colors = 'no'

3.2.5 Parameter: *completed*

Like the *side* parameter, the *completed* parameter is used when the *formation* parameter is one of the following options: *wedge*, *square*, or *diamond*. The allowed values for the completed parameter are *yes* or *no*. Figures 15, 16 and 17 show the *wedge*, *square*, and *diamond* formations, with parameter *completed* = *'yes'*, respectively. Figures 7, 9 and 11 show the *wedge*, *square*, and *diamond* formations with the parameter *completed* = *'no'*.

The parameterization used for the scenario of Fig. 15 is presented in Listing 1.6.

Listing 1.6 Example of *completed* parameter

```
swarm:
   formation: 'wedge' #no, wedge, square or diamond
   completed: 'yes' #when used formation
```

3.2.6 Parameter: *leader*

When the *formation* parameter is one of the options of *wedge*, *square*, or *diamond*, a leader may be necessary. The allowed values for the *leader* parameter are *yes* or *no*. Figures 18, 19, and 20 present examples of the respective *wedge*, *square*, and *diamond* formations, with the *leader* = *'yes'* parameter. As can be seen in Fig. 18–20, the leader has a format different from the other robots and always has the color red. The *swarm.yaml* file for Fig. 18 was configured as Listing 1.7.

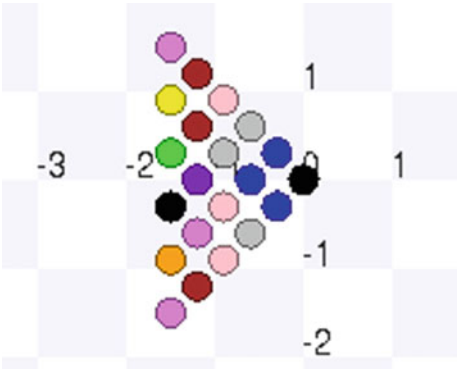


Fig. 15 Scenario generated with the parameter completed = 'yes' and formation = 'wedge'

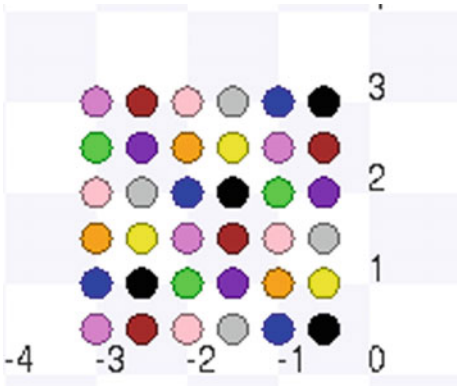
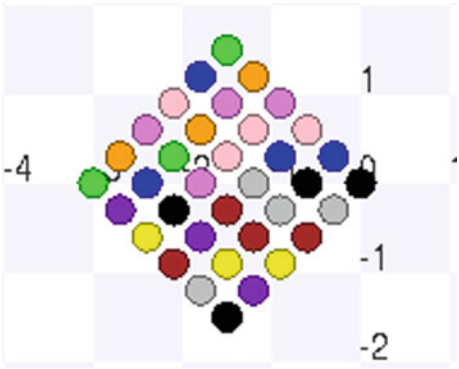


Fig. 17 Scenario generated with the parameter completed = 'yes' and formation = 'diamond'



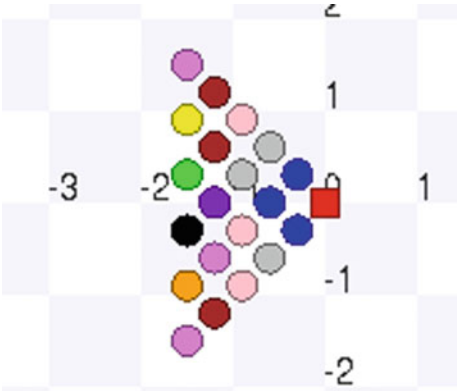


Fig. 18 Scenario generated with the parameter leader = 'yes' and formation = 'wedge'

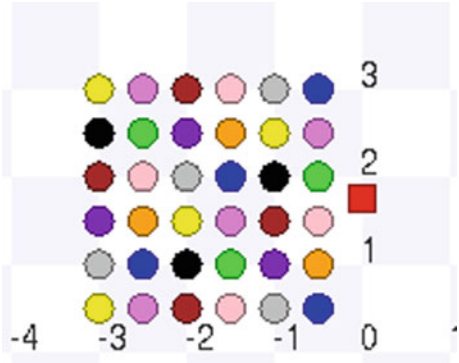
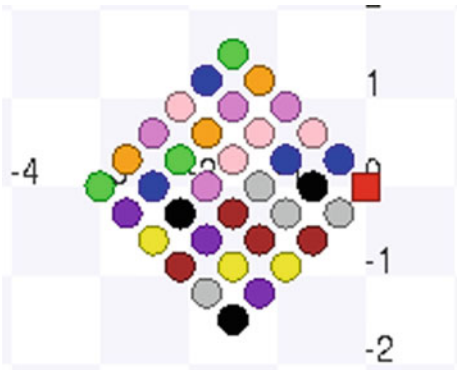


Fig. 20 Scenario generated with the parameter leader = 'yes' and formation = 'diamond'



Listing 1.7 Example of *leader* parameter

```
swarm:
    formation: 'wedge' #no, wedge, square or diamond
    leader: 'yes'  #yes or no
```

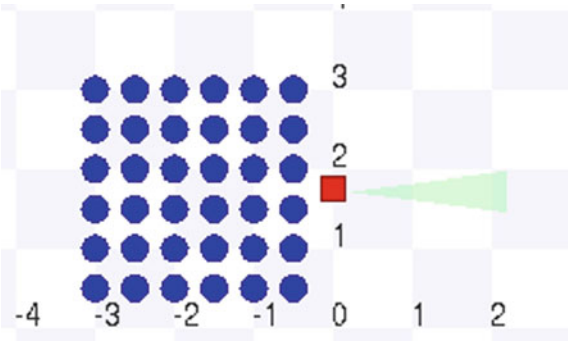


Fig. 21 Scenario generated with the parameter leader = 'yes' and leader_sonar_sensor = 1

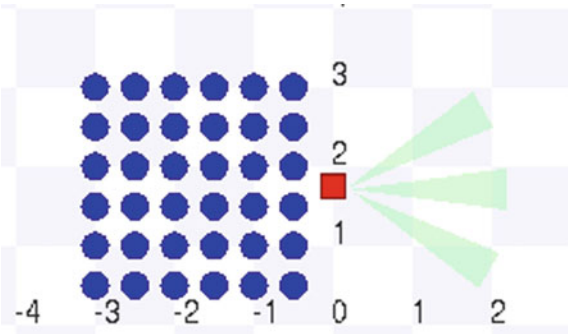


Fig. 22 Scenario generated with the parameter leader = 'yes' and leader_sonar_sensor = 3

3.2.7 Parameter: *leader_sonar_sensor*

If the *leader* parameter has a value of *yes*, then it can be configured with one or three sonar sensors. The *leader_sonar_sensor* parameter allows the value *0* to be entered when no sonar sensor will be used, *1* when one sonar sensor is used, and *3* when three sonar sensors are used for the leader robot. Figures 21 and 22 present scenarios where the leading robot has one sonar sensor and three sonar sensors, respectively. The configuration used for the scenario of Fig. 21 is presented in Listing 1.8.

Listing 1.8 Example of *leader_sonar_sensor* parameter

```
swarm:
    formation: 'square' #no,wedge,square or diamond
    leader: 'yes'  #yes or no
    leader_sonar_sensor: 1 #0, 1  or 3
```

3.2.8 Parameter: *leader_laser_sensor*

If the *leader* parameter is set to *yes*, it is also possible to choose to use a laser sensor. In this case, the *leader_laser_sensor* parameter only allows the value *no* to indicate that a laser sensor will not be used, or *yes* to indicate that a laser sensor will be used. It is not possible to use a sonar and laser sensor at the same time, and it is only necessary

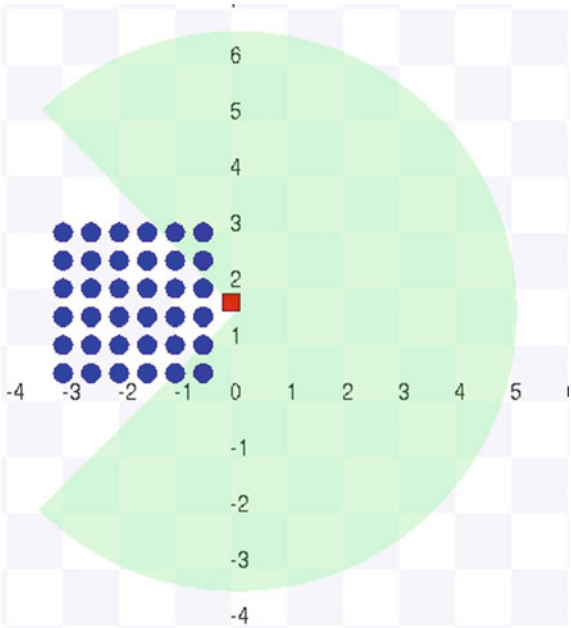


Fig. 23 Scenario generated with the parameter leader = 'yes' and leader_laser_sensor = 1

to choose one of the two types of sensors. Figure 23 presents a scenario where the leader has a laser sensor. The *swarm.yaml* file for Fig. 23 is shown in Listing 1.9.

Listing 1.9 Example of *leader_laser_sensor* parameter

```
swarm:
    formation: 'square' #no,wedge,square or diamond
    leader: 'yes'  #yes or no
    leader_laser_sensor: 'yes' #yes or no
```

3.2.9 Parameter: *other_robots_sonar_sensor*

The *other_robots_sonar_sensor* parameter allows the other robots of *wedge*, *square*, or *diamond* formations to have one sonar sensor or three sonar sensors. This parameter is also used when the *formation* parameter is *no*. The value *0* is still allowed to indicate that no sonar sensor will be used in the robots. Figure 24 presents a scenario with the other robots having one sonar sensor, and Fig. 25 presents the same scenario with the other robots having three sonar sensors. The configuration used for the scenario in Fig. 24 is presented in Listing 1.10.

Listing 1.10 Example of *other_robots_sonar_sensor* parameter

```
swarm:
    other_robots_sonar_sensor: 1 #0, 1  or 3
```

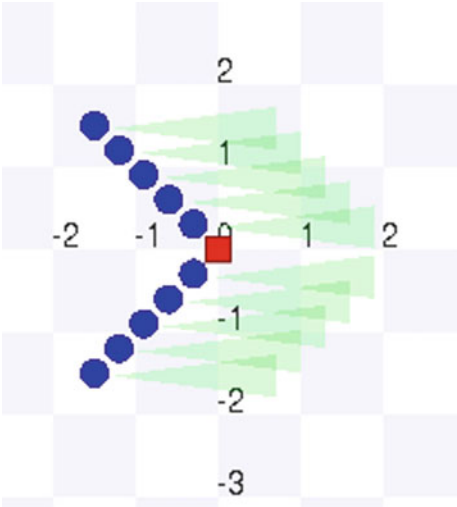


Fig. 24 Scenario generated with the parameter other_robots_sonar_sensor = 1

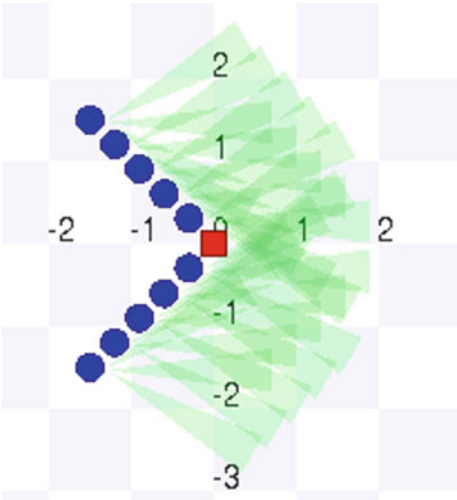


Fig. 25 Scenario generated with the parameter other_robots_sonar_sensor = 3

3.2.10 Parameter: *other_robots_laser_sensor*

The parameter *other_robots_laser_sensor* should be used when it is necessary to use a laser sensor in the other robots of *wedge*, *square*, or *diamond* formations, or in robots when no formation is used. It is not possible to use sonar and laser sensors at the same time in robots. Figure 26 presents a scenario where the other robots have a laser sensor in the wedge formation. The configuration of the *swarm.yaml* file is as in Listing 1.11.

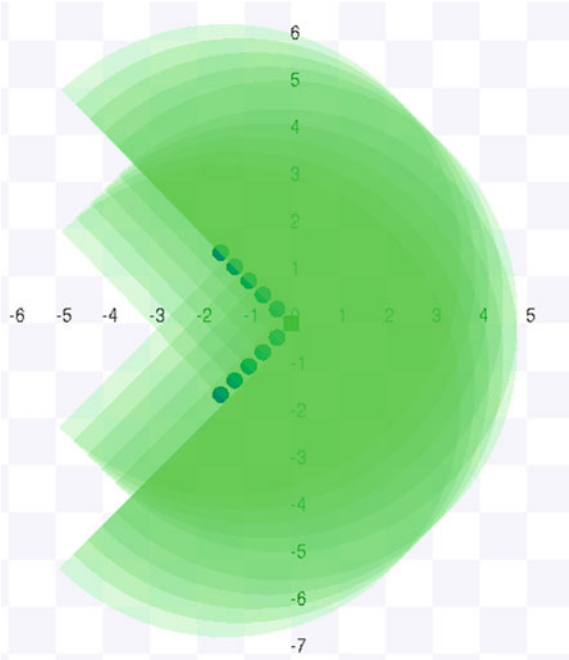


Fig. 26 Scenario generated with the parameter other_robots_laser_sensor = 'yes'

Listing 1.11 Example of *other_robots_laser_sensor* parameter

```
swarm:
    other_robots_laser_sensor: 'yes' #yes or no
```

3.2.11 Parameter: *scenario*

The *scenario* parameter allows the user to choose which image is most suitable for the task that the robots need to solve. The allowed values are *stadium*, *forest*, or *hospital*. The generated image is proportional to the number of robots chosen.

If the *stadium* value is chosen, a blank delimited image is generated, as shown in Fig. 27. The configuration for this scenario is elaborated in Listing 1.12.

Listing 1.12 Example of *scenario* = *'stadium'* parameter

```
swarm:
    scenario: 'stadium' #stadium, forest or hospital
```

If *forest* is chosen, some circles will be drawn around the formation to represent the trees. The larger the formation, the greater the number of circles drawn around it. Figure 28 presents the scenario generated for a smaller formation, and Fig. 29 presents the scenario generated for a larger formation. The *swarm.yaml* file for Figs. 28 and 29 is shown in Listing 1.13.

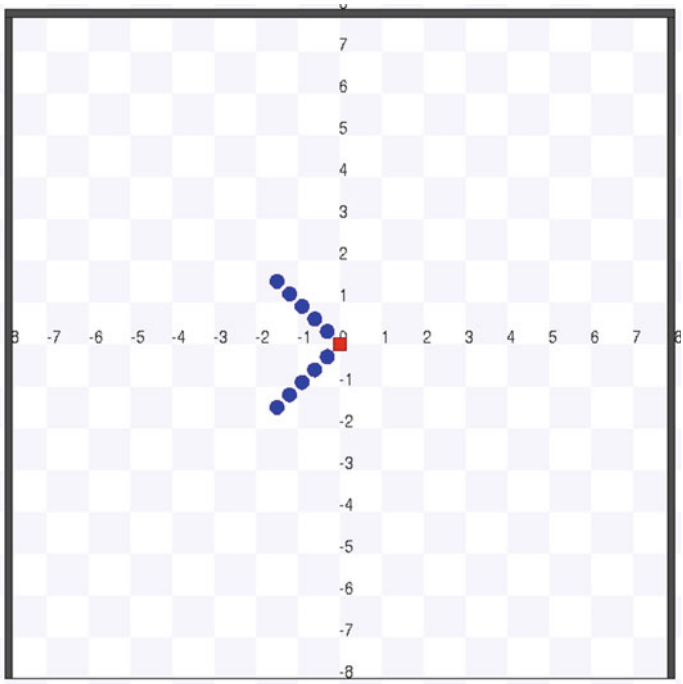


Fig. 27 Scenario generated with the parameter scenario = 'stadium'

Listing 1.13 Example of *scenario = 'forest'* parameter

```
swarm:
    scenario: 'forest' #stadium, forest or hospital
```

Finally, the *hospital* scenario is one of the options. In this scenario, rooms are drawn around the formation. The size of the rooms designed is proportional to the size of the formation. Figures 30 and 31 present, respectively, the scenario generated for a smaller formation and a larger formation. The configuration for the scenarios of Figs. 30 and 31 is shown in Listing 1.14.

Listing 1.14 Example of *scenario = 'hospital'* parameter

```
swarm:
    scenario: 'hospital' #stadium, forest or hospital
```

It is important to note that the user cannot choose the number of obstacles, or the size and number of rooms, in the *forest* and *hospital* scenarios, respectively. This is done automatically.

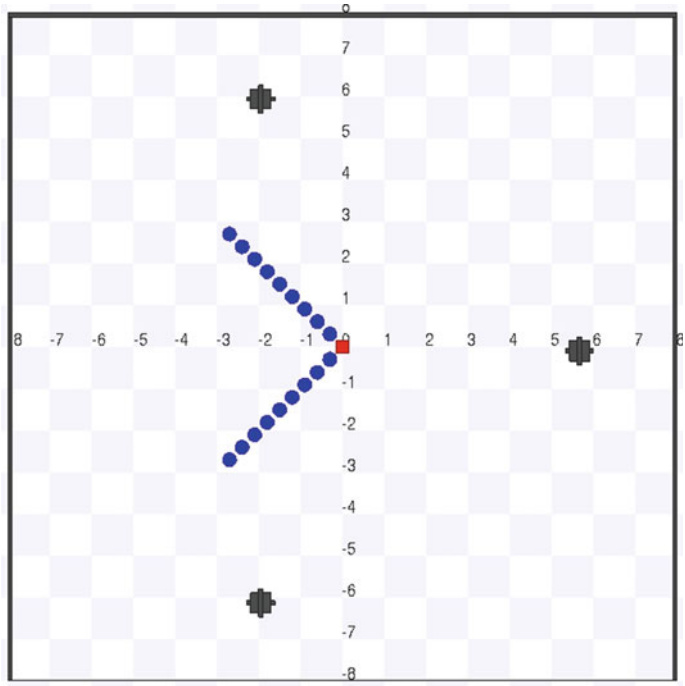


Fig. 28 Scenario generated with the parameter scenario = 'forest' and side = 10

3.2.12 Parameter: *position_controller*

The *position_controller* parameter accepts two values: *yes* or *no*. If it is *no*, then only the *controller.launch* file is generated at the end of processing, as explained at the beginning of the section:

```
<?xml version="1.0"?> <launch>
  <!-- ROSCORE -->
    <master auto="start"/>
  <!--  Stage Simulator -->
    <node pkg="stage_ros" type="stageros" name="stageros" args="$(find swarm_stage_ros)/
          launch/custom.world" respawn="true" output="log"/>
</launch>
```

If the value of the *position_controller* parameter is *yes*, then a node is created for each robot in the formation to call the program *posctrl* that will receive as an argument the name of the robot. This program should control the position of each robot. An example implementation of this program is presented in Sect. 4.1. In addition, at the end of the *controller.launch* file, a node is created to call the *controller* program, which will receive as an argument the number of robots in the formation. This program should pass the positions of each of the robots. An example implementation of this program is also presented in Sect. 4.1. The tree of ROS nodes generated by ROS is shown in Fig. 32.

An example of a *controller.launch* generated for the scenario of Fig. 33 is as follows:

```
<?xml version="1.0"?>
<launch>
  <!-- ROSCORE -->
    <master auto="start"/>
  <!--  Stage Simulator -->
    <node pkg="stage_ros" type="stageros" name="stageros" args="$(find swarm_stage_ros)/
          launch/custom.world" respawn="true" output="log"/>
        <node name="robot0" pkg="swarm_stage_ros" type="posctrl" args="robot_0" output="
              screen"/>
        <node name="robot1" pkg="swarm_stage_ros" type="posctrl" args="robot_1" output="
              screen"/>
        <node name="robot2" pkg="swarm_stage_ros" type="posctrl" args="robot_2" output="
              screen"/>
        <node name="robot3" pkg="swarm_stage_ros" type="posctrl" args="robot_3" output="
              screen"/>
        <node name="robot4" pkg="swarm_stage_ros" type="posctrl" args="robot_4" output="
              screen"/>
        <node name="robot5" pkg="swarm_stage_ros" type="posctrl" args="robot_5" output="
              screen"/>
        <node name="robot6" pkg="swarm_stage_ros" type="posctrl" args="robot_6" output="
              screen"/>
        <node name="robot7" pkg="swarm_stage_ros" type="posctrl" args="robot_7" output="
              screen"/>
        <node name="robot8" pkg="swarm_stage_ros" type="posctrl" args="robot_8" output="
              screen"/>
        <node name="robot9" pkg="swarm_stage_ros" type="posctrl" args="robot_9" output="
              screen"/>
        <node name="robot10" pkg="swarm_stage_ros" type="posctrl" args="robot_10" output
              ="screen"/>
        <node name="robot11" pkg="swarm_stage_ros" type="posctrl" args="robot_11" output
              ="screen"/>
        <node name="robot12" pkg="swarm_stage_ros" type="posctrl" args="robot_12" output
              ="screen"/>
        <node name="robot13" pkg="swarm_stage_ros" type="posctrl" args="robot_13" output
              ="screen"/>
        <node name="robot14" pkg="swarm_stage_ros" type="posctrl" args="robot_14" output
              ="screen"/>
        <node name="robot15" pkg="swarm_stage_ros" type="posctrl" args="robot_15" output
              ="screen"/>
        <node name="controller" pkg="swarm_stage_ros" type="controller" args="16" output
              ="screen"/>
</launch>
```

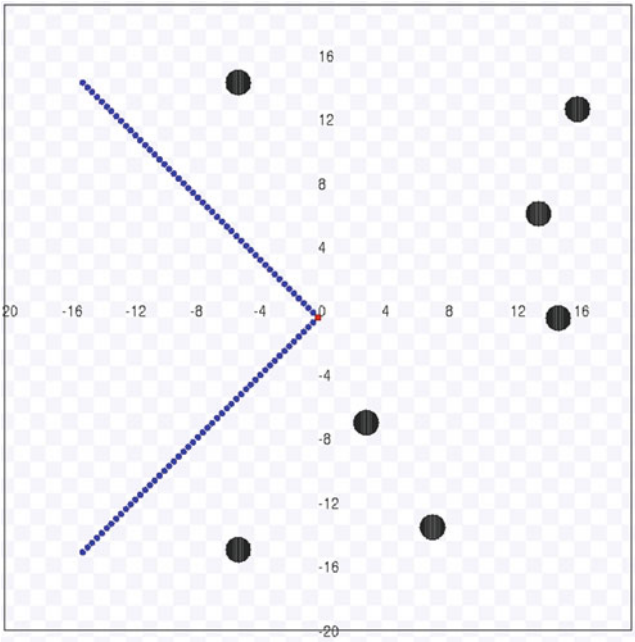


Fig. 29 Scenario generated with the parameter scenario = ‘forest’ and side = 50

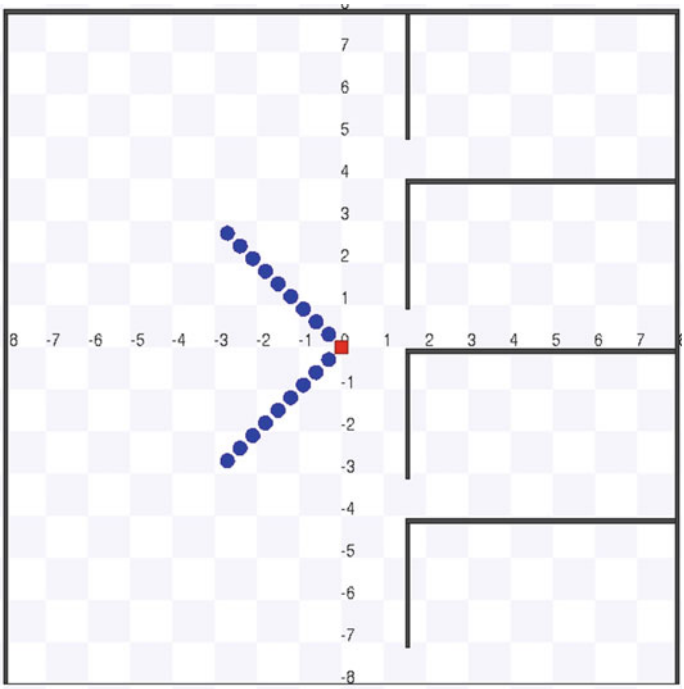


Fig. 30 Scenario generated with the parameter scenario = 'hospital' and side = 10

The swarm.yaml file for the scenario in Fig. 33 is presented in Listing 1.15.

Listing 1.15 Example of *position_controller* parameter

```
swarm:
   position_controller: 'yes' #yes or no
```

3.2.13 Parameter: *reference*

The *reference* parameter allows a choice of whether the position of the robot will be relative, that is, based on the odometry or global position of the robot relative to the point of origin of the map. The possible values for this parameter are *relative* or *global*. The scenario for both values is the same; what changes is the *custom.world* file. If the *relative* value is chosen, then for the leader, the robot *turtlebot* is used. For the other robots, the *roomba* is used, where the field *odom_error* specifies how the odometry is obtained, as follows:

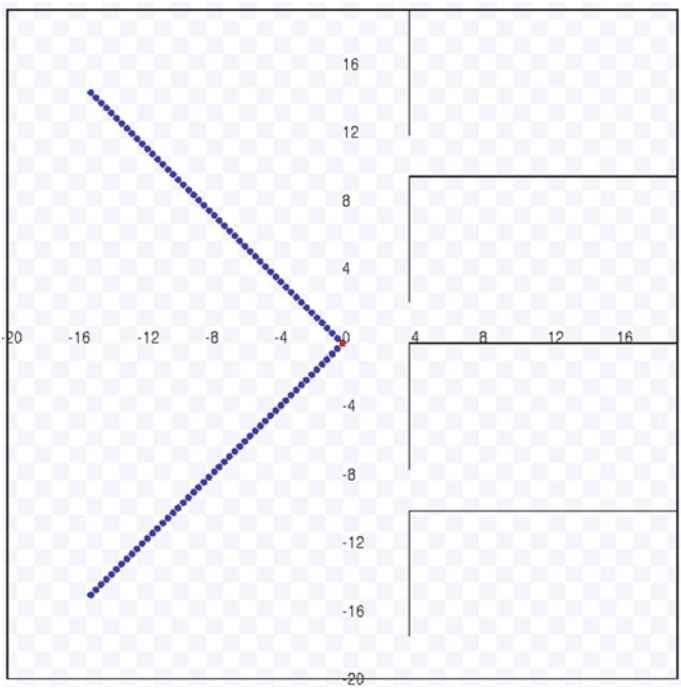


Fig. 31 Scenario generated with the parameter scenario = ‘hospital’ and side = 50

```
define turtlebot position (
  pose [ 0.0 0.0 0.0 0.0 ]
  odom_error [0.03 0.03 999999 999999 999999 0.02]
  size [ 0.3 0.3 0.10 ]
  origin [ 0.0 0.0 0.0 0.0 ]
  gui_nose 1
  drive "diff"
  color "gray"
)
define roomba position
(
  size [0.33 0.33 0.1]
  # this block approximates the circular shape of a Roomba
  block(
    points 16
    point[0] [ 0.225 0.000 ]
    point[1] [ 0.208 0.086 ]
    point[2] [ 0.159 0.159 ]
    point[3] [ 0.086 0.208 ]
    point[4] [ 0.000 0.225 ]
    point[5] [ -0.086 0.208 ]
    point[6] [ -0.159 0.159 ]
    point[7] [ -0.208 0.086 ]
    point[8] [ -0.225 0.000 ]
    point[9] [ -0.208 -0.086 ]
    point[10] [ -0.159 -0.159 ]
    point[11] [ -0.086 -0.208 ]
    point[12] [ -0.000 -0.225 ]
    point[13] [ 0.086 -0.208 ]
    point[14] [ 0.159 -0.159 ]
    point[15] [ 0.208 -0.086 ]
    z [0 0.1]
  )
  # odometry error model parameters,
  # only used if localization is set to "odom"
  odom_error [0.03 0.03 0.00 0.05]
  color "gray50"
)
```

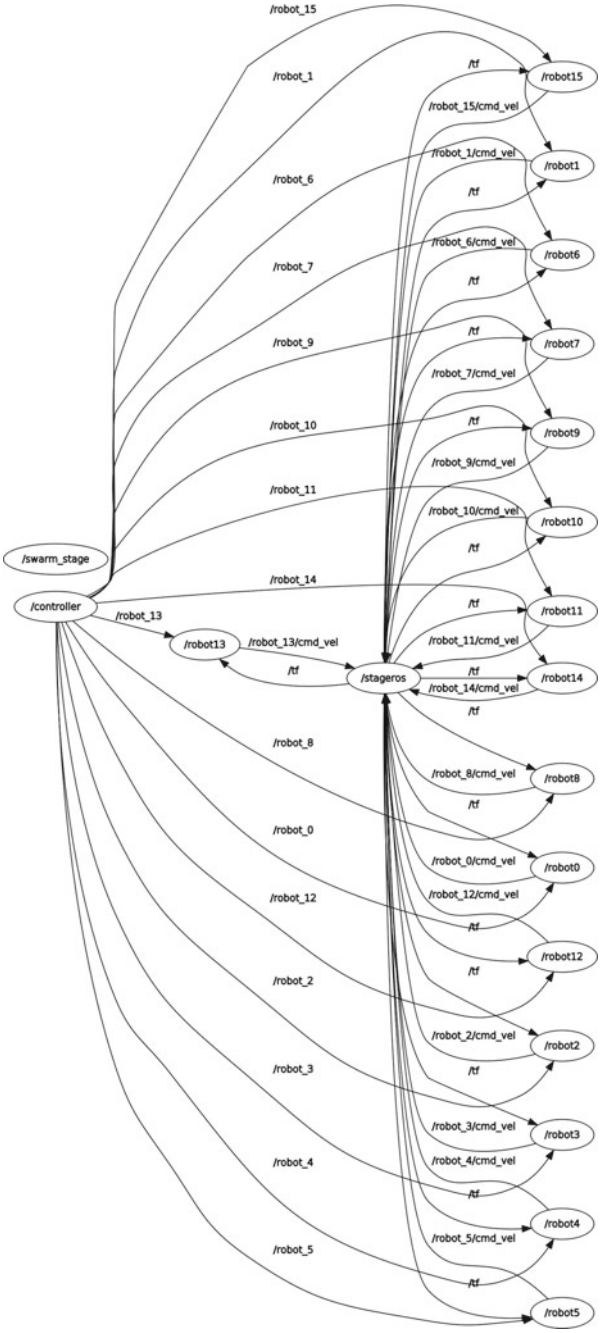


Fig. 32 The tree of ROS nodes generated by ROS to Fig. 33

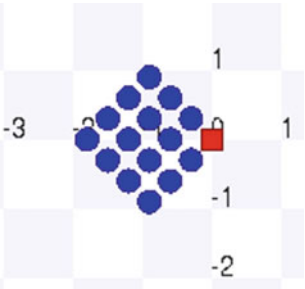


Fig. 33 Scenario generated with the parameter position_controller = 'true'

If the *reference* parameter has the *global* value, then the robot *turtlebot_gps* is used for the leader and *roomba_gps* is used for the other robots, where the field *localization* specifies the global position:

```
define turtlebot_gps position (
  pose [ 0.0 0.0 0.0 0.0 ]
  localization "gps"
      localization_origin [0 0 0 0]
  size [ 0.3 0.3 0.10 ]
  origin [ 0.0 0.0 0.0 0.0 ]
  gui_nose 1
  drive "diff"
  color "gray"
)
define roomba_gps position
(
  size [0.33 0.33 0.1]
  # this block approximates the circular shape of a Roomba
  block(
    points 16
    point[0] [ 0.225 0.000 ]
    point[1] [ 0.208 0.086 ]
    point[2] [ 0.159 0.159 ]
    point[3] [ 0.086 0.208 ]
    point[4] [ 0.000 0.225 ]
    point[5] [ -0.086 0.208 ]
    point[6] [ -0.159 0.159 ]
    point[7] [ -0.208 0.086 ]
    point[8] [ -0.225 0.000 ]
    point[9] [ -0.208 -0.086 ]
    point[10] [ -0.159 -0.159 ]
    point[11] [ -0.086 -0.208 ]
    point[12] [ -0.000 -0.225 ]
    point[13] [ 0.086 -0.208 ]
    point[14] [ 0.159 -0.159 ]
    point[15] [ 0.208 -0.086 ]
    z [0 0.1]
  )
  localization "gps"
      localization_origin [0 0 0 0]
  color "gray50"
)
```

Listing 1.16 Example of *reference* parameter

```
swarm:
   reference: 'relative' #relative(odom) or global
```

For the parameterization in Listing 1.16, the following *custom.world* file is generated:

```
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/map.inc"
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/robots.inc"
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/sensors.inc"
floorplan (name "cave" size [16.000 16.000 0.800] pose [0 0 0 0]
bitmap
"/home/user/catkin_ws/src/swarm_stage_ros/launch/map/image.png")
turtlebot (name "r0" pose [0.00 0.00 0 0 ]  color "red") roomba
(name "r1" pose [-0.30 0.30 0 0 ]  color "blue") roomba (name "r2"
pose [-0.30 -0.30 0 0 ]  color "blue") roomba (name "r3" pose [-0.60
0.60 0 0 ]  color "blue") roomba (name "r4" pose [-0.60 -0.60 0 0 ]
color "blue") roomba (name "r5" pose [-0.90 0.90 0 0 ]  color
"blue") roomba (name "r6" pose [-0.90 -0.90 0 0 ]  color "blue")
roomba (name "r7" pose [-1.20 0.60 0 0 ]  color "blue") roomba (name
"r8" pose [-1.20 -0.60 0 0 ]  color "blue") roomba (name "r9" pose
[-1.50 0.30 0 0 ]  color "blue") roomba (name "r10" pose [-1.50
-0.30 0 0 ]  color "blue") roomba (name "r11" pose [-1.80 0.00 0 0 ]
color "blue")
```

If *reference* = *'global'* is chosen, then the following *custom.world* file is generated:

```
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/map.inc"
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/robots.inc"
include
"/home/user/catkin_ws/src/swarm_stage_ros/launch/include/sensors.inc"
floorplan (name "cave" size [16.000 16.000 0.800] pose [0 0 0 0]
bitmap
"/home/user/catkin_ws/src/swarm_stage_ros/launch/map/image.png")
turtlebot_gps (name "r0" pose [0.00 0.00 0 0 ]  color "red")
roomba_gps (name "r1" pose [-0.30 0.30 0 0 ]  color "blue")
roomba_gps (name "r2" pose [-0.30 -0.30 0 0 ]  color "blue")
roomba_gps (name "r3" pose [-0.60 0.60 0 0 ]  color "blue")
roomba_gps (name "r4" pose [-0.60 -0.60 0 0 ]  color "blue")
roomba_gps (name "r5" pose [-0.90 0.90 0 0 ]  color "blue")
roomba_gps (name "r6" pose [-0.90 -0.90 0 0 ]  color "blue")
roomba_gps (name "r7" pose [-1.20 0.60 0 0 ]  color "blue")
roomba_gps (name "r8" pose [-1.20 -0.60 0 0 ]  color "blue")
roomba_gps (name "r9" pose [-1.50 0.30 0 0 ]  color "blue")
roomba_gps (name "r10" pose [-1.50 -0.30 0 0 ]  color "blue")
roomba_gps (name "r11" pose [-1.80 0.00 0 0 ]  color "blue")
```

For the two options, the scene of Fig. 34 is obtained.

3.2.14 Parameter: *publish_tf*

If the *reference* parameter is *relative*, it is possible to choose between the *yes* and *no* values for the *publish_tf* parameter. If *yes*, then the *tf.launch* file is created:

```
<?xml version="1.0"?> <launch>
  <param name="pub_map_odom_transform" value="true"/>
  <param name="map_frame" value="map"/>
  <param name="base_frame" value="base_frame"/>
  <param name="odom_frame" value="odom"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_0" args="0.00 0.00 0 0 0
        0 /map robot_0/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_1" args="-0.30 0.30 0 0 0
         0 /map robot_1/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_2" args="-0.30 -0.30 0 0
        0 0 /map robot_2/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_3" args="-0.60 0.60 0 0 0
         0 /map robot_3/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_4" args="-0.60 -0.60 0 0
        0 0 /map robot_4/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_5" args="-0.90 0.90 0 0 0
         0 /map robot_5/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_6" args="-0.90 -0.90 0 0
        0 0 /map robot_6/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_7" args="-1.20 0.60 0 0 0
         0 /map robot_7/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_8" args="-1.20 -0.60 0 0
        0 0 /map robot_8/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_9" args="-1.50 0.30 0 0 0
         0 /map robot_9/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_10" args="-1.50 -0.30 0 0
         0 0 /map robot_10/odom 100"/>
  <node pkg="tf" type="static_transform_publisher" name="robot_11" args="-1.80 0.00 0 0
        0 0 /map robot_11/odom 100"/>
</launch>
```

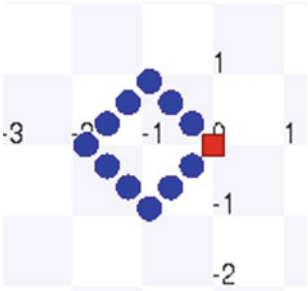


Fig. 34 Scenario generated with the parameter reference = 'relative' or reference='global'

That is, for each robot, a transformation occurs between its relative position to its overall position on the map. In addition, at the end of the *controller.launch* file, there is a call from the *tf.launch* file:

```
<?xml version="1.0"?> <launch>
  <!-- ROSCORE -->
    <master auto="start"/>
  <!--  Stage Simulator -->
    <node pkg="stage_ros" type="stageros" name="stageros" args="$(find swarm_stage_ros)/
          launch/custom.world" respawn="true" output="log"/>
        <node name="robot0" pkg="swarm_stage_ros" type="posctrl" args="robot_0" output="
              screen"/>
        <node name="robot1" pkg="swarm_stage_ros" type="posctrl" args="robot_1" output="
              screen"/>
        <node name="robot2" pkg="swarm_stage_ros" type="posctrl" args="robot_2" output="
              screen"/>
        <node name="robot3" pkg="swarm_stage_ros" type="posctrl" args="robot_3" output="
              screen"/>
        <node name="robot4" pkg="swarm_stage_ros" type="posctrl" args="robot_4" output="
              screen"/>
        <node name="robot5" pkg="swarm_stage_ros" type="posctrl" args="robot_5" output="
              screen"/>
        <node name="robot6" pkg="swarm_stage_ros" type="posctrl" args="robot_6" output="
              screen"/>
        <node name="robot7" pkg="swarm_stage_ros" type="posctrl" args="robot_7" output="
              screen"/>
        <node name="robot8" pkg="swarm_stage_ros" type="posctrl" args="robot_8" output="
              screen"/>
        <node name="robot9" pkg="swarm_stage_ros" type="posctrl" args="robot_9" output="
              screen"/>
        <node name="robot10" pkg="swarm_stage_ros" type="posctrl" args="robot_10" output
              ="screen"/>
        <node name="robot11" pkg="swarm_stage_ros" type="posctrl" args="robot_11" output
              ="screen"/>
        <node name="controller" pkg="swarm_stage_ros" type="controller" args="12" output
              ="screen"/>
        <include file="$(find swarm_stage_ros)/launch/tf.launch"/>
</launch>
```

The parameterization chosen is shown in Listing 1.17.

Listing 1.17 Example of *publish_tf* parameter

```
swarm:
   reference: 'relative' #relative(odom) or global
   publish_tf: 'yes' #yes or no
```

The scenario generated was that shown in Fig. 34, and the *tf* tree generated for this scenario is shown in Fig. 35.

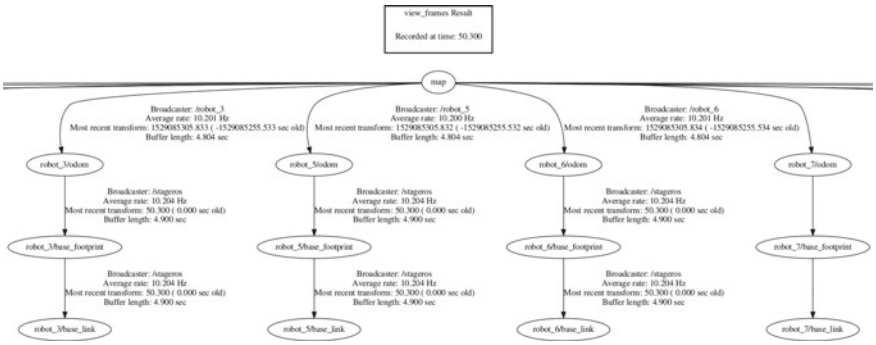


Fig. 35 The *tf* tree generated for scenario of Fig. 34

3.2.15 Parameter: *run*

The *run* parameter accepts *yes* or *no* values. This parameter indicates whether the *controller.launch* file is automatically executed at the end of the packet execution. If the value of this parameter is *no*, then it is necessary to run the following command manually:

```
roslaunch swarm_stage_ros controller.launch
```

3.3 Summary

To simplify the understanding of each parameter required in the *swarm.yaml* file, Table 1 presents a summary containing the values allowed by the parameter and a brief description. The next section presents two experiments using the *swarm_stage_ros* package.

4 Application Examples of Swarm Stage Package

This section presents two examples of the *swarm_stage_ros* package application. The first example is the position control of a *square* formation with 20 robots. The second example deals with the position control and detouring around obstacles for a *wedge* formation with 13 robots. Other examples in industrial environments can be found in the works [24, 25].

The purpose of these demonstrations is to present how the *swarm_stage_ros* package simplifies working with swarms of robots and focuses on the application rather than the required configurations.

Table 1 Parameters of the file *swarm.yaml*

Parameter	Possible values	Description
formation	no, wedge, square or diamond	Is the type of formation to be used by the swarm of robots
robots	> 0	The number of robots to be used when the type of formation chosen is “no”. For other types of formation, this parameter is ignored
random_colors	yes or no	If “yes" is chosen, then the colors of the robots will be random
side	> 0	The number of robots for each side of the chosen formation. If no formation is chosen, then this field is ignored
completed	yes or no	Used when some type of formation is chosen. When “yes” is chosen, then the formation is filled with robots.
leader	yes or no	If a formation is chosen, then an additional robot can be added as the leader
leader_sonar_sensor	0, 1 or 3	The leader may have zero, one, or three sonars
leader_laser_sensor	yes or no	A laser sensor can be added to the leader. Only one type of sensor is allowed for the leader
other_robots_sonar_sensor	0, 1 or 3	Robots can have zero, one, or three sonars
other_robots_laser_sensor	yes or no	A laser sensor can be added to the robots. Only one type of sensor is allowed for the robots
scenario	stadium, forest or hospital	The scenario for the simulation can be chosen. In the case of the forest scenario, trees are introduced as obstacles at random. In the case of the hospital, rooms are introduced
position_controller	yes or no	If the value of the position controller parameter is “yes", then a node is created for each robot of the formation, to call the program posctrl, which will receive as an argument the name of the robot. This program should control the position of each robot
reference	relative or global	Allows to choose whether the position of the robot will be relative, that is, based on the odometry or global position of the robot relative to the point of origin of the map
publish_tf	yes or no	If the reference parameter is relative, it is possible to choose between the “yes" and “no" values for the publish tf parameter. If “yes", then the tf.launch file is created and for each robot a transformation is made between its relative position to its overall position on the map

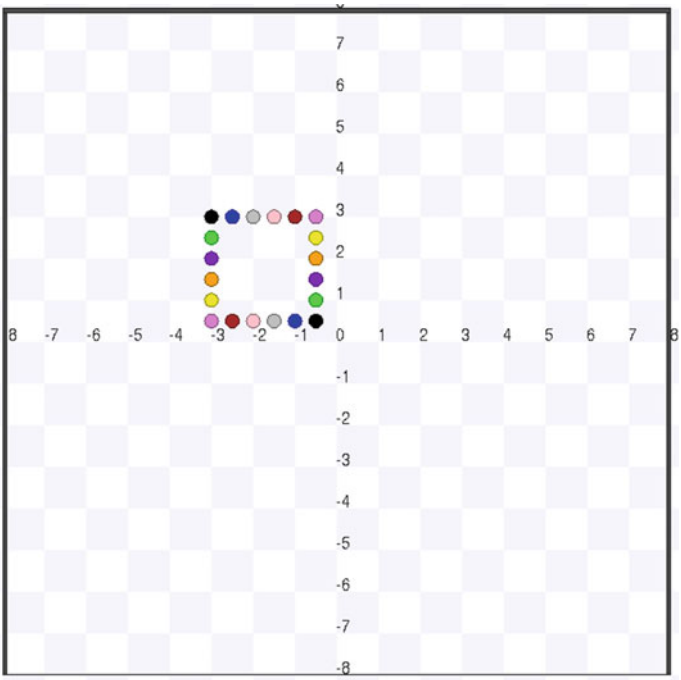


Fig. 36 Virtual experimentation scenario for the position control experiment

4.1 Formation

The first example application addresses the position control of 20 robots in a square formation. For this, the 20 robots need to form the design of a cross.

The *swarm.yaml* file that gave rise to the scenario is shown in Listing 1.18.

Listing 1.18 Parameterization for the position control environment

```
swarm:
   formation: 'square' #no, wedge, square or diamond
   robots: 0 #used to "no" formation
   random_colors: 'yes' #yes or no
   side: 6 #number of robots per side
   completed: 'no' #when used formation
   leader: 'no'  #yes or no
   leader_sonar_sensor: 0 #0, 1  or 3
   leader_laser_sensor: 'no' #yes or no
   other_robots_sonar_sensor: 0 #0, 1  or 3, use this field for formation = 'no'
   other_robots_laser_sensor: 'no' #yes or no
   scenario: 'stadium' #stadium, forest or hospital
   position_controller: 'yes' #yes or no
   reference: 'relative' #relative(odom) or global
   publish_tf: 'yes' #yes or no, use this field when reference = "relative"

run: yes
```

The scenario generated from the parameterization of Listing 1.18 is shown in Fig. 36.

As the *position_controller* parameter is *yes*, at the end of processing, the *controller.launch* file is generated with a node calling the *posctrl* program for each robot

and a node calling the *controller* program. Thus, the *controller* program is written as follows:

```
#include <ros/ros.h> #include "MainController.cpp" int main(int
argc, char** argv) {
  ros::init(argc, argv, "Controller");
  ros::NodeHandle n;
  MainController ctrl;
  int robotsNumber;

  robotsNumber = atoi(argv[1]);

  ctrl.setNode(n);
  ctrl.mainControl(robotsNumber);

  ros::spin();
  return 0;
}
```

That is, it takes as a parameter the number of robots and passes it to the *mainControl()* method. The *MainController* class is elaborated as follows:

```
1  #include <ros/ros.h>
2  #include "turtlesim/Pose.h"
3  class MainController{
4  private:
5      ros::NodeHandle n_;
6      ros::Publisher * pub_;
7      turtlesim::Pose * pos_;
8  public:
9      void setNode(ros::NodeHandle n){
10         n_ = n;
11     }
12     //Controls all actions in this class
13     void mainControl(int n){
14             char name[50];
15             int i;
16             //A vector for storing publishers for robots
17             pub_ = (ros::Publisher *)malloc(n * sizeof(ros::Publisher));
18             //Computes the positions
19             setPositions(n);
20             //Each robot on each side of the square receives its coordinates to form one side of the cross
21             for (i=0; i<(n/4); i++){
22                 snprintf(name, 50, "%s%d", "/robot_", i);
23                 pub_[i] = n_.advertise<turtlesim::Pose>(name, 1000);
24                 ros::Duration(10.0).sleep();
25                 pub_[i].publish(pos_[i]);
26             }
27             for (i=(n/4)-1; i>=0; i--){
28                 snprintf(name, 50, "%s%d", "/robot_", i+(n/4)*3);
29                 pub_[i+(n/4)*3] = n_.advertise<turtlesim::Pose>(name, 1000);
30                 ros::Duration(10.0).sleep();
31                 pub_[i+(n/4)*3].publish(pos_[i+(n/4)*3]);
32             }
33             for (i=(n/4)-1; i>=0; i--){
34                 snprintf(name, 50, "%s%d", "/robot_", i+(n/4));
35                 pub_[i+(n/4)] = n_.advertise<turtlesim::Pose>(name, 1000);
36                 ros::Duration(10.0).sleep();
37                 pub_[i+(n/4)].publish(pos_[i+(n/4)]);
38             }
39             for (i=(n/4)-1; i>=0; i--){
40                 snprintf(name, 50, "%s%d", "/robot_", i+(n/4)*2);
41                 pub_[i+(n/4)*2] = n_.advertise<turtlesim::Pose>(name, 1000);
42                 ros::Duration(10.0).sleep();
43                 pub_[i+(n/4)*2].publish(pos_[i+(n/4)*2]);
44             }
45     }
46 };
```

Therefore, the *mainControl()* method of the *MainController* class counts the *setPositions()* method (line 19) to calculate the position of each robot to form the cross. After that, the *mainControl()* method executes a for (lines 21–44) to each side of the robot to form the cross so that there are no collisions between the robots. Thus, each side of the square formation forms an edge of the resulting cross. It is also important to note that the position of each robot is published in a topic with the name of the robot, for example, /robot_0.

Finally, there is the implementation of the *posctrl* program. It is written as follows:

```
#include "ros/ros.h" #include <sstream> #include
<tf/transform_listener.h> #include "string.h" #include
"turtlesim/Pose.h" #include "PositionController.cpp"

int main(int argc, char** argv)
{
    std::string nodeName;
    std::stringstream ss;
    ss << argv[1];
    nodeName = ss.str();

    ros::init(argc, argv, nodeName);
    ros::NodeHandle n;
    PositionController posctrl;

    posctrl.setNode(n);
    posctrl.setTopics(nodeName);
    posctrl.mainControl();

    ros::spin();
    return 0;
}
```

That is, the *posctrl* program needs the *PositionController* class. This class has the *mainControl()* method, which subscribes to the topic with the name of the robot, for example, /robot_0, and publishes in its *cmd_vel* topic. Upon receiving a message, the *subCallback()* method is invoked. This method is responsible for receiving the desired position to which the robot should move. For this, it calculates the linear and angular distance from the point where the robot is currently on the map. With these values, it is possible to calculate the linear and angular velocities that the robot needs to execute to reach its destination.

To run this example, you must use the *swarm.yaml* file from Listing 1.18 and run the following command line:

```
$ roslaunch swarm_stage_ros swarm_stage.launch
```

The result obtained is shown in Fig. 37.

4.2 Obstacle Avoidance

This second example proposes a navigation method focused on multirobot systems with the cooperative interaction between its agents. The motivation is the employment of a group of mobile robots to execute cleaning services in an industrial plant with specific group requirements, aiming to maintain the agents' formation to expand the cleaning area. The group is composed of 13 vacuum cleaner robots, where the leader is equipped with a 2D laser scanner and followers do not have sensors. An intelligent fuzzy controller is designed to regulate the pose (position and orientation) of each robot, and to control the obstacle detouring behavior of the leader robot.

4.2.1 Multi-robot Formation to Cleaning Tasks

An automatic cleaning service is commonly executed by vacuum cleaner robots, which are simple agents composed of primitive perception systems. Thus, these

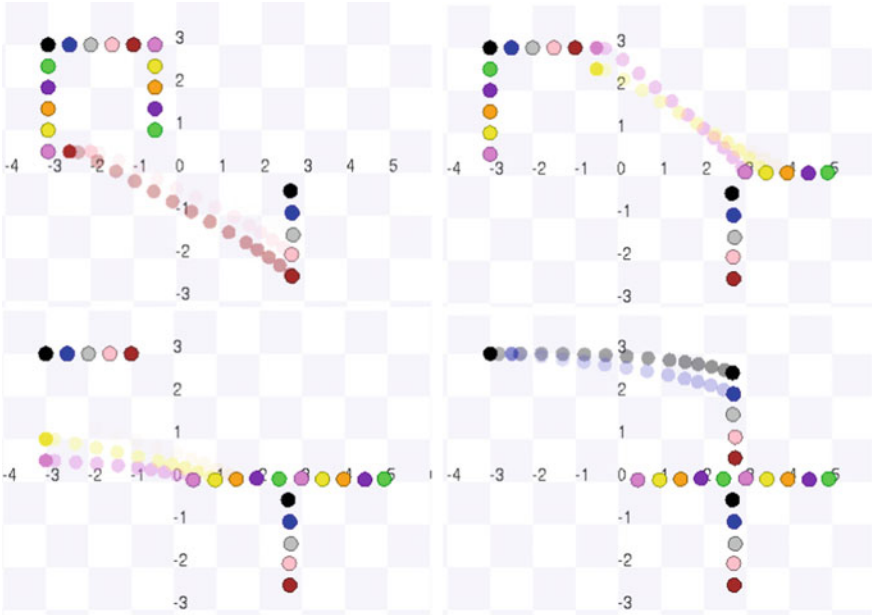


Fig. 37 Formation square becoming a cross

capabilities of detecting, avoiding and overcoming obstacles is very limited, allowing only retreat behavior. A particular class of vacuum cleaner is equipped with perception systems (such as a 2D laser scanner) with enable intelligent behaviors. However, these features significantly increase the robot's cost, making it impossible to apply to all agents.

The formation strategy adopted for the multirobot system is the wedge $\wedge$, where the apex is the robot leader and the followers are designed as the wall, with one pair of robots at each level. This format was selected so that the dirt that is not absorbed at one level can be pushed to the next level by the rotational movement of the brushes, always to the outside of the formation, as seen in Fig. 38.

The leader is responsible for specifying the formation motion, aiming to complete the cleaning task and avoiding obstacles. Followers must maintain the formation by pursuing the leader and staying inside the range of the leader's laser scanner.

4.2.2 Multi-robot Navigation with Formation Requirements

The position of the follower robots is designed based on the global position of the leader robot, aiming to maintain formation in a $\wedge$ shape, as shown in Fig. 39. The follower robots are organized at an angle of 130° about the leader, where the first level after the leader has 0.4 m of distance and the others have 0.6 m.

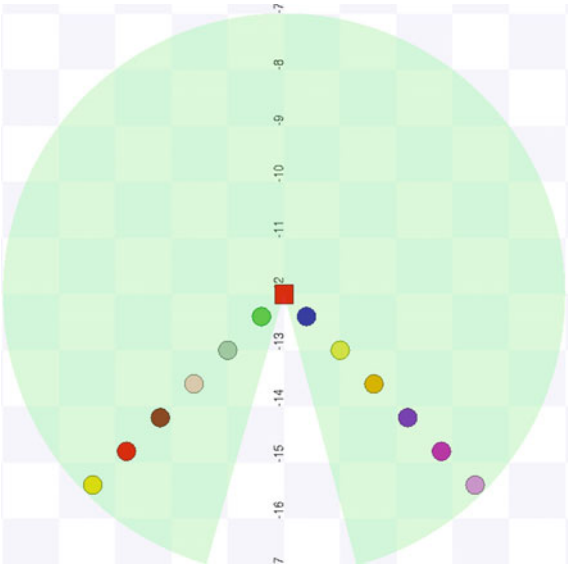


Fig. 38 Standard formation in ∧

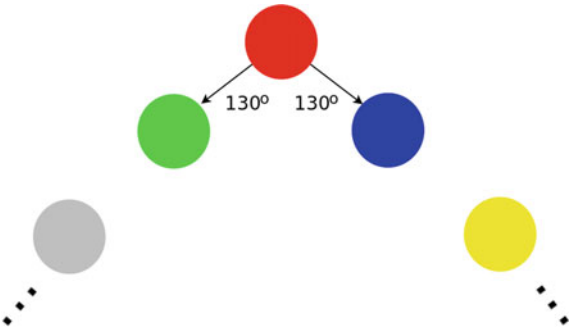


Fig. 39 Wedge shape ∧ for multi-robot systems

The guidance of the leader is considered to maintain formation over a global curve. Thus, the rotation of the followers in relation to the angle θ of the leader can be determined as

$$x = d.\cos(\alpha), \tag{1}$$

$$y = d.\sin(\alpha). \tag{2}$$

where d is the distance between the follower and leader to maintain formation, and α is the sum of the leader's angle θ with the opening in degrees necessary to maintain the configuration, for example, 130°. In addition, it adds up to the x_l and y_l of the leader robot. Using x and y resulting from Eqs. 1 and 2 obtains the new position of each follower.

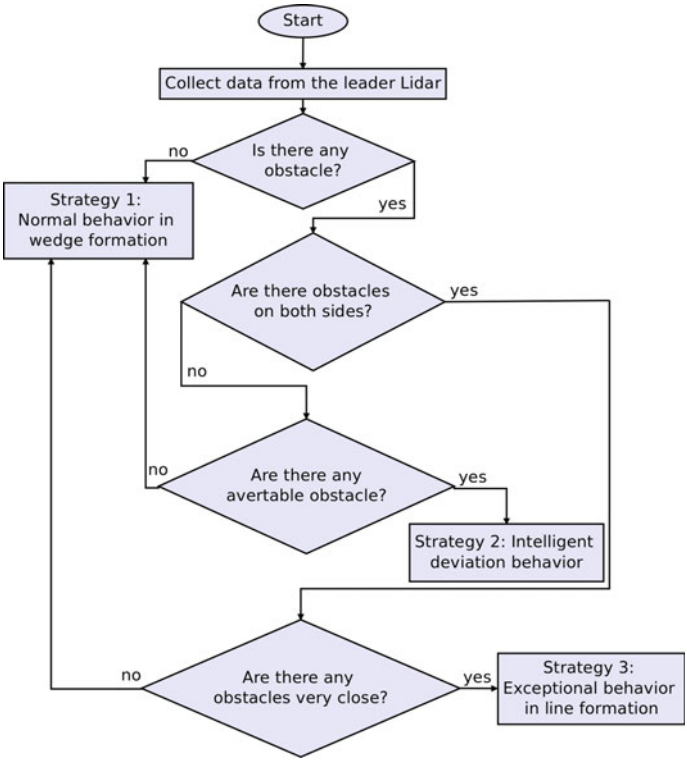
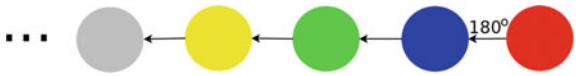


Fig. 40 Formation management

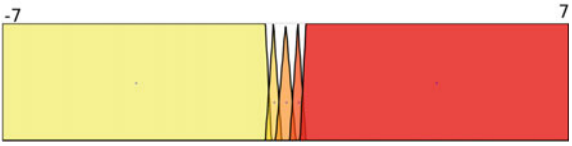
Fig. 41 Strategy of line formation



It is not always possible to follow the training strategy presented in Fig. 39 since, in this way, robots occupy a larger space and collisions can occur in different manners. As only the leader robot has a laser sensor and, consequently, control of obstacle deviation, additional strategies have been developed, as presented in Fig. 40. If the obstacle is detected to the right of the leader robot, then the formation moves 45° to the left. Likewise, if the obstacle is detected to the left, the formation shifts 45° to the right. If there is an obstacle on both sides of the leader robot, then a line formation is adopted. The positions of the follower robots are configured at 180° and distances of 0.3 m for the first robot and 0.6 m between the others robot, as shown in Fig. 41.

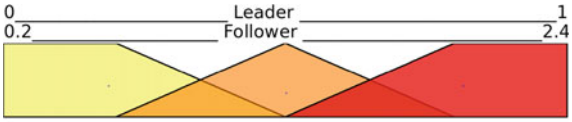
Fig. 42 Inputs of the Fuzzy position controller

(a) Input of linear distance.

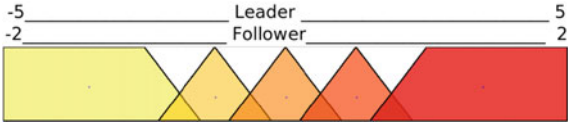


(b) Input of angular distance.

Fig. 43 Outputs of the Fuzzy position controller



(a) Output of linear velocity.



(b) Output of angular velocity.

4.2.3 The Strategy of the Leader Robot

A fuzzy controller is applied to control the global position of the leader robot to achieve the desired points (or task goals) in the environment. The controller receives as inputs the linear and angular distances between the objective and the position of the leader and generates as outputs the linear and angular speeds of the leader to reach the desired location. The Euclidean distance is an input that is expressed by the linguistic variables Near (N), Medium (M), and Far (F), as can be seen in Fig. 42a. The other input is the angular distance expressed by linguistic variables VeryRight (VR), Right (R), Front (F), Left (L), and VeryLeft (VL), as shown in Fig. 42b.

As output, the linear velocity is expressed through linguistic variables Low (L), Medium (M), and High (H), as shown in Fig. 43a. The angular velocity is expressed as TurnVeryRight (TVR), TurnRight (TR), Front (F), TurnLeft (TL), and TurnVeryLeft (TVL), as seen in Fig. 43b.

Finally, Table 2 presents the rules basis of the fuzzy controller. The system uses as conjunction the minimum, disjunction the maximum, activation the minimum and centroid as defuzzification.

Table 2 Rules of Fuzzy position controller

	N	M	F
VR	TVR, L	TVR, M	TVR, H
R	TR, L	TR, M	TR, H
F	F, L	F, M	F, H
L	TL, L	TL, M	TL, H
VL	TVL, L	TVL, M	TVL, H

Another fuzzy system was designed to deviate from obstacles. This system has as input how much to the right or left the closest object is to the leader, and the Euclidean distance between the robot and object. A 2D laser sensor (LIDAR) provides a vector of values that indicate whether the robot is detecting obstacles. When there is no obstacle, the value returned at each vector position is 5 m, which is the detection capacity configured for the leader robot's sensor. If there is an obstacle, then the value returned is less than the maximum value.

The lowest value of the LIDAR sensor is used to determine the object closest to the leader robot. The obstacle localization (X and Y) is obtained through the vector position of the minimum value found, minimum sensor angle, and sensor angle increment. Thus, the object side of the robot is determined with the signal of object coordinates where, if the value of X is negative, the obstacle is to the right of the leader robot, and if it is positive, the obstacle is to the left. The object side is one of the inputs to the fuzzy obstacle deviation system because the deviation should occur to the right if the obstacle is to the left, and to the left if the obstruction is on the right.

The linguistic variables for this input are VeryRight (VR), Right (R), Front (F), Left (L), and VeryLeft (VL), as shown in Fig. 44a. The second input of the fuzzy system is the Euclidean distance between the robot and the detected obstacle, with the linguistic variables VeryNear (VN), Near (N), Far (F), and VeryFar (VF) (Fig. 44b). The output is the angular velocity of the leader robot to deviate from the obstacle, which has as language variables TurnVeryRight (TVR), TurnRight (TR), Front (F), TurnLeft (TL), and TurnVeryLeft (TVL) (Fig. 44c).

The rule base of the fuzzy deviation system is presented in Table 3, which uses as a minimum, disjunction the maximum and activation the minimum and centroid to defuzzification.

4.2.4 The Behavior of Follower Robots

Unlike the leader robot, the follower robots do not have obstacle detection sensors. Thus, the follower robots receive only X and Y coordinates of the positions where they must navigate, considering them as positions free of obstacles. Thus, only one fuzzy position controller was developed for the follower robots.

(a) Fuzzy input of obstacle side to the deviation controller.

(b) Fuzzy input of Euclidean distance to the deviation controller.

(c) Fuzzy output of angular velocity to the deviation controller.

Fig. 44 Inputs and outputs for the Fuzzy obstacle deviation controller

Table 3 Rule base for the fuzzy obstacle deviation controller

	VN	N	F	VF
VR	TVL	TVL	TL	F
R	TL	TL	TL	F
F	TL	TL	F	F
L	TR	TR	TR	F
VL	TVR	TVR	TR	F

The fuzzy position controller of the followers receives as input the linear Euclidean distance and the angular distance between the point obtained from the leader and the current position of the follower. The controller has as outputs the linear and angular speeds that the follower must execute to reach the given point by the leader. The input of the linear Euclidean distance is expressed by linguistic variables as Near (N), Medium (M), and Far (F), as can be seen in Fig. 42a. The angular distance input is expressed by linguistic variables as VeryRight (VR), Right (R), Front (F), Left (L), and VeryLeft (VL), as shown in Fig. 42b. The linear velocity output is written as linguistic variables Low (L), Medium (M), and High (H) (Fig. 43a), and the output of the angular velocity is expressed as TurnVeryRight (TVR), TurnRight (TR), Front (F), TurnLeft (TL), and TurnVeryLeft (TVL), as shown in Fig. 43b.

Table 2 presents the rule base of the fuzzy position controller of the follower robots. This system uses as conjunction the minimum, disjunction the maximum and activation the minimum and centroid to defuzzification. Note that the position control of the followers is similar to that of the leader robot, having only distinct values of the leader for the linguistic variables of the outputs (linear and angular velocities).

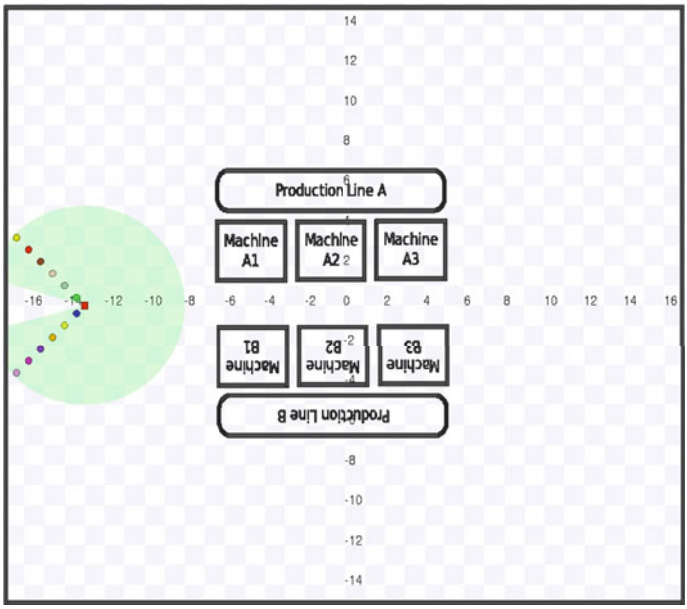


Fig. 45 Virtual experimentation scenario with the multi-robot system

Thus, the linear and angular velocities of the follower are adjusted in relation to the linear and angular velocities of the leader.

4.2.5 Experimental Evaluation

The approach evaluation is performed with a group of mobile robots to execute the cleaning services in an industrial plant with specific group requirements, aiming to maintain the agents' formation to expand the cleaning area. The group is composed of 13 vacuum cleaner robots and a leader equipped with a 2D laser scanner. The followers do not have any sensors, as shown in Fig. 45.

4.2.6 Experiment with Obstacles on both Sides

The first experiment aims to verify the multirobot system behavior when obstacles exist to both the right side and left side of the leader robot. In this case, the formation will converge to a queue arrangement, according to the strategy presented in Fig. 41. The result obtained is shown in Fig. 46, where the multirobot system adapts to suppress the detected obstacles.

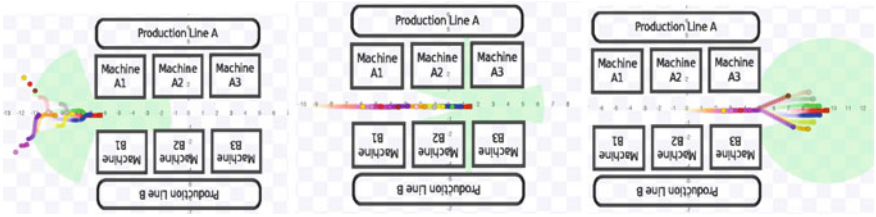


Fig. 46 Results of experiment with obstacles on both sides

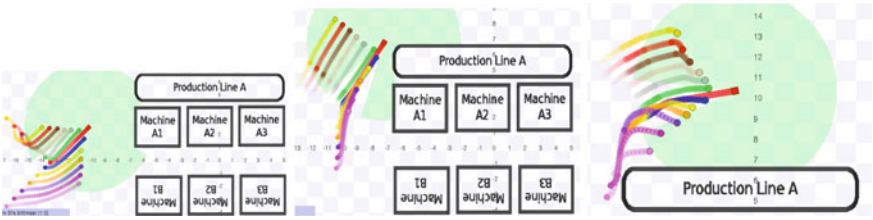


Fig. 47 Results of experiment with obstacles on right side

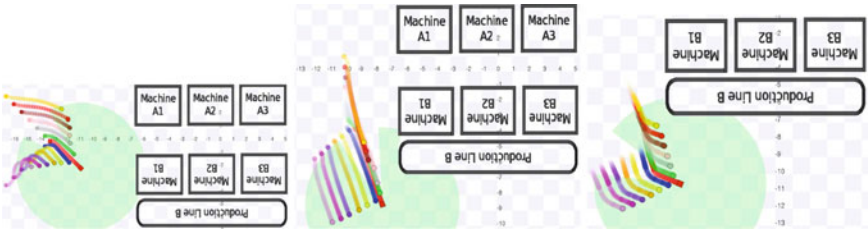


Fig. 48 Experiment with obstacles on left side of leader

4.2.7 Experiment with Obstacles on Right Side of Leader

The second experiment will evaluate the obstacle deviation when barriers occur to the right side of the leader robot. The intelligent behavior promotes formation adaptation, shifting the multirobot system 45° to the left, as shown in Fig. 47.

4.2.8 Experiment with Obstacles on Left Side of Leader

This experiment focuses on the evaluation of intelligent avoidance behavior of left lateral obstacles. The avoidance procedure ensures a formation shifting 45° to the right, as illustrated in Fig. 48.

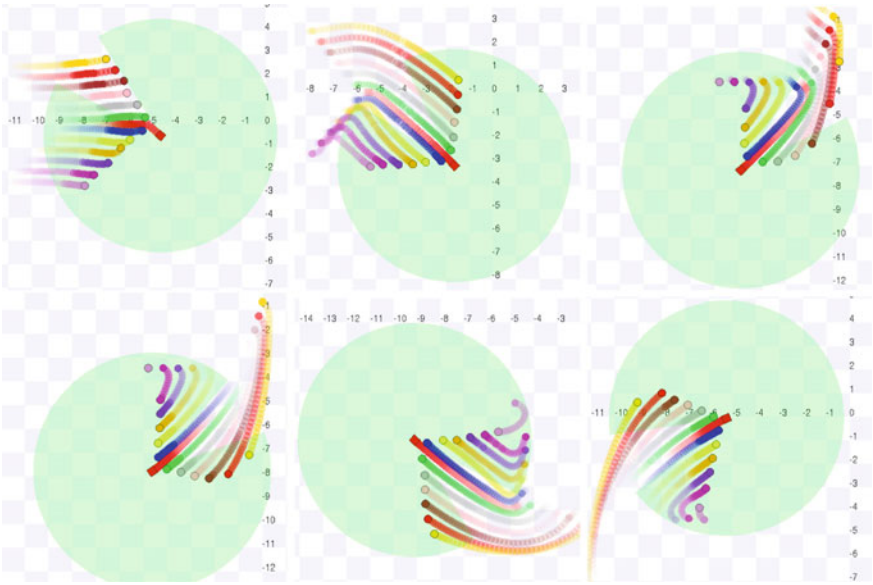


Fig. 49 ∧ formation regulation over angular motions

4.2.9 Experiment of Formation Maintaining

The last experiment is a demonstration of formation maintaining capability over rigorous angular motion in a multirobot system. This kind of variation is widely employed in cleaning tasks and cannot disturb the system formation. Thus, the experiment is developed in a free area where all robots are in linear motion and should execute an angular motion to change their global orientation by 180°, as seen in Fig. 49.

5 Computational Analysis

Table 4 lists a computational analysis of the use of RAM in kilobytes, evaluating a robot swarm with square formation and without a leader. The first column of the table shows the number of robots used in the experiment. The second column presents the use of RAM using a scenario without the use of sensors. The third column presents the results using a sonar sensor, the fourth column with three sonar sensors, and the fifth column with a laser sensor only. The last column displays the result using the *position_controller* parameter with a value of *true*. In this case, the *reference* was used as *relative* and *publish_tf* as *yes*.

It is noticed that the increase in RAM is more significant when the increment of sensors occurs for the number of robots from 240. In this case, the consumption of

Table 4 Computational analysis (kilobytes)

# robots	without sensor	with sonar	with 3 sonars	with LIDAR	with pose controller
1	127.043	127.427	127.418	127.286	132.049
12	129.635	130.082	130.004	130.328	166.381
120	152.797	156.964	160.457	158.293	668.327
240	177.930	193.840	202.133	195.538	1.100.053

RAM increases around 9% when added to the scenario one sonar sensor, 14% with three sensors, and 10% with one laser sensor.

When using the position controller, we can see that the increase in the use of RAM is already significant from 12 robots, where there is an increase of 28% in relation to the scenario without any control or sensor. This becomes to 337% when the number of robots is 120 and 518% when the number of robots is 240.

Already the memory consumption, for the same scenario, just now watching the number of robots suffers a significant increase from 120 robots. In the sensorless situation, when using 120 robots, there is a 20% increase of memory over the scenario with one robot. Even for the scene with 240 robots, the growth is 40%. For the sensor scenarios, a more remarkable increase is seen with three sonars. By contrast, for 120 robots, in relation to the scenario with 1 robot, there is an increase of 26% in memory consumption. For 240 robots, this percentage increases to 59%.

Finally, in analyzing the position controller for the number of robots, considering a scenario with 1 robot for a situation with 12 robots, we already have a 26% increase in memory consumption. This increase reaches 406% when the number of robots is 120 and 733% when the number of robots is 240.

6 Conclusions

Experimentation with robot swarms can be a time-consuming and tedious task if there is no automated help to set up the simulation environment and to begin its execution. This chapter aimed to show a novel ROS package to solve this issue. This package performs easy environmental parameterization through a quick setup.

Two examples are discussed to demonstrate the potential of the presented package: a simpler one with the objective of only controlling the position of a robot swarm, and another that demonstrates this virtual platform when applied to complex scenarios with specific approaches, as in the control of a swarm formation. In this last example, a fuzzy position controller was developed for the leader's intelligent behavior. This method is based on the linear and angular distance of the point that one wishes to reach, having as output the linear and angular velocity of the leader. Another fuzzy controller is developed for obstacle deviation. The object that is closest to the leader

is taken into account for performing the deviation, where the output of the controller is the angular velocity to be executed by the leader to deviate from the obstacle. For this behavior, the leader was equipped with a 2D LIDAR perception system. The position of the follower robot is determined by the leader since they are not equipped with sensors. Each follower has a direction and fixed position in the formation, both determined by the leader robot. The formation may vary between the composition initially determined in $\wedge$. If there is an obstacle on the right, then the position of the robots is adjusted 45° to the left. If there is a leftward obstacle, then the position of the followers is changed 45° to the right to avoid a collision. Finally, followers will adapt to the line formation if there are obstacles to both the right and left. A step-by-step video was developed to demonstrate how to build a robot swarm with the *swarm_stage_ros* package. This is available on YouTube.[3]

An estimate of computational cost is discussed to allow users to predict the required resources for a specific experiment. However, the robot quantity is dependent on the computer capacity to execute individual resources since each robot will require more computational it. For example, to simulate a swarm with 120 robots, all with sonar sensors and position control, the required RAM usage is 668.327 KB.

The *swarm_stage_ros* package is very versatile and can support research studies and educational activities even in complex situations. Future works will demonstrate the development of more scenarios and the possibility of subgroups of robots, with the potential of a leader for each subset. The integration of types of sensors besides the laser and sonar are possible.

References

1. M. Rubenstein, C. Ahler, N. Hoff, A. Cabrera, R. Nagpal, Kilobot: a low cost robot with scalable operations designed for collective behaviors. Robot. Auton. Syst. **62**(7), 966–975 (2014), Reconfigurable Modular Robotics
2. Y.K. Lopes, A.B. Leal, T.J. Dodd, R. Groß, *Application of Supervisory Control Theory to Swarms of e-puck and Kilobot Robots* (Springer International Publishing, Cham, 2014), pp. 62–73
3. I. Navarro, F. Matía, An introduction to swarm robotics, in *ISRN Robotics* (2013), 10 pages
4. G. Beni, From swarm intelligence to swarm robotics, in *Swarm Robotics* (2005), pp. 1–9
5. A.J.C. Sharkey, N. Sharkey, The application of swarm intelligence to collective robots, in *Advances in Applied Artificial Intelligence* (2006)
6. E. Sahin, S. Girgin, L. Bayindir, A.E. Turgut, *Swarm Intelligence: Introduction and Applications* (Springer, 2008), Swarm Robotics, pp. 87–100
7. G. Ben, J. Wang, Swarm intelligence, in *Seventh Annual Meeting of the Robotics Society of Japan* (1989), pp. 425–428
8. E. Sahin, Swarm robotics: from sources of inspiration to domains of application, in *Swarm Robotics Workshop: State-of-the-Art Survey* (2005), pp. 10–20
9. M. Yogeswaran, S.G. Ponnambalam, Swarm robotics: an extensive research review, in *Advanced Knowledge Application in Practice*, ed. by I. Fuerstner (InTech, 2010)
10. A.J.C. Sharkey, Robots, insects and swarm intelligence. Artif. Intell. Rev. **26**(4), 255–268 (2006)

[3] https://www.youtube.com/watch?v=Utz85Wnm1QU&feature=youtu.bes.

11. M. Larsson, Why do fish school? Curr. Zool. **58**(1), 116–128 (2012)
12. A.M. Deshpande, Robot swarm based on ant foraging hypothesis with adaptive lèvy flights, Master's thesis, University of Cincinnati (2017)
13. W. Blake, Drag reduction from formation flight flying aircraft in bird-like formations could significantly increase range, DTIC Document, Tech. Rep. (2002)
14. A. Ravankar, A.A. Ravankar, Y. Kobayashi, T. Emaru, On a bio-inspired hybrid pheromone signalling for efficient map exploration of multiple mobile service robots. Artif. Life Robot. **21**(2), 221–231 (2016)
15. J. Zelenka, T. Kasanický, I. Budinská, A self-adapting method for 3d environment exploration inspired by swarm behaviour, in *Advances in Service and Industrial Robotics*, ed. by C. Ferraresi, G. Quaglia (Springer International Publishing, Cham, 2018), pp. 493–502
16. A.R. Shirazi, Bio-inspired self-organizing swarm robotics, Ph.D. dissertation, University of Surrey (2017)
17. X. Yi, A. Zhu, S.X. Yang, C. Luo, A bio-inspired approach to task assignment of swarm robots in 3-d dynamic environments. IEEE Trans. Cybern. **47**(4), 974–983 (2017)
18. W. Dai, H. Lu, J. Xiao, Z. Zheng, Task allocation without communication based on incomplete information game theory for multi-robot systems. J. Intell. Robot. Syst. (2018)
19. K. Garapati, J.J. Roldán, M. Garzón, J. del Cerro, A. Barrientos, A game of drones: game theoretic approaches for multi-robot task allocation in security missions, in *ROBOT 2017: Third Iberian Robotics Conference. ROBOT*, vol. 693 (2017)
20. D. Wu, G. Zeng, L. Meng, W. Zhou, L. Li, Gini coefficient-based task allocation for multi-robot systems with limited energy resources. IEEE/CAA J. Automatica Sinica **5**(1), 155–168 (2018)
21. B. Kartal, E. Nunes, J. Godoy, M. Gini, Monte Carlo tree search for multi-robot task allocation, in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)* (2016)
22. R. Vaughan, Massively multi-robot simulation in stage. Swarm Intell. **2**(2), 189–208 (2008)
23. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, Mason: a multiagent simulation environment. Simulation **81**(7), 517–527 (2005)
24. V.C. Kalempa, M.A.S. Teixeira, A.S. de Oliveira, Versatile and massive experimentation of robot swarms in industrial scenarios, in *Robot, Fourth Iberian Robotics Conference*, ed. by M.F. Silva, J. Luís Lima, L.P. Reis, A. Sanfeliu, D. Tardioli (Springer International Publishing, Cham, 2020), pp. 279–290
25. L. Piardi, V.C. Kalempa, M. Limeira, A.S. de Oliveira, P. Leitão, Arena - augmented reality to enhanced experimentation in smart warehouses. Sensors **19**(19) (2019)

Vivian Cremer Kalempa received the graduation in Computer Science at Santa Catarina State University (UDESC), Brazil, and Master degree in Computer Science at Santa Catarina Federal University (UFSC), Brazil. Currently she is a PhD student at Federal University of Technology—Parana (UTFPR) and professor at UDESC. Her research has been focused on intelligent production systems, collaborative robots and autonomous mobile robot.

Marco Antonio Simoes Teixeira received the B.Sc degree in Information Systems from Santa Catarina State University, Brazil, in 2014, the M.Sc. degree in Electrical and Computer Engineering from Federal University of Technology—Parana, Brazil, in 2017. He is currently Ph.D student in Electrical and Computer Engineering at Federal University of Technology—Parana (UTFPR). His research interests comprise Mobile Robots, in the areas of perception and intelligent systems.

Andre Schneider de Oliveira is Adjunct Professor at Federal University of Technology—Parana (UTFPR), member of Advanced Laboratory of Robotics and Embedded Systems (LASER) and Laboratory of Automation and Advanced Control Systems (LASCA). He received his Ph.D. degree in engineering of automation and systems (2011) with the thesis focused on differential kinematics through dual-quaternions for vehicle-manipulator systems. In 2007, he received his M.Sc. degree in mechanical engineering, concentrate in force control of rigid manipulators, at Federal University of Santa Catarina (UFSC). His research interests lie in the areas of Robotics, Mechatronics

and Automation with particular focus in navigation and localization of mobile robots; autonomous and intelligent systems; perception and environment identification; cognition, and deliberative decisions; human-interaction and navigation control.

João Alberto Fabro holds a degree in Informatics (bachelor's degree) from the Federal University of Paraná (UFPR-1994), a master's degree in Electrical Engineering from the State University of Campinas (UNICAMP-1996), a doctorate in Electrical Engineering and Industrial Informatics from the Federal Technological University of Paraná (UTFPR-2003). He was a professor at UNIOESTE (State University of Western Paraná) from 1998 to 2007. Since 2008, he is a professor at the Federal University of Technology—Paraná, based in the Department of Informatics—Curitiba Campus. He is an Associate Professor. From 2012 to 2013, he was the Head of the M.Sc. course on Applied Computing of UTFPR. In 2014, Prof. Dr. Fabro was visiting fellow at the Faculty of Engineering of the University of Porto, Portugal. Has experience in Computer Science, focusing on Artificial Intelligence, acting on the following subjects: Autonomous Mobile Robotics, Computational Intelligence (Neural Networks, Fuzzy Systems, Evolutionary Computing) and Educational Technologies. Is the coordinator of Research Groups on Computational Intelligence and the LASER-Advanced Laboratory of Embedded Systems and Robotics (http://laser.dainf.ct.utfpr.edu.br), at UTFPR. Is a collaborator of LIACC-Laboratory of Artificial Intelligence and Computer Science, University of Porto, Portugal (https://liacc.fe.up.pt). Since 2009, he has participated in several national, Latin American and international robotics championships (Robocup), both in robot soccer and service robot categories (Robocup @ Home), both as competitor and organizer. He is member of the special comission on Robotics of the Brazilian Computer Society.

Autonomous Systems

Lessons Learned Building a Self Driving Car on ROS

Nicolò Valigi

Abstract Autonomous vehicles (AVs) capable of safe operation on urban roads are a complex problem that involves many subdomains of Robotics. This use case chapter presents the lessons that the authors learned while using the ROS framework to support the AV development efforts of Cruise, originally acquired by General Motors in 2016, and now an independent company. Scaling ROS to this level of complexity and team size brings an interesting viewpoint to its design decisions and can help guide the future direction of the community. After briefly introducing the hardware and software stack on a modern AV, the chapter begins with a discussion of ROS' defining abstraction, publish/subscribe, and how it impacts system design and performance. Afterwards, we share our point of view about the complementary tools in the ROS ecosystem: visualization, data recording, and the build system. We conclude with the fundamental task of simulation and automated testing.

1 Introduction

In 2019, we have a fairly clear idea of the role of a "Robotics framework" in the Open Source community: it must enable code reuse, provide building blocks for standard navigation and planning tasks, and ship with convenient developer tooling. The resounding success of ROS in the academic and R&D world is testament to its fulfilling this niche.

However, automotive-grade AV development has a different focus and thus a different set of Engineering tradeoffs. When a company is developing a single product, flexibility and modularity take a back seat to reliability and predictability of performance. Requirements on data processing, testing, and automation scale beyond the usual needs of the typical research lab in academia. Tooling needs to be accessible to Labeling and Quality Assurance teams as well, not just to Scientists and Engineers.

AV companies have attracted well over $1bn in funding every year for the past few years, gathering enormous momentum to re-think the robotics framework for the next generation of Scientists and Engineers. This case study chapter investigates

N. Valigi (✉)
Mir Srl Medical International Research, Via del Maggiolino 125, 00155 Roma, Italy

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_5

the feasibility, strengths and weaknesses of ROS as a platform for AV development in the context of proprietary product development.

AVs hold a great potential for increased road safety, decreased urban traffic, and improved quality of life. Several manufacturers have already commercialized semi-autonomous vehicles capable of driving safely in restricted scenarios, like highways or dedicated roads. However, the end goal of fully autonomous driving in urban areas remains somewhat elusive, with a handful of companies vying for the first public release of the product.

Cruise, based in San Francisco, is one of such companies. Founded in 2013, it was later acquired by General Motors in 2016, then spun off as a separate entity in 2018. It has received several billion dollars in funding towards the goal of developing a full-stack AV solution, benefiting from General Motors' expertise in automotive manufacturing. In this case study chapter, the authors discuss the lessons they learned while working as Software Engineers at Cruise and implementing a full-stack AV product based on ROS. This chapter discusses the strengths and weaknesses of ROS, expanding on its necessary (and sometimes painful) evolution from a prototyping framework to a production platform.

AV development is an important test bed for the future of the open-source robotics ecosystem (and of ROS in particular) for a variety of reasons. First of all, stringent safety requirements dictate high standards of software quality and reliability, spurring the development of "production ready" frameworks that can benefit other sectors too. Huge market potential and financial resources mean that AVs can use the most powerful hardware available, including multiple CPUs and datacenter-class GPUs, leading to new challenges in system design and architecture.

This book chapter is divided into 4 sections. The first discusses the design and implementation of AVs as robotics systems, including a description of the typical sensor suite and autonomy stack. The second section explores the strengths and weaknesses of the core ROS abstractions when applied to AV-specific development. The third section looks at the complementary tooling in the ROS ecosystem, like visualization and data recording. The last section ties all these topics together by introducing the challenge of large scale simulation and testing.

Unless otherwise noted, all information in this chapter refers to ROS 1 as that is the platform the authors have more experience on. While some specific implementation details have been improved for ROS 2, the authors believe that this systems-level overview will nonetheless be valuable for practitioners and open-source contributors alike.

2 Related Work

Due to its widespread use in the academic community, many open-source autonomous vehicle research projects are based on ROS [25], including Autoware [18] and Baidu's Apollo [13]. ROS is also widely used for education in the field of autonomous vehicles, for example in Udacity's Self Driving Car Nanodegree [27].

Accurate information about commercial use is harder to come by but can be inferred from online job postings. Both Cruise and Zoox, two of the leading players in the space, mention ROS in their careers page. Engineers from BMW have also spoken at conferences about the use of ROS on the company's AVs [4]. Apex AI [2] is a company building a proprietary, certified, framework for AV development, albeit based on ROS 2.

3 Autonomous Vehicles as Robots

AVs are complex systems because of their need to safely perform the large variety of behaviors needed for safe urban driving. This section provides a brief introduction to the main hardware and software features of modern AVs, with no reference to specific implementations [24, 32].

3.1 Hardware Components

Just like any other robot, hardware components of AVs perform 3 main roles: sensing, computation, and actuation. Since modern consumer cars are manufactured with a drive-by-wire architecture, the AV stack can control the vehicle by connecting to OEM equipment. Therefore, little custom hardware is required for actuation. For the purposes of this chapter, the computing platform is almost always based on commodity hardware with desktop or server CPUs.

In other words, most of the effort in hardware development is concentrated on the sensor suite. Given the state of today's sensor technology, no single sensor modality is enough to guarantee the performance needed for safe urban operation. As a consequence, the typical AV sensor architecture includes several different families of sensors that complement each other. While the AV industry is still experimenting with several different configurations of the sensor suite, this section will nevertheless try to give the reader an introduction to the most typical designs (Fig. 1).

A large part of the industry has converged on LIDARs as the central sensor modality because of their affinity with the requirements of urban driving. LIDAR is a ranging sensor based on near-infrared light that provides a 360-degree view of the environment in the form of a point cloud. The typical automotive LIDAR sensor sweeps a beam of 64 rays over the full 360 degrees 10 or 20 times a second, provid-

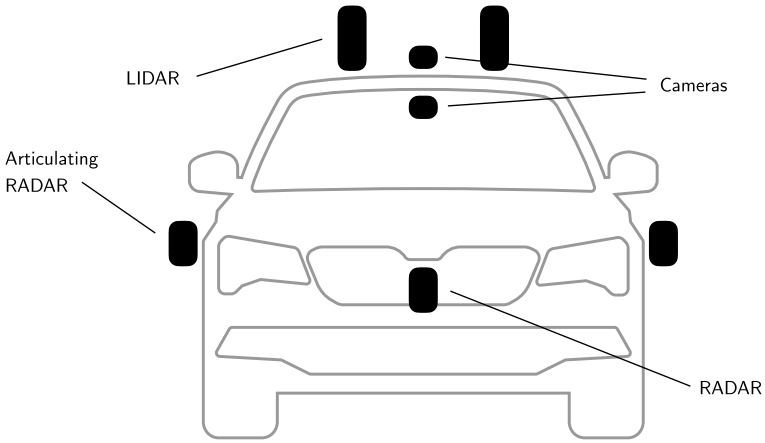


Fig. 1 The sensor suite on a modern AV covers several distinct modalities, each optimized for a specific task in urban driving

ing a constant stream of 1k points per second. Because of their unique ability to provide high-resolution 3D data with reliable range information, LIDARs have been a staple of AV designs ever since the original DARPA challenge in 2004. LIDAR point clouds are versatile, and commonly used for localization, object detection, and low-level obstacle avoidance (thanks to their relatively low false positive detection rate).

Cameras are a great complement to LIDARs thanks to their high spatial resolution and color sensitivity. Because of their limited field of view, the typical AV features 5–10 cameras looking outwards at different angles. Camera images are often the main input to object classification, but are also commonly used for traffic light detection and fused probabilistically with LIDAR data for obstacle detection. Because of their higher resolution, vision-based object detection has the potential to see further away from the vehicle than LIDAR. From the point of view of system design, cameras have higher data throughput than LIDAR, and thus require more expensive post-processing, which is often executed on GPUs or ASICs.

RADARs are similar in operation to LIDARs, but use radio waves instead of light. Since the wavelength of radio waves is higher than that of near-infrared used by LIDARs, RADARs generally have a higher range. This directly translates into an increase in the allowed reaction time for the AV stack, which is beneficial for safety, and allows increasing travel speed. Longer-wavelength signals are also less sensitive to adverse weather conditions, like rain and fog, which typically degrade the quality of LIDAR measurements. Furthermore, both antennas and signal processing electronics are simpler in RADARs, making them cheaper than LIDARs. On the other hand, data produced by RADARs is typically of lower accuracy and resolution, and is also affected by spurious returns by nearby buildings and other static obstacles.

Inertial Measurement Units (IMUs) and wheel encoders are proprioceptive sensors that measure, respectively, acceleration and angular velocity, and wheel angular velocity. When corrected with lower-frequency vision and LIDAR data, they're fundamental for measuring high-frequency dynamics needed by the low-level controls subsystem.

Sensors are connected to compute units through a variety of interfaces. High-bandwidth, low rate sensors can be connected through an Ethernet interface, since the packet processing overhead through the Linux kernel does not add significant latency. This is the case for cameras and LIDARs. Low-bandwidth, high rate sensors like IMUs, wheel encoders, and other inputs from the car Engine Control Unit (ECU) are usually connected through the CAN bus installed by the vehicle OEM. From the point of view of the software architecture, separate connections are not a substantial difference, because all drivers publish incoming data as ROS messages.

All sensors should be synchronized to a common time base to improve the performance of algorithms that merge data from multiple sensors (e.g. state estimation). Otherwise, even small time shifts can cause significant inaccuracies when the AV is moving at normal traffic speed. As long as the acquisition timestamp is stored as part of the message payload, downstream consumers can correctly associate each message to an instant in time, independent of the delays added by intermediate processing steps.

The simplest approach is to time stamp messages on arrival to the driver node. However, such timestamps are affected by high jitter because the physical communication layer, hardware interrupts, and OS scheduling each add variable and unpredictable delays. Messages from networked sensors using UDP can be timestamped more accurately at the kernel level using the dedicated feature of the Linux network stack.

Generally speaking, the best option is to push the timestamping responsibilities to the hardware/firmware layer by sharing a common clock to all devices. This can be achieved with the GPS Pulse-Per-Second (PPS) signal which is synchronized with atomic clocks installed on GPS satellites. In the absence of a common clock line, sensors can be synchronized over commodity networks using the Network Time Protocol (NTP) and the Precision Time Protocol (PTP) [16].

Safety is a prime concern in the design and development of AV hardware. As is common in the automotive industry, suppliers have been investing heavily in the production of sensors and compute units that achieve safety certification for autonomous use on the road (for example, ISO ASIL-D) [23]. Therefore, the AV hardware market is expected to become increasingly specialized in the next few years, leaving software as the primary differentiating element among AV companies on the market. The next section provides a brief introduction about the typical architecture of AV software.

3.2 Software Components

The software architecture of AV software follows the typical setup of most robotics systems, and can be divided in *Perception*, *Planning*, and *Control* stacks. The Perception stack interprets incoming sensor data and uses it to build a consistent vision of the world, including both persistent and transient features of the environment (e.g. both road curbs and walking pedestrians). The Planning component makes high-level decisions about the desired trajectory of the vehicle to accomplish its mission (i.e. from the start to the end location, accounting for intersections and parked vehicles). The Control component receives the plan and computes the appropriate control signals (e.g. steering and throttle) so that the vehicle can track the desired trajectory and passengers are comfortable.

The Perception stack has two conceptually different roles: Localization and Environmental Perception. Localization is one of the most basic requirements of all robots, and AVs are no exception. Since required accuracies for safe road operation are higher than achievable with GPS, AVs need to rely on additional sensors for accurate localization. Localization is often implemented by collecting LIDAR point clouds and using them to build city-scale 3D maps using offline map optimization algorithms. The maps are then loaded on the vehicle and used for real-time localization by matching the local point clouds collected by the LIDAR against the pre-built map. Since the outputs of the Localization stacks are used everywhere else in the system, it is accuracy is critical.

While the 3D map can provide strong priors on permanent features of the environment, like the surface area of roads and the presence of traffic lights, safe road operation also requires that the vehicle be capable of identifying transient objects that might share the road. These include other cars, pedestrians, cyclists, etc., than can not be mapped beforehand for obvious reasons, and need to be tracked using live sensor data. This is where *Environment Perception* comes in. This component applies probabilistic models to the sensor data to estimate the position, velocity, and future behavior of other road users.

The *Planning stack* makes high level decisions about the trajectories that the vehicle should follow, according to the desired destination, the rules of traffic, and the behavior of other road users. A conceptual model of the Planning stack includes three main components (in decreasing levels of abstraction): Mission Planning, Behavioral Planning, and Motion Planning. Mission Planning deals with the highest-level actions of the vehicle: choosing a destination, pulling up, and pulling over to stop. Given the overall mission plan, Behavioral Planning decides which actions are the safest and most appropriate for a given traffic situation: whether to overtake, change lanes, or stop at a traffic light. Motion Planning is the lowest level of Planning, where behaviors are synthesized into smooth trajectories with known path, velocity, and acceleration.

Finally, the Controls stack computes the appropriate values of the control commands (e.g. steering wheel and throttle) that should be applied to the vehicle to follow the computed trajectory as accurately and comfortably as possible. The outputs of

the Controls stack are consumed by the OEM electronics on the vehicle that manage the physical actuators.

3.3 Data Flow

To a first approximation, the data flow across the stack follows a linear pattern where sensor data is first fed into the Perception and Localization stacks. The Planning stack receives the information about detected actors on the road and computes the optimal trajectory towards the destination. Finally, the Controls stack turns the trajectory into command actions by using Localization as a feedback signal.

This simplification quickly breaks down as we consider that road driving is highly correlated over time. Decisions made at time t have a strong effect on those that will be made at time $t + 1$. For example, lane change maneuvers should not be aborted midway just because a traffic light has turned yellow. Likewise, the behavior of other road users is highly auto-correlated over time, as it is relatively rare to encounter human drivers that swerve or act impulsively.

At the very least, this means that the linear data flow model must be augmented by keeping a memory of past state. For the most part, each component can track its state separately, for example the Perception module might record the last known position of each cyclist as it receives incoming LIDAR data to update it. However, there are also higher-level sources of feedback effects across the whole stack. For instance, decisions made by the AV are very likely to affect the future behavior of other road participants, to the extent that predicting their actions will need to take into account the current plan being executed by the AV.

Besides feedback loops, it is common to encounter road situations calling for tight integration between Planning and Perception stacks, going beyond the basic linear dataflow. A typical example is the vehicle having to creep up at an intersection to gain better visibility into oncoming traffic, thus coupling the perception and planning stacks to improve performance. Such requirements increase the complexity of the data flow patterns in the system.

Further details about the algorithms used for localization, perception, and planning are not relevant to the discussion in this chapter. Interested readers can refer to [24, 32] for a general review of the field.

4 ROS as a Framework for AV Development

This section covers the core components of the ROS framework: the publish/subscribe message bus (Sect. 4.1) and the configuration tooling (Sect. 4.4). We also share our challenges with two aspects that affect many real-world deployments of ROS: real-time execution (Sect. 4.2) and determinism (Sect. 4.3).

4.1 The Publish/subscribe Abstraction

The first concepts that each novice Roboticist encounters when starting to learn ROS are *nodes*, *topics*, and the *publish/subscribe* (*pub/sub*) model. Together, they form what is referred to as the ROS *computational graph*: independent components that communicate with each other by sending *messages* through named channels (*topics*). Messages are defined using a human-readable schema that can be consumed by code-generation tools that support a dozen languages (C++, Python, Javascript, LISP, etc.).

The pub/sub functionality is at the core of the ROS framework, and therefore heavily used in open-source ROS code. Since the ROS distribution comes with default message schemas for common tasks in robotics (poses, point clouds, and so on), pub/sub enables modularity because different components of the stack can communicate with each other without code changes. Since communication is mediated through the pub/sub system, developers can use debug tools to trace and record messages flowing through the stack. The structure of the computational graph can be queried and manipulated at runtime, for example to spawn new nodes. Nodes implemented in different programming languages can communicate transparently through the pub/sub mechanism.

An important implementation detail of the runtime is that, by default, each node runs in its own OS process. This means that crashes are isolated to a single node, and the stack enjoys built-in concurrency and parallelism thanks to the OS scheduler. Since the default ROS pub/sub implementation uses the TCP stack, a ROS computational graph can transparently span multiple physical machines. This applies to both wired and wireless networks, and allows scaling to multiple physical machines for more computation power. A common configuration enabled by this feature is having a lightweight rover for data collection backed by a more powerful stationary server, or swarms of robots that share sensor data and maps for localization.

Despite all these attractive properties, even small-scale projects will soon run into limitations in both the concept and the implementation of pub/sub in ROS. In our experience with a complex AV stack, challenges with the pub/sub machinery have been the greatest source of concern with ROS. The remainder of this section illustrates these shortcomings.

4.1.1 Serialization and Data Transfer Overhead

Since each ROS node runs in a separate process, and thus a different memory address space, data sent between nodes needs to be serialized and go through an Inter Process Communication (IPC) mechanism. The default IPC mechanism is based on TCP/IP and therefore forces message data to flow through the kernel networking stack. This causes multiple unnecessary copies and context switches from user to kernel space.

To avoid this issue, the community has produced alternative IPC implementations, including some that use shared memory rather than the network stack (for example

[28]). Wang et al. [29] paired shared memory with a custom serialization algorithm to further reduce latency.

Another option to avoid the IPC overhead altogether is to use *nodelets* instead of nodes. Nodelets are a built-in extension to the ROS API that follows a similar API to nodes, with the crucial difference that multiple nodelets can run within a single OS process. Nodelets running within a single process share a certain number of worker threads where message callbacks are scheduled. Because threads share the same memory address space, nodelets can "send" messages to each other simply by sharing pointers.

Nevertheless, nodelets do not completely eliminate message-passing overhead. To maintain compatibility with the process-based ROS API, the ROS runtime still needs to maintain queues of message pointers that must be protected by mutexes for correct multi-threaded operation.

On paper, one of the most interesting aspects of communication based on TCP/IP is that a single ROS computation graph can transparently spawn multiple physical machines connected with commodity hardware (WiFi or Ethernet). As long as all nodes are configured to point to the same *ROS master*, they can send and receive messages with no changes to user code. However, this functionality is useful for little more than prototypes and debugging because of reliability and performance concerns.

The TCP transport achieves reliability at the expense of unbounded latency, potentially interfering with timing requirements. The UDP transport offers bounded latency but no guarantees about message delivery. The responsibility to gracefully handle these situations therefore falls on the application code. Overall, this means that stacks distributed across multiple machines are forced to implement their own protocol on top of ROS anyway. Developers need to bring problem-specific knowledge about how nodes are distributed across machines to ensure reliability and performance.

4.1.2 Poor Use of the CPU

Initializing the ROS runtime in a C++ process (e.g. `ros::init`) spawns at least 5 threads to handle message passing, timers, and housekeeping. Therefore, a medium-sized stack with several sensors will often spawn a hundred or more threads.

Running such a high number of threads negatively impacts both *latency* and *throughput* of the overall stack. The simplified model in Fig. 2 helps describe the effect of the number of threads on the latency of the system.

Nodes spend much of their inactive time in the blocked state, waiting to receive data from other nodes through the network stack. Network sockets are handled by the ROS runtime library, and are serviced in a "subscription thread", which is also managed by the ROS runtime. As soon as a message arrives, the kernel wakes up the thread, which deserializes the incoming data and puts the message into a FIFO queue. Callback threads, normally sleeping on a condition variable, are woken up and start running user code on the messages being popped from the queue.

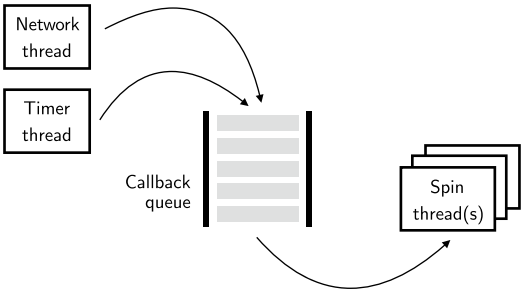


Fig. 2 In the C++ ROS runtime, the network thread waits for data to come in from the sockets, deserializes incoming messages, and puts them on a queue. The spin threads pop new items from the callback queue and execute them. Not shown are mutexes and condition variables for thread safety

When a new message is published, network sockets receive data and the corresponding subscriber nodes become *runnable*. Since ROS does not provide facilities to explicitly set the priority of subscriptions, callbacks triggered by high-priority messages might be delayed while less important tasks are executed. Developers can get control of this process by creating multiple callback queues. Each queue can be configured to be serviced by a different thread, which can thus be assigned a different OS priority.

Nevertheless, having so many OS-level tasks creates unnecessary coupling between the framework and the OS, since execution patterns depend on the behavior of the OS scheduler. Section 4.3 will expand on the impact of scheduling and latency on the overall performance of the system.

Having too many threads also reduces the overall throughput of the stack because CPU time is wasted on *context switching* from one thread to another and from user to kernel space. While a context switch only takes 1–2 μs [5], it also reduces the effectiveness of CPU caches as the two threads are likely to be using different portions of system memory.

4.1.3 Distributed System Mindset

At first glance, message passing protects from well-known concurrency bugs like race conditions and deadlocks. Developers do not need to concern themselves with writing multi-threaded code in each node, and yet the stack as a whole can still benefit from parallelism. Behind this apparent convenience, however, lies the fact that ROS forces developers to adopt a distributed systems mindset even when not necessary.

In the absence of shared memory, each node maintains its own copy of the system state using message subscriptions. This seems to work fine until it does not, and transient bugs start to plague the stack. A common case happens when different

components do not receive messages at the same time, and thus operate on conflicting versions of the shared state.

Debugging also becomes more difficult, because common tools generally do not have good support for systems that span multiple processes. Most ROS developers know how to implement condition variables and other concurrency primitives using ROS topics. By doing so, however, they lose the ability to use language-level debuggers like `gdb`, and have to rely on ROS-specific tools such as `rostopic echo`.

4.1.4 Conclusion

None of the afore-mentioned shortcomings are intrinsic to ROS or ROS programs themselves. Usage of pub/sub is by no means enforced or required at the framework level: it is certainly possible to include all logic into a single node, and use language-level facilities for communication and synchronization. Nevertheless, heavy use of pub/sub is encouraged by the documentation and widely adopted in the community, leading many developers to embrace it as the "right" way to build robotics application under ROS. By sharing our experience, we hope to raise awareness about the downsides of heavily distributed designs.

It is no doubt that the modularity and concurrency achieved by message passing are important for rapid development. However, it is our experience that the apparent simplicity of ROS primitives tends to nudge developers into over-reliance on a distributed architecture.

In a few words, our recommendation is to "make libraries, not nodes". ROS patterns tend to couple computation with communication, following an unwritten law where units of reusable code are expected to communicate using ROS topics, and not function calls. As we discussed above, this tends to cause performance and usability problems.

Instead of a multitude of nodes competing for CPU time, the goal is to strive to reduce the number of independent flows of execution, maintaining tight control over CPU resources at all times. Language-level facilities for concurrency are often more performant and sophisticated than ROS alternatives, especially in the era of C++11 `std::thread`, `std::future`, and coroutines.

4.2 Real-Time

Generally speaking, robotic systems fit the definition of *real-time* computing: safe operation of the robot requires that tasks are completed within a fixed time budget. In other words, computation has *temporal* as well as functional requirements. A control input that is numerically correct but is computed too late is effectively useless.

While real-time is often confused with low-latency, the two terms are not interchangeable [22]. Low-latency is a vague term for systems that react *fast* to new

inputs. Real-time is about providing *guarantees* for the response time, irrespective of its value. Control software for a nuclear power plant might not be "low-latency", as some of the tasks might only need to run every few seconds, but is real-time because overshooting the time budget can cause system failures.

The software stack for AVs matches both definitions. It must be real-time because unbounded delays can cause accidents on the road. Furthermore, end-to-end latency from perception to vehicle control must be roughly comparable to human drivers for safe operation at typical driving speeds. This section discusses why ROS does not fit these requirements and introduces alternative system architectures for safe operation.

The vast majority of facilities provided by ROS are not suitable for real-time usage:

Dynamic memory allocation Both runtime structures and APIs make pervasive use of dynamically allocated memory (i.e. `new` and `shared_ptr`). Any dynamic allocation can cause the process to block for unpredictable lengths of time as it requests new blocks of free memory from the kernel. The embedded systems community has extensive experience in programming without dynamic memory allocations. The same techniques could conceivably be used to reimplement the ROS runtime avoiding runtime allocations. However, some of the user-facing APIs rely on dynamically-allocated memory. Therefore, many API changes to reduce dynamic allocation would require updates to user code as well.

Inter-process communication As mentioned in the previous section, the default mechanism for Inter-Process Communication (IPC) in ROS is TCP. TCP is a stream-based protocol that achieves reliability at the cost of unbounded latency. Features like acknowledgements and retransmissions are unsuitable for a real-time system.

Priority management None of the ROS pub/sub facilities allow setting the relative priority between data flows. This makes it possible for low-priority messages to steal CPU time away from high-priority messages. Callback and publication queues are all FIFO, which means that newly-arrived high-priority messages can not preempt existing tasks, even if they are lower priority.

After this discussion, it would seem impossible to use any of the ROS features in systems with real-time requirements. However, most robotic systems can be architected such that not all components have the same safety requirements. In the case of AVs, route planning is not under the same strict timing requirements as path following on low-level controls. The AV can operate safely even if higher-level behaviors are computed and executed at a lower frequency. The vehicle might miss a lane change that would shorten the route, but this is not a safety-critical issue as long as the vehicle is still stopping at red lights.

This pattern of mixing real-time and non real-time processes aims at getting the best of both worlds: the flexibility and ease of development of ROS with the ability to incrementally migrate code to satisfy real-time constraints. The low-level controller loop of the PR2 robot follows this approach (and is available as open-source code

[12, 14]). To achieve the goal of running the loop at 1 kHz, most ROS features are explicitly avoided. Integration with the rest of the ROS computational graph is achieved through a separate thread that communicates with the controller loop through a lock-free queue to avoid blocking the real-time thread. Modularity and extensibility are achieved through a plug-in architecture that loads binary code into the same process.

The expedient of running a separate thread with no ROS components is not enough to achieve real-time operation. Support from the OS is also needed. In Linux, this means selecting an appropriate scheduling algorithm, using a real-time kernel and pinning tasks to specific cores.

The first step is to change the scheduling policy for the real time thread. The default scheduler in Linux is not a good fit for real-time tasks because it is designed for fair allocation of CPU time (according to task priority) and makes no guarantees about deadlines. In other words, the scheduler might allow low priority tasks to run even when higher-priority tasks are available as long as the long-term (100 ms) distribution of CPU time respects the priorities.

Luckily, he Linux kernel also supports two POSIX-compliant real time scheduling classes, `SCHED_FIFO` and `SCHED_RR`. Tasks allocated to these two classes take priority over all other tasks in the system, guaranteeing that time critical tasks can always get a slice of CPU time. On the other hand, this means runaway `SCHED_FIFO` or `SCHED_RR` tasks can starve other tasks on the same CPU core. Furthermore, interaction between multiple real-time tasks is not well-behaved: high-priority tasks can block lower priority tasks indefinitely because the scheduler is based purely on task priorities.

These shortcomings in `SCHED_FIFO` and `SCHED_RR` motivated the introduction of `SCHED_DEADLINE`, a completely different scheduling class that takes priority over all other tasks [1]. Tasks assigned to `SCHED_DEADLINE` specify *runtime* and *period* values (i.e. they declare that they intend to be using *runtime* CPU time every *period* time). Since the timing requirements are configured independently for each task, the scheduler can guarantee that high-priority tasks using unexpected amounts of CPU time do not starve lower-priority tasks (a form of isolation). The PR2 controller loop described above uses `SCHED_DEADLINE` for its real-time thread.

Even when using real-time scheduling, the default Linux kernel does not guarantee real-time operation. This is because some kernel tasks can not be preempted, and can thus delay user-space code indefinitely (including real-time tasks). It is not uncommon to observe delays in the order of many milliseconds in loaded systems. These issues can be solved using the `RT_PREEMPT` patch to the Linux kernel. This set of changes, slated for inclusion in future releases, reworks kernel internals to make many more kernel operations preemptable. These changes prevent kernel-space tasks from blocking user-space code. Recent reviews and real-world experiments [1] suggest that carefully tuned `RT_PREEMPT` systems are within a 2–5x factor in latency guarantees compared to platform running special-purpose real-time kernels.

On multi-core systems, it is beneficial to assign certain critical tasks to their dedicated CPU cores (*pinning*). While this has no effect on the real-time guarantee per-se, it helps reduce latency in the general case because code can make more

efficient use of CPU caches. On a related node, power management features on modern CPUs can create unexpected delays as the silicon throttles its frequency to conserve power. Since robots must be designed to accommodate maximum power dissipation anyway, most applications should disable these features, and configure Linux to use a power governor that prevents the CPU from throttling down.

A combination of `SCHED_DEADLINE` scheduler, `RT_PREEMPT` kernel, and core pinning might be sufficient for the requirements of certain robotic applications, especially after careful tuning and validation [15]. However, it is also possible to delegate the most critical components of the system to a secondary processor running a real-time OS. This improves isolation, and makes it easier to validate that the real-time components are working as expected, while leaving more freedom to engineers working on higher-level behavior. Relying on a co-processor also opens up more hardware choices because some reliability requirements no longer apply to the main processor. Further discussion of safety architectures and certification is beyond the scope of this work.

Even systems that adopt the "coprocessor" model can stand to benefit from the real-time tools described in this section. Most importantly, the latency and reliability of driver and actuator nodes can be noticeably improved when by using a real-time scheduling class. More consistent latency numbers reduce the likelihood of overflowing buffers leading to data loss, even if the platform does not satisfy all the requirements for real-time operation.

Thankfully, the ROS community has already realized the importance of real-time guarantees for the coming generation of robots, and significant efforts are already underway to make ROS 2 capable of real-time operation [17, 19].

4.3 Determinism

As we have seen in Sect. 4.2, it is hard for ROS-based nodes to meet timing deadlines. It is nevertheless desirable to at least be able to make guarantees about the *outputs* of the system. We call this property *determinism*. A deterministic system produces the same outputs when provided with the same inputs. Unfortunately, the design and implementation of ROS make it hard to build deterministic systems [20]. This section discusses why this is the case and presents possible avenues for mitigation of the problem.

Determinism is a sensible property for any robotic system. During development, lack of determinism causes automatic tests to be unreliable and burdens developers with false-positive failures. It also makes it hard to measure the effect of changes to the codebase because variations in performance could also be attributed to randomness. On the road, determinism is required for safety by ensuring that real-world behavior matches offline testing and is repeatable over time.

At a conceptual level, non-determinism can arise both at the *algorithm* or at the *system* level. Many common algorithms in robotics have randomized components, for example particle filters, RANSAC, and graph search algorithms. These algorithms

will produce different results when run on the same inputs because their execution is dependent on the results of Random Number Generators (RNGs). We will not be discussing non-determinism at the level of individual algorithms because there are well-known strategies to tackle this challenge (seeding the RNGs, making sure that their use is consistent across different execution paths, etc.).

For the purposes of this section, our definition of determinism is practical, rather than academic. We are satisfied that a system is deterministic if multiple runs on the same recorded sensor data produce the same outputs. The typical example involves running the same 1-min road scenario several thousand times and checking that the resulting car trajectory is identical in all runs. A more extensive test would also simulate additional CPU/GPU load to ensure that the stack is robust to unexpected increases in computational requirements. In other words, we are usually not interested in the reproducibility of internal states of the system as long as the final outputs are consistent.

At the system level, non-determinism means that the stack produces non-reproducible results even when all of the algorithms it uses are deterministic. This happens because of *timing* effects that arise during the interactions between different nodes. In a ROS stack not explicitly designed for determinism, the normal variability in the execution time of operations can lead to changes in the *values* of the outputs, and not just in the time that it takes to compute them.

Indeed, some degree of variability in the execution times of individual tasks is expected (and unavoidable) on modern hardware due to the large impact of CPU caches and memory accesses. It is the responsibility of the framework to protect application developers from these details and provide a platform that guarantees deterministic execution of algorithms. ROS provides no such facilities, leading most stacks to become annoyingly non-deterministic as soon as they grow beyond a handful of nodes.

Figure 3 shows one example of how variability in timing (*jitter*) can cause non-determinism. Tasks A and B are both scheduled to run at the same, fixed, frequency (for example, 20 Hz). Node B subscribes to messages that A published on each tick. The dark shaded areas represent the execution time of the nodes. In this example, their widths are roughly equal, representative of well tuned algorithms with consistent running times. The lightly shaded areas represent how long the node has to wait in runnable state before the OS finally schedules it to run on a CPU. Their widths are much more variable because of the combined effect of communication and scheduling delays.

The first two iterations of nodes A and B produce the expected result as values computed by A are available to B before it is scheduled to run (message passing is represented by arrows). In the third iteration, however, an especially long scheduling delay for A causes B to not receive A's output in time and thus forces B to use an old value and produce a different output. As a result, the complete system is not deterministic even if both of its component nodes are.

Non-determinism is related, but not equivalent to real-time. Real-time is about making sure that tasks are scheduled and completed before a deadline. In particular,

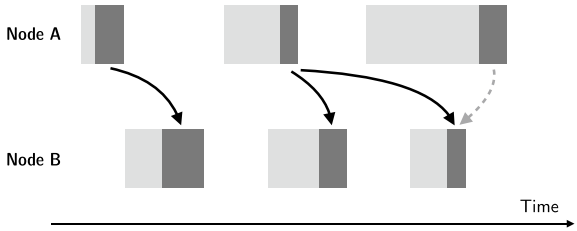


Fig. 3 In the third iteration, node A is scheduled to run after node B, therefore B is forced to reuse a previous value of A's output

execution times can still be jittery and thus potentially lead to non-determinism of the kind shown in Fig. 3.

However, common sense and practical experience suggest that many of the techniques commonly employed to develop real-time systems have the side-effect of increasing determinism too. Priority-aware scheduling, less locking and dynamic memory allocation in critical sections all contribute to eliminate potential sources of jitter and thus make real-time systems more likely to be deterministic as well.

In general, the 3 most important sources on non-determinism in the typical ROS stack are:

- *execution reordering*, as shown in Fig. 3. The two nodes have a data dependency (B subscribes to data published by A) which should in theory be respected because they are scheduled to run at the same frequency and in the right order. However, jitter can break this data dependency and force B to use old data if it does not receive the input from A in time.
- *non-atomic message delivery*. Since publishers and subscribers are connected peer-to-peer through independent network sockets, it is impossible to predict the order in which different subscribers are going to receive messages from different topics, let alone when the OS will schedule them to run on the CPU. Different components might therefore be using different versions of the same data, some of which are stale. This problem is typical of distributed systems like distributed databases, and naturally causes non-determinism.
- *dropped data* caused by overflowing subscriber queues on nodes that are falling behind on their callbacks. In many cases, nodes are only interested in having the "latest" data point. In ROS, this is usually achieved by limiting the size of the subscriber queue, so that older messages are dropped to make space for newer ones. Jitter influences which messages are dropped and therefore couples computation with timing in ways that are difficult to predict and control.

While all of these patterns are visible even on a two-node system, they become drastically more significant as the number of nodes in the system increases.

So far, we have only discussed non-determinism in terms of its effect on the *outputs* of the stack. However, non-determinism also affects the *end-to-end latency* of the system, as expressed by the time it takes for new sensor data to be reflected in the control outputs. Even if the outputs are produced at a constant frequency (as in node B in Fig. 3), actual end-to-end latency is affected when non-determinism

forces stale data to be used. Effectively, computation time allocated to downstream components (i.e. node B) is wasted because upstream components (i.e. node A) are unexpectedly late.

Non-deterministic behavior is hard to reproduce by definition, making it harder for developers to even become aware of its existence. The rest of this section shares some of the insights we gained while trying to make ROS more deterministic.

4.3.1 Towards Deterministic Systems

In light of the discussion about reordering and aliasing effects, the first solution that comes to mind is to reduce jitter at the algorithm level, extending determinism to *timing* as well as the outputs themselves. Many algorithms are inherently constant-time, such as Deep Learning models and Kalman filters. Others can be made so by making the implementation aware of passing time, so that the algorithm can tune itself to respect its time budget (for example by adjusting the number of iterations).

However, constant-time algorithms alone can not guarantee system determinism. OS scheduling, communication, and multi-threaded synchronization delays are nevertheless unpredictable and thus partially defeat the purpose of constant-time algorithms. Even in the absence of these macroscopic effects, modern hardware has deep pipelines and large caches that make it largely impossible to predict in advance how many CPU cycles are needed for a code section.

On the other hand, back-of-the-envelope calculations suggest that CPU execution jitter is in the order of nanoseconds whereas OS overhead is in the order of microseconds. Most of the components in a robotic stack operate at 5–50 Hz, around 3–6 orders of magnitude slower. This is the reason why relatively few people are concerned about determinism in ROS: there is ample headroom for jitter to impact execution patterns before its effect becomes visible at the application level.

Regardless, a well-designed architecture and framework should void amplifying jitter by 3 orders of magnitude. The remainder of this section introduces measures that we have found beneficial to achieve this goal.

The first idea is to eliminate rate-triggered nodes (e.g. those using `ros::Rate`) in favor of data-triggered computation (triggered by message subscriptions). Relying on rate-triggered nodes effectively introduces a first-order (albeit implicit) dependency on the passage of time, rather than on the world state. Behavior begins to be affected by *when* messages arrive as well as *what* they contain. A stack with many rate-triggered nodes is especially prone to execution reordering issues, and generally hard to debug because it is not immediate to trace the flow of data across nodes. Lütkebohle [21] presents tracing and debugging tools that can be used to track down this type of misbehavior.

Of course, not all nodes can be converted to data-triggering. The most notable exceptions are sensor drivers, which typically publish data at fixed intervals driven by the hardware. This is actually quite fortunate because timing effects can (and should) be limited to the boundaries of the stack (the sensor drivers). All other components should derive their sense of timing from sensor data, either directly from

their timestamps, or indirectly through intermediate computation that conserves the timestamp.

In other words, data-triggered architectures encourage thinking about *computation*, and not *communication* as often emphasized in ROS. The question becomes "what should the stack do when this new piece of data becomes available", and not "what nodes should be connected together"?

The natural conclusion is to create a new execution platform that is based on *data* dependencies between nodes and not on *delivery timing* as in conventional message passing. Such a platform is presented in [9], albeit limited to offline testing and not on-road use.

Once the data dependencies across different nodes of described are described explicitly, rather than implicitly through the pub/sub configuration, it is possible to express the computation graph as a Directed Acyclic Graph (DAG) of pure algorithms, rather than concurrent threads as in plain ROS. The idea is to maintain pub/sub as a low-level communication primitive but augment it with a dependency-aware framework layer that helps developers make their algorithms deterministic by default.

The OROCOS [8] framework for real-time robotics follows a similar concept, by introducing *Data Flow ports* that the runtime uses to guarantee that data is shared between nodes in constant time. While ROS schemas only define the structure of messages, OROCOS schemas also apply to the interface of nodes and define all of their inputs and output topics.

Together with a *scheduling policy* that specifies how nodes should react to new incoming data (following the ideas in [9]), this provides all the information needed for an executor to schedule tasks across CPU cores while respecting their data dependencies and thus maintaining output determinism even in the presence of jitter.

This section introduced the problem of deterministic behavior which we believe is a core requirement for both production and development. While ROS does not actively prevent the creation of deterministic systems, some of the implementation details and widespread patterns in the community do little to help application-level developers. Therefore, we discussed measures to improve determinism in ROS, as well as more conceptual ideas about rethinking the execution platform.

4.4 Node Lifecycle and Configuration

Since a complete ROS system is composed of dozens of nodes and nodelets, ROS users need a tool to start up all of these components. `roslaunch` is ROS' answer to this problem. It starts and manages the constellation of OS processes required to operate a ROS stack according to the definitions expressed in an XML file (the *launch file*). Launch files can include and extend each other, reducing the effort needed to describe a complete system.

Launch files are also often used to configure the *ROS parameter server*, a global key-value store used for shared node configuration. Nodes can read parameters by

using the ROS client API (e.g. `roscpp` or `rospy`), which makes calls to the parameter server through XML-RPC over HTTP.

Together, `roslaunch`, `rosparam`, and launch files provide a basic solution to the problem of managing the *lifecycle* of nodes. While nodes spend most of their time in the "running" state, they first need to be configured before they can start processing data. Furthermore, a subset of nodes might temporarily linger in an "errored" state without affecting other nodes. The ROS API does not provide facilities to share this kind of lifecycle state information with `roslaunch`, leaving developers to implement customized solutions at the application level. It should be noted that ROS 2 does provide facilities for lifecycle management [6].

Lifecycle management is not especially important during nominal operation of the stack because a fairly monolithic system like an AV should never need to start or restart nodes on the fly. The lack of a lifecycle API is most problematic during stack startup, when `roslaunch` spawns all nodes in parallel, without considering the dependency relations between them. Before they can initialize properly, nodes might need to allocate system resources (like memory) or wait from data coming from other nodes. This means that many nodes are likely to spend a significant amount of time in the "errored" state, generating many false negatives that make it harder to debug startup failures.

Similar problems also affect partial or total error conditions. `roslaunch` does not monitor node health beyond checking that the process is still alive. Therefore, many common failure modes like deadlocks or infinite loops might still go undetected. Most ROS developers are forced to implement their own health checking components, implemented either as a periodic timer on the nodes or as a frequency check on published topics. All things considered, monitoring individual instances of nodes is fairly straightforward.

It is more complex to monitor the overall health of the stack as error conditions propagate downstream from node to node. Some of these conditions might be unrecoverable and must be handled by activating a fallback plan that slows the vehicle down as safely as possible and pulls over. Others failures might be transient and tolerated for a while to allow the subsystem time to recover automatically. Ultimately, there is little the framework can do to solve these challenges in a generic and reusable way.

Problems with `roslaunch` begin even before the stack is launched. The XML stack definition is very dynamic, and can pull configuration options from a variety of places (including arbitrary Bash and Python snippets). However, vehicle hardware is usually fairly stable across revisions, causing this flexibility to show up mostly in terms of bugs and missing opportunities for static checks. In practice, it is necessary to add an additional higher-level description format that can be statically checked for consistency, and reduces the verbosity and duplication that often plagues XML launch files.

As in most configuration formats, design decisions revolve around the tradeoff between expressivity, verbosity, and complexity. This means that expressive and succinct stack description formats can be adequate as long as their expansions can be checked statically. On the opposite side of the spectrum to `roslaunch` are

orchestration tools that provide a full-blown Python API, like [7] and ROS 2's in-progress implementation [31].

A core feature of ROS is that the computation graph is dynamic and can be altered during the operation of the robot. Subscriptions can be created or destroyed during nominal operation just like they can during the configuration stage. Taking advantage of this feature is rarely a good idea in a production system, and indeed most of the open-source code groups all subscriptions during node initialization. Indeed, it is possible to take this idea even further, and completely forbid subscriptions after nodes are done with configuration. This restriction enables many sanity checks on the structure and connections in the computational graph. The same goes for parameters read from the ROS parameter server. In general, tighter boundaries between the configuration and runtime phases pay dividends in terms of debuggability and reliability.

In conclusion, the main message of this section is that the flexibility and configurability that were core design goals of ROS in the early days naturally show their limitations when applied to a production system. Some of these issues are just (fixable) implementation shortcomings. Others require a deeper look at the role of the framework in a production safety-critical application.

5 Complementary Tooling

A large fraction of the value of ROS as a prototyping framework is the extensive set of open-source tools that the community has built for visualization, logging, and debugging. This section describes the ones that we have found to be the most challenging to adapt for large-scale AV development.

5.1 Data Logging and Schema Evolution

Any AV development team will need powerful Data Engineering tools to track progress, identify and reproduce challenging road situations, and extract training data for Machine Learning. This section investigates how the default tooling in ROS handles this need, and how it can be extended.

The base ROS distribution comes with `rosbag`, a widely used data recording and replay tool. Thanks to the flexibility of the pub-sub model, the rosbag recorder acts as a standard node and subscribes to a specified list of topics, recording any published messages to disk in the ROS bag format.

Because of high resolution cameras and LIDARs, the aggregated data rate from sensors can easily reach 1GB/s or more. It is uncommon to run the `rosbag` recorder as nodelet because crashes in the main processing code would also bring down the recorder, making it harder to use the logs for debugging. On the other hand, running

the recorder(s) in a separate process means that logging incurs serialization and IPC overhead, which can be significant at such high data rates.

Besides these performance issues, ROS has an inadequate Data Engineering story because data logging uses the same serialization format as standard pub/sub. The ROS message serialization format (as implemented in the `genmsg` family of projects) has low overhead, both in computation time and memory. The generated code is also convenient to use, because it uses mutable STL types in C++, which can serve double-duty as in-memory data structures. ROS messages are several times faster than comparable alternatives, most notably Protobuf, the open-source serialization format by Google. Due to their simplicity, ROS messages also compare favorably with zero-copy serialization formats, like Cap'n'Proto and Flatbuffers.

The main drawback of the ROS serialization format is that serialized representations are bound to the version of the message definition (i.e. *schema*) that generated them. In other words, changing a message definition makes all previously recorded messages unreadable. This is because `rosbag` records the MD5 hash of the message definition stored in the bag and refuses to deserialize a message if it does not match the version embedded in the executable. Therefore, even the most trivial changes, like adding a new constant definition, will prevent bags from being read at all.

On the other hand, this design choice has effectively frozen the basic ROS data types since the first release, spawning a huge ecosystem of compatible software that uses consistent data types for basic concepts in Robotics (e.g. positions).

Many other serialization formats allow some form of *forwards compatibility*: as long as schema changes follow certain restrictions, newer versions of the deserialization code can still read serialized buffers generated by older code. A common design to achieve this goal is implemented in Protobuf. Fields contents are indexed by a lookup table, so that missing or unknown fields can be represented with no issues. By contrast, the ROS serialization format closely packs field data along the wire for simplicity and performance and therefore has no room for such additional metadata.

The lack of forwards compatibility causes issues both for offline data processing and for day-to-day development. Data recorded during road testing must be processed to extract useful metrics like the number of miles traveled, or the frequency of specific events on the road. Such processing often happens on pools of cloud machines. The code that powers the backend should ideally be independent from the vehicle codebase to reduce coupling and infrastructure headaches. There are 3 main alternatives to achieve this goal, each with its own set of tradeoffs:

- deserialize messages using Python, which does not need access to the schema at build time,
- write C++ code to deserialize messages on the fly given their schema and construct a runtime representation of the message that can be used by C++ code, albeit with a less convenient syntax compared to standard classes,
- migrate rosbags and messages to a schemaless format, like Avro or Parquet, that are a better fit for the ecosystem of big-data tools.

The lack of forward compatibility also makes it annoying to test a newer version of the code on older recorded data. If any of the message definitions have changed since

the data was recorded, the ROS runtime will refuse to deserialize messages even if the on-wire format remained identical. The solution offered in the ROS distribution is to write *bag migrations*, Python scripts that leverage dynamic typing to read messages in the old schema, and produce their equivalents in the new schema after appropriate conversion. While the default implementation of rosbag migrations reads and writes bag files on disk, it is possible to extend it to perform on-the-fly migration, avoiding the need for massive scratch space.

When combined with appropriate tooling, a combination of the afore-mentioned solutions is acceptable for both backend jobs and developer workflows. A more drastic solution is to replace ROS' native serialization format with an alternative that supports forwards schema evolution. Indeed, the Apollo open source self-driving car project developed a `roscpp` plugin that adds support for Protobuf-encoded messages in the standard pub/sub ecosystem [3]. The main drawback of this approach is that requires potentially extensive changes to client code as well, most notably in some of the core ROS debug tooling (e.g. `rostopic`).

5.2 Visualization

The ROS ecosystem includes a wide variety of visualization tools, ranging from generic ones included in the core distribution, to specialized programs available from the wider community.

The workshorses of the ROS visualization story are `rviz` and the `rqt` suite. The former is a 3D tool often used to visualize 3D models, trajectories, and point clouds. `rviz` can be extended by writing C++ plugins that hook into its 3D rendering pipeline, and can thus visualize new data types or add new UI behavior. While rviz is focused on spatial data, the rqt suite provides the building blocks for visualization of scalar and 2D data. It provides many Qt-based widgets for plotting, drawing text, and displaying images. The upstream distribution also includes several standalone tools that leverage these widgets for a specific task (e.g. `rqt_image_view` for displaying image topics).

Overall, the suite of open-source visualization tools is fairly complete and modular and can be adapted to a variety of tasks. On the other hand, this flexibility imposes a fair amount of effort to set up consistent views and workflows that can be shared by many Engineers. Furthermore, visualization tools are strongly coupled to the running ROS master and the message schemas, thus running into many of the same issues that we have discussed with regards to data storage. In addition to that, reliance on the full ROS distribution means that the tools can only run on compatible Linux workstations with ROS installed.

These shortcomings led Cruise to develop Webviz, a unified visualization tool that runs in the browser. The application is open source and available on GitHub [30]. Webviz follows a similar philosophy to the `rqt` family of tools: topics of different types of data are visualized in different *panels* that can be moved and resized within the browser window to adapt to specific visualization needs (see Fig. 4 for an example

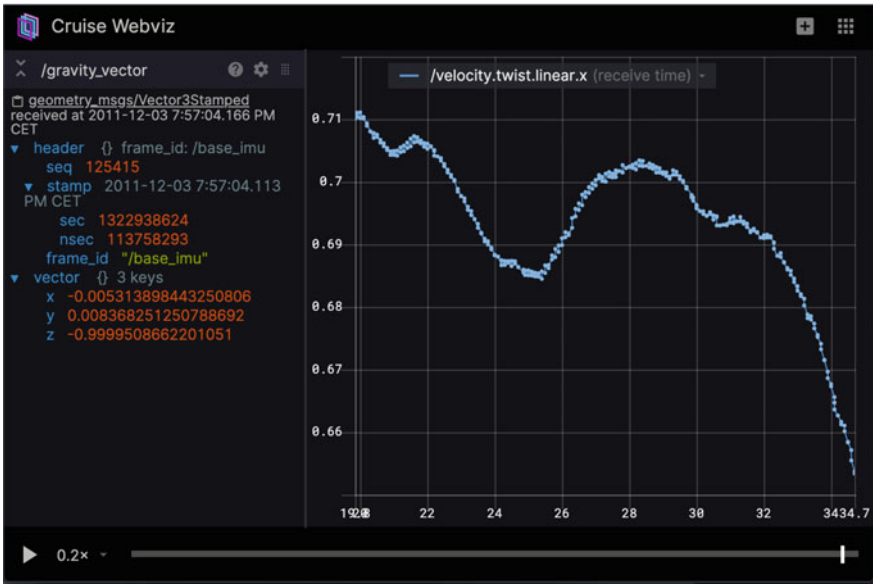


Fig. 4 `webviz` is a browser-based visualization tool that integrates the functionality of `rviz` and the `rqt` family of tools. it is extensible in Javascript and doesn't need a ROS installation

view). New panels and visualization plugins can be written in Javascript and therefore shared between users with no need for compilation or installation.

Unlike upstream ROS tooling, Webviz is not tied to specific versions of the message schemas, since it can leverage the dynamic typing features in Javascript. As a result, it runs on any machine with a web browser, without having to install the ROS distribution first. Webviz can read ROS bag files with no server-side support (deserialization runs within the browser) or connect to a running ROS master using the Websockets protocol, similarly to the `rosbridge_suite` in the main distribution.

5.3 Build System

One of the first concepts that ROS developers learn is that they should partition their stack in packages. Each *package* is a collection of nodes, data assets, scripts, and message definitions that solves a specific problem. Each package can contain code in different languages, and multiple packages can be stored and built together in a single *workspace*.

From the point of view of the build system, this composability is achieved through ROS' build system: `catkin`. `catkin` is a set of conventions and wrappers around CMake, a widely used open source build system in the C/C++ world. What `catkin` adds to CMake is the ability to compose a single workspace out of multiple ROS

packages, and track their dependency relationships so that source code can be built in the correct order. Catkin also handles ancillary tasks like code generation and installation, and has built-in support for the primary languages in the ROS ecosystem (C++ and Python). Catkin is one of the main enabling factors that brings flexibility into the ROS ecosystem: packages from multiple sources and repositories can be dropped into a single workspace and integrate seamlessly.

Catkin is a good fit for a heterogeneous environment like an open-source community but does not work as well in industry. Robotics companies often have a monolithic codebase ("monorepo") and deploy to a single, tightly controlled platform. Cruise migrated from Catkin to Bazel, the open-source version of Google's internal build system. Google uses Bazel to handle their massive repository of billions of lines of code and thousands of developers.

The main benefits are faster builds thanks to remote caching of build artifacts, and a workflow refined by Google over many years of working with a monorepo. Compared to Catkin (even with `ccache` support), Bazel cut full build times on developer workstations from an hour to a few minutes, thanks to remote caching and better scheduling on parallel hardware. The flexible build definition language embedded in Bazel allows code generation for messages, packaging, and integration testing that is compatible with the ROS ecosystem. Thanks to an optimized in-memory state, no-op builds on a large workspace with hundreds of Catkin packages dropped from several minutes (most of which are spent running wrapper scripts) to just a few seconds.

Bazel also allows dispatching build actions to a remote pool of workers (similar to `distcc`) where they can be executed with greater parallelism. There are several open-source implementations of such remote runners, as well as some hosted solutions to avoid managing cloud infrastructure. Remote execution and caching are also supported for tests and allow skipping running tests when none of the source files they depend on have changed. This is especially important for Continuous Integration (CI) environments because it saves large amounts of CPU time on large test suites.

Bazel and Catkin follow diametrically opposed philosophies about development environments. Catkin values configurability, extensibility, and integration with the OS distribution. Bazel is designed with the goal of complete isolation from the host environment (all dependencies are built from source) and more finely grained dependencies between components in the workspace. Unfortunately, this means that a migration from Catkin to Bazel is not straightforward because large parts of the ROS runtime rely on Catkin's implementation details (most notably, the core libraries in `rospack`, `rospkg`, `pluginlib`). ros-bazel code repository [26] is an open-source starting point for such a migration.

6 Simulation and Testing

Any framework that wants to be adopted in safety-critical systems needs to have a robust story around automated testing and simulation. This section discusses how ROS satisfies these needs from the point of view of industry and how to improve the situation.

6.1 The Need for Simulation

For the past several years, the leading AV companies have reported that they have driven thousands of miles autonomously on public roads in California [10]. If we were to include miles driven in simulation, however, the counts would be orders of magnitude greater. Full-stack simulations are a fundamental tool for the development of AV stacks, even more so than other robotic systems.

In fact, road testing has several important limitations that imply that any serious AV effort is bound to rely heavily on simulation:

- Many subsystems in modern AVs are based on Machine Learning. Statistically speaking, road testing is not a great fit for validating these systems, as the rare edge cases that challenge Machine Learning models happen infrequently on real roads. it is much easier to simulate and reproduce them in virtual environments.
- Road testing is expensive, because human safety drivers must be present and engaged at all times, ready to take over in case the AV stack malfunctions or is unable to proceed safely.
- Simulation allows Engineers to try out exploratory code without the safety concerns involved in road testing.

Overall, even as AV companies are working to increase the number of miles they drive on public roads to prepare for public release of their products, simulation will remain the most important validation and development tool for thousands of Engineers.

Automated testing conventionally includes a spectrum of activities that range from unit testing individual algorithms to full-stack tests that even exercise the sensor hardware. By definition, there is little to say about unit testing and ROS, because that category of tests is designed to be independent of the underlying platform.

On the other hand, integration tests can benefit from support by the framework, as their aim is to validate the interaction between different components of the system. For practical reasons, most integration tests are usually designed not to require sensor and actuator hardware, let alone a real vehicle. This means that some subcomponents need to be either simulated or their outputs loaded from data recorded on the actual vehicle.

Typically, simulators are used to provide the system with a realistic physical model of the vehicle for full-stack tests, whereas sensor data is loaded from a recorded bag.

However, pre-recorded sensor data limits the usefulness and scope of tests. As soon as the behavior of the vehicle deviates from what was recorded, sensor data becomes useless (e.g. the camera view points are different) and the whole scene must be discarded. This makes it harder to establish consistent baselines over which to track improvements, because scenarios must be refreshed over time.

To alleviate this issue, many AV companies have developed more complete simulation stacks that can also produce realistic sensor data using 3D rendering techniques similar to those used in video games. Since the simulation engine has full knowledge of the world state and contains generative models for sensor data, it can react consistently to changes in behavior of the simulated vehicle (e.g. by rendering the correct camera view). Dosovitskiy et al. [11] is an open-source implementation of this idea.

A comprehensive integration testing plan includes tests that exercise each functionality in isolation (e.g. localization, behavioral planning, ..) with fixed inputs, as well as broader tests that validate how information flows from one component to another (e.g. from trajectory planning to controls). All these scenarios lead to a dazzling amount of possibilities for the configuration of the framework during testing. Because of its roots as a prototyping tool for non safety-critical applications, ROS does not cater especially well to any of these workflows.

6.2 ROS Simulation Tooling

Indeed, the blueprint for integration testing in ROS is fairly crude. It involves `roslaunch` to spin up a portion of the stack, and `rosbag` to replay recorded data. Since the ROS API natively supports fetching the time from a `/clock` topic published by `rosbag` during the replay, nodes can easily sync up to the shared simulated clock. This means that timestamps can be compared across runs of the same data file as long as nodes use `ros::Time::now()` instead of fetching the current timestamp using native OS APIs (eg `libc`'s `time()`). ROS-friendly simulators like *gazebo* are configured by remapping topic subscriptions and publications so that nodes can consume simulation outputs instead of recorded data.

This simplistic approach quickly shows its limitations on more complex stacks for the following reasons:

- The framework has no knowledge of the data dependencies between different subcomponents (e.g. localization, object detection, etc.). Therefore, topic remappings needs to be managed manually to choose whether to use recorded data or simulated inputs for each node. As data travels downstream through the computational graph, the number of such combinations grows exponentially, thus limiting the number of configurations that can be tested.
- Simulations and tests are affected by determinism issues even more so than operation on the road, often because the execution platform is not managed as carefully as the real robot (e.g. shared cloud machines instead of dedicated hardware on the vehicle).

- In large scale deployments supporting thousands of developers, it is often useful to run tests at higher or lower speeds than possible on real hardware. Faster runs are useful to reduce the time it takes to test a subsystem of the complete robot. Slower runs make it possible to run batch simulations on slower, cheaper hardware than installed in the robot. ROS natively supports neither. Slowing down the replay rate easily hides situations where the overall system is running out of CPU time. Increasing the replay rate over a certain (unpredictable) threshold causes subsets of nodes to miss ticks or use stale data.

6.3 Towards a Better Simulation Platform

The challenges described above are closely connected to the themes we explored throughout this chapter:

- Reducing the reliance on the pub/sub model in favor of explicit data dependencies between components of the stack (as described in Sect. 4.3 and [9]) makes simulation both more performant and more flexible. Because the code describes the data flow explicitly, the stack's operation does not depend on message delivery timing. Therefore, simulations can run at higher or lower rates without loss of determinism.
- When stack configuration is tightly integrated with the data-flow architecture, simulation authors can freely mix recorded and simulated data without having to manage topic remappings manually.
- A build system with granular dependency tracking like Bazel can automatically cache test and simulation results, greatly reducing the cost and time needed for validating code changes.
- Customized and web-based visualization tooling makes it possible to expose simulation and test results to the whole organization and not only to Engineering teams. As driving behaviors grow more sophisticated, it often becomes difficult to design tests with the clear pass/fail conditions commonly used for automated testing. Integrating visualization with testing tooling can aid developers better understand and judge the behavior of the system under test.

7 Conclusion

In this use case chapter, we discussed the strengths and weaknesses of the ROS framework and ecosystem in the context of the development of an Autonomous Vehicle.

Regarding the core framework functionality of ROS, we discussed limitations in terms of performance and reliability. Some of these limitations are due to implementation details and thus can be mitigated by re-implementing or optimizing certain

components. Others are more fundamental and call for the implementation of higher-level abstractions that maintain performance and ease of development across larger stacks.

With complementary tooling, the impedance mismatch between community and proprietary use is again one of flexibility against optimizations for specific use cases. Thankfully, the modular nature of ROS enables the development of replacements that can be shared back to the community.

In conclusion, teams that are attracted to the ubiquity of ROS and choose it to implement their prototype will find it hard to quit the platform altogether as their needs evolve. However, they still can gradually wean off their dependence on ROS, starting with the components that have the most stringent performance and reliability requirements. With some forethought, the usage of ROS can be confined to the "boundaries" of the platform like communication, serialization, and interaction with the third-party code in the ecosystem.

References

1. J. Altenberg, Introduction to realtime linux, https://www.youtube.com/watch?v=BKkX9WASfpI
2. Apex.ai homepage (2019), https://apex.ai
3. Apollo project design doc: Native Support with Protobuf, https://github.com/ApolloAuto/apollo-platform/blob/master/ros/docs/design/native_support_with_protobuf.md
4. Automated Driving with ROS at BMW (2016), https://www.ros.org/news/2016/05/michael-aeberhard-bmwautomated-driving-with-ros-at-bmw.html
5. E. Bendersky, Measuring context switching and memory overheads for Linux threads (2018), https://eli.thegreenplace.net/2018/measuring-context-switching-and-memoryoverheads-for-linux-threads/
6. G. Biggs, T. Foote, ROS 2 Design Documents: Managed Node, http://design.ros2.org/articles/node_lifecycle.html
7. A. Bockenkamp, roslaunch2: versatile, flexible and dynamic launch configurations for the robot operating system, in *Robot Operating System (ROS): The Complete Reference*, ed. by A. Koubaa, vol. 4 (Springer International Publishing, Berlin, 2020), pp. 165–181
8. H. Bruyninckx, Open robot control software: the OROCOS project, in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, May 2001, pp. 2523–2528. https://doi.org/10.1109/ROBOT.2001.933002
9. B. Cairl, Deterministic, asynchronous message driven task execution with ROS, presentation at ROSCon 2018 (2018), https://roscon.ros.org/2018/presentations/ROSCon2018_deterministic_asynchronous_message_driven_task_execution.pdf
10. California DMV Autonomous Vehicle Disengagement Reports (2018), https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2018
11. A. Dosovitskiy et al., CARLA: an open urban driving simulator, in *Proceedings of the 1st Annual Conference on Robot Learning* (2017), pp. 1–16
12. S. Glaser, E. Berger, W. Meeussen, pr2 controller manager ROS wiki page, https://wiki.ros.org/pr2_controller_manager
13. H. Fan et al., Baidu Apollo EM Motion Planner (2018), arXiv:1807.08048 [cs.RO]
14. S. Glaser, realtime tools ROS wiki page, http://wiki.ros.org/realtime_tools
15. C.S.V. Gutiérrez et al., Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications (2018), arXiv:1808.10821 [cs.OS]

16. C.S.V. Gutiérrez et al., Time Synchronization in modular collaborative robots (2018), arXiv:1809.07295 [cs.RO]
17. C.S.V. Gutiérrez et al., Towards a distributed and real-time framework for robots: evaluation of ROS 2.0 comLessons learned building a self driving car on ROS 29 munications for real-time robotic applications (2018), arXiv:1809.02595 [cs.RO]
18. S. Kato et al., An open approach to autonomous vehicles. IEEE Micro **35**(6), 60–68 (2015)
19. J. Kay, A.R. Tsouroukdissian, Real-time control in ROS and ROS 2.0, presentation at ROSCon 2015 (2015), https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf
20. I. Lütkebohle, Determinism in Robotics, presentation at ROSCon 2017 (2017), https://roscon.ros.org/2017/presentations/ROSCon20ROS.pdf
21. I. Lütkebohle, Why and how to achieve deterministic timing with ROS, presentation at ROS-Industrial Conference 2016 (2016), https://static1.squarespace.com/static/51df34b1e4b08840dcfd2841/t/58383609725e25680b8df592/1480078865773/ROS-I-Conf2016-day1-07-luetkebohle.pdf
22. P.E. McKenney, 'Real Time' vs.'Real Fast': How to Choose?
23. NVIDIA Drive AGX Origin (2019), https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orinadvanced-software-defined-platform-for-autonomousmachines
24. S.D. Pendleton et al., Perception, planning, control, and coordination for autonomous vehicles, in *Machines 5.1* (2017). ISSN: 2075-1702. https://doi.org/10.3390/machines5010006, https://www.mdpi.com/2075-1702/5/1/6
25. M. Quigley et al., ROS: an open-source Robot Operating System, vol. 3 (2009)
26. ros-bazel code repository (2019), https://github.com/nicolov/ros-bazel
27. Self Driving Car Engineer Nanodegree homepage (2019), https://www.udacity.com/course/self-driving-carengineer-nanodegree--nd013
28. shm_transport ROS wiki page, http://wiki.ros.org/shm_transport
29. Y.-P. Wang et al., TZC: efficient inter-process communication for robotics middleware with partial serialization (2018), arXiv:1810.00556 [cs.RO]
30. webviz code repository (2019), https://github.com/cruise-automation/webviz
31. W. Woodall, ROS 2 Launch, presentation at ROSCon 2018 (2018), https://roscon.ros.org/2018/presentations/ROSCon2018_launch.pdf
32. E. Yurtsever et al., A survey of autonomous driving: common practices and emerging technologies (2019), arXiv:abs/1906.05113

Nicolò Valigi is a Software Engineer with experience in self-driving cars and drones. He holds a Master's Degree in Aerospace Engineering and has worked on localization, navigation, and Machine Learning algorithms for perception. He is also interested in the low-level programming and high-level design that make robotic systems robust, performant, and pleasant to develop on.

Landing an Autonomous UAV on a Moving Platform Using only a Front Facing Camera

George Stavrinos

Abstract This use case chapter goes through a series of packages developed for the author's MSc Dissertation that allows an Unmanned Aerial Vehicle (UAV) to land on a moving platform, using only a front facing camera. The implemented system was run and tested on both ROS Indigo and ROS Kinetic and includes software that implements landing zone detection [32], velocity estimation [35], global [30] and local planning [31], as well as simulation [33]. At first, a typical way of running a simulated environment is presented, to allow readers to test its functionality and decide if they want to deepen their understanding of the implemented system. Subsequently, each package is presented separately, highlighting the rationale behind each implementation. After all packages have been analyzed, Concluding, some future work is presented, highlighting the weakness of the implemented software and how it can be improved, along with a summary of the system's features and advantages.

Keywords Unmanned aerial vehicle · UAV · Autonomous landing

1 Introduction

The field of autonomous Unmanned Aerial Vehicles (UAVs) has been a very active research topic for more than a decade now. UAVs are not only used in research, they are a new way for companies to transport lightweight cargo, for fire safety organizations to monitor distant areas and for photography enthusiasts to take just the right shot.

With UAVs being so popular in a plethora of different applications, it is quite obvious that their cost can widely vary depending on their built-in hardware and software capabilities. Most of them are equipped with sensors that are needed for a specific task. For example, a consumer grade UAV that its intended use is for aerial

G. Stavrinos (✉)
Faculty of Technology, De Montfort University, The Gateway & Mill Lane, Leicester LE2 7DR, UK
e-mail: gstavrinos@protonmail.com
URL: http://gstavrinos.github.com

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_6

photography is equipped with just an Inertial Measurement Unit (IMU) for stabilization, a Global Positioning System (GPS) sensor for a return-to-home functionality and, of course, a front-facing Red-Green-Blue (RGB) camera. A UAV targeted for research would be equipped with many more sensors, like a laser scanner, a Red-Green-Blue-Depth (RGBD) camera, a vertical RGB camera and other sensors. All these would be attached on a much more damage resilient frame to endure the test phase of the study.

This study focuses on implementing an effective method for UAVs to land on moving platforms with the absolute minimum of required sensors, to reduce cost. The method presented enables a consumer grade quadrotor drone to land on a target that is moving with a constant speed using only a front facing (horizontal) RGB camera.

In order to implement an autonomous landing system that is solely based on a front-facing camera, a series of ROS packages were developed:

- Landing zone detector based on Augmented Reality (AR) tags. [32].
- Transformation velocity estimator to estimate the landing zone's velocity [35].
- Global planner to plan a path towards a future position of the landing zone [30].
- Local planner to plan a path based on the global plan, that will enable the robot to approach and land on the landing zone [31].
- Complementary simulation software to enhance the testing phase of the study [33].

All the mentioned ROS packages have been tested on ROS Indigo and ROS Kinetic and work independently following the distributed ROS philosophy to easily interchange nodes with others. This is also a reason why these packages are offered in separate repositories, to allow the user to choose which ones are needed, without having to download and meet several dependencies.

In Fig. 1 we can see the main difference between this study and all previous studies on the subject of autonomous landing on a moving target. This major difference is the position and the orientation of the RGB camera. In this study we are using low cost UAVs that are only equipped with a front-facing camera like shown in the left part of the figure. In contradiction to a down-facing camera as shown on the right of the figure, a UAV equipped only with a front-facing camera does not constantly have the landing platform on its field of view (FOV), hence it has to estimate the platform's position. For a successful autonomous landing on a moving platform where the robot does not have visual contact with the platform itself during the landing process, the robot has to estimate the trajectory of the moving platform at the moment it achieves visual contact with it, and then plan a path towards a future position of its target (Fig. 2). An illustration of this procedure can be seen in Fig. 3.

In the following sections the implemented packages are going to be presented in detail, followed by installation and execution instructions. Before diving into the implementation details though, a brief background section is needed in order to help the reader understand important topics discussed later.

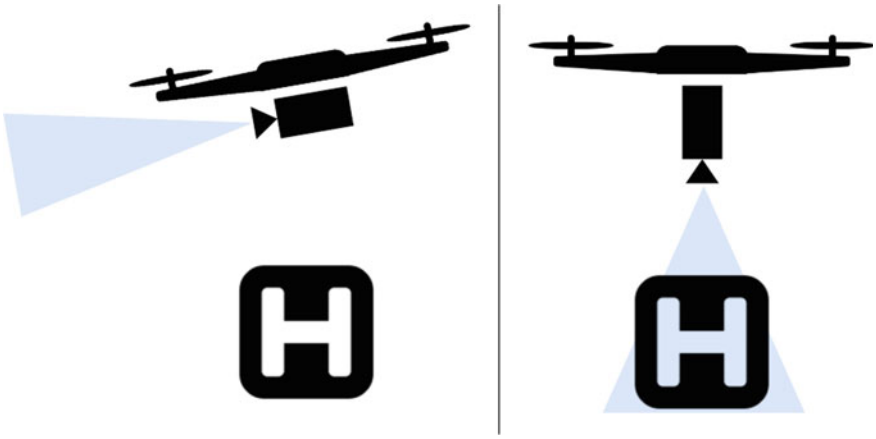


Fig. 1 Camera FOV in different quadcopter setups with left being the forward direction

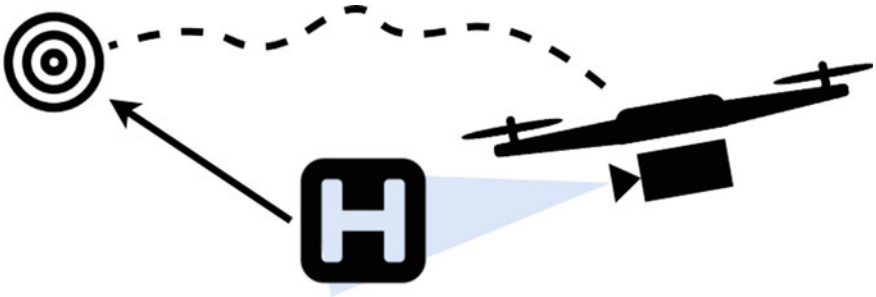


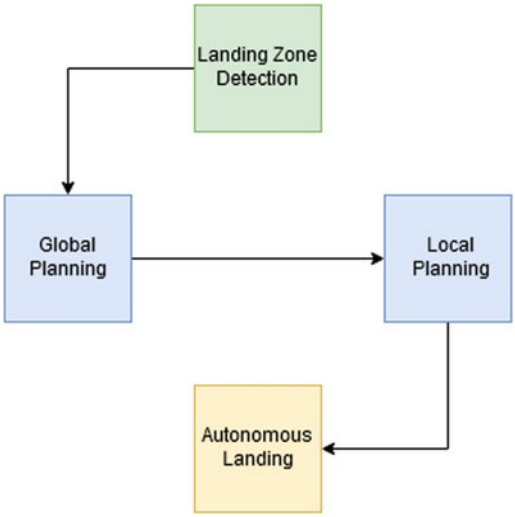
Fig. 2 The general idea of the project. The dotted line represents the drone's planned path, and the arrow the moving platform's estimated trajectory

2 Background

As with almost all interesting ideas, research has been conducted in the topic of autonomous UAVs landing on moving targets. The problem with existing methods is that they require sensors that are not common in the widely spread and less expensive consumer grade UAVs, like vertical RGB cameras. As discussed in the previous section, the lack of a vertical (down-facing) camera creates the need for an estimation of the movement of the target, since it is not visible during the landing procedure.

In order to achieve such task, a thorough look into the state of the art is required. The whole system can be divided into four groups: The Landing Zone Detection, the Global Planning, the Local Planning and the Autonomous Landing. The generalised system's inflormation flow is shown in Fig. 3.

Fig. 3 General system



2.1 Autonomous Landing

Research on autonomous landing on moving platforms using a down-facing camera, can be seen, for example, in the work of Saripalli et al. [25, 26], where an autonomous helicopter is used to land on a H-shaped helipad. Tracking is done on the symbol itself but is limited to one axis. In these implementations a vertical camera is used along with a GPS to keep track of the location of the moving platform and the UAV respectively. Although these studies provide methods that are not directly applicable to this project, the motion planning guidelines could be used to create a new implementation of local planning software.

The method by Benavidez et al. [4] unlike other implementations, utilizes fuzzy logic to control a UAV. The hypothesis of this paper is very similar to the hypothesis of this study regarding the absence of obstacles but again, uses a vertical camera to achieve the landing of the vehicle.

There are also other implementations like in [5, 11, 37] that use vertical cameras to track and land on moving platforms which cannot be applied directly to consumer grade hardware, but can offer details of UAV kinematics, since especially the paper by Beul et al. [5] is related to fast autonomous landing, where kinematics play a very vital role in trajectory estimations.

2.2 Landing Zone Detection

Apart from existing methods, the literature also offers solutions on the landing zone detection problem. Although some of the implemented systems are not ROS compat-

ible, are of vital importance in understanding computer vision, pattern recognition and pose estimation.

Helipad detection for accurate UAV pose estimation by means of a visual sensor, is a published article by Patruno et al. [24], that introduces a new robust way to detect helipads using computer vision. The implemented algorithm provides the pose (position and orientation) of the target platform by extracting features from the generic "H" symbol of helipads. It starts by extracting the marker from the input image, and then computes in real time the pose estimation of the platform. Since the source code of this paper is not publicly available yet, it can only be used for understanding key concepts of this study.

A Library for Virtual and Augmented Reality (ALVAR) [1] was developed by the VTT Technical Research Centre of Finland Ltd. It is written in portable C++ making it lightweight and compatible with various devices, from Windows and Linux, to Android and iOS.

It offers many image-based applications like blob detection and pose estimation. Among its many features, it includes a fast and precise tracker of Augmented Reality (AR) markers.

ALVAR is a very powerful library for which a ROS wrapper called ar_track_alvar has been developed, making ALVAR fully compatible with ROS [2]. At this point it should be noted that the paper by Benavidez et al. [4] discussed earlier utilizes ar_track_alvar for landing zone detection.

Until now, we have seen literature regarding already implemented software for detection and pose estimation of the landing zone, as well some studies that consider UAV dynamics. We have not discussed about planning yet, and the literature offers a wide variety of three dimensional global and local planners.

2.3 Global Planning

A global planner is the software that creates a path from the starting point (the robot's position) to the goal point. Its role is vital, so thorough investigation of the best of 3D global planners is needed. What should be noted is that for this project we do not need a global planner that can create paths that avoid obstacles, since in our hypothesis we assume that there are no obstacles in the area between the robot (UAV) and the moving platform. It should also be highlighted that we are looking for a global planner that not only creates a 3D path but also estimates the goal point, while the target is moving. Since looking for a global planner that also generates a goal point is too specific, we are going to consider global planners that fit our other requirements (working in 3D space and being ROS compatible) and be prepared to develop a plugin for them that could feed them with an estimated goal point.

Efficient Two-phase 3D Motion Planning for Small Fixed-wing UAVs paper implemented by Hwangbo et al. [17], proposes a way to autonomously navigate a fixed-wing UAV. This implementation does not only include a global planner, but also a local planner which we will discuss later in this section.

What is interesting about the global planner developed for the fixed-wing UAV is that it does not transform the 3D world into a 2D environment to plan on. It uses the A* algorithm to find the shortest path between a set of goal points in a 3D discretized set of nodes (grid). The global planner is responsible for creating nodes based on the kinematic capabilities of the fixed-wing UAV.

This method is excellent for fixed-wing UAVs but the kinematic model of a fixed-wing does match the one of a quad copter, which is the most widespread type of consumer grade UAVs. Although this global planning method can not be directly integrated to this project, the overall rationale is of vital importance for the reader.

The ROS community keeps developing interesting and useful packages, and a lot of effort has gone into global planning implementations. Currently there are mainly two packages that fully implement global planning methods: the navfn package [20], and the global_planner package [13].

The navfn package implements a Dijkstra search algorithm [9] in 2D space. The global_planner package completely replaces the first one, since by setting a single parameter to ‘ ‘true” the global planner behaves just like the one in the navfn package. Global_planner except for the Dijkstra search algorithm, implements A* [14] and offers more parameters to experiment with.

The problem with both of those excellent packages, is that they support only 2D planning. There have been some efforts to support 3D planning in ROS [16] but they are no longer actively supported. New packages have been implemented to replace the old unsupported ones, but they are still under development and not ready for use [21].

2.4 Local Planning

A local planner is a software that is responsible for generating velocity commands for a robot, in order for it to follow the path generated by the global planner. It is extremely important for the software that sends commands to the hardware, to be reliable and precise. For this project we need a local planner that not only sends velocity commands to the UAV, but also sends a landing signal to it, when it is located just above the moving platform. Looking for such a local planner is too specific and the current state of the art can not provide us with exactly what is needed, but studying robust local planner implementations will definitely help on understanding some basic concepts.

3D Local Trajectory Planner for UAV developed by Sasiadek and Duleba [27] is an innovative local planning implementation. This method consists of two phases: the decision and the trace mode that allow it to effectively follow the global path and avoid obstacles. Of course, avoiding obstacles in not part of this study, but the rational of this implementation remains relevant.

As stated earlier, this method is based on a two phase structure. The first phase is the decision mode. In this phase the planner disregards any visible obstacles and creates the most effective local path (3D velocity commands) that leads the robot to

the next point of the global path, based on the kinematic capabilities of the vehicle. The second phase is the trace mode. This phase is triggered when an obstacle blocks the path defined in decision mode. The trace mode creates an alternative path based on the one generated in decision mode. The trace mode is always triggered in a distance that will not make the robot completely stop to avoid collision with an obstacle.

The paper by Sasiadek and Duleba provide an effective local planning method that focuses on obstacle avoidance. The implementation even solves the problem of moving obstacles and provides enough information to recreate the algorithms. Unfortunately the source code is not publicly available.

Efficient Two-phase 3D Motion Planning for Small Fixed-wing UAVs [17] has been discussed earlier in this section.

The global planner of the method by Hwangbo et al., uses a grid to generate a path. The nodes of this grid are not dense enough to allow for navigation without a local planner. The local planner developed uses tree search on all possible motion states of the fixed-wing UAV. The method tries to utilize future states to generate smooth curves due to the limited maneuverability of fixed-wing UAVs.

As stated in the global planning section, this paper is very innovative and effective for fixed-wing UAVs, but it is not directly applicable to quadcopter UAVs. Since the source code is also not available for this method, the paper is the only source of information for information on the local planner's implementations.

ROS offers out-of-the-box navigation solutions for ground vehicles that depend on 2D navigation. Some of the local planners are dwa_local_planner [10] which is a ROS compatible implementation of the *The dynamic window approach to collision avoidance* algorithm [12] and teb_local_planner [36] which a ROS compatible implementation of the *Timed Elastic Band* method [18].

Both of those ROS packages provide 2D local planning implementations. Autonomously navigating quadcopters using 2D software is not impossible, but it would not provide us with optimal solutions. We are not going to delve deeper into those two packages even if they are sophisticated solutions for 2D navigation. Using them for this project would provide a foundation to build on, but their complexity and capabilities in 2D navigation surpass the goals of this project.

As expected, investigating into the state of the art did not provide us with a direct solution to our research problem, but based on previous work and the needs of the project a new series of software was developed. The following sections provide a thorough look at all the implemented packages.

3 Testing (and Installing) the Packages

The implemented packages have been tested on ROS Indigo and ROS Kinetic. There is no specific configuration needed to use the packages with a real robot, but when used in simulation, Gazebo 7 or newer [28] is needed, which is not installed by default in ROS Indigo. The process of installing Gazebo 7 on ROS Indigo though, is as easy as uninstalling the previous (Gazebo 2) version, and installing Gazebo 7

through the binary files distributed by the OSRF [23] repositories. If users are using other versions of ROS, they would have to make sure that all the dependencies of the packages are met, and that they are prepared to fix compatibility problems that may arise.

The packages needed are all available through the Github platform and are the following:

1. bebop_gazebo [33]
2. ar_helipad [32]
3. tf_velocity_estimator [35]
4. aerial_local_planner [31]
5. aerial_global_planner [30]

Installing these packages follows the same procedure as with every ROS package, meaning that the users would have to clone the repositories on their catkin workspace and then build them with the catkin_make (or catkin build) command.

3.1 Running the System

Before moving on how to use the packages, it should be highlighted once more that the planning packages assume that there are no obstacles between the robot and the moving platform's trajectory.

You need to run one executable for each of the packages mentioned above. The order of execution is the following: (Note: each command should be executed in a separate terminal).

– roslaunch bebop_gazebo bebop_moving_helipad.launch
– roslaunch ar_helipad bebop_helipad_individual.launch
– rosrun tf_velocity_estimator velocity_estimator.py
– rosrun aerial_local_planner local_planner.py
– rosrun aerial_global_planner global_planner.py

WARNING! The first command runs the Gazebo simulation for the Parrot Bebop. Gazebo often fails to start the controllers successfully. If you get the "Failed to start controllers: controller/position, controller/velocity, controller/attitude" message, please kill the process and try again, before running the rest of the commands.

3.2 Parameters

In the bebop_moving_helipad.launch file in the bebop_gazebo package, you can set a series of parameters to experiment with different setups. These parameters are:

1. Helipad starting position on the X axis.

2. Helipad starting position on the Y axis.
3. Helipad position on the Z axis.
4. Helipad V_x (velocity on the X axis).
5. Helipad V_y (velocity on the Y axis).
6. Seconds before the helipad starts moving (increase this if you don't have the time to run all the necessary commands before the helipad starts moving).

The configuration of the ar_helipad package allows you to modify the expected landing zone. More specifically, the expected AR marker can be set along with its size, and if the landing zone contains a bundle of markers or a single marker.

The tf_velocity_estimator package uses two parameters. The first one is the name of the target tf, which is "/helipad" by default, and the size of the velocity estimations to be saved for future processing, which defaults to 10.

The global planner can be configured for different robot capabilities, since the max velocity can be modified. At this point, it has to be noted, that the odometry frame and the robot frame are hardcoded to "/odom" and "/base_link" respectively, but this is something that might change in the future.

The local planning software can also be modified for different robots. The mathematical formulas used to determine the commanded velocity are parameterized based on the minimum and maximum velocity of the UAV. The local planning software can also be modified in the terms of landing tolerance. As we discussed earlier, the local planner uses some conditions to determine where the robot should land on the moving platform. The distances to be checked can be modified in order to achieve better performance or slightly variant goals.

4 The System in Detail

Landing an autonomous UAV on a moving platform using only a front facing camera can be achieved with a set of software that is listed below:

1. Landing Zone Detector: The software responsible for detecting the position of the moving platform relatively to the robot.
2. Landing Zone Velocity Estimator: The software responsible for estimating the velocity of the moving platform, based on the positions provided by the landing zone detector.
3. Global Planner: The software responsible for generating a path leading to a future position of the moving platform.
4. Local Planner: The software responsible for sending velocity commands to the robot in order to follow the planned path.

Before analysing each of the various parts of the system, some assumptions and contraints have to be highlighted. The list below shows such limitations along with a small description:

1. The landing zone is moving with constant speed and on a straight line: These limitations are part of the system's first iterations, which will be addressed in future

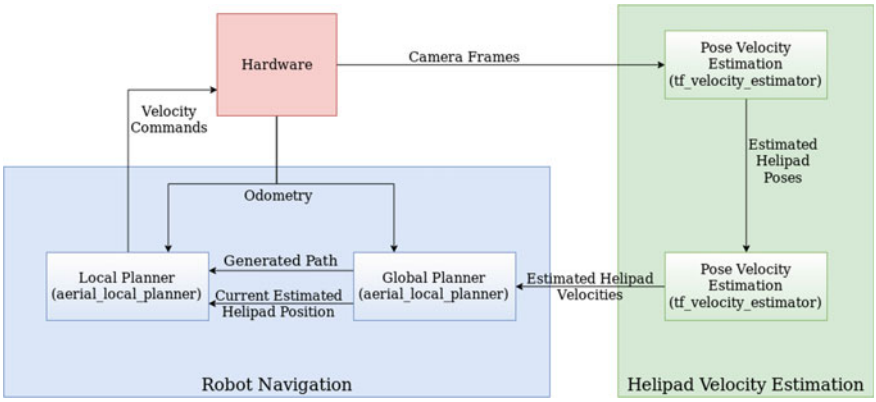


Fig. 4 The architecture of the overall system

work and allow it to predict future positions of the landing zone (complementatry to the already implemented velocity estimation).

2. There are no obstacles between the robot and the landing zone: This is a limitation mostly because consumer grade quadcopters are not equipped with any kind of 3D sensors that could help them detect obstacles. That being said, future work includes investigation on monocular obstacle detection and avoidance.

In the rest of this section we are going to analyse the system's node architecture and focus on each of the four software packages. Additionally, we are going to discuss about the simulation software that was developed to enhance the testing process.

4.1 System Node Architecture

The implementation of a landing system for a UAV using only a front facing camera on a moving platform, consists of various software parts that combined can lead to the desired result. Figure 4 analyses the architecture of the whole system. The colour coding allows for easier separation of hardware related nodes (red), navigation related nodes (blue) and landing zone related nodes (green). The arrows represent information flow, with each arrow label presenting the basic type of information transfered. Figure 5 shows the ROS graph using the rqt_graph command. This graph provides a great way to visualise the data transfered via ROS topics, along with all nodes involved.

4.2 Landing Zone Detection

Cameras, just like every other sensor, produce data that contain noise. Camera noise may not be a problem when the frames are used just for video shooting, but when

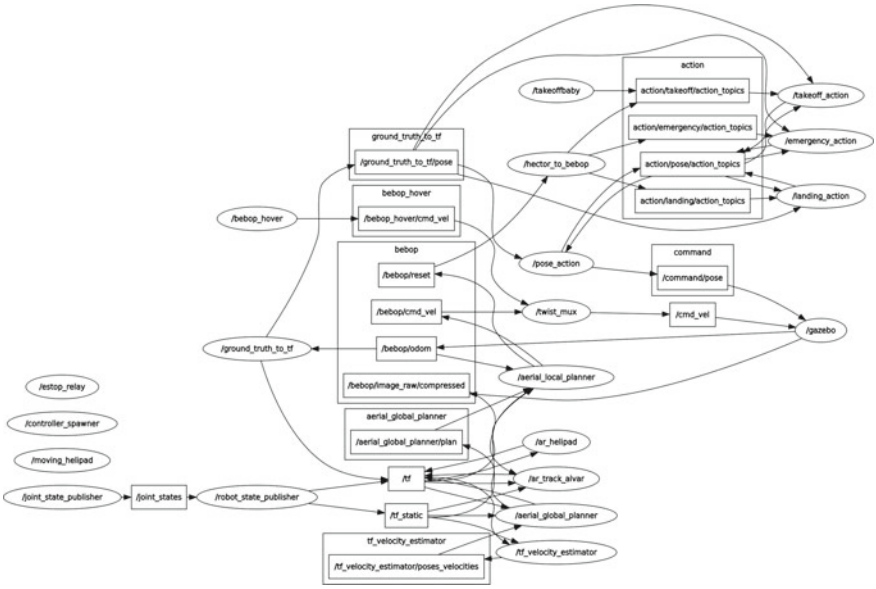


Fig. 5 rqt_graph with Nodes/Topics (active) graph type of the whole system.

each frame is used to generate an estimation, the accuracy of each specific value (pixel) is important. This is why camera calibration is needed before using the sensor for estimation-based processing. The ROS community has provided a calibration package, called camera_calibration [7] which is a wrapper for OpenCV calibration software [22]. OpenCV is the biggest open source library for computer vision, and supports mainly C++ and Python. The implemented software is based on research on pixel dissimilarity measurement [6] and image rectification [15].

This calibration software is used to detect the type of image noise that each camera sensor is subject to. Noise is categorized in two distortion types: the radial distortion and the tangential distortion. Radial distortion, which is the main cause of noise, can be distinguished in the barrel distortion and the pincushion distortion. Visually, the barrel distortion makes the center of the image appear closer, and the pincushion distortion makes the same for the edges of the image.

The camera distortion is a factor that cannot be changed, but due to its permanency the calibration software is able to generate a series of coefficients that can be used to rectify the image. Image rectification is the process of removing image distortion and make everything appear on a flat surface. In order to generate those coefficients, the algorithm needs to use an image with known dimensions as a reference. The most effective reference image for such task is a checkerboard.

The process of calibration is very simple. The first thing that has to be done is to have a checkerboard printed. Then the exact measurement of each rectangular cell and the number of cells for both rows and columns have to be set in the parameters of the algorithm in order for it to be able to calculate the distortion coefficients. The

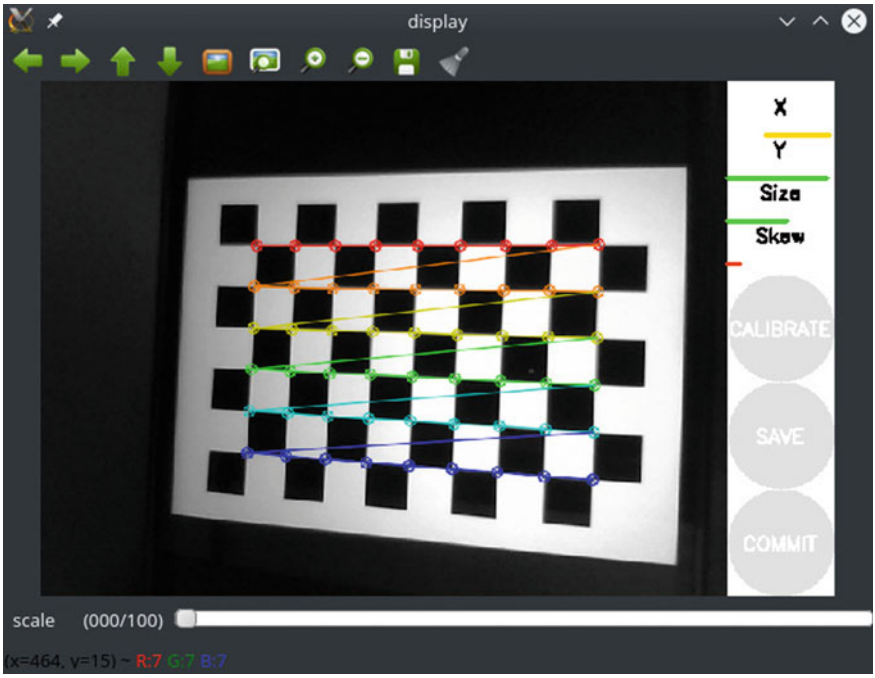


Fig. 6 The graphical user interface of the camera calibration software

last thing that has to be done is to slightly alter the position and the rotation of the checkerboard while keeping it inside the camera frame, to assist the algorithm in calculating the coefficients by using various positions and angles. Figure 6 shows a screenshot of the camera calibration software provided by the camera_calibration ROS package.

The ar_track_alvar [2] ROS package is used to detect the moving platform. This package can be used with stereo/depth cameras and monocular ones but both types should be calibrated first using the method described earlier.

The ar_track_alvar package uses Augmented Reality (AR) markers for its detection algorithm. Those AR markers consist of black and white blocks like a checkerboard, but with a variety of patterns. An example of three AR tags is shown in Fig. 7.

ar_track_alvar offers two ways of detection. The first detection mode consists of a single marker for which the package reports its estimated pose. The second detection mode consists of multiple markers that are used as a bundle, to increase accuracy. For this mode, one of the markers is considered to be the "master marker" which means that ar_track_alvar will always report the estimated position of that marker, even if only other markers of the bundle are visible.

The landing zone can consist of two (bundled) AR markers, one on the upper left and one on the bottom right corner. This setup is used to increase the accuracy. Because, by default, the ar_track_alvar package reports the position of the master

Fig. 7 Three Augmented Reality markers

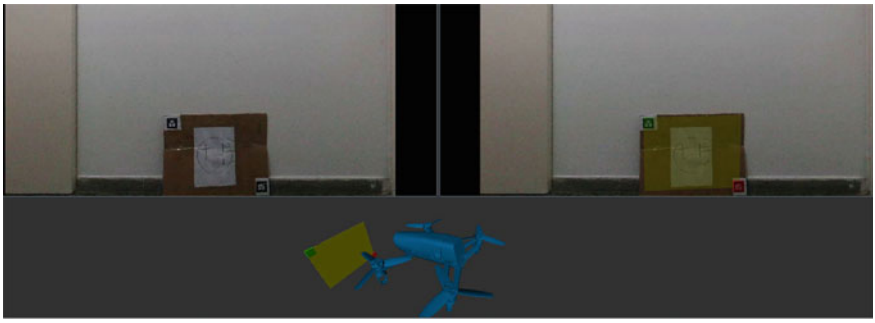


Fig. 8 Screenshot from rviz. The upper left image is the raw image from the camera, and the upper right image is the same frame with superimposed visualization detection data provided by the ar_helipad package. The lower part of the figure shows the quadcopter and the relative estimated position of the helipad

marker, a software that translated the estimated position from the marker to the center of the helipad was developed. This ROS package, called ar_helipad [32], offers a generalized way to create bundles of two AR markers set up in two opposite corners and have their center reported with each detection, only by changing a set of parameters. The package also has visualization capabilities for easier evaluation of the parameters given. Figure 8 shows a cropped screenshot from rviz, the main visualization tool for ROS. The upper left image is the raw image from the camera, and the upper right image is the same frame with superimposed visualization detection data provided by the ar_helipad package. The lower part of the figure shows the quadcopter and the relative estimated position of the helipad.

Although AR marker bundles offer higher accuracy, this is only true in close distances. In distances more than one meter, which is almost always the case in this project, the markers become too small to be detected. One solution would be to make the helipad and the two markers bigger. This solution is not really favourable for the simple reason that at some point, there would be limitations of the maximum size of the landing zone, so a more robust solution is required.

This is why a new prototype helipad was designed that consisted solely from a big AR marker. The helipad remains at the exact same size as with the bundled setup, but this time the AR marker basically is the landing zone. The ar_helipad

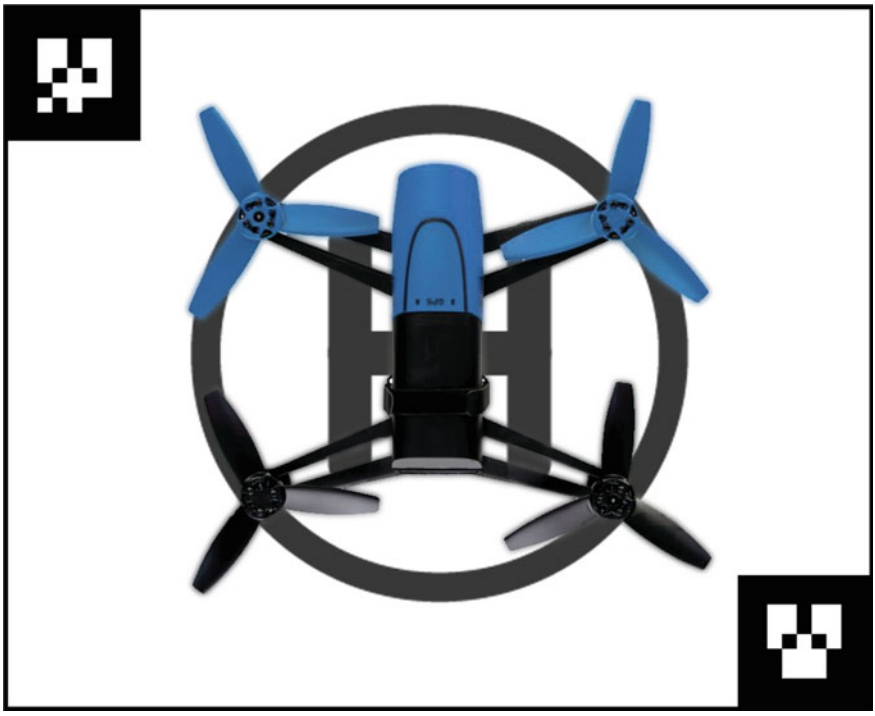


Fig. 9 Helipad-Quadcopter relative size

ROS package supports this kind of setup too, reporting the center of the AR marker. This functionality is also available through the ar_track_alvar package for a single marker detection mode, but the information of the helipad size was added, which can be used for both visualization and estimation purposes. Figure 9 shows a scaled size comparison of the Parrot Bebop and a bundled helipad that was used for testing. The helipad is just about 1.5 times bigger than the UAV itself.

4.3 Landing Zone Velocity Estimation

With the helipad detection implementation available, the next step is to use a ROS node that can estimate the velocity of the detected object. For this purpose a generalized ROS package was developed, called tf_velocity_estimator [35].

When a detection is registered by the ar_helipad ROS package a transformation ("tf" in ROS's language) message is published, which is received by the velocity estimator. The tf_velocity_estimator package offers a set of parameters that can be used to estimate the velocity of any tf.

The tf_velocity_estimator package aggregates incoming pose estimations provided by the ar_helipad, and relays them along with velocity estimations. The size

of the aggregated data can be set via the available package parameters in order to provide the needed flexibility to run on hardware with a variety of computational power. The velocity estimations are based on the physics formula of constant speed:

$$v = \frac{\Delta x}{\Delta t} \tag{1}$$

where Δx is the position difference between two observations and Δt is the time difference between two observations.

The velocity estimator generates velocity estimations for the three dimensional environment of the robot, thus the previous equation is divided into three, one for each axis:

$$v_x = \frac{\Delta x_x}{\Delta t} \tag{2}$$

$$v_y = \frac{\Delta x_y}{\Delta t} \tag{3}$$

$$v_z = \frac{\Delta x_z}{\Delta t} \tag{4}$$

where Δx_x is the position difference between two observations on the x axis, Δx_y is the position difference between two observations on the y axis, Δx_z is the position difference between two observations on the z axis and Δt is the time difference between two observations.

It should be highlighted that although the velocity estimator offers the ability to calculate velocity in three dimensional space, in this project the landing zone moves only perpendicularly to the Z axis ($v_z = 0$).

4.4 Global Planning

In the previous section, it was described how the implemented system can detect the moving platform and estimate its velocity using the ar_helipad and the tf_velocity_estimator ROS packages, respectively.

The information provided with the help of the previously described packages should be enough for the global planning software to generate a path towards a selected future position of the moving platform. The global planning software is a ROS package called aerial_global_planner [30]. It uses the current position of the robot given by the odometry, the latest position of the moving platform and the latest set of estimated velocities generated by the transformation velocity estimation package.

Before generating a path, the global planner has to choose the best velocity estimation of the moving platform from a set of estimations provided by the

tf_velocity_estimator. Since the helipad is moving with constant speed, the perfect estimations should all be the same. Computer vision just like human vision, is not absolutely perfect though, and the global planning software has to remove noisy estimations from the set before determining the estimated velocity. Algorithm 1 shows the procedure of determining the velocity estimation through the series provided by the velocity estimation software.

```
Input array vx;
Input array vy;
meanvx = Calculate mean of vx;
meanvy = Calculate mean of vy;
stabilized = false;
while not stabilized do
    stabilized = true;
    if size of vx > 3 then
        if the first value of vx is not close to meanvx then
            stabilized = false;
            Delete the first element of vx;
            meanvx = Calculate mean of vx;
        end
        if the last value of vx is not close to meanvx then
            stabilized = false;
            Delete the last element of vx;
            meanvx = Calculate mean of vx;
        end
    end
    if size of vy > 3 then
        if the first value of vy is not close to meanvy then
            stabilized = false;
            Delete the first element of vy;
            meanvy = Calculate mean of vy;
        end
        if the last value of vy is not close to meanvy then
            stabilized = false;
            Delete the last element of vy;
            meanvy = Calculate mean of vy;
        end
    end
end
```

Algorithm 1: Calculate the mean value of the estimated velocity of the moving platform, by removing potential noisy estimations

In short, this algorithm firstly calculates the mean for the set of vx and vy values. Then, in an attempt to eliminate noisy data, discards all values that their difference with the mean value is larger than a threshold. This procedure is repeated and new mean values are calculated when a value is removed from the set, until all remaining

values are close enough to the mean, or there are only three values left. Based on those calculations the global planning software also provides the current estimated position of the moving platform.

The generated global path is always a straight line that connects the latest robot position with a selected future moving platform position. The generated path is a straight line due to the project's assumption of obstacle-free navigation.

The strategy of selecting a goal point is based on the selection of the point that requires the lowest velocity. Algorithm 2 shows the strategy of the global planner in order to achieve such task.

```
time = seconds after last observation;
minVel = maxVel;
for t in range [time-20] with step 0.1 do
    pos = position of the moving helipad after t seconds;
    vel = needed velocity to reach pos after t seconds;
    if vel < minVel then
        minVel = vel;
        goal = pos;
    end
end
```

Algorithm 2: Lowest required velocity point selection

The algorithm calculates a future position of the moving platform and the required velocity to reach that position in a specific time frame. These calculations are based on the physics formula of constant speed. Given the assumption that the helipad is not moving on the Z axis, the two dimensional future position of the moving platform is calculated with the following equations:

$$x'_{hx} = x_{hx} + v_{hx} \cdot \Delta t \tag{5}$$

$$x'_{hy} = x_{hy} + v_{hy} \cdot \Delta t \tag{6}$$

where x_{hx} is the last detected position of the helipad on the x axis, x_{hy} is the last detected position of the helipad on the y axis, v_{hx} is the estimated velocity of the helipad on the x axis, v_{hy} is the estimated velocity of the helipad on the y axis, x'_{hx} is the future position of the helipad on the x axis after t seconds and x'_{hy} is the future position of the helipad on the y axis after t seconds.

The required velocity of the robot to reach a future position of the moving platform in the three dimensional space is the calculated using the following equations:

$$v'_{rx} = \frac{x_{rx} - x'_{hx}}{\Delta t} \tag{7}$$

$$v'_{ry} = \frac{x_{ry} - x'_{hy}}{\Delta t} \tag{8}$$

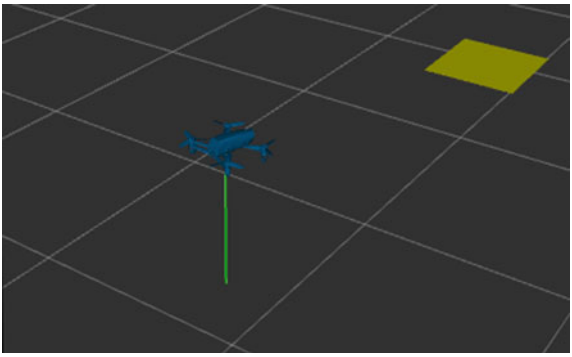


Fig. 10 The path (in green) generated by the global planning software while the landing zone is moving towards the robot, as shown in rviz

$$v'_{rz} = \frac{x_{rz} - x_{hz}}{\Delta t} \tag{9}$$

where x_{rx} is the current position of the robot on the x axis, x_{ry} is the current position of the robot on the y axis, x_{rz} is the current position of the robot on the z axis, v'_{rx} is the required velocity of the robot on the x axis, v'_{ry} is the required velocity of the robot on the y axis, v'_{rz} is the required velocity of the robot on the z axis, x'_{hx} is the future position of the helipad on the x axis after t seconds and x'_{hy} is the future position of the helipad on the y axis after t seconds.

Figure 10 shows the path generated by the global planning strategy while the quadcopter is floating and a landing zone is moving towards it.

4.5 Local Planning

The local planning software called aerial_local_planner [31] is a ROS package designed to receive the generated path by the global planning software and send velocity and landing commands to the robot in order for it to effectively navigate to the end of the planned path and land.

The developed local planning software receives the following information:

1. The path generated by the global planning software.
2. The current position of the robot provided by the hardware odometry estimation.
3. The current position of the helipad provided by the global planning software.

The first two pieces of information are used so that the local planning software can generate the velocity commands that are sent to the robot. The velocity of the robot consists of four parts: The three dimensional linear velocity and the angular velocity on the yaw rotational axis. All four velocities are calculated using the formulas below:

$$\begin{cases} v_0 = 2 \cdot \sqrt{(x_r - x_h)^2 + (y_r - y_h)^2} \cdot \cos\left(\tan^{-1}\left(\frac{y_h - y_r}{x_h - x_r}\right) - yaw_r\right) \\ v_x = v_0,\ v_0 \in [v_{min}, v_{max}] \\ v_x = v_{min},\ v_0 \in (-\infty, v_{min}) \\ v_x = v_{max},\ v_0 \in (v_{max}, +\infty) \end{cases} \tag{10}$$

where v_0 is the estimated robot velocity on the currently investigated axis, v_x is the velocity command on the x axis that is going to be sent to the robot, v_min is the minimum velocity command that can move the robot, v_max is the maximum velocity command that the robot can reach, yaw_r is the current orientation of the robot in the yaw axis, x_r is the current position of the robot on the x axis, y_r is the current position of the robot on the y axis, x_h is the current estimated position of the helipad on the x axis and y_h is the current estimated position of the helipad on the y axis.

$$\begin{cases} v_0 = 2 \cdot \sqrt{(x_r - x_h)^2 + (y_r - y_h)^2} \cdot \sin\left(\tan^{-1}\left(\frac{y_h - y_r}{x_h - x_r}\right) - yaw_r\right) \\ v_y = v_0,\ v_0 \in [v_{min}, v_{max}] \\ v_y = v_{min},\ v_0 \in (-\infty, v_{min}) \\ v_y = v_{max},\ v_0 \in (v_{max}, +\infty) \end{cases} \tag{11}$$

where v_0 is the estimated robot velocity on the currently investigated axis, v_y is the velocity command on the y axis that is going to be sent to the robot, v_min is the minimum velocity command that can move the robot, v_max is the maximum velocity command that the robot can reach, yaw_r is the current orientation of the robot in the yaw axis, x_r is the current position of the robot on the x axis, y_r is the current position of the robot on the y axis, x_h is the current estimated position of the helipad on the x axis and y_h is the current estimated position of the helipad on the y axis.

$$v_z = 2 \cdot (z_h - (z_r - 0.1)) \tag{12}$$

where v_z is the velocity command on the z axis that is going to be sent to the robot, z_r is the current position of the robot on the z axis and z_h is the current estimated position of the helipad on the z axis.

$$v_{yaw} = \left(\tan^{-1}\left(\frac{y_h - y_r}{x_h - x_r}\right) - yaw_r\right) \cdot 10 \tag{13}$$

where v_{yaw} is the angular velocity command on the yaw rotational axis that is going to be sent to the robot, yaw_r is the current orientation of the robot in the yaw axis, x_r is the current position of the robot on the x axis, y_r is the current position of the robot on the y axis, x_h is the current estimated position of the helipad on the x axis and y_h is the current estimated position of the helipad on the y axis.

Equations 10 and 11 calculate the velocity on the x and y axes. They use the two dimensional distance on the x and y axes between the robot and the moving platform, as well as the heading of the robot relatively to the position of the target.

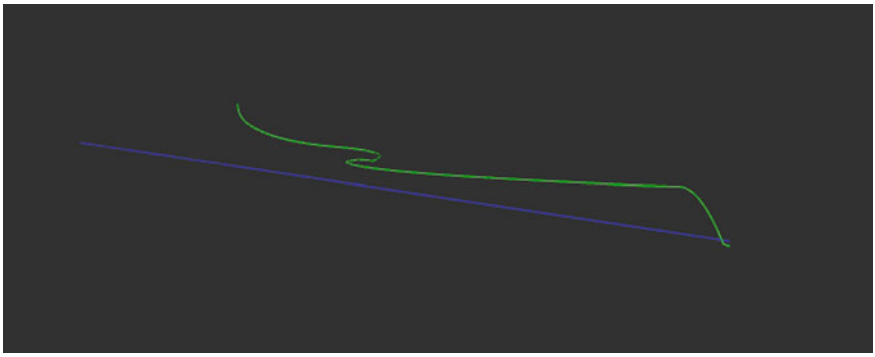


Fig. 11 Visualization of a successful landing (Moving platform's trajectory in blue and robot's trajectory in green).

Equation 12 calculates the velocity on the z axis and uses only the z axis distance (difference) between the robot and the moving platform. The -0.1 on the equation ensures that the robot never gets closer than 10 cm to the future position of the moving platform (on the z axis).

Equation 13 calculates the angular velocity on the yaw rotation axis. It uses only the heading of the robot relatively to the position of the target.

With the four mathematical equations presented earlier, the robot is able to autonomously navigate to the target point specified by the global planning software. The only thing that remains is to land on the moving platform. The navigation problem we are investigating is not only a problem of space, but also time. When the robot arrives to the target point, it does not necessarily mean that the moving platform is at the desired point too. This leads us to the last input information that the local planning software receives, the current estimated position of the moving platform. The local planning software constantly checks for a set of conditions. These conditions are the following:

- The two dimensional distance between the robot and the moving platform on the x and y axes is less than 20 cm.
- The distance (difference) in the z axis between the robot and the moving platform is less than 20 cm.
- The robot is above the moving platform.

When all of the three conditions presented above are met, the landing command is sent, and no more velocity commands can be executed by the robot.

Figure 11 shows the trajectory of the moving helipad in blue and the trajectory of the robot in green, until the successful landing of the latter on the former.

4.6 Gazebo Simulation

The best ROS compatible simulator is the Gazebo Simulator [28]. Gazebo offers 3D physics simulation for a variety of sensors and robots, and it can be customized enough to allow for plugins that offer underwater and air drag simulation.

Gazebo is bundled into the full installation package of ROS, making in widely available in robotics simulation. A lot of work has gone into making Gazebo a great tool for simulating real robots. Many robot manufacturers offer ROS packages specifically designed for Gazebo support.

Although a Parrot Bebop simulation software is available for programmers, called Sphinx [29] that uses Gazebo as a basis, it is not compatible with ROS and is pretty limited in terms of customizability. The Parrot Bebop Gazebo simulation package (bebop_gazebo) [33] is a ROS package developed specifically to assist the testing phases of this project, but can be used by anyone interested in simulating this quadcopter. The developed ROS package is based on another quadcopter simulation package developed by the Technische Universität Darmstadt called hector_gazebo [8]. The bebop_gazebo package is basically a modified hector_gazebo with some additions for specific hardware capabilities. More specifically, the changes that have been applied to the hector_gazebo ROS package are discussed in the following paragraphs.

Firstly, Unified Robot Description Format (URDF) files have replaced the original ones to reflect the physical properties of the Parrot Bebop quadcopter. URDF is a file format used to describe robots. This format is compatible with ROS and Gazebo, so it is an effective tool to visualize and simulate a robot. URDF offers dimension related parameters for visualization and collision detection, as well as physics related properties, like inertia and mass. It is compatible with a variety of gazebo plugins to enhance its usage and make the simulation more accurate. The bebop_gazebo URDF specifically includes the following features:

1. Mass properties to realistically simulate gravity effect.
2. Size properties that reflect the real hardware.
3. A 3D model of the Parrot Bebop as shown in Fig. 12, for realistic visualization and collision detection.
4. The exact camera position to accurately simulate the transformation between estimations and the robot itself.
5. Sensor plugins for camera and GPS that simulate the properties of the hardware (resolution, frames per second etc.)
6. Aerodynamics and propulsion plugins that are able to simulate air dynamics, and drag or even generate gusts of wind.
7. An odometry plugin to simulate the odometry estimations provided by the Parrot Bebop out of the box.

All of the topics from the hector_gazebo package have been remapped to be exactly the same as the ones in the Parrot Bebop ROS driver, the bebop_autonomy

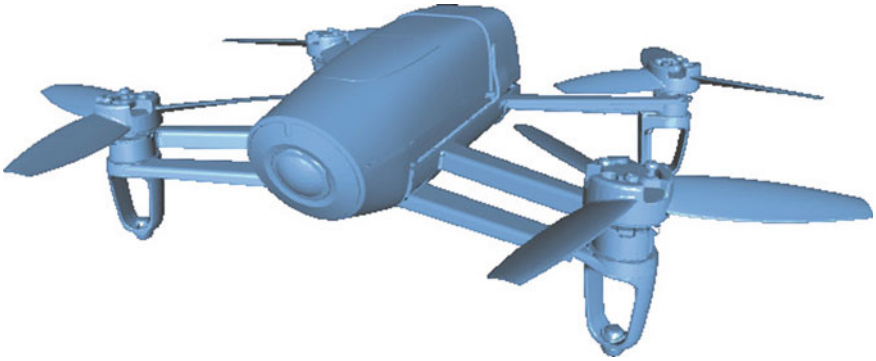


Fig. 12 The 3D model used to simulate the Parrot Bebop

package [3]. This change ensures a seamless transition between the real hardware and the simulated one.

The last step in order to make the simulated quadcopter behave like the real one, was to add some extra software called actions. The Parrot Bebop SDK and the bebop_autonomy ROS driver offer the ability to send commands to the robot for three specific tasks:

1. Taking off
2. Landing
3. Emergency Landing

It is of vital importance to have those three commands implemented in the simulated environment, especially the two landing ones, because the Parrot Bebop has a built-in feature that disables it from touching the ground when the landing or emergency landing commands have not been received. This safety feature requires an additional emergency landing command to be sent by the the local planning software.

The take off and landing actions were already implemented by the hector_gazebo package, so they were used without any changes (apart from the topic names that enable the actions). When the landing command is received, the robot slowly decreases height until it touches the ground. The emergency landing action was not implemented by that package though, so it had to be developed from scratch for the purposes of this project. On the real hardware, when the emergency landing command is received the motors are immediately shut down, making the robot fall from any height. A simple action software was developed that enabled the simulated hardware to stop its motors and make the quadcopter free fall. The importance of such action should be clear enough by studying the local planning implementation.

Apart from the Parrot Bebop simulation, a moving platform simulation was needed for testing purposes. This is why two platform models were created, along with software that offers parameters to set the platform's starting position (x, y, z) and velocity properties (vx, vy). Figure 13 shows the two platform models. It should be

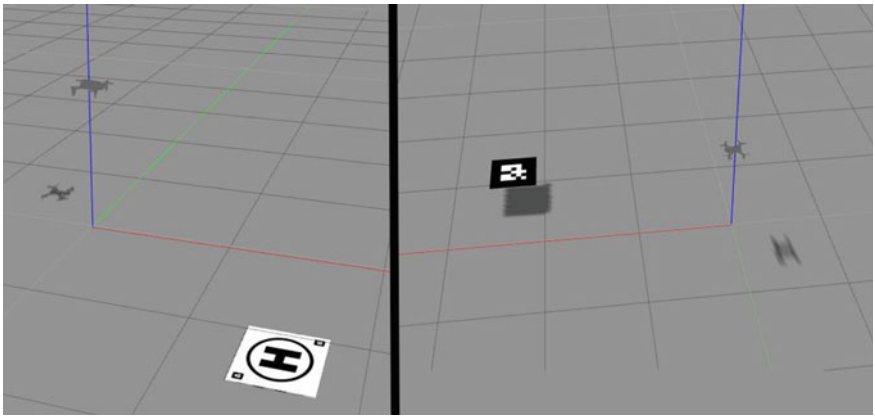


Fig. 13 Two different moving platforms used for testing in Gazebo Simulator

noted that in the image on the left, the AR tags on the edges of the platform are the ones detected by the robot and not the "H" symbol.

5 Leasons Learned

This chapter, apart from the technical details and the overall discussion around the subject, should also be used as an example on how to use ROS nodes, when to use already implemented methods, when to expand them and when to implemented new ones.

The use case presented in this chapter is a great example of all the above. A lot of effort has gone into expanding ROS packages, using already existing ones but also implemented others from scratch.

This project started with the acquisition of a consumer grade quadcopter (Parrot Bebop) for which, at the time, there was no ROS compatible simulated hardware. At first, all tests where run on the real hardware, but this was time consuming and stressing, not only due to battery life and potential hardware damage, but also because outdoors tests were required. Developing a simulation environment greatly improved the experimentation phase both in terms of time and consistency.

Apart from the enhancement of the project due to simulation availability, studying and analysing already implemented ROS nodes, boosted the performance of the system either because they were used with minor or no changes at all, or because they provided a great baseline on how to implement new software. For example, the already implemented local planning software, provided great programming material for the development of the new local planner used in the described system.

Other problems encountered during the implementation of the whole system, included data transfer delays or noisy results/estimations. Both problems were not

apparent from the beginning, but once detected additional code had to be implemented to compensate for such occasions, or use related ROS packages.

In general, the experience gained from this project that should be transfered to other ROS users indicates that users should first do some research on already implemented ROS compatible software. If the already implemented software seems to solve their problem, they should take the time to study and experiment with it. It will be worthwhile in the long run. If the already implemented software partly solves their problem, they should check if it allows for parameterisation that could shape it around the solution of their problem. If this is not possible or the results are not good enough, then and only then should they start developing their one software. It is also advised to use already implemented software as a baseline for node architecture. Finally, even if simulation does not surely imply real world solution, it is a vital step during experimentation that can greatly improve the whole development and testing process.

6 Conclusion

The results of the work done enabled consumer grade UAVs equipped with just an IMU and a forward facing camera to land on moving platforms. The landing was achieved without direct visual contact with the platform, and was based on estimations that were generated while the moving target was briefly inside the field of view of a front facing camera.

The implemented landing system is modular, allowing for easy substitution of software packages. Some of its parts were already implemented modules that were integrated into the system and others were developed from scratch. The final seamless software combined the two to achieve the goals of the project. Additionally, a simulation ROS package was developed to enhance the testing phase of the research.

The resulting system offers packages that could help other researchers and robot enthusiasts who use ROS. The simulation package provides an easy way to experiment with methods that could damage the hardware without such risks. It also allows for more experimentation time since it eliminates the flying time limitation due to battery life. Additionally, Gazebo simulation enables developers to experiment with various virtual environments before running their methods on real UAVs.

Since this project makes extensive use of the Robot Operating System, an effort was made to help the ROS community benefit from all the implemented software by composing a tutorial page on the official ROS wiki page [34]. This tutorial shows the basic usage of the implemented system and shows videos that demonstrate a series of the simulated results.

The methods implemented are effective, but there is room for improvement. A major part of the system's robustness is affected by the moving platform velocity estimation method. Based on the literature review, apart from the ar_track_alvar package, another, still in early stages, method [24] has been implemented that could

be used in the future in an attempt to enhance the effectiveness of the landing system's overall detection/velocity estimation.

Apart from improvements, the project could benefit from various added capabilities. Such additional capabilities would make the system even more useful for real life scenarios and expand its usage beyond controlled environments. In short, future research should attempt to make the implemented system work without the assumptions defined for this project.

The first one could be the additional capability of obstacle avoidance. This is a very difficult task though, since the obstacle detection has to be developed using only the sensors available in consumer grade UAVs which are very limited. An interesting approach [19] has been implemented by Jeff Michels et al., in which reinforcement learning is used to train the system for monocular camera depth estimation.

The other assumptions of this project was the constant speed of the moving platform. Researchers could work into developing prediction algorithms that could estimate a future position of the helipad just by briefly observing its current trajectory. Of course, this is not a trivial task, since unpredictable movements should also be considered, like the moving platform itself having to avoid an obstacle.

In addition, the system could be expanded to allow consumer grade UAVs to land on moving platforms that are floating on water, which means that they suffer from unpredictable (but sometimes periodic) displacement in the z axis due to wave movement.

To conclude, in this chapter we presented a system that enables consumer grade UAVs to land on a moving target. The novelty of the system is that it does not require the UAV to be equipped with a vertical camera. With only a front facing camera and without visual contact during landing, the system uses pose estimations of its target to succesfully land on it. Although the implemented ROS packages offer a robust way to achieve the defined task, improvements and additional features can further enhance the capabilities of the overall system, as we discussed earlier.

References

1. A. Alvar, Library for virtual and augmented reality (2016)
2. A.T. Alvar, ROS wiki page (2018). http://wiki.ros.org/ar_track_alvar
3. bebop_autonomy. ROS wiki page (2018). http://wiki.ros.org/bebop_autonomy
4. P.J. Benavidez, J. Lambert, A. Jaimes, M. Jamshidi, Landing of a quadcopter on a mobile base using fuzzy logic, in *Advance Trends in Soft Computing* (Springer, Berlin, 2014), pp. 429–437
5. M. Beul, S. Houben, M. Nieuwenhuisen, S. Behnke, Fast autonomous landing on a moving target at mbzirc, in *Mobile Robots (ECMR), 2017 European Conference on*, (IEEE, Paris, 2017), pp. 1–6
6. S. Birchfield, C. Tomasi, A pixel dissimilarity measure that is insensitive to image sampling. IEEE Trans. Pattern Anal. Mach. Intell. **20**(4), 401–406 (1998)
7. camera_calibration, ROS wiki page (2018). http://wiki.ros.org/camera_calibration
8. T.U. Darmstadt, hector_gazebo github repository (2018). https://github.com/tu-darmstadt-ros-pkg/hector_gazebo
9. E.W. Dijkstra, A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)

10. dwa_local_planner, ROS wiki page (2018). http://wiki.ros.org/dwa_local_planner
11. D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, D. Scaramuzza, Vision-based autonomous quadrotor landing on a moving platform, in *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics*, Shanghai, China, pp. 11–13, (2017)
12. D. Fox, W. Burgard, S. Thrun, The dynamic window approach to collision avoidance. IEEE Robot. & Autom. Mag. **4**(1), 23–33 (1997)
13. global_planner, ROS wiki page (2018). http://wiki.ros.org/global_planner
14. P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
15. R.I. Hartley, Theory and practice of projective rectification. Int. J. Comput. Vis. **35**(2), 115–127 (1999)
16. A. Hornung, M. Phillips, E.G. Jones, M. Bennewitz, M. Likhachev, S. Chitta, Navigation in three-dimensional cluttered environments for mobile manipulation, in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (2012)
17. M. Hwangbo, J. Kuffner, T. Kanade, Efficient two-phase 3d motion planning for small fixed-wing uavs, in *Robotics and Automation, 2007 IEEE International Conference on* (IEEE, Nadi, 2007), pp. 1035–1041
18. M. Keller, F. Hoffmann, C. Hass, T. Bertram, A. Seewald, Planning of optimal collision avoidance trajectories with timed elastic bands. IFAC Proc. Vol. **47**(3), 9822–9827 (2014)
19. J. Michels, A. Saxena, A.Y. Ng, High speed obstacle avoidance using monocular vision and reinforcement learning, in *Proceedings of the 22nd International Conference on Machine learning* (ACM, New York, 2005), pp. 593–600
20. Navfn, ROS wiki page (2018). http://wiki.ros.org/navfn
21. octomap_server, ROS wiki page (2018). http://wiki.ros.org/octomap_server
22. OpenCV_calibration, OpenCV webpage (2018). https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
23. OSRF, Open Source Robotics Foundation official page (2018). https://www.openrobotics.org/
24. C. Patruno, M. Nitti, E. Stella, T. D'Orazio, Helipad detection for accurate uav pose estimation by means of a visual sensor. Int. J. Adv. Robot. Syst. **14**(5), 1729881417731083 (2018)
25. S. Saripalli, J.F. Montgomery, G.S. Sukhatme, Vision-based autonomous landing of an unmanned aerial vehicle, in *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, vol. 3 (IEEE, Washington, DC, 2002), pp. 2799–2804
26. S. Saripalli, G.S. Sukhatme, Landing on a moving target using an autonomous helicopter, in *Field and Service Robotics* (Springer, Berlin, 2003), pp. 277–286
27. J.Z. Sasiadek, I. Duleba, 3d local trajectory planner for uav. J. Intell. Robot. Syst. **29**(2), 191–210 (2000)
28. G. Simulator, Gazebo Simulator webpage (2018). http://gazebosim.org/
29. Sphinx, Sphinx Parrot Bebop Simulator official page (2018). https://developer.parrot.com/docs/sphinx/index.html
30. G. Stavrinos, aerial_global_planner github repository (2018a). https://github.com/gstavrinos/aerial_global_planner
31. G. Stavrinos, aerial_local_planner github repository (2018b). https://github.com/gstavrinos/aerial_local_planner
32. G. Stavrinos. ar_helipad github repository (2018c). https://github.com/gstavrinos/ar_helipad
33. G. Stavrinos, bebop_simulator github repository (2018d). https://github.com/gstavrinos/bebop_simulator
34. G. Stavrinos, Package Tutorial ros wiki page (2018e). http://wiki.ros.org/Tutorials/Landing%20an%20autonomous%20UAV%20on%20a%20moving%20platform%20using%20only%20a%20front%20facing%20camera
35. G. Stavrinos, tf_velocity_estimator github repository (2018f). https://github.com/gstavrinos/tf_velocity_estimator
36. teb_local_planner, ROS wiki page (2018). http://wiki.ros.org/teb_local_planner
37. J. Thomas, J. Welde, G. Loianno, K. Daniilidis, V. Kumar, Autonomous flight for detection, localization, and tracking of moving targets with a small quadrotor. IEEE Roboti. Autom. Lett. **2**(3), 1762–1769 (2017)

George Stavrinos received a BSc degree in Computer Science from Athens University of Economics and Business in 2014 and a MSc in Intelligent Systems and Robotics from De Montfort University in 2018. He is currently pursuing his second MSc in Space Science, Technologies and Applications in the National Observatory of Athens. He is currently working as a researcher in RoboSKEL Lab in the National Centre of Scientific Research "Demokritos" in Athens, Greece. His main research interests are robots under zero-gravity (microgravity) conditions (rovers or humanoids), autonomous spacecrafts, flying robots (autonomous UAVs), navigation autonomy, dynamics and kinematics analysis and motion planning.

Simulation Frameworks

Integrating the Functional Mock-Up Interface with ROS and Gazebo

Ralph Lange, Silvio Traversaro, Oliver Lenord, and Christian Bertsch

Abstract The *Functional Mock-up Interface* (FMI) is a widely used industry standard for exchange and co-simulation of dynamic models as *Functional Mock-up Units* (FMU). It is supported by more than 100 modeling and simulation tools. In this chapter, we present two implementations of FMI that bridge the gap between these tools and the ROS and Gazebo community: First, the *fmi_adapter* package for running/simulating FMUs in ROS nodes, from https://github.com/boschresearch/fmi_adapter_ros2. Second, the *gazebo-fmi* package for integrating FMUs with Gazebo, from https://github.com/robotology/gazebo-fmi. After an introduction to the FMI standard, this chapter provides step-by-step, hands-on examples for both packages, followed by interface descriptions and selected implementation details. In addition to these tutorial-style sections, the chapter also provides comprehensive descriptions of two use-cases. First, it explains how the fmi_adapter enabled a convenient model-based control design workflow for a self-driving vehicle for industrial logistics. Second, it reports on the simulation of electrical actuators in Gazebo from a Modelica model.

Keywords Functional mock-up interface · FMI · FMU · ROS · ROS 2 · Gazebo · fmi_adapter · gazebo-fmi

R. Lange (✉) · O. Lenord · C. Bertsch
Robert Bosch GmbH, Robert-Bosch-Campus 1, 71272 Renningen, Germany
e-mail: ralph.lange@de.bosch.com

O. Lenord
e-mail: oliver.lenord@de.bosch.com

C. Bertsch
e-mail: christian.bertsch@de.bosch.com

S. Traversaro
Italian Institute of Technology, Via Morego, 30, 16163 Genova, Italy
e-mail: silvio.traversaro@iit.it

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_7

1 Introduction

The *Functional Mock-up Interface* (FMI) is an industry standard for exchange and co-simulation of dynamic models, supported by more than 100 modeling and simulation tools [1], including Dymola, MATLAB/Simulink, OpenModelica, SimulationX, and Wolfram System Modeler. Such tools allow exporting dynamic models as *Functional Mock-up Units* (FMU). Technically, an FMU is a ZIP file containing the differential equations as a shared library and/or C source code together with an XML-based description of the variables, parameters and model structure, i.e. derivatives, initially unknown variables, etc. For co-simulation [2], a numerical solver may be integrated in the FMU.

A number of tools and libraries, including MATLAB/Simulink, Dymola, SimulationX, MapleSim, AMESim, GT-Suite, and Simpack, allow importing FMUs to compose larger models from existing ones and to simulate them together. This enables to integrate different models from (graphical and textual) modeling languages and tools. The use of a binary format even allows sharing of models while protecting intellectual property. The FMI standard is used in a number of industries and actively developed by an international consortium as explained in the history section of the FMI website [1].

In robotics, dynamic models are used for many aspects including advanced control of mobile platforms and manipulators, modern motion planning algorithms, and object tracking, to name a few. The *fmi_adapter* package [3, 4] provides a straightforward integration of the FMI standard with ROS and ROS 2. It offers a generic ROS node as well as a C++ class to load an FMU at runtime and to map the major FMI concepts to ROS and vice-versa. This allows integrating an FMU with a complex dynamic model and solver into a ROS-based system with just a few lines of C++ code or launch code.

For example, for a self-driving intra-logistics vehicle, we created a path filter to avoid high bore friction at the vehicle's caster wheels. Both, the path filter and the underlying kinematic model to estimate the orientations of the wheels were implemented in the Modelica language and exported as an FMU. By the fmi_adapter package, this FMU could be integrated with the ROS-based motion planner and motion controller of the vehicle with only 25 lines of code.

Dynamic models also play a crucial role when testing robotic systems in simulation. Robotic simulators such as AirSim, Gazebo, MORSE, OpenRAVE, SynCity, V-REP, and Webots, are generally based on physics engines for rigid body dynamics. In advanced applications, there is often the necessity to integrate custom dynamic models, which cannot be expressed using the underlying physics engine. As an example, consider the simulation of the low-level actuator dynamics, compliance, and delays in legged robotics [5], that is often critical in reinforcement learning approaches to robot control [6, 7]. The obvious solution is co-simulation, i.e. combining the generic robotics simulator and its physics engine with a custom dynamic model and solver. The FMI standard has been developed exactly for this use-case,

and the *gazebo-fmi plugins* [8] provide an easy-to-use integration of FMUs into the Gazebo simulator.

For example, the gazebo-fmi plugins allow co-simulating an FMU that models an electrical actuator of the iCub humanoid robot [9], including the actuator's back electromotive force (EMF) and the rotational inertia, directly with Gazebo.

The goals of this chapter are three-fold:

1. to teach the basics of the FMI by giving a comprehensive introduction to the FMI standard and corresponding tools,
2. to overcome any entry hurdle in the use of FMI technology with ROS by providing hands-on tutorials and documentations on the fmi_adapter package and the gazebo-fmi plugins, and
3. to inspire for practical use of the FMI in ROS-based systems and Gazebo-based simulations by reporting on advanced applications from our labs.

The remainder of this chapter is structured along those goals. Section 2 gives an overview to the goals and history of the FMI standard, before describing the contents of an FMU and related tools. Section 3 explains all steps to setup the fmi_adapter package on ROS or ROS 2 and to run the examples provided by the fmi_adapter_examples package. Then, details on the interface and the implementation are given. Analogously, Sect. 4 describes all necessary steps to build or install the gazebo-fmi plugins and to run a first demo application before explaining the interface and selected implementation details. Thereafter, Sect. 5 reports on the use of the fmi_adapter in the mentioned autonomous intra-logistics vehicle and on the implementation of the electrical actuator model based on gazebo-fmi. The chapter concludes in Sect. 6 with final remarks.

2 Overview to FMI

A crucial task in the development of robotic systems and software intensive systems in general, is the validation of the overall system behavior including software and hardware. This implies to shift the scope of simulation from components to systems and from single technical domains to multi-physical simulations with interfaces to software components.

As different simulation tools may be best suited for a particular task and varying preferences between departments, companies, and organizations exist, it is crucial to be able to exchange and to co-simulate models from different simulation tools. In this context, model exchange refers to the case of integrating an encapsulated model from one into another simulation environment, while co-simulation refers to the coupling of simulation models together with their individual numerical solver using a master algorithm and accepting a certain numerical error due to the coupling. This demand led to the development of the FMI standard.

In this section, we first give an introduction to the history of FMI, followed by a technical overview. Then, we present important FMI tools, before we discuss limitations. The section concludes with an outlook on further developments.

2.1 History of FMI

The FMI standard was developed in the publicly funded ITEA3 project MODELISAR from 2008 to 2011, which had been initiated by Daimler and others to create a tool-independent standard for model exchange and co-simulation [10]. Before FMI, one had to use or develop proprietary, bilateral tool interfaces, with a rapidly growing effort due to the large number of possible combinations. Alternatively, one had to use a proprietary de-facto standard like S-Functions from MATLAB/Simulink [11] within its limitations.

The name "Functional Mock-up Interface" is motivated from the "Digital Mock-up", which is commonly used for the integration of CAD data from the different parts and subsystems to an overall CAD assembly of a vehicle. This idea is transferred to the domain of system simulation in the sense of a complementary "functional" representation of the system.

The FMI 1.0 was released in 2010, see overview paper [12]. This first version of the standard was rapidly adopted by many simulation tools and was revised in 2014 by the improved version FMI 2.0 [13], as described in [14].

Since the MODELISAR project has ended, FMI is further developed as a project within the Modelica Association [15], a non-profit organization under Swedish law dedicated to the coordinated standardization in the field of systems engineering. FMI has been adopted very fast by industry, academia, and tool vendors [16].

Current developments within the FMI project focus on FMI 3.0, while in the publicly funded ITEA3 project EMPHYSIS [17] the extension of FMI for the deployment to embedded systems is being developed. More details are given in Sect. 2.5.

2.2 Functional Mock-Up Units: Technical Overview

A *Functional Mock-up Unit* (FMU) is the artifact exchanged between different simulation tools. Technically, an FMU is a ZIP file [18] containing the following elements:

- A model and interface description in form of an XML file named `modelDescription.xml`.
- A model representation with either
  - one or multiple platform specific binary representation(s) (e.g., Windows 64 bit or Linux 32 bit), or
  - a source code representation in C.

Fig. 1 Model exchange versus co-simulations FMUs

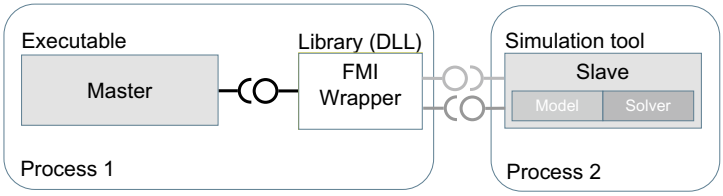


Fig. 2 Tool wrapper FMU

- An optional documentation folder.
- An optional resources folder containing, for instance, data files.

As FMI defines a model interface, there is always an *exporting tool* that generates the FMU and an *importing tool* that executes the FMU during a simulation run.

2.2.1 FMU Kinds

In FMI 1.0 and FMI 2.0, two kinds of FMUs are distinguished as illustrated in Fig. 1:

- *Model Exchange (ME) FMUs* describe a model, typically consisting of ordinary differential equations, where the numerical solver is not included in the model but must be provided by the importing tool.
- *Co-Simulation (CS) FMUs* contain both the model and a suitable solver. The communication with the importing tool takes place at discrete *communication time instances* only. The orchestration of multiple FMUs during the simulation process by passing the signals between the FMUs (possibly with filtering and extrapolation) and calling the individual FMUs in the right order at a certain time step, is the responsibility of a *master algorithm* provided by the importing tool. Please note that the FMI standard does not specify the master algorithm itself.

Since FMI 2.0, the standardized description of both kinds of FMUs have been combined. Hence, an FMU may contain both kinds of FMUs at once.

2.2.2 Self-contained Versus Tool-Wrapper FMUs

An FMU is from the basic idea a self-contained simulation model that does not have external dependencies, but is shipped with everything that is needed for execution

(including necessary libraries). However, external dependencies are allowed and can be signaled by a flag in the XML file. This can in the extreme case lead to the situation that an FMU is just a wrapper that communicates with the instance of an installed simulation tool to simulate the model as depicted in Fig. 2. Such an FMU is referred to as *tool wrapper FMU*. Especially for the deployment to real-time targets or software environments like ROS, self-contained FMUs are beneficial.

2.2.3 C-API and Calling Sequence

The FMI standard defines an application programming interface (API) in the C programming language. In FMI 2.0 [13] it consists of 25 + 9 (ME) resp. 10 (CS) function prototypes. Examples for such function prototypes are

```
fmi2Component fmi2Instantiate(fmi2String instanceName,
       fmi2Type fmuType, fmi2String fmuGUID,
       fmi2String fmuResourceLocation,
       const fmi2CallbackFunctions* functions,
       fmi2Boolean visible, fmi2Boolean loggingOn);
```

for instantiation of an FMU,

```
fmi2Status fmi2DoStep(fmi2Component c,
       fmi2Real currentCommunicationPoint,
       fmi2Real communicationStepSize,
       fmi2Boolean noSetFMUStatePriorToCurrentPoint);
```

for performing a simulation step for a co-simulation FMU, and

```
fmi2Status fmi2GetReal(fmi2Component c,
       const fmi2ValueReference vr[], size_t nvr,
       fmi2Real value[]);
```

for getting the value of a variable form the FMU.

For each FMU kind, a state machine defines the allowed calling sequence of the functions. For example, for a Co-Simulation FMU this could look like the following pseudocode:

```
fmi2Component c = fmi2Instantiate(...)
fmi2SetupExperiment(c, ..., relTol, ...)
fmi2EnterInitializationMode(c)
fmi2SetReal(c, ...)
fmi2ExitInitializationMode(c)
...
(loop)
  fmi2SetReal(c, ...)
  fmi2DoStep(c, t=0.1, dt=0.1, ...)
  fmi2GetReal(c, ...)
(end loop)
...
fmi2Terminate(c)
fmi2FreeInstance(c)
```

2.2.4 Model Description File

An FMU's model description file specifies all necessary interface information about the model required by the importing tool to properly integrate the model in terms of variables, parameters, structural dependencies and settings for model exchange and/or co-simulation. As an example, we consider the `modelDescription.xml` file from the `DampedPendulum.fmu`, which is provided by the fmi_adapter_examples package explained in the next section. This FMU implements a simple simulation model of a damped pendulum with configurable length, damping ratio and gravity constant:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription fmiVersion="2.0" modelName="dampedPendulum"
       description="..."
       guid="{192455ca-e123-595a-b9df-f95fe079b4ef}">

<CoSimulation modelIdentifier="dampedPendulum"
       canHandleVariableCommunicationStepSize="true">
  ... optional list of source files ...
</CoSimulation>

... some logging settings ...

<DefaultExperiment stepSize="0.002"/>
```

This first part of the XML refers to the FMI standard 2.0, provides the name of the FMU and a GUID (Globally Unique Identifier) string that is used for a unique instantiation and consistency check between the model description (XML) and the model representation (source code and/or binary). It also states that this FMU is a Co-Simulation FMU that can not only handle fixed but also variable communication time steps, as indicated by the capability flag `canHandleVariableCommunication-StepSize`. A crucial parameter for the simulation is the default step size, which is set to 2 ms in this example.

The next section of the xml file describes the model variables by specifying their names, attributes, and initial values:

```
<ModelVariables>
  <ScalarVariable name="a" valueReference="0" causality="output"
         variability="continuous"
         description="Angular displacement (in radians)">
    <Real/> <!-- index="1" -->
  </ScalarVariable>
  <ScalarVariable name="der(a)" valueReference="1"
         description="Angular velocity">
    <Real derivative="1"/> <!-- index="2" -->
  </ScalarVariable>
  <ScalarVariable name="der(a,2)" valueReference="2"
         description="Angular acceleration">
    <Real derivative="2"/> <!-- index="3" -->
  </ScalarVariable>
  <ScalarVariable name="l" valueReference="3"
```

```
28          causality="parameter"
29          variability="fixed" initial="exact"
30          description="Length of the pendulum (in meter)">
31     <Real start="1.0"/> <!-- index="4" -->
32   </ScalarVariable>
33   ... similar tags for damping ratio and gravity constant ...
34 </ModelVariables>
```

Finally, the structural dependencies between the known and unknown variables of the model are defined. The `Derivatives` refer those variables required to compute the corresponding state variables during a simulation step. The `InitialUnknowns` define the variables to be computed during the initialization. The variables are referred to by their `index`, which is a unique number within the XML file applied in increasing order starting from 1.

```
33 <ModelStructure>
34   <Derivatives>
35     <Unknown index="2"/>
36     <Unknown index="3"/>
37   </Derivatives>
38   <InitialUnknowns>
39     <Unknown index="2"/>
40     <Unknown index="3"/>
41   </InitialUnknowns>
42 </ModelStructure>
43
44 </fmiModelDescription>
```

In this example the two variables with index 2 and index 3, respectively `der(a)` and `der(a,2)`, are both declared as unknown derivative and as well as unknown initial variable.

2.3 *FMI Tools*

There is a rapidly growing number of (simulation) tools that support the FMI standard. The official list can be found in the tools section of the FMI website [1]. In May 2019, 132 tools were listed. Many other tools (e.g., in-house simulation tools of companies) exist beyond the list.

2.3.1 Introspection Tools and FMU Cross Check

Even though there is no tool available that can 100% guarantee the compliance with the FMI standard, there are very good tools available to support the users in performing compliance checks. One must distinguish between checking exported FMUs (i.e., whether the exported FMUs comply with the FMI standard) and checking the import and simulation of FMUs (i.e., whether the importing tools comply with the

Filter

Legend: planned supported cross-check passed

How do I add or update my tool?

			FMU Export		FMU Import	
Name	License	Platforms	Co-Simulation	Model Exchange	Co-Simulation	Model Exchange
20-sim	$		1.0 2.0		1.0 2.0	
20-sim 4C	$				1.0 2.0	1.0 2.0
@Source	$			1.0		
Adams	$		1.0 2.0		1.0 2.0	1.0 2.0
AGX Dynamics	$		1.0 2.0			
Altair Activate	$		2.0	2.0	1.0 2.0	1.0 2.0
AMESim	$		1.0 2.0	1.0 2.0	1.0 2.0	1.0 2.0
ANSYS CFX	$				2.0	
ANSYS DesignXplorer	$			1.0		
ANSYS SCADE Display	$		1.0 2.0	1.0 2.0		
ANSYS SCADE Suite	$		1.0 2.0	1.0 2.0		
ANSYS Simplorer	$		1.0 2.0	1.0 2.0	1.0 2.0	1.0 2.0
AUTOSAR Builder	$		1.0 2.0	1.0		

Fig. 3 FMI cross-check results displayed in the tools section of the FMI website [1]

standard). In the first case, checking the export of FMUs, one can use the following two tools:

- *FMI Compliance Checker* [19]
- *FMPy* Python toolbox [20]

In addition, the *FMI Cross-Check* [21] measures the FMI maturity level of the importing and exporting tools (also using the FMPy Python toolbox) by validating uploaded FMUs. The results of such checks are displayed in the tools section of the FMI website [1]. Exporting tools provide FMUs together with reference solutions, while importing tools report on the results of the simulation. If a minimum set of FMUs is simulated successfully, the corresponding tool gets a green badge as displayed in Fig. 3.

In the repository of the FMI Cross-Check, one can find hundreds of example FMUs. The *Test FMUs* by Dassault Systèmes [22] are of particular interest as they provide checks of importing tools inspired by the concept of *reference FMUs* developed in [23]. Furthermore, a long list of different FMI supporting tools is provided with:

- Commercial tools: Modelica-based simulation tools (e.g., Dymola, SimulationX, and MapleSim) as well as other simulation tools (e.g., MATLAB/Simulink, AMESim and GT-Suite).
- Open-source tools: Simulation tools (e.g., OpenModelica), SDKs (e.g., FMU-SDK), test FMUs, and supporting tools for FMU generation and simulation for many programming languages like C, Java, and Python.

2.4 Limitations of FMI

Due to its design with the possibility for black-box model exchange, FMUs hide internal information of the model. When splitting overall systems into multiple FMUs, it is very important to carefully partition the system and to carefully select a proper co-simulation master algorithm to avoid numerically difficulties in case of tight couplings.

The currently released versions FMI do not support structured signal interfaces. Hence, interfaces consist of scalars only and there is no semantics of physical connectors, which can easily lead to a poor graphical readability of models with a large number of inputs and outputs. Also does FMI addresses "only" the interface problems, not the underlying numerical problems. Again, note that master algorithms are not part of the FMI standard.

Some of these issues will be improved by the next FMI release as described in the next subsection.

2.5 Future Trends in the Development and Usage of FMI

The version 3.0 of the FMI standard is currently under development. It will provide the following new features, cf. FAQ section of the FMI website [1]:

- *Support for ports and icons:* Help the user to build consistent systems from FMUs and render the systems more intuitively with better representation of structured ports (e.g., buses and physical connectors) in the model description file.
- *Array variables:* Allow FMUs to communicate multi-dimensional variables of variable size using structural parameters.
- *Clocks and hybrid co-simulation:* Introduce clocks for synchronization of variable changes across FMUs. Allows co-simulation with event handling.
- *Binary data type:* Adds an opaque binary data type to FMU variables to allow, for instance, the efficient exchange of complex sensor data.
- *Intermediate variable access:* Allow access to input and output values between communication time instances to enable an enhanced numerical stability by advanced co-simulation master algorithms or to disclose relevant subsystem behavior for analysis purposes. (See [24] for more information on master algorithms. A master algorithm that could be implemented using this new FMI 3.0 feature is described in [25].)
- *Better support for source code FMUs:* Adding more information to the model description file to improve automatic import of source code FMUs.
- *Numeric variable types:* Adds 8, 16, 32 and 64-bit signed and unsigned integer and single-precision floating-point variable types to improve efficiency and type safety when importing or exporting models from the embedded, control and automotive domains.

While FMI was originally particularly developed for modeling and simulation on non-real-time systems, FMI is also successfully used on real-time systems like Hardware-in-the-Loop (HiL) systems. FMUs have been ported prototypically to embedded Electronic Control Units (ECUs), as demonstrated in [26]. This has inspired the standardization effort *FMI for Embedded Systems* (eFMI) within the ITEA 3 EMPHYSIS Project, see [27]. An additional approach of using FMI within control software is the usage of FMI within ROS-based applications, which is described in the next section.

3 The fmi_adapter Package

The fmi_adapter is a ROS package for wrapping co-simulation FMUs into ROS nodes. It has been implemented for ROS 1 as well as for ROS 2 in the C++ programming language. Binary distributions are currently available for ROS Melodic and for ROS 2 Crystal, Dashing, and Eloquent.

The fmi_adapter package aims at providing a mapping between the most important functions of the FMI 2.0 Co-Simulation interface and corresponding ROS concepts and types as depicted in the following table:

FMI	ROS
input variable	subscriber
output variable	publisher
state variable	*no explicit counterpart*
parameter initialization	parameter server
simulation time	ROS clock
communication step-size	timer

By this mapping, FMUs can be integrated with ROS without the necessity to use FMI types in the ROS C++ code. Fmi_adapter is intended neither to implement the whole FMI 2.0 API nor to provide the rich set of introspection functions as, for instance, FMI Library [28] or FMI4cpp [29]. For advanced use-cases with FMUs, ROS developers are referred to such libraries.

The fmi_adapter package allows for two types of usage named *node-based use* and *library-based use* in the following. Accordingly, it provides an executable ROS node as well as a shared library. In node-based use, the fmi_adapter node is called directly with a path to an FMU. The node creates a subscriber for each input variable of the FMU and a publisher for each output variable. Then, it runs—i.e., simulates—the FMU according to ROS time.

The library-based use gives much more control about the integration of this FMU in a ROS node. For this type of usage, the fmi_adapter package provides a C++ class `fmi_adapter::FMIAdapter`, which is instantiated with the path to an FMU and wraps this FMU. This allows translating complex ROS message types to the input types of the FMU and vice-versa for the output types. In addition, the class

provides member functions to introspect the FMU. Finally, this type of usage allows for multiple FMUs inside a single node.

In this section, we first describe how to install or build the fmi_adapter package for ROS 1 and ROS 2, followed by instructions on how to setup a free modeling tool for creating own FMUs. Then, we explain how to run the sample applications provided by the fmi_adapter_examples package and how to create and run an own FMU in node-based use. Thereafter, we give details on library-based use and the interface provided by the `FMIAdapter` class. The section is concluded with a description of the architecture of the fmi_adapter package and selected implementation details.

3.1 ROS Environment Configuration

The fmi_adapter package and the associated fmi_adapter_examples package are shipped with all dependencies. Therefore, they can be installed or built directly on current vanilla ROS 1 or ROS 2 installations. There is one important limitation for the fmi_adapter_examples package: The two provided sample FMUs (`DampedPendulum.fmu` and `TransportDelay.fmu`) have been built for AMD64 architecture only. We tested them with Ubuntu 16.04 and Ubuntu 18.04 successfully.

3.1.1 Binary Installation

On Debian, Ubuntu, and related Linux distributions, the fmi_adapter package for ROS Melodic and the associated example package can be installed using apt with the command:

```
sudo apt install ros-melodic-fmi-adapter
sudo apt install ros-melodic-fmi-adapter-examples
```

Similarly, the binary packages for ROS 2 Dashing can be installed by:

```
sudo apt install ros-dashing-fmi-adapter
sudo apt install ros-dashing-fmi-adapter-examples
```

3.1.2 Building from Source

To build the fmi_adapter package with its examples for ROS 1 from source, we recommend using the usual Catkin Command Line Tools [30]. This also allows building the package for older ROS versions such as Kinetic and Lunar. Detailed instructions are given in the README.md of the corresponding repository [3]. Please note that during the build, the *FMILibrary* from JModelica.org [28] is being downloaded and built as a third-party library using CMake's `externalproject_add` command [31].

Similarly, the ROS 2 version of the fmi_adapter package can be built from source using the usual colcon tool [32]. Further instructions are given in the README.md

of the repository [4]. Unlike in the ROS 1 version, the FMILibrary is not downloaded and built by the build file of the fmi_adapter package itself, but by the build file of a separate vendor package named *fmilibrary_vendor*.

3.1.3 Installation of Modeling Tools

The fmi_adapter_examples package provides two FMUs (`DampedPendulum.fmu` and `TransportDelay.fmu`) as well as a Modelica file (`DampedPendulum.mo`), from which a third one can be created.

Several tools may be used for the latter, as explained in Sect. 2. In the next subsection, we explain this procedure by the example of OpenModelica. Please navigate to the Download page of the OpenModelica website [33] for detailed instructions on how to install it from pre-built Debian packages. The website also provides instructions for installations under Windows and macOS as well as for building from source.

The two prebuilt FMUs `DampedPendulum.fmu` and `TransportDelay.fmu` distributed in the fmi_adapter_examples package have been implemented using the FMU SDK by QTronic. To follow the next subsections, there is no need to install this SDK. Nevertheless, if you are interested, this SDK can be downloaded from the QTronic website as a ZIP file [34]. Once unzipped, several further sample FMU projects can be found at `fmu20srcmodels`. They can be build using the `build_fmu` script in the same folder.

3.2 First Applications by Node-Based Use

As a first example, we run the `DampedPendulum.fmu` from the fmi_adapter_examples package located in the `share` folder. Thereafter, we run a more advanced example with the two provided FMUs `DampedPendulum.fmu` and `TransportDelay.fmu` before we simulate a damped pendulum FMU created from the `DampedPendulum.mo` file.

3.2.1 Simple Damped Pendulum Example on ROS 1

The examples package provides a launch file to run the prebuilt `DampedPendulum.fmu`. Simply call

```
roslaunch fmi_adapter_examples simple_damped_pendulum.launch
```

from the command line. This launches two nodes: First, the fmi_adapter node with the `DampedPendulum.fmu` as value for the `fmu_path` parameter. Second, rqt_plot with argument `/fmi_adapter_node/a`. This last argument makes rqt_plot to visualize the data published by the fmi_adapter node on the topic `/fmi_adapter_node/a`, which is the angle of the simulated pendulum. Figure 4 shows the output by rqt_plot.

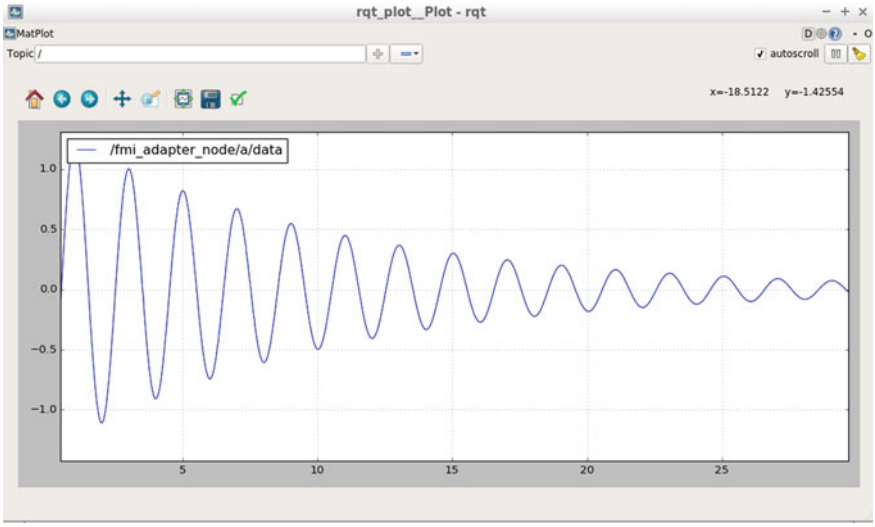


Fig. 4 Output by rqt_plot for the angle of the simple damped pendulum

The launch file `simple_damped_pendulum.launch` consists of the following six lines:

```
1 <launch>
2   <include file="$(find fmi_adapter)/launch/fmi_adapter_node.
      ↪ launch">
3     <arg name="fmu_path" value="$(find fmi_adapter_examples)/
        ↪ DampedPendulum.fmu" />
4   </include>
5   <node name="rqt_plot" pkg="rqt_plot" type="rqt_plot" args="/
      ↪ fmi_adapter_node/a" />
6 </launch>
```

The lines 2–4 start the fmi_adapter node by calling a tiny, generic launch file from the fmi_adapter package with the path to the `DampedPendulum.fmu`. It is also possible to refer directly to the fmi_adapter node by replacing the lines 2–4 with:

```
2   <node name="fmi_adapter_node" pkg="fmi_adapter" type="node">
3     <param name="fmu_path" value="$(find fmi_adapter_examples)/
        ↪ DampedPendulum.fmu" />
4   </node>
```

The name of the topic for the pendulum's angle is composed from the name of the fmi_adapter node (which is set to `fmi_adapter_node` in line 2) and the name `a` of the output variable for the pendulum's amplitude (the angle in radians) specified in the `modelDescription.xml` file of `DampedPendulum.fmu`.

Of course, the fmi_adapter node may be also invoked directly using `rosrun`. For example, assuming a standard installation of the fmi_adapter_examples package, the damped pendulum simulation can be started by:

```
rosrun fmi_adapter node _fmu_path:=/opt/ros/melodic/share/
    ↪ fmi_adapter_examples/DampedPendulum.fmu
```

3.2.2 Simple Damped Pendulum Example on ROS 2

On ROS 2, the simple damped pendulum example can be invoked by

```
ros2 launch fmi_adapter_examples simple_damped_pendulum.launch.py
```

analogously to ROS 1. The output may be retrieved on a second command prompt by

```
ros2 topic echo /a
```

which should give an output like

```
data: -1.0744451949191
data: -1.067165037971358
data: -1.0571404911146913
```

and so on. Note that the node name is not prepended to the topic name as in ROS 1.

As in the ROS 1 version, the fmi_adapter package provides a generic launch file, which is called from `simple_damped_pendulum.launch.py` with the `DampedPendulum.fmu` as value for the `fmu_path` parameter. The following listing shows this Python-based launch code.

```
1  ... some import statements ...
2
3  def generate_launch_description():
4      fmu_path = (ament_index_python.packages.
           ↪ get_package_share_directory('fmi_adapter_examples') +
           ↪ '/share/DampedPendulum.fmu')
5
6      fmi_adapter_description = launch.actions.
           ↪ IncludeLaunchDescription(
7          launch.launch_description_sources.
               ↪ PythonLaunchDescriptionSource(
8              ament_index_python.packages.
                   ↪ get_package_share_directory ('fmi_adapter')
9              + '/launch/fmi_adapter_node.launch.py'),
10         launch_arguments={'fmu_path': fmu_path}.items())
11
12     description = launch.LaunchDescription()
13     description.add_action(fmi_adapter_description)
14
15 return description
```

The path to the FMU file is determined in line 4. The lines 7–9 load the launch file from the fmi_adapter package. For ROS 2, the fmi_adapter node has been implemented as a *ManagedNode* [35], which provides a standardized runtime lifecycle. The generic `fmi_adapter_node.launch.py` file automatically switches the node from the initial state *unconfigured* to *inactive* and then to *active*.

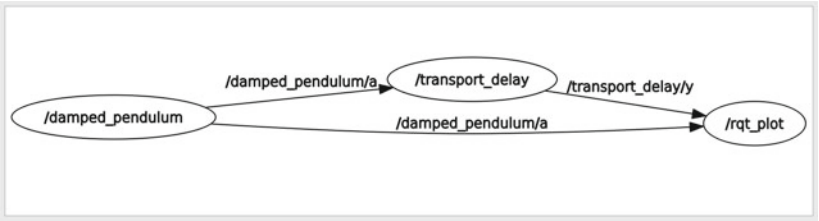


Fig. 5 Screenshot of rqt_graph for damped pendulum with transport delay example

3.2.3 Damped Pendulum with Transport Delay Example

The examples package provides a second launch file `damped_pendulum_with_transport_delay.launch`, which creates two instances of the fmi_adapter node with the damped pendulum FMU and a simple transport delay FMU. The latter gets the output of the pendulum FMU as an input and delays it by 2.33 s as depicted in Fig. 5. On ROS 1, call

```
roslaunch fmi_adapter_examples
    ↪ damped_pendulum_with_transport_delay.launch
```

to launch this example. As with the simple damped pendulum, rqt_plot is opened and visualizes both, the output of the damped pendulum FMU and the output after the transport delay.

On ROS 2, the corresponding command is

```
ros2 launch fmi_adapter_examples
    ↪ damped_pendulum_with_transport_delay.launch.py
```

but does not provide a visualization. Here, the launch file does not call the generic `fmi_adapter_node.launch.py` launch file from the fmi_adapter package but directly starts two instances of the fmi_adapter node. In doing so, the lifecycles of the two instances are switched synchronously from unconfigured to inactive and then to active.

3.2.4 Running Your Own Damped Pendulum FMU

The fmi_adapter_examples package contains a Modelica file of a damped pendulum model created with OpenModelica. If you performed a binary installation of the package, you may open the file by

```
OMEdit /opt/ros/melodic/share/fmi_adapter_examples/DampedPendulum
    ↪ .mo
```

or rather

```
OMEdit /opt/ros/dashing/share/fmi_adapter_examples/share/
    ↪ DampedPendulum.mo
```

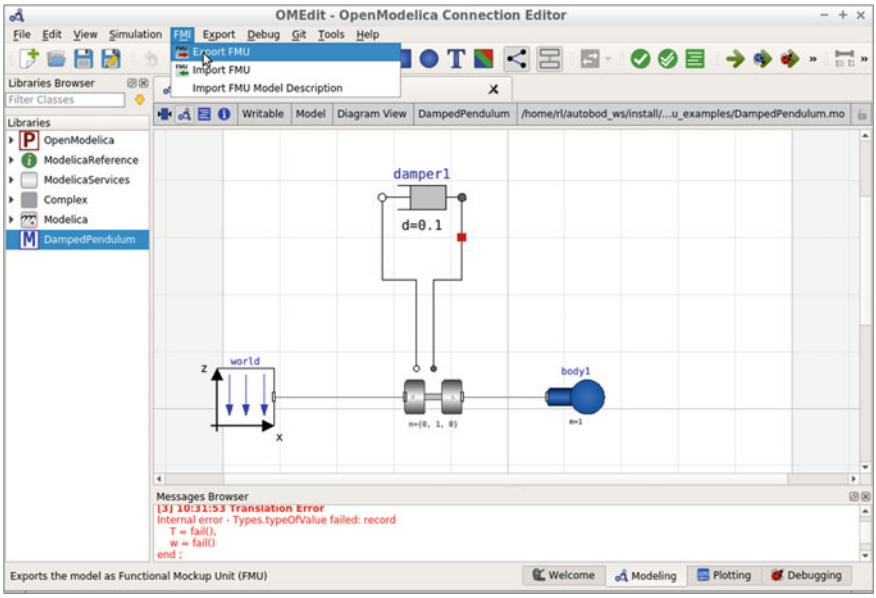


Fig. 6 Screenshot of OpenModelica connection editor with DampedPendulum.mo model

in the OpenModelica Connection Editor. To export an FMU from this file, perform the following steps (cf. also Fig. 6):

- Click on the DampedPendulum model in the project tree on the left.
- Navigate to Tools → Options → FMI and ensure that Version=2.0, Type=Co-Simulation and Platforms=Dynamic is selected.
- Then click File → Export → FMU.[1]
- The path of the resulting FMU file is shown in the message browser at the bottom of the window, typically `/tmp/OpenModelica_[user]/OMEdit/DampedPendulum.fmu`, where `[user]` is your user name.

Once the FMU has been exported, you may use the generic launch file of the fmi_adapter package to launch it by

```
roslaunch fmi_adapter fmi_adapter_node.launch fmu_path:=/tmp/
    ↪ OpenModelica\_[user]/OMEdit/DampedPendulum/DampedPendulum.
    ↪ fmu
```

or, alternatively, using rosrun, by

```
rosrun fmi_adapter node _fmu_path:=/tmp/OpenModelica\_[user]/
    ↪ OMEdit/DampedPendulum/DampedPendulum.fmu
```

The angle of the pendulum is published on the topic `/fmi_adapter_node/revolute1_angle`.

[1] In versions before 1.14, navigate to the menu FMI and click Export FMU.

ROS 2 does not support to pass arguments directly to launch files. For `ros2 run`, the parameters have to be passed in a YAML file. Therefore, create a file `my_params.yaml` as follows:

```
fmi_adapter_node:
    ros__parameters:
        fmu_path: '/tmp/OpenModelica\_[user]/OMEdit/DampedPendulum
            ↪ /DampedPendulum.fmu'
```

Then, start the fmi_adapter node by

```
ros2 run fmi_adapter fmi_adapter_node __params:=my_params.yaml
```

Nothing will happen at this point yet, since the node will start in the lifecycle state *unconfigured*. To change the state to *active* call

```
ros2 lifecycle set /fmi_adapter_node configure
ros2 lifecycle set /fmi_adapter_node activate
```

from a second terminal. Now, the pendulum's angle is published on the topic `/revolute1_angle`.

3.3 Interface Description and Library-Based Use

Next, we first explain the whole interface of the fmi_adapter node—i.e., for node-based use—before we provide details on the library-based use.

3.3.1 Node-Based Use

From the previous examples, you already learned most interface elements in node-based use. In detail, these elements are:

- *fmu_path:* The most important interface element is the mandatory parameter `fmu_path`, which takes the path to the FMU to simulate in this node.
- *Subscribers for input variables:* The fmi_adapter node reads the model description of the FMU and creates a subscriber for each input variable of the FMU. The subscribers are named as the corresponding input variables, where special characters that are not allowed in ROS names are replaced with an underscore. Currently, the fmi_adapter supports 64-bit floating-point variables only—and creates subscribers of type `std_msgs/Float64` accordingly.
- *Publishers for output variables:* Analogously to subscribers, a publisher is created for each output variable of the FMU.
- *Parameters for FMU parameters and start values:* The fmi_adapter node also reads any FMU parameter and variable start values from potential ROS parameters. For this purpose, it queries the ROS parameter server for entries that match the name of a variable or parameter of the FMU (again replacing special characters by underscore). For each such entry being found, the value is read from the ROS

parameter server and the correspondent FMU parameter or variable is initialized with that value.

- *Parameters for solver step-size and update period:* The optional parameter `step_size` specifies the duration in seconds to be used by the FMU's solver as solving step-size. If this parameter is given, it overwrites the step-size specified in the FMU's model description. Note that the ROS parameter `step_size` supports fixed values only, whereas the FMU's model description may also specify a variable step-size. The ROS parameter `update_period` controls the frequency in which the output variables of the FMU are read and published on the corresponding topics. By choosing the update period higher than the step-size, the output rate can be reduced without affecting the solving precision.

3.3.2 Library-Based Use

In library-based use, the class `fmi_adapter::FMIAdapter` is instantiated with the path to an FMU, to wrap this FMU for further use with ROS data types and other ROS concepts. In addition to the FMU path, the constructor also takes an optional argument `stepSize`, which may be used to overwrite the step-size for the FMU's solver specified in the FMU's model description.

As explain above, the fmi_adapter package aims to map the major FMI concepts only and not to implement and map the whole FMI 2.0 API. The first simplification being made by the `FMIAdapter` class are the runtime states of an FMU. The FMI standard defines a set of runtime states for FMUs by a hierarchical state-machine [13, p. 103]. The major such states for co-simulation are *instantiated*, *InitializationMode*, *slaveInitialized*, and *terminated*. The `FMIAdapter` distinguishes between two states only: *InitializationMode*, in which parameters and initial values of variables can be set, and *slaveInitialized*, in which the actual simulation steps may be computed. The other two states are switched automatically in the constructor and destructor, respectively.

The Boolean member function `FMIAdapter::isInInitializationMode()` allows querying whether the FMU is still in initialization mode or has already entered the actual simulation state.

During initialization mode, the FMIAdapter provides the function `setInitialValue(..)` to set initial variable values as well as parameters by means of their name. The names of all variables and parameters can be obtained by the functions `getAllVariableNames()`, `getInputVariableNames()`, `getOutputVariableNames()`, and `getParameterNames()`. The parameters and variables may be also initialized from the ROS parameter server using the function `initializeFromROSParameters()`. This function queries the ROS parameter server for entries that match the name of a variable or parameter of the FMU as described above in node-based use. On ROS 2, the function `declareFromROSParameters()` has be to called first, to make potential parameter values set during launch available to the node.[2]

[2]The `declare_parameter` function was introduced in rclcpp in ROS 2 Dashing Diademata.

An `FMIAdapter` instance also allows querying whether the FMU's solver supports a variable communication step-size (by `canHandleVariableCommunicationStepSize()`) and to obtain the default step-size (by `getDefaultExperimentStep()`).

By invoking the function `exitInitializationMode(..)`, the `FMIAdapter` instance is set to *slaveInitialized* state. This function takes a ROS timestamp as argument. This timestamp defines the offset between the FMU's solver time (which typically starts at zero) and the ROS time (which is either the system time or a simulated time). This offset is used as clock translation between ROS and the FMU in all further function calls.

Values for the input variables can be set programmatically by the following function:

```
FMIAdapter::setInputValue(variable-name, timestamp, value)
```

The second argument specifies the ROS timestamp from which on the given value is valid. For each input variable, the `FMIAdapter` stores the timestamp-value pairs as an input trajectory for the FMU. Thus, `setInputValue` may be called arbitrarily in advance of solving. By the constructor flag `interpolateInput`, it can be controlled whether the input trajectory is interpolated linearly between the given timestamp-value pairs or whether it is considered as a step function (i.e., piecewise constant).

The FMU simulation can be advanced by a single step by `doStep()`, which returns the ROS timestamp until that the FMU solver has advanced to. This function may be also called with user-defined step-size value instead of the default step-size from the `FMIAdapter` constructor or the FMU. The function `doStepsUntil(..)` advances the simulation to a given ROS timestamp (modulo step-size). During the computation, the solver reads the input values from the input trajectories according to their timestamps. This feature of input trajectories allows translating between different sampling/sensor rates easily.

After each simulation step, `getOutputValue(..)` allows retrieving the current value of an output variable by means of its name.

3.3.3 Code Example for Library-Based Use

As an example for library-based use, we consider at node that runs both sample FMUs (damped pendulum and transport delay) together synchronously. The following code snippets show the node's main function—here for ROS 1, a port to ROS 2 is straightforward.

As a first step, the paths of the two FMUs are read from the parameter server, where exception handling in case of missing parameter values is omitted here. Then, the `FMIAdapter` class is instantiated for each FMUs using the same step-size of 20 ms.

```
1 #include <ros/ros.h>
2 #include <std_msgs/Float64.h>
```

```
#include <fmi_adapter/FMIAdapter.h>

int main(int argc, char** argv) {
  ros::init(argc, argv, "fmi_adapter_examples_integrated_node");
  ros::NodeHandle n("~");

  std::string pendulumFmuPath;
  n.getParam("pendulum_fmu_path", pendulumFmuPath);
  std::string delayFmuPath;
  n.getParam("delay_fmu_path", delayFmuPath);

  ros::Duration stepSize(0.02);
  fmi_adapter::FMIAdapter pendulumAdapter(pendulumFmuPath,
      ↪ stepSize);
  fmi_adapter::FMIAdapter delayAdapter(delayFmuPath, stepSize);
```

Next, some parameters of the FMUs are set manually. Here, we set the damping ratio d of the pendulum to 0.001 and the length of pendulum to 25 m. The delay is set to 5.5 s. Furthermore, the initial angle of the pendulum a is set to 1.3 rad.

```
  pendulumAdapter.setInitialValue("d", 0.001);
  pendulumAdapter.setInitialValue("l", 2.0);
  delayAdapter.setInitialValue("d", 5.5);

  pendulumAdapter.setInitialValue("a", 1.3);
```

By calling `exitInitializationMode(..)` with the same timestamp, both instances are put to the *slaveInitialized* state with exactly the same offset between the solver times and ROS time.

```
  ros::Time now = ros::Time::now();
  pendulumAdapter.exitInitializationMode(now);
  delayAdapter.exitInitializationMode(now);
```

In this example, the simulation steps of the FMUs are triggered by a timer, but they could be triggered likewise by a subscriber or service. Both solvers are advanced in a synchronized fashion in the timer's callback function (a lambda expression). Due to the dependency between the two FMUs, first the damped pendulum FMU is advanced, then the transport delay. In between, the pendulum's current angle is obtained from the damped pendulum FMU and fed into the transport delay FMU. As a last step, the callback publishes the current angle as well as the delayed angle. As usual, the actual computation starts with `ros::spin()`.

```
  ros::Publisher anglePub = n.advertise<std_msgs::Float64>("angle
      ↪ ",1000);
  ros::Publisher delayedAnglePub = n.advertise<std_msgs::Float64
      ↪ >("delayed_angle", 1000);

  ros::Timer timer = n.createTimer(stepSize, [&](const ros::
      ↪ TimerEvent& event) {
    pendulumAdapter.doStepsUntil(event.current_expected);
    double angle = pendulumAdapter.getOutputValue("a");
    delayAdapter.setInputValue("x", event.current_expected, angle
        ↪ );
```

```
    delayAdapter.doStepsUntil(event.current_expected);
    double delayedAngle = delayAdapter.getOutputValue("y");
    std_msgs::Float64 angleMsg;
    angleMsg.data = angle;
    anglePub.publish(angleMsg);
    std_msgs::Float64 delayedAngleMsg;
    delayedAngleMsg.data = delayedAngle;
    delayedAnglePub.publish(delayedAngleMsg);
  });

  ros::spin();
}
```

3.4 Architecture and Implementation Details

Internally, fmi_adapter is based on the FMI Library [28], which is an implementation of the FMI 1.0 and 2.0 standard in the C programming language. Figures 7 and 8 illustrate the architectural layering for node-based use and library-based use, respectively. From the first figure, it can be seen that the generic fmi_adapter_node is based directly on the `FMIAdapter` class and makes intensive use of introspection functions to determine the names of the subscribers, publishers, and parameters to create.

3.4.1 Use of FMI Library

The fmi_adapter package hides the FMI Library largely from the ROS developer. All FMI-specific types are translated to ROS types as explained in the introduction to this section. The `FMIAdapter` header file does not include any header from the FMI Library. Nevertheless, it provides access to the variable type

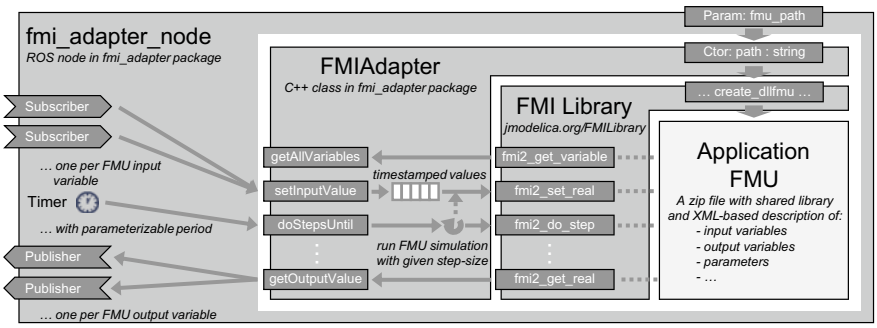


Fig. 7 Architecture diagram for node-based use

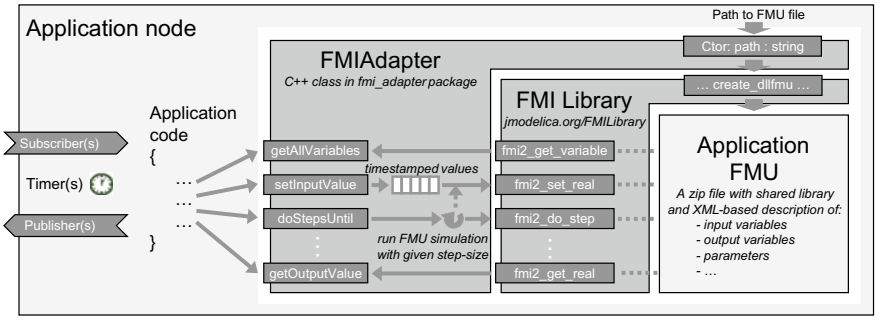


Fig. 8 Architecture diagram for library-based use

`fmi2_import_variable_t` of the FMI Library, but only as a pointer to forward-declaration. The use of this pointer type allows avoiding repeated name resolution of the variables for computational efficiency.

The FMI Library sources are not included in the fmi_adapter package but downloaded on the fly from `jmodelica.org` (using CMake's `externalproject_add` function) when building it. For ROS 2, a separate vendor package named *fmilibrary_vendor* has been introduced for this purpose.

3.4.2 Time Offset

As explained above, an `FMIAdapter` instance starts the FMU simulation from time zero on and stores the offset to ROS time to translate between the two clock representations. In fact, an FMU may also support other start times. We nevertheless decided for the explicit offset and the start time zero since the FMI 2.0 standard represents time by a floating-point type, which implies a significant loss of precision when using the system clock for the FMU time.

3.4.3 Special Characters in Names

The FMI standard supports various characters in the variable names that are not allowed in parameter or topic names in ROS and ROS 2. Therefore, the static function `FMIAdapter::rosifyName(..)` has been introduced to replace these characters by underscores. An `FMIAdapter` instance automatically applies this function to all FMU variable and parameter names. The same underscores have to be inserted when referring to such names in source code, launch files, or parameter YAML files.

3.4.4 Parameter Declaration in ROS 2

In ROS 2 Dashing, a parameter declaration mechanism was introduced in rclcpp. Amongst others, this mechanism allows to detect typos in parameter names and type errors at runtime. When using the automatic initialization of FMU parameters and start values from ROS parameters by the `initializeFromROSParameters()`, this requires declaring those ROS parameters first. Since their names depend on the FMU, the function `declareROSParameters(..)` has been introduced in the `FMIAdapter` class. This function should be called once before `initializeFromROSParameters()`.

Once a parameter is declared with a specific type by rclcpp's `declare_parameter(..)` function, it cannot be distinguished from the default value whether this value has been set explicitly (by launch or other nodes) or not. This, however, is necessary since an FMU may provide an internal mechanism to compute the start value for a variable without a specific default value. Therefore, `FMIAdapter::declareROSParameters(..)` declares all parameters with the data type `PARAMETER_NOT_SET`. By means of this data type, it can later decide on the parameter values that were given explicitly, to pass only those ones to the FMU.

3.4.5 Supported FMU Data Types

The FMI 2.0 standard defines six basic data types (`fmi2Real`, `fmi2Integer`, `fmi2Boolean`, `fmi2Char`, `fmi2String`, and `fmi2Byte`), which are mapped to the corresponding C types in the FMI Library. Currently, the fmi_adapter node and the `FMIAdapter` class support `fmi2Real` only, which is the type double in C/C++. We deem this sufficient for many practical applications, but other types could be added easily.

4 The gazebo-fmi Plugins

Gazebo is a 3D simulation environment for robotics, which is integrated in ROS binary distributions [36]. From the point of view of physics simulation, Gazebo implements an abstraction interface over several rigid body dynamics physics engines: a modified version of Open Dynamics Engine (ODE), Bullet, Simbody, and DART.

For several applications, the dynamics captured by this kind of simulations are not enough to achieve the necessary level of simulation accuracy with respect to the real system. Examples of these applications are quadruped control, both model-based [5] and reinforcement learning based [6, 7]. Other examples include the aerodynamics of a quadcopter and the dynamics of an underwater autonomous vehicle [37]. In most of these cases, the authors addressed the problem by augmenting the Gazebo simulator (or a simulator with similar characteristics) with manually coded C++ plugins that

co-simulate the necessary dynamics together with the usual multibody simulation of Gazebo. This approach requires a lot of C++ expertise and profound knowledge of the API and architecture of Gazebo, whereas it prevents to easily exploit any existing modeling environment. A different approach was used in the case of [38] where the dynamics model used in the Gazebo plugin was not manually programmed in C++, but was rather generated from a Matlab model.

Gazebo-fmi is a set of plugins for the Gazebo simulator that permit to import co-simulation FMUs in Gazebo to use them to perform co-simulation with the main Gazebo multibody simulation. The plugins integrate the FMUs in the main Gazebo physics thread. Therefore, in co-simulation terminology, the Gazebo physics thread acts as *co-simulation master* while the loaded FMUs with their solvers and models are *co-simulation slaves*. This permits to use any existing software that supports FMI export for modeling the behavior of interest, and just integrate it in Gazebo by writing an appropriate configuration file, without the need of writing any custom C++ code. In particular, as of version 0.2, gazebo-fmi supports integration of FMUs in Gazebo using two plugins: `gazebo-fmi-actuator` to simulate actuator dynamics and `gazebo-fmi-single-body-fluid-dynamics` to simulate the approximate interaction between a single body and a surrounding fluid, such as air or water.

The *actuator dynamics* refers to the relation between the input to the physical motor used in the mechanical model (voltage in the case of electric motors, and other quantities for pneumatic and hydraulic motors), the position, velocity, and acceleration of output shaft of the transmission connected to the motor and the actual effort transmitted by the output shaft on the mechanical structure. Out of the box, the physics engines used in Gazebo consider just the transmission output effort as an input to the simulation, and they only simulate basic joint friction behavior. By using gazebo-fmi's actuator plugin, it is possible instead to simulate the actuator dynamics such as reflected inertias, velocity/torques curves and series elastic actuators using an FMU.

The actuator plugin can be used transparently with the joint of any Gazebo model, regardless of how the joint is controlled: using the Gazebo internal PID controller or using ROS specific control plugin such as gazebo_ros_pkgs. This is possible by exploiting the Gazebo event system and by ensuring that the plugin is always running after the Gazebo PID update, but before the main Gazebo physics update, as explained in Sect. 4.4.

The *single body fluid dynamics* refers to the approximate models that are used for real-time simulations of vehicles that move in the air or in water: when a complete Computational Fluid Dynamics (CFD) simulation of the body-fluid interaction is not feasible, simplified models that just capture lift and drag for aerodynamics or drag, buoyancy, and added masses for hydrodynamics are used. By using gazebo-fmi's single body fluid dynamics plugin, it is possible to simulate these behaviors using an FMU.

This chapter describes the version 0.2 of gazebo-fmi. For updated information, please refer to the gazebo-fmi repository [8].

4.1 ROS Environment Configuration

4.1.1 Building from Source

Although it can be used in a ROS environment, gazebo-fmi is a C++ project that does not depend directly on any ROS or ROS 2 library. For this reason, as of version 0.2, the recommended way of using it is building it from source. Gazebo-fmi's *build system* is based on plain CMake, so it can be built as a standalone CMake project or using any *build tool* that support plain CMake, such as catkin-tools [30] or colcon [32]. For a definition of *build system* and *build tool*, see [39].

There are two main dependencies for the gazebo-fmi plugins. The first one is the package containing development libraries of Gazebo, usually available in Ubuntu in a deb package called `libgazebo[ver]-dev`, where `[ver]` is the major version of Gazebo, either `7`, `8`, `9`, or `10` for gazebo-fmi v0.2 [40]. The second dependency is the FMI Library from JModelica.org [28]. This library is not included in the official Ubuntu software repository, but can be downloaded and built as a third-party library using CMake's `FetchContent` module [41] if the `USE_SYSTEM_FMILIBRARY` CMake option is set to `OFF`.

On a system in which Gazebo and its development libraries are installed, compiling the library without any build tool can be done with the following steps

```
git clone https://github.com/robotology/gazebo-fmi
cd gazebo-fmi
mkdir build && cd build
cmake CMAKE_INSTALL_PREFIX=<install\_prefix> ..
make
make install
```

where `<prefix>` is the installation prefix for the library. Once installed, the directory `<install_prefix>/lib` should be added to the Gazebo environment variable `GAZEBO_PLUGIN_PATH` so that the gazebo-fmi plugins can be found by Gazebo, and any directory that contains FMUs that should be loaded by the Gazebo should be added to the `GAZEBO_RESOURCE_PATH` environment variable.

Further instructions are given in the README.md of the gazebo-fmi repository [8].

4.1.2 Installation of Modeling Tools

Similarly to what discussed in Sect. 3.1, a complete discussion of all the modeling tools (cf. tools section of the FMI website [1]) that support FMI and can be used with the gazebo-fmi plugins is beyond the scope of this chapter.

Among all the different existing tools, for continuous integration and provide examples usage, we use the open source Modelica environment OpenModelica. Please navigate to the Download page of the OpenModelica website [33] for detailed instructions on how to install it from pre-built Debian packages. The OpenModelica

website also provides instructions for installations under Windows and macOS as well as for building from source.

4.2 First Applications

In this section, we will discuss basic examples shipped with gazebo-fmi. To run these examples, it is not necessary to configure the `GAZEBO_PLUGIN_PATH` and `GAZEBO_RESOURCE_PATH` environment variable of Gazebo manually, but it is sufficient to source the `setup-examples.sh` script, which will set all relevant environment variables, by:

```
source <install_prefix>/share/gazebo-fmi/examples/setup-examples.
    ↪ sh
```

As first example, we run a Gazebo world that contains two pendulum models, available at `gazebo-fmi/plugins/actuator/examples/damped_pendulum/damped_pendulum.world`. This world contains two pendulum models whose geometric and inertial parameters are identical to the Modelica pendulum used in Sect. 3.2. Discussion on the details of how the pendulum are modeled using the Simulation Description Format (SDF) of Gazebo are out of scope here, the interested readers can refer to Gazebo tutorials [42], such as the simple_gripper tutorial.

The difference between the models is in their joint connecting the pendulum to the fixed base. In the first model `pendulum_damped_via_sdf`, the joint is defined as

```
<joint name="joint" type="revolute">
  <parent>world</parent>
  <child>pendulum</child>
  <axis>
    <xyz>0.0 1.0 0.0</xyz>
    <dynamics>
      <damping>0.1</damping>
      <friction>0</friction>
      <spring_reference>0</spring_reference>
      <spring_stiffness>0</spring_stiffness>
    </dynamics>
  </axis>
</joint>
```

In contrast, in the second model `pendulum_damped_via_fmi`, the joint is defined as

```
<joint name="joint" type="revolute">
  <parent>world</parent>
  <child>pendulum</child>
  <axis>
    <xyz>0.0 1.0 0.0</xyz>
    <dynamics>
      <damping>0.0</damping>
      <friction>0.0</friction>
```

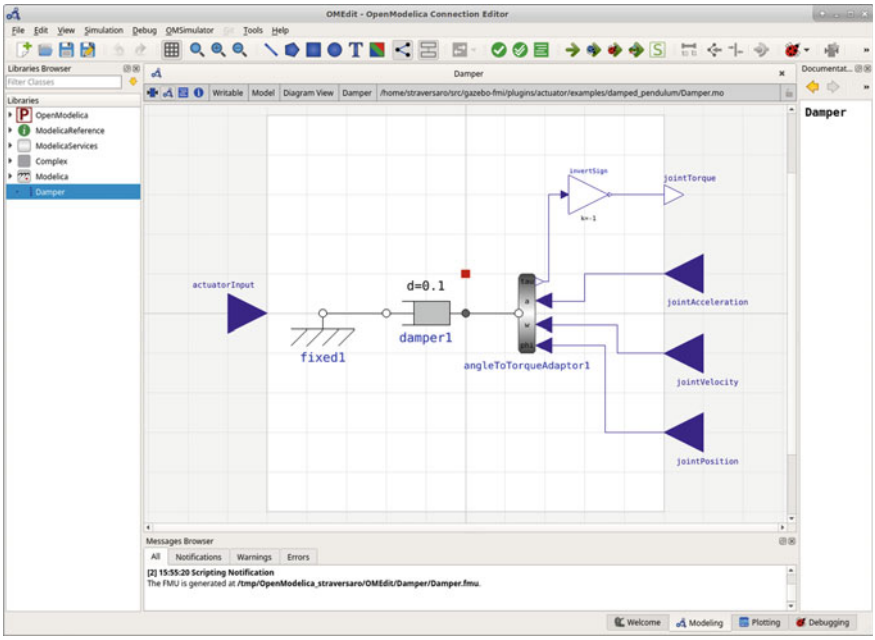


Fig. 9 Screenshot of OpenModelica connection editor with `Damper.mo` model

```
      <spring_reference>0</spring_reference>
      <spring_stiffness>0</spring_stiffness>
    </dynamics>
  </axis>
</joint>
<plugin name="damper" filename="libFMIActuatorPlugin.so">
   <actuator>
     <name>joint_damper</name>
     <joint>joint</joint>
     <fmu>Damper.fmu</fmu>
   </actuator>
</plugin>
```

The joint of the first model has a damping of 0.1, specified via SDF. The joint of the second model has a SDF damping of 0.0, but references a custom FMI actuator model named `Damper.fmu`. This FMU file should be generated from the `gazebo-fmi/plugins/actuator/examples/damped_pendulum/Damper.mo` Modelica model shown in Fig. 9. This model has been obtained by taking the damper used in Sect. 3.2 and copying it in a "shell" model that exposes the variable necessary be the actuator plugin, that are discussed in Sect. 4.3.

To run the world, there is no need to generate the FMU manually: If the OpenModelica compiler `omc` was installed in the system when the project was configured, the `Damper.fmu` file will be automatically generated by the CMake build. It will then

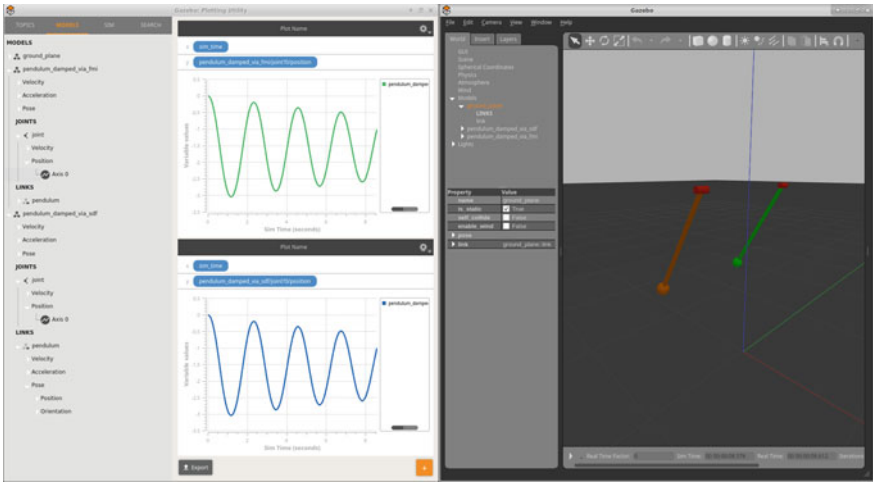


Fig. 10 Output by Gazebo for the angle of the two damped pendulums

be sufficient to run, as long the `setup-examples.sh` has been correctly sourced, the following command:

```
gazebo -u -e simbody damped_pendulum.world
```

The positional argument `damped_pendulum.world` is the example world, while the additional option `-u` is necessary to start Gazebo in paused mode, to permit to configure the Gazebo plot window (accessible via the Windows → Plot menu) to plot the variables of interest. In particular, for the sake of the analyzing the experiment, it is recommended to plot the joint positions for the two pendulums, plotting respectively the `pendulum_damped_via_sdf/joint?0/position` and the `pendulum_damped_via_fmi/joint?0/position` quantities. Figure 10 shows the output of Gazebo plotting the positions of the pendulum joints.

The additional option `-e simbody` is used to ensure that Gazebo uses Simbody [43], a physics engine that generally has lower numerical errors compared to the default physics engine, which is a modified version of the Open Dynamics Engine (ODE). In this particular example, running the simulation with Simbody ensures that the two pendulums are in sync, while running the example with ODE results in small differences that quickly make the two pendulum go out of sync.

4.3 Interface Description

In this section, for the sake of clarity, only the input/output interface of each plugin is discussed. Links to the complete documentation are provided for users interested in the additional details.

Fig. 11 Empty Modelica model that exposes the variables required for an FMU to be loaded by the actuator plugin

4.3.1 Actuator Plugin

The actuator plugin loads all FMUs that have at least the following four input variables and one output variable:

FMU variable name	Type	Description	Causality
`actuatorInput`	Real	Actuator input	`input`
`jointPosition`	Real	Joint position	`input`
`jointVelocity`	Real	Joint velocity	`input`
`jointAcceleration`	Real	Joint acceleration	`input`
`jointTorque`	Real	Joint torque	`output`

The units of the variables change depending of whether the joint is of *revolute* or *prismatic* type. The convention followed for units is the Gazebo's one of always using SI units.

An empty Modelica model that respects the interface required for an FMU to load by the actuator plugin is shown in Fig. 11.

A complete documentation of the parameters support by the `gazebo-fmi-actuator` plugin is given in the README.md of the gazebo-fmi repository [8].

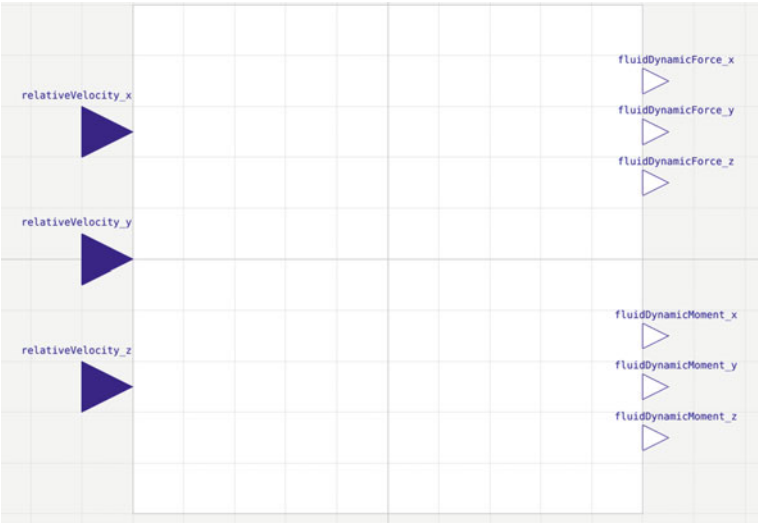


Fig. 12 Empty Modelica model that exposes the variables required for an FMU to be loaded by the single-body-fluid-dynamics plugin

4.3.2 Single Body Fluid Dynamics Plugin

The single body fluid dynamics plugin loads FMUs that have at least the following three input variables and the six output variables:

FMU variable name	Type	Description	Causality
`relativeVelocity_x`	Real	Body-medium relative velocity (X)	`input`
`relativeVelocity_y`	Real	Body-medium relative velocity (Y)	`input`
`relativeVelocity_z`	Real	Body-medium relative velocity (Z)	`input`
`fluidDynamicForce_x`	Real	Force exerted on the body (X)	`output`
`fluidDynamicForce_y`	Real	Force exerted on the body (Y)	`output`
`fluidDynamicForce_z`	Real	Force exerted on the body (Z)	`output`
`fluidDynamicMoment_x`	Real	Moment exerted on the body (X)	`output`
`fluidDynamicMoment_y`	Real	Moment exerted on the body (Y)	`output`
`fluidDynamicMoment_z`	Real	Moment exerted on the body (Z)	`output`

All the linear velocity components are expressed in $\frac{\mathrm{m}}{\mathrm{s}}$, the linear force components are expressed in N, and the moment components are expressed in Nm. All the quantities are expressed with the body orientation and with respect to the body origin.

An empty Modelica model that respects the interface required for an FMU to load by the single-body-fluid-dynamics plugin is shown in Fig. 12.

A complete documentation of the parameters support by the `gazebo-fmi-single-body-fluid-dynamics` plugin is given in the README.md of the gazebo-fmi repository [8].

4.4 Architecture and Implementation Details

Gazebo can be extended by six different types of plugins, which are C++ classes that inherit from a specific C++ interface. In particular, Gazebo supports *World*, *Model*, *Sensor*, *System*, *Visual*, and *GUI*. Each type of plugin is used to customize a different aspect of the simulation. As the gazebo-fmi plugins are used to augment Gazebo models with custom dynamical systems and mainly access model information, they are implemented as Model plugins. For more information on Gazebo plugins, see Gazebo's hello_world tutorial [42].

From the execution point of view, the entry point of all plugins is the `Load()` function, which is called by the Gazebo main physics simulation thread once when the entity that contains the plugin is loaded. During the load phase, the gazebo-fmi plugin searches for the specified FMU in the directory given in `GAZEBO_RESOURCE_PATH` and loads it using the FMI Library. If there is an error in the configuration, the specified FMU is not found, or there is an error during loading, the plugin prints an error message. This message is visible by launching Gazebo with the `-verbose` option. Note that the Gazebo model is still loaded as Gazebo does not expose any overall abort functionality in the case the loading a plugin fails.

Once the FMU is loaded, the co-simulation pattern requires that an FMU co-simulation step is performed at any Gazebo's physics engine update-step, and that after the co-simulation step data is exchanged between the FMU and the Gazebo physics engine. This interaction is implemented via events. The Gazebo Event System is an API that permits to register callbacks for specific events of the simulation cycle. These callbacks are executed by the Gazebo physics thread, ensuring proper synchronization between the Gazebo physics update step and the FMI integration. Examples of events include `Events::worldUpdateBegin`, used for callbacks that need to executed during the begin of a Gazebo physics update step; `Events::beforePhysicsUpdate`, used for callbacks that are executed after the collision are detected, but before the physics's engine integrator computed the new positions and velocities of the model, and `Events::worldUpdateEnd`, for code that needs to be executed at the end of the update step. Most Model plugins, including middleware-specific plugins such as the one contained in `gazebo_ros_pkgs`, typically just execute their periodic code during the `Events::worldUpdateBegin` callback.

Depending on the specific gazebo-fmi plugin, the actual FMI step and exchange of input/output data with the Gazebo physics engine is done during different events.

4.4.1 Actuator Plugin

The goal of the actuator plugin is to be used transparently with the joint of any Gazebo model, regardless of how the joint is controlled: using the Gazebo internal PID controller or using middleware specific control plugin such as gazebo_ros_pkgs or gazebo-yarp-plugins.

All this software control the behavior of the joints by calling the function `Joint::SetForce` in the main Gazebo physics thread, as part of the callback of the `Events::worldUpdateBegin` event. For this reason, the actuator plugin code runs *after* the joint control code has run in the `events::worldUpdateBegin` event, but *before* the actual physics engine is updated.

This is achieved by running in the callback of the event `events::beforePhysicsUpdate`. The actuator plugins reads the value that was set in the `Joint::SetForce` function and it passes it to the FMU as the `actuatorInput`. It then runs the FMU simulation, and it substitutes the value that was set in the `Joint::SetForce` with the `jointTorque` output of the FMU. For more information, this logic can be found in the `FMIActuatorPlugin::BeforePhysicsUpdateCallback` function code.

4.4.2 Single Body Fluid Dynamics Plugin

The single body fluid dynamics plugin code is called during the `events::worldUpdateBegin` event. The plugin reads the velocity of the link to which the plugin is associated and the velocity of the wind, and computes from these two quantities the relative velocity, which is then passed to the FMU. The computed force/torque is then added back to the link using Gazebo APIs to add external forces to links.

The mechanism of the plugin was inspired by the Gazebo plugin `LiftDragPlugin`, described in the Gazebo's Aerodynamics tutorial [42].

5 Use-Cases

In this section, we report on the use of the fmi_adapter in a ROS-based autonomous intra-logistics vehicle and on the implementation of an electrical actuator model based on gazebo-fmi.

5.1 FMU-Based Controller for a Self-driving Vehicle in Intra-logistics

Industrial logistics is an important application area of autonomous robotics, which gave rise to many new innovative products and solutions such as the *ActiveShuttle* from Bosch Rexroth [44], which is considered in this use-case. In the last years, a number of elaborate algorithms for task scheduling, coordination and path planning for fleets of self-driving vehicles in such applications have been proposed. Prerequisite to apply these strategies is a reliable vehicle motion control. Trajectories commanded by the planner need to be properly executed by the drive platform to ensure that the goals of the mission are met in time and space.

Model-based design is a well-established approach to develop and implement motion control strategies. Model-in-the-loop (MiL) simulations allow the early validation and test of the controller design against a plant model. Based on a physical model of the plant, dedicated controllers such as model predictive control (MPC) or feedback linearization can be designed that take the physical properties and system dynamics into account.

Exporting the validated controller model directly from the modeling and simulation environment as an FMU allows utilizing the fmi_adapter to integrate the model-based control function directly into a ROS-based software architecture. This generic and efficient workflow to enhance the capabilities of a robotic system with an advanced control function is demonstrated in the following by the self-driving intralogistics vehicle *ActiveShuttle DevKit*, which is the predecessor of today's product.

The analysis of various operation scenarios revealed that the transitions between moving straight and turning on the spot are prone to high bore friction (cf. Fig. 14) at the caster wheels. Typically, it is assumed that the passive caster wheels follow the commanded vehicle trajectory perfectly without interference. In practice, however, it can be observed that this is not necessarily the case. After standstill, a significant counteracting force occurs when the current orientation of the caster wheels is not compliant with the new commanded direction of motion.

To avoid locking under these potentially critical conditions, the *Path Filter* control function was developed avoiding high bore friction at the caster wheels. The self-contained control function was then integrated with the existing ROS-based motion planner and motion controller to improve the robustness and overall performance of the differential drive vehicle.

5.1.1 Model-Based Control Design and Validation in Modelica

Following the outlined model-based control design process, a plant model of the ActiveShuttle DevKit was developed to replicate the undesired behavior of differential drive vehicles with caster wheels.

The plant model is implemented in the Modelica language based on the open-source PlanarMechanics library [45]. The model takes the kinematic structure of the six-wheeled ActiveShuttle DevKit into account, as illustrated in Fig. 13. Lateral, longitudinal, and rotational inertial forces in the xy plane are considered by the planar body at the center of gravity. The distribution of the normal gravitational force to the kinematically overconstrained ground contact forces at the wheels is resolved by taking the chassis kinematics and orientation of the caster wheels into account.

In order to account for the physical effect of bore friction at the caster wheels properly, a tire model has been developed considering a contact patch as illustrated in Fig. 14.

The approach of [46] to describe the bore friction characteristic, see Fig. 15 on the left, has been improved as described in [47] by the characteristic in Fig. 15 on the right. In contrast to the other implementation does this model account for a continuous

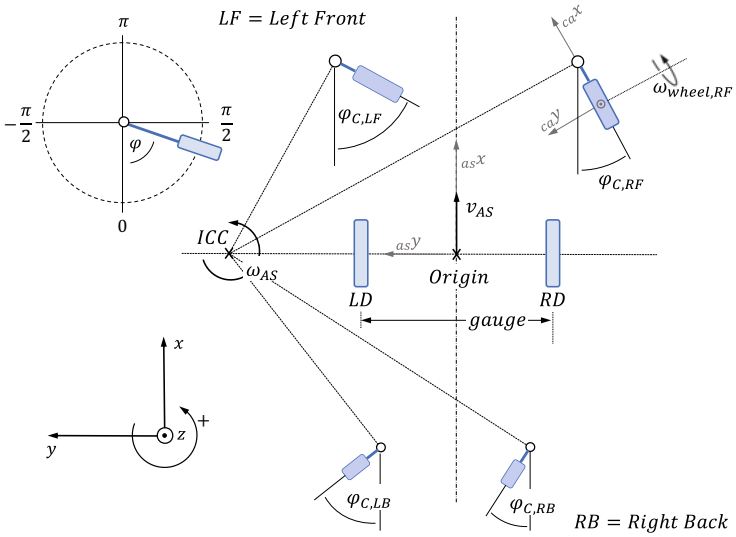


Fig. 13 ActiveShuttle DevKit motion state of driven and caster wheels

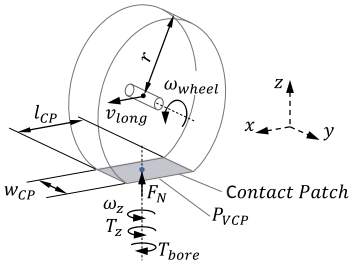


Fig. 14 Properties of the tire road contact considered in the wheel model

decrease of the bore torque with increasing rolling velocity ω_{wheel}, which is crucial to properly model the locking condition with ω_z and ω_{wheel} close to zero.

Based on this caster wheel model and the plant model of the ActiveShuttle DevKit, the critical operation scenarios were successfully simulated and analyzed, as depicted in Fig. 16.

While moving straight all caster wheels are facing forward with the instantaneous center of curvature (ICC) being located in infinite lateral distance from the origin according to Fig. 13. If the vehicle is stopped and a turn on the spot is introduced, the new position of the ICC will be at the origin. Accordingly, the orientations of the caster wheels are no longer compliant with the commanded motion state. This implies that all caster wheels have to be turned by an angle of $\Delta\phi_i$. As the rolling velocities of the caster wheels are initially zero, the maximum bore torque applies at all four caster wheels and has to be overcome simultaneously in addition to the inertial torque of the vehicle. Under the given geometric relations of the ActiveShuttle DevKit, the counteracting friction forces applied at the mountings of the caster wheels have a

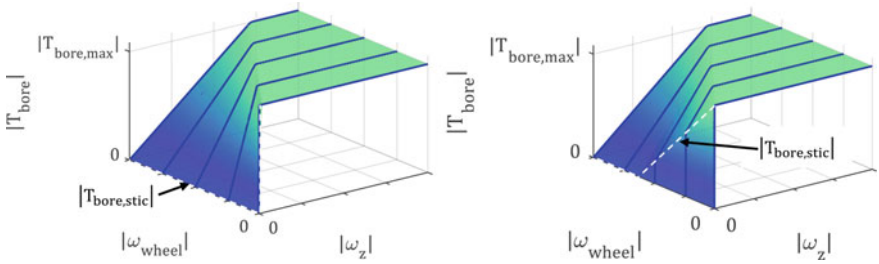


Fig. 15 Bore friction characteristics on the left as proposed by [46], on the right the enhanced variant as applied to the ActiveShuttle DevKit plant model

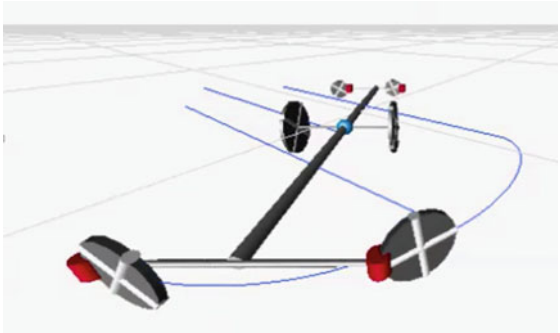


Fig. 16 3D animation of the simulated turning on the spot of the ActiveShuttle DevKit with *path filter* in Dymola

longer lever arm than the driving forces at the driven wheels. Under full load and for high stiction coefficients (rubber tires on concrete floor) the counteracting forces lead to high peak current of the drives and can even exceed the power limits of the drives.

Based on this simulation model, a control strategy to enhance the existing control architecture was developed and tested. The basic idea behind the *Path Filter* is to circumvent the high bore friction torque for $\omega_{wheel} \approx 0$ by forcing the motion controller to follow a trajectory that first sets the caster wheels into a rolling motion before introducing the turn.

Following this approach, the Path Filter can be implemented as a function with the commanded left and right drive velocities as inputs and the corrected command velocities as outputs. The input velocities are used to estimate the motion state of the caster wheels $[\phi_i, \omega_i]$ in order to determine the gap $\Delta\phi$ between the current and the desired orientation of the caster wheels. The following relation is then used to determine the corrected motion state of the caster wheels, which is finally projected back onto the velocities of the driven wheels and provided as filtered command value

$$\varphi_{C,fil} = \hat{\varphi}_C + k \cdot \Delta\varphi_C \tag{1}$$

to the motion controller, with

$$k = min\left(1, |\frac{\omega_C}{\omega_{max}}|\right). \tag{2}$$

When the ActiveShuttle DevKit is at standstill, the motion controller is forced to start moving in the direction of the current orientation of the caster wheels $\hat{\varphi}_C$. With increasing rolling velocity of the caster wheels ω_C, the influence of the correcting term is decreased linearly to zero for $\omega_C \geq \omega_{max}$. In other words, the path filter describes a trajectory in the bore friction characteristic of the caster wheels that avoids the peak friction torques (cf. Fig. 15). ω_{max} represents the slope of k. Small values reduce the time Δt that is necessary to transfer between estimated and desired state, but at the same time describe a trajectory in the bore friction characteristic that encounters higher bore torques. Hence, the choice of ω_{max} can be seen as a trade-off between deviation from the commanded trajectory and required effort.

5.1.2 Controller Integration Using the fmi_adapter

In order to integrate the previously described path filter functionality into the existing ROS-based navigation architecture, the fmi_adapter has been utilized as described in Sect. 3. Therefore, the tested and validated function has been exported directly from the simulation environment. This was performed simply by opening the path filter block of the Modelica model and using the FMU for Co-Simulation export command.

In the navigation architecture of the ActiveShuttle DevKit, drive commands are represented as `TwistStamped` messages (from the `geometry_msgs` package) representing the lateral and rotational velocity in the corresponding fields `twist.linear.x` and `twist.angular.z`. This is a very common representation in ROS for velocity commands for differential drive robots. In the navigation stack of the ActiveShuttle DevKit, the desired velocity is sent as twist message from the *path tracker* node to the *engine driver* node on a topic named `/velDes`. The path tracker node determines the desired velocity based on the input from the global *path planner*. The engine driver node translates the twist command into motor speed commands for the left and right drive.

In order to consume the previously exported *Path Filter FMU*, a new node named `PathFilter` is added using the fmi_adapter library. This new node consists of one function `main` with only 25 lines of code. The path filter node is placed between the path tracker and the engine driver node, as depicted in Fig. 17, by receiving a `TwistStamped` message on the `/velDes` topic and publishing the resulting filtered velocity command outputs as `TwistStamped` message published on a new `/velDesFil` topic.

The `PathFilter` node feeds the Path Filter FMU with the values of `twist.linear.x` and `twist.angular.z` using the `setInputValue` interface. The FMU is then updated to the current time. The resulting filtered velocity commands are read through the `getOutputValue` interface from the FMU and published.

Integrating the `PathFilter` node in the existing architecture, required only two lines in the corresponding ROS launch file to be changed: A new line for the `Path-`

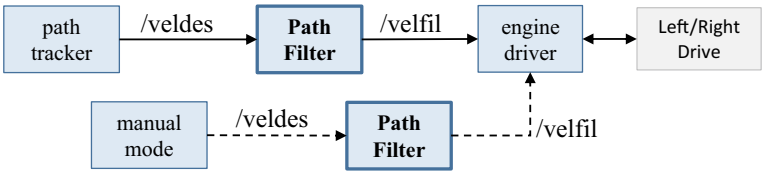


Fig. 17 ROS graph with the path filter FMU node integrated into the ActiveShuttle DevKit navigation architecture (solid lines) and the test setup (dashed lines)

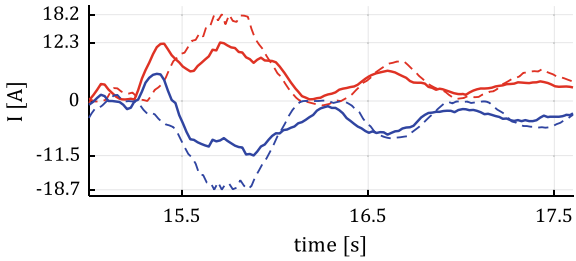


Fig. 18 Motor currents left/right (red/blue) with path filter (solid) and without path filter (dashed) for a turning-on-the-spot movement

`Filter` node was added and the input topic for the engine driver node changed to `/velDesFil`.

5.1.3 Application and Test Results

For test purposes, a simplified setup of the control architecture has been applied using the *manual* mode, as depicted in Fig. 17 (dashed lines). In this setup the `PathTracker` node is deactivated and the desired motion is published to the `/velDes` topic by a node that is interfaced with a joystick. This way it was straightforward to try out the following motion profile:

forward → stop → turn→ stop → forward → stop → backward

The profile was driven with and without path filter. The ActiveShuttle DevKit was fully loaded with 150 kg, which leads to a total weight of ≈200 kg. The relevant topics `/velDes`, `/velDesFil`, and `/enginedata` were recorded with the `rosbag` tool. The latter topic holds the messages with the measured motor speeds and currents. In addition, video recordings documented the movement of the caster wheels.

The measured motor currents for the first part of the motion profile (turning on the spot) are plotted in Fig. 18. Comparing the solid and dashed curves shows that the total effort necessary to perform the turn on the spot is significantly reduced by the path filter. The peak currents are reduced by 30% from 18 to 12 A. Consequently, also the oscillations are significantly smaller which can also be observed in the video recordings.

5.1.4 Conclusions

The use-case of extending the motion controller of a self-driving vehicle for intra-logistics demonstrates an efficient model-based design workflow for ROS enabled by the fmi_adapter. Through the fmi_adapter, it is possible to directly integrate a control function designed and validated in a simulation environment as FMU into an existing ROS architecture. This allows taking full advantage of a simulation environment to analyze the system behavior, try out different control strategies, develop and parameterize a control function and verify its proper function in a safe environment before applying the approach to the actual robot.

Using the fmi_adapter package, a minimal implementation effort is required to integrate the control function exported as FMU from the simulation environment as node into a ROS control architecture.

The application and test of the path filter on the ActiveShuttle DevKit show that the proposed small enhancement of the motion controller can

1. reduce the risk of entering a locking condition when trying to perform a turn on the spot after stand still by reducing the peak currents,
2. reduce the power consumption of turns on the spot,
3. reduce oscillations, and
4. increase the durability of the hardware components by reducing the jerk.

The example of the path filter impressively highlights how much the performance of a robot can be improved by considering the impact of physical effects (in this case bore friction) in the control software, and how the fmi_adapter enables efficient model-based design workflows for ROS relying on open standards.

With the up-coming eFMI (FMI for Embedded Systems) standard [48], currently developed in the ITEA3 project EMPHYSIS (Sept. 2016–Aug. 2020) [17], this approach will also be applicable to real-time systems and devices with very limited computation power such as microcontrollers or other embedded devices, enabling new workflows for model-based control and diagnosis function development [27].

5.2 Electrical Actuator Simulation Using gazebo-fmi Actuator Plugin

The physics engines used by Gazebo are specialized simulation tools that simulate the dynamics of rigid body systems, i.e., systems composed by rigid bodies (called *links*) interconnected by joints. To simulate joints that are actuated such as the one of robots, these engines take as one of their simulation inputs the force applied on the joint, which is usually called joint *effort*, or joint *torque* for revolute joints. For this reason, any joint position controller that is implemented in Gazebo to drive the joint uses as its output the joint effort, which is then passed to the simulation engine. With respect to real robots, this implies that the simulation is neglecting how the effort is

actually applied on the joint in the real robot, for example via electric or hydraulic motors.

This has two main implications: First, some relevant aspects of the dynamics of the overall system may be missing from the simulation, as noted in [5–7]. Second, unless the robot system implements some sort of joint torque feedback, the typical joint controller running on a real robot does not directly set the effort in the joint but rather computes some value for the actuator-specific input signal such as *voltage* or *current* for electric motors. This means that the gains used and tuned in the Gazebo joint controllers cannot be directly used on the real robot, as the order of magnitude and units may be completely different. For example, for a voltage-controlled electrical actuator mounted on a revolute joint the proportional gain tuned in Gazebo is expressed as $\frac{\mathrm{Nm}}{\mathrm{rad}}$, while the corresponding gain on the real robot is expressed in $\frac{\mathrm{V}}{\mathrm{rad}}$.

Modeling at least some basic aspects of the actuation dynamics permits to reduce, with limited modeling effort, the simulation-to-reality gap and the portability of gains between the simulated robot and the real robot.

5.2.1 Simple Electrical Actuator

For an example of this use-case, we will discuss the simple electrical actuator example shipped with `gazebo-fmi` itself, available at `gazebo-fmi/plugins/actuator/examples/simple_electrical_actuator`.

While for the sake of simplicity this example simulates just a one degrees-of-freedom robot, the order of magnitude of the parameters and the elements of the models have been chosen based on the actuator models used for humanoid robot balancing and walking, in particular on the actuators of the iCub humanoid robot [9, 49].

This directory contains an example Gazebo world that contains two one degrees of freedom manipulators. The models are identical, with the difference that one (called `no_actuator_model`) uses the standard Gazebo actuation model, while the other (called `simple_electrical_actuator_model`) is using a simple model of an electrical actuator that is contained in the `SimpleElectricalActuator.fmu`, which is generated from the `SimpleElectricalActuator.mo` Modelica model.

The Modelica model for the simple electrical actuator is shown in Fig. 19. Its `actuatorInput` is the voltage of the motor, expressed in Volt. Directly after the input, the voltage is limited between -48 and $+48$ V by a limiter. After that, the voltage drives a simple electrical circuit composed by a resistor with a resistance of $1\,\Omega$, and a back electromotive force (EMF) component with a motor gain of $0.1\,\frac{\mathrm{Nm}}{\mathrm{A}}$. The mechanical output of the EMF drives a rotational inertia of $10^{-5}\,\mathrm{kg\,m^2}$, which represents the inertia of the motor's rotor. The inertia then drives a ideal gear box with a reduction ratio of 100, which is then connected to a damper with a damping coefficient of $0.25\,\frac{\mathrm{Nm\,s}}{\mathrm{rad}}$, which is connected to the Gazebo model itself.

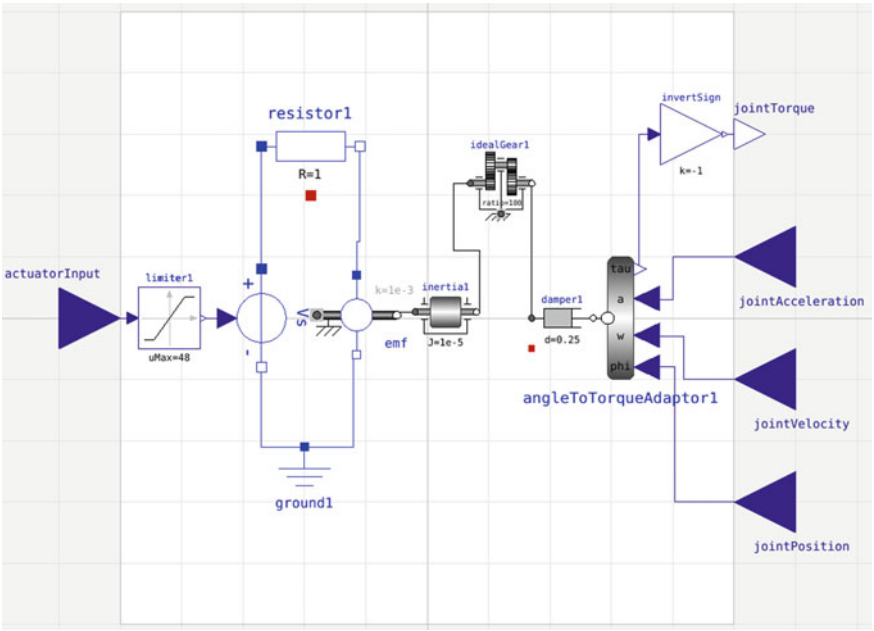


Fig. 19 Modelica model for a simple electrical actuator

5.2.2 Run the Example

As discussed in Sect. 4.2, to run this example, it is not necessary to manually configure the `GAZEBO_PLUGIN_PATH` and `GAZEBO_RESOURCE_PATH` environment variable of Gazebo, but it is sufficient to directly source the `<install_prefix>/share/gazebo-fmi/examples/setup-examples.sh`. The script will set all the environment variables necessary to run the examples shipped with gazebo-fmi:

```
source <install_prefix>/share/gazebo-fmi/examples/setup-examples.
    ↪ sh
```

At this point, you can launch the simulation using the following command:

```
gazebo simple_electrical_actuator.world
```

The two models will start in the 0 position. You can change the target position of the internal Gazebo joint controllers using the `gz joint` command:

```
gz joint -m no_actuator_model -j joint --pos-p 10.0 --pos-d 3.0
    ↪ --pos-t 1.0 && gz joint -m
    ↪ simple_electrical_actuator_model -j joint --pos-p 10.0
    --pos-d 3.0 --pos-t 1.0
```

The `pos-t` option specifies the target of the joint position controller, while `pos-p` and `pos-d` specify the position and derivative gains of the internal Gazebo PID joint controller. Interestingly, due to the use of the low-level electrical actuator

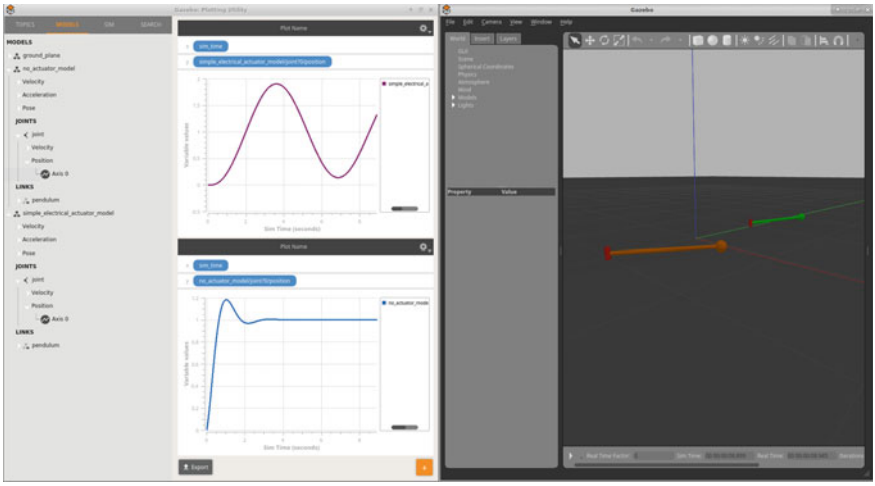


Fig. 20 Plot in Gazebo for the angle of the two controlled joints

model, the gains that are able to stabilize with minimal overshoot and oscillations the `no_actuator_model` are not a good fit instead for the `simple_electrical_actuator_model`, as shown in Fig. 20. The idea is that in this way the Gazebo model, as long as the electrical actuation is similar to the real model, can be used for the tuning (at least preliminary) of the gains of the real joint torque controllers.

To set a different set of gains that instead is able to reduce the oscillations on the `simple_electrical_actuator_model`, execute the following command:

```
gz joint -m simple_electrical_actuator_model -j joint --pos-p
    ↪ 109.0 --pos-d 60.0 --pos-t 1.0
```

6 Final Remarks

In this chapter, two approaches to bridge the gap between classical simulation tools and ROS and Gazebo have been presented, based on the open FMI standard. Two practical use-cases of FMI have illustrated the many benefits of the proposed integrations in terms of enhanced system analysis and simulation capabilities, reusability of models, and increased development efficiency.

In the next years, with the upcoming FMI 3 standard [1] and the development of FMI for Embedded Systems (eFMI) [27], the Functional Mock-up Interface will become even more relevant for robotic systems and software engineering. The fmi_adapter is a first step to allow ROS developers to benefit from these developments. More features are desirable. Similarly, more gazebo-fmi plugins are imaginable, for example to simulate complex surface properties in mobile robotics. Therefore, we cordially invite to contribute to the corresponding repositories.

References

1. Modelica Association: Functional mock-up interface (FMI) website. https://fmi-standard.org/. Accessed 10 Aug 2019
2. C. Gomes, C. Thule, D. Broman, P.G. Larsen, H. Vangheluwe, Co-simulation: a survey. ACM Comput. Surv. **51**(3), 49:1–49:33 (2018)
3. Robert Bosch GmbH: fmi_adapter package for ROS. https://github.com/boschresearch/fmi_adapter. Accessed 10 Aug 2019
4. Robert Bosch GmbH: fmi_adapter package for ROS 2. https://github.com/boschresearch/fmi_adapter_ros2. Accessed 10 Aug 2019
5. M. Neunert, T. Boaventura, J. Buchli, Why off-the-shelf physics simulators fail in evaluating feedback controller performance – a case study for quadrupedal robots, in *Advances in Cooperative Robotics: Proceedings of the 19th International Conference on CLAWAR 2016* (World Scientific, 2016), pp. 464–472
6. J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, M. Hutter, Learning agile and dynamic motor skills for legged robots. Sci. Robot. **4**(26) (2019)
7. J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, V. Vanhoucke, Sim-to-real: learning agile locomotion for quadruped robots, in *Proceedings of Robotics: Science and Systems XIV* (2018)
8. Italian Institute of Technology: gazebo-fmi plugins. https://github.com/robotology/gazebo-fmi/. Accessed 10 Aug 2019
9. G. Metta, L. Natale, F. Nori, G. Sandini, D. Vernon, L. Fadiga, C. von Hofsten, K. Rosander, M. Lopes, J. Santos-Victor, A. Bernardino, L. Montesano, The iCub humanoid robot: an open-systems platform for research in cognitive development. Neural Netw. **23**(8), 1125–1134 (2010)
10. ITEA: Modelisar impact story. https://itea3.org/project/impact-stream/modelisar-impact-story.html. Accessed 21 Apr 2019
11. MathWorks, Inc.: What is an S-function? https://de.mathworks.com/help/simulink/sfg/what-is-an-s-function.html. Accessed 29 Apr 2019
12. T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, S. Wolf, The functional mockup interface for tool independent exchange of simulation models, in *Proceedings of the 8th International Modelica Conference, Dresden, Germany* (Linköping University Electronic Press, Linköpings universitet, 2011)
13. Modelica Association: Functional mock-up interface for model exchange and co-simulation (Standard) (2014). https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf. Accessed 29 Apr 2019
14. T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, A. Viel, Functional mockup Interface 2.0: the standard for tool independent exchange of simulation models, in *Proceedings of the 9th International Modelica Conference, Munich, Germany* (Linköping University Electronic Press, Linköpings universitet, 2012)
15. Modelica Association: Modelica website. https://www.modelica.org/. Accessed 10 Aug 2019
16. C. Bertsch, E. Ahle, U. Schulmeister, The functional mockup interface – seen from an industrial perspective, in *Proceedings of the 10th International Modelica Conference, Lund, Sweden* (2014), pp. 27–33
17. ITEA: 15016 EMPHYSIS embedded systems with physical models in the production code software. https://itea3.org/project/emphysis.html. Accessed 25 May 2019
18. PKWARE Inc.: ZIP specification. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT. Accessed 21 Apr 2019
19. Modelon AB: FMI compliance checker (FMUChecker). https://github.com/modelica-tools/FMUComplianceChecker. Accessed 10 Aug 2019
20. Dassault Systèmes: FMPy. https://github.com/CATIA-Systems/FMPy. Accessed 10 Aug 2019
21. Modelica Association: FMI cross check rules. https://github.com/modelica/fmi-cross-check/. Accessed 26 May 2019

22. Dassault Systèmes: Test FMUs. https://github.com/CATIA-Systems/Test-FMUs. Accessed 10 Aug 2019
23. C. Bertsch, A. Mukbil, A. Junghanns, Improving interoperability of FMI-supporting tools with reference FMUs, in *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic* (2017), pp. 533–540
24. J. Bastian, C. Clauss, S. Wolf, P. Schneider, Master for co-simulation using FMI, in *Proceedings of the 8th International Modelica Conference, Dresden, Germany* (2011), pp. 115–120
25. M. Benedikt, D. Watzenig, J. Zehetner, A. Hofer, EPCE – a nearly energy-preserving coupling element for weak-coupled problems and co-simulation, in *Proceedings of the International Conference on Computational Methods for Coupled Problems in Science and Engineering* (2013), pp. 1–12
26. C. Bertsch, J. Neudorfer, E.A.S.S. Arumugham, K. Ramachandran, A. Thuy, FMI for physical models on automotive embedded targets, in *Proceedings of the 11th International Modelica Conference, Versailles, France* (2015), pp. 43–50
27. O. Lenord, F. Jarmolowitz, W. Kiesenhofer, K. Rath, T. Obertopp, Towards an integrated tool chain from physical models to diagnosis functions, in *Proceedings of the 12th MODPROD Workshop 2018 on Cyber-Physical Systems of Systems, Linköping, Sweden* (2018)
28. JModelica.org: FMI library. https://jmodelica.org/. Accessed 10 Apr 2019
29. SFI Offshore Mechatronics Research Centre: FMI4cpp. https://github.com/NTNU-IHB/FMI4cpp/. Accessed 10 Apr 2019
30. Open Source Robotics Foundation: Catkin command line tools. https://catkin-tools.readthedocs.io/. Accessed 26 Apr 2019
31. Kitware, Inc. and Contributors: CMake documentation – modules: ExternalProject. https://cmake.org/cmake/help/latest/module/ExternalProject.html. Accessed 26 Apr 2019
32. D. Thomas, Colcon – collective construction. https://colcon.readthedocs.io/. Accessed 26 Apr 2019
33. Open Source Modelica Consortium: OpenModelica. https://openmodelica.org/download/download-linux. Accessed 10 Aug 2019
34. QTronic: FMU SDK. https://www.qtronic.de/en/fmu-sdk/. Accessed 10 Aug 2019
35. Open Robotics: Management of nodes with managed lifecycles. https://index.ros.org/doc/ros2/Tutorials/Managed-Nodes/. Accessed 28 May 2019
36. N. Koenig, 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)*, vol. 3 (2004), pp. 2149–2154
37. M.M.M. Manhães, S.A. Scherer, M. Voss, L.R. Douat, T. Rauschenbach, UUV simulator: a Gazebo-based package for underwater intervention and multi-robot simulation, in *OCEANS 2016 MTS/IEEE Monterey* (2016)
38. J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, O. von Stryk, Comprehensive simulation of quadrotor UAVs using ROS and Gazebo, in *Simulation, Modeling, and Programming for Autonomous Robots*, ed. by I. Noda, N. Ando, D. Brugali, J.J. Kuffner (Springer, Berlin, 2012), pp. 400–411
39. D. Thomas, Contributors: a universal build tool. https://design.ros2.org/articles/build_tool.html. Accessed 27 Apr 2019
40. Open Source Robotics Foundation: Install Gazebo using Ubuntu packages. http://gazebosim.org/tutorials?tut=install_ubuntu. Accessed 27 Apr 2019
41. Kitware, Inc. and Contributors: CMake documentation – modules: FetchContent. https://cmake.org/cmake/help/v3.14/module/FetchContent.html. Accessed 27 Apr 2019
42. Open Source Robotics Foundation: Gazebo tutorials. http://gazebosim.org/tutorials. Accessed 10 Aug 2019
43. M.A. Sherman, A. Seth, S.L. Delp, Simbody: multibody dynamics for biomedical research. Procedia IUTAM **2**, 241–261 (2011)
44. Bosch Rexroth: ActiveShuttle – setting your intralogistics in motion. https://www.boschrexroth.com/en/xc/products/product-groups/assembly-technology/topics/intralogistics/template-neuprodukt-seite-6. Accessed 30 May 2019

45. D. Zimmer, PlanarMechanics library 1.4.0 (2017-01-12). https://github.com/dzimmer/PlanarMechanics. Accessed 31 May 2019
46. D. Zimmer, M. Otter, Real-time models for wheels and tyres in an object-oriented modelling framework. Veh. Syst. Dyn. (2010)
47. N. Schröder, O. Lenord, R. Lange, Enhanced motion control of a self-driving vehicle using Modelica, FMI and ROS, in *Proceedings of the 13th International Modelica Conference, Regensburg, Germany* (Linköping University Electronic Press, Linköpings universitet, 2019)
48. EMPHYSIS Project: EMPHYSIS – functional mock-up interface for embedded systems. https://emphysis.github.io/. Accessed 25 May 2019
49. G. Nava, D. Pucci, F. Nori, Momentum control of humanoid robots with series elastic actuators, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 2185–2191

Ralph Lange joined Bosch Corporate Research in 2013. He is currently leading an internal research project on methods, tools, and frameworks for systems and software engineering in robotics. In this context, he is also principal investigator in the EU project OFERA, which brings ROS 2 on microcontrollers in an open-source initiative named micro-ROS. Ralph actively follows the development of ROS 2. Amongst others, he researches execution management and run-time reconfiguration concepts for ROS 2. Before joining Bosch, Ralph worked as a software engineer for TRUMPF Machine Tools on a graphical programming system for punching and punch-laser machines. He obtained his Ph.D. in the field of scalable context-aware systems in 2010 at the Institute for Parallel and Distributed Systems of the University of Stuttgart, Germany.

Silvio Traversaro joined the Italian Institute of Technology (IIT) as a Postdoctoral Researcher in 2017. As part of the iCub Tech facility at IIT, he is currently working on industrial and commercial applications of robotics research in several sectors, including automation for the metal processing industry. Furthermore, he collaborates with the Dynamic Interaction Control research line at IIT, working on software architectures and estimation techniques for legged walking, aerial humanoid robotics, and reinforcement learning. He obtained his Ph.D. in Robotics from the University of Genoa, working at the Italian Institute of Technology on dynamics modeling, estimation, and identification applied to the iCub humanoid robot.

Oliver Lenord joined Bosch Corporate Research in 2016. He is currently leading the European publicly funded ITEA3 project EMPHYSIS aiming to develop the eFMI (FMI for Embedded Systems) standard. He is founding member and previous chairman of the OSMC (Open Source Modelica Consortium). He is also supporting the systems and software engineering research project in robotics in terms of model-based design methods. From 2011 until 2016, he worked as product manager of the Mechatronics Concept Designer at Siemens PLM in Cypress, California, USA. After his Ph.D. in the field of robotics at the Institute of Mechatronics at the University of Duisburg-Essen, Germany, his first position in the industry was at Bosch Rexroth AG, where he was leading a simulation and engineering software development group for hydraulic applications and industry automation.

Christian Bertsch joined Robert Bosch GmbH in 2001. Within Bosch Corporate Research, he is currently leading a research project on advanced model-based functions and their numerical realization on embedded systems. There he also investigated the extension of FMI towards embedded systems—as will be standardized in the ITEA3 EMPHYSIS project. He is a personal member of the Modelica Association, represents Bosch in the FMI Steering Committee, and actively participates in the further development of the FMI Standard. Within Bosch, he coordinates the FMI activities both internally with training and consulting, and in collaboration with tool vendors. Before joining Bosch, he graduated in Mathematics and Physics with a focus on numerical mathematics at the University of Heidelberg, Germany.

An ARVA Sensor Simulator

Jonathan Cacace, Nicola Mimmo, and Lorenzo Marconi

Abstract Robot Operating System (ROS) framework supports different kind of sensors, typically used to develop robotic applications, such as vision and depth sensors, laser scanners and so on. In this chapter, we present a new ROS package designed to simulate ARVA transceiver sensors: `arva_sim`. ARVA is a French acronym which stands for *Appareil de Recherche de Victims en Avalanche* and represents the forefront technology adopted in Search & Rescue operations to localize victims of avalanches buried under the snow. In order to simulate its behavior, this package provides two Gazebo plugins: the transmitter and receiver. The aim of this chapter is to describe the mathematical and theoretical background of the transceiver, discussing its implementation and integration with ROS. To demonstrate the accuracy of the proposed sensor model, we present a simulation scenario in which an Unmanned Aerial Vehicle (UAV) equipped with the transceiver sensor performs a basic S&R pattern using the output of ARVA system. It is worth nothing that the proposed ROS package, `arva_sim`, represents the first simulation model of an ARVA transceiver system and can be useful for the developer to design, test and benchmark faster and smarter search strategies to speed up rescue missions in case of avalanches.

Keywords ARVA · Search & Rescue · Gazebo ROS · Unmanned aerial vehicle

J. Cacace
Department of Electrical Engineering and Information Technology, University of Naples "Federico II", Via Claudio, 21, 80125 Naples (NA), Italy
e-mail: jonathan.cacace@unina.it

N. Mimmo (✉) · L. Marconi
Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi", University of Bologna, Via Risorgimento, 2, 40136 Bologna (BO), Italy
e-mail: nicola.mimmo2@unibo.it

L. Marconi
e-mail: lorenzo.marconi@unibo.it

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_8

1 Introduction

In the future, Search & Rescue (S&R) missions will benefit from the use of fast and smart mobile robots. Indeed, the effectiveness of robots is twofold: from one hand, they can support rescuers to improve the efficiency of search operations and, on the other hand, they can even substitute humans thus decreasing the risks associated to operations in harsh environments (fires, contaminated sites, ...). To this end, a variety of ground [1–3] and aerial [4, 5] robotic platforms, characterized by different capabilities and equipment, have been developed to support S&R operations in different scenarios such as urban disasters, mining accidents, explosions in radiation sites, etc.

A further important challenge in S&R robotics is the one addressed by the European projects *SHERPA* [6] and *AIRBorne* [7] which aim to provide a feasible and effective robotic solution to support alpine S&R teams in rescuing avalanche victims. The avalanche S&R scenarios are particularly challenging due to the tight time constraint and the harshness of the environment. Indeed, the rescue time in avalanche accidents is crucial because the survival probabilities of people buried under the avalanches decrease rapidly over the burial time. On the other hand, the search operations are performed by teams of rescuers scanning the avalanche site by walking on the irregular avalanche surface, with the risk of inducing a further avalanche. Unfortunately, the peculiarities of this S&R domain make practically impossible the usage of common technologies like phones and ultra-wide band localization systems. For this reason a special system, called ARVA, acronyms of the French "Appareil de Recherche de Victimes", is currently used and it represents one of the forefront technologies nowadays used in S&R operations in case avalanches [8].

The ARVA devices, depicted in Fig. 1a, consist of two main elements, i.e. a *transmitter* and a *receiver*, which are operated alternatively. Excursionists and skiers who use the ARVA normally set it in the transmitting mode so that, in the accidental case of avalanche, the system is already in the right operative mode. On the other hand, rescuers operate their ARVA devices in the receiving mode to be able to detect the transmitter' signal. In the receiver mode, the ARVA devices provide information about the electromagnetic field, emitted by the transmitter, which is exploited to

(a) ARVA transceiver. (b) Human rescuer

Fig. 1 The ARVA transceiver (left). This device can operate both as a transmitter and receiver. The working mode is selected by using a physical switch button. The right Figure depicts a human rescuer involved in a S&R mission using ARVA transceiver

guide the rescuer toward the victim. The working principle of this sensor is based on the detection of the magnetic low-power pulses, emitted by the transmitter. This information is usually made available to the rescuers in terms of magnetic distance and magnetic direction to the victim.

To allow for multiple victims search, each ARVA device in the receiver mode is able to detect more than one transmitter simultaneously (the most common devices can detect up to four different signals).

Alpine rescuers, exploiting the ARVA technology for the search operations, adopt a three-step strategy consisting in the scan of the avalanche area for the detection of the first valid signal, the approach to the transmitter location, and the so called *pin-pointing*. The three-step search technique is detailed in the manuals of the ARVA devices [9, 10]. In the first step, the rescue team navigates the avalanche site by looking for a valid ARVA signal, with a formation and a strategy that depend on several aspects such as the number of rescuers, the kind of avalanche and the number of the missing people. Later, after having detected a valid ARVA signal, rescuers start following the electromagnetic flux lines emitted by the ARVA transmitter, and visualized on the screen of the ARVA receivers (see Fig. 1b). Finally, when the rescuers are sufficiently close to the transmitter, they slowly move the receiver in order to precisely localize the victim buried under the snow by minimizing the magnetic distance provided by the ARVA device. This latter phase is called *pin-pointing*.

Despite the huge amount of sensors supported by ROS simulation tools [11], a method to simulate the ARVA transceiver has not yet been developed. Then, the aim of this work is twofold and consists in presenting the `arva_sim`, a ROS package implementing the ARVA system, and in interfacing it with the Gazebo simulator [12]. Installing the `arva_sim` package, a new couple of plugins will be available in the Gazebo simulations. The main goal of the `arva_sim` package is to allow researchers and developers to equip simulated robots with ARVA devices, in order to test and benchmark efficient and autonomous search strategies.

Moreover, Unmanned Aerial Vehicles (UAVs) are extensively employed in S&R operations in several domains [13–15], with a particular focus on the alpine scenarios, [16–19] demonstrate how S&R operations can benefit from the use of autonomous aerial robots to survey the environment and collect evidences about the position of a missing person.

For this reason, this work also proposes a S&R test case in agreement with the guidelines of the European project *AIRBorne*. In detail, a simulated UAV equipped with the proposed ARVA sensor is used to autonomously perform the standard human made search operations.

This chapter is organized as follows: Sect. 2 provides background information about the mathematical model of the ARVA system. In Sect. 3, the description of the software architecture of the `arva_sim` package is presented by discussing its configuration, inputs and outputs. In Sect. 4 the `arva_sim` is tested in a simulated S&R mission in which an autonomous UAV exploits the ARVA system. Finally, Sect. 5 concludes this work with some remarks on possible future developments.

2 Background

This section presents the theoretical background of the ARVA system. In particular, after the introduction of the notation, the mathematical models of both the *receiver* and *transmitter* operative modes are described. The proposed models are also comprehensive of electromagnetic interference and distortions due to the environment. In addition, this section reports a brief overview about some analytical tools needed to properly understand the ARVA mathematical model. Finally, the working principle of commercial ARVA devices is detailed.

2.1 Notation

The set of natural numbers is identified by the symbol $\mathbb{N}$, the set of real numbers is indicated by the letter $\mathbb{R}$ whereas, the symbol $\mathbb{Z}$ represents the set of integer numbers. Given a vector space $\mathbb{R}^n$ this paper indicates subspaces and subsets with calligraphic letters, e.g. $\mathcal{X} \subseteq \mathbb{R}^n$, whereas a vector belonging to $\mathcal{X}$ is indicated by a lowercase letter, i.e. $x \in \mathcal{X}$. Furthermore, matrices are indicated by means of capital letters, with $X \in \mathbb{R}^{n \times m}$ representing a rectangular real matrix of n rows by m columns.

Three reference systems are used to derive the mathematical model. The inertial reference frame is indicated by $\mathcal{F}_i(O_i, x_i, y_i, z_i)$, which is centered at O_i, with the z_i axis aligned with the gravity and opposite in direction, the x_i and y_i axes are oriented toward the geographic East and North respectively. The second reference frame is attached to the ARVA transmitter and it is denoted by $\mathcal{F}_t(O_t, x_t, y_t, z_t)$, with x_t, y_t and z_t aligned with the three antennae constituting the device. Finally, the third reference frame, $\mathcal{F}_r(O_r, x_r, y_r, z_r)$, centered at O_r is attached to the receiver with the three axes oriented as the three antennae of the ARVA device.

The positions of the transmitter and the receiver are indicated by ${}^{\#}p_t$ and ${}^{\#}p_r$ with the superscript $\# \in \{i, r, t\}$ indicating the reference frame of description.

The relative rotation from $\mathcal{F}_r$ to $\mathcal{F}_i$ is indicated by the matrix ${}^{i}R_r \in SO(3)$ whereas the matrix ${}^{i}R_t \in SO(3)$ rotates a vector described in $\mathcal{F}_t$ to $\mathcal{F}_i$. The $SO(3)$ generator is the matrix $S(\cdot)$ which indicates a skew symmetric matrix.

Finally, a subset $\mathbf{x} \subset \{x, y, z\}$ of coordinates, with cardinality indicated by $n = |\mathbf{x}|$, is defined. Then, the operator which projects the vectors in $\mathbb{R}^3$ onto the subspace of coordinates $\mathbf{x}$ is indicated by $L_{\mathbf{x}} : \mathbb{R}^3 \mapsto \mathbb{R}^n$.

2.2 Analytical Tools

The contents of this Section are based on theoretical tools which ranges from the mathematics to the physics. In detail, the reader who is not familiar with matrices and vectors, vector spaces, norms and linear transformation can read any book on the

topic *linear algebra*. Two remarkable examples of these book are represented by [20, 21]. On the other hand, the description of the ARVA system is based on knowledge on *electromagnetic fields* which are well described in several books and, as example, in [22]. Finally, the relative position, rotation and dynamics of the receiver with respect to the transmitter is described by the *classical mechanics*. One interesting reading on this subject is represented by the book [23].

2.3 The ARVA System

The ARVA transceivers commercially available have two working modalities, namely they can work as receivers or as transmitters, with a manual switch used to commute between the two modes. The sensor has a very compact case that make it easily transportable and wearable. Before starting their activities, experienced skiers switch the worn sensor in the transmitter mode. In case of accident, companions not buried by the avalanche, or rescuers who reach the disaster area, switch their device in the receiver mode and start searching the victim by following the flux lines of the transmitter. This paragraph goes through the main physical principles of the sensor and its main features that are instrumental to the development of the automatic search algorithm presented in Sect. 4.

2.3.1 ARVA in the Transmitter Mode

The ARVA system relies on a transmitter device that generates a pulsed magnetic field of 457 kHz. The generated magnetic dipole, indicated by the vector m (see Fig. 2), is aligned with the x_t axis of $\mathcal{F}_t$ with an amplitude that is approximately constant and only affected by the charging state of the battery device. The electromagnetic vector field generated by the dipole is indicated by h. Its strength is clearly affected by the

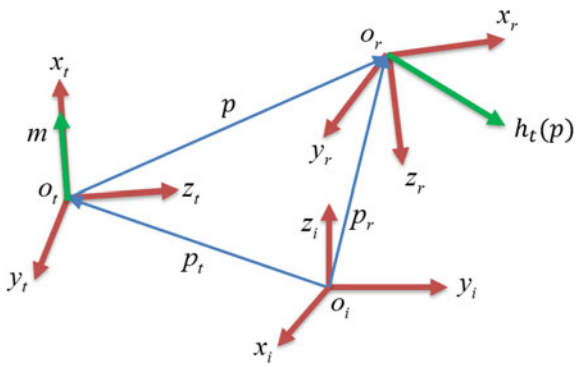


Fig. 2 Reference frames definition

under-snow depth of the transmitter and by the kind of snow (iced, soft, …). Denoting by (x, y, z) the components of the vector ${}^t p_r$ expressed in $\mathcal{F}_t$ (i.e. ${}^t p_r = \mathrm{col}(x, y, z)$), it turns out that a mathematical model of the magnetic vector field is given by (see [24])

$$ {}^t h({}^t p_r) = \frac{\|m\|}{4\pi \| {}^t p_r \|^5} \mathrm{A}({}^t p_r) \tag{1} $$

where

$$ \mathrm{A}({}^t p_r) := \begin{bmatrix} 2x^2 - y^2 - z^2 \\ 3xy \\ 3xz \end{bmatrix} $$

Moreover, the norm of the magnetic field ${}^t h$ is given by:

$$ \| {}^t h \| = \frac{\|m\|}{4\pi \| {}^t p_r \|^3} \sqrt{1 + 3 \frac{x^2}{\| {}^t p_r \|^2}} \, . \tag{2} $$

The flux lines described by (1) are depicted in Fig. 3 (red lines) whereas the equation (2) has been exploited to depict the iso–power lines, superposed to the flux lines in Fig. 3 (blue lines). As can be see in Fig. 3 both the flux lines and the

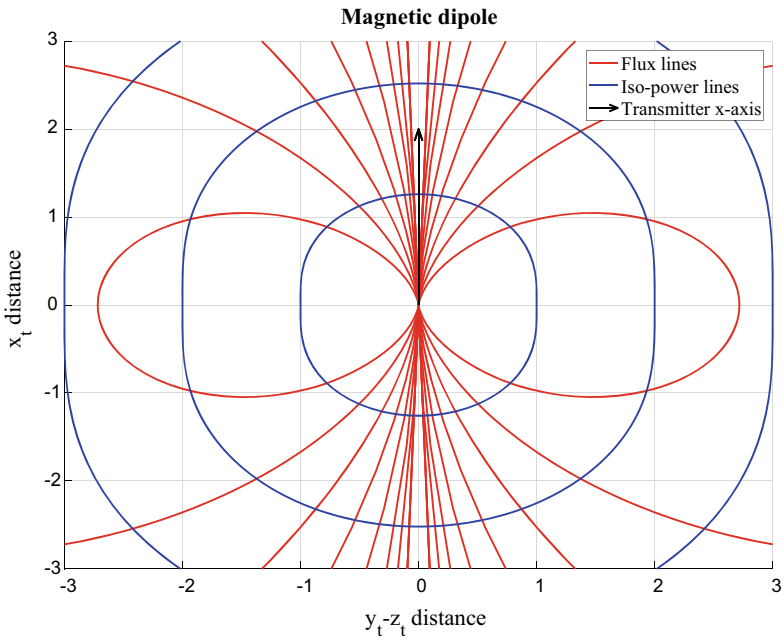


Fig. 3 Flux lines and iso-power lines for a theoretical magnetic dipole

iso–power curves are symmetric with respect to the axis x_t and, moreover, each flux line intersects several iso-power lines and vice-versa. Finally, the nominal electromagnetic field ${}^{i}h(\cdot)_0$, expressed in the inertial coordinate frame, is obtained by a roto-translation of the field ${}^{t}h(\cdot)$ by the matrix ${}^{i}R_t$:

$$ {}^{i}h_0 = {}^{i}R_t \, {}^{t}h\left({}^{t}R_i({}^{i}p_r - {}^{i}p_t)\right). \tag{3} $$

2.3.2 ARVA in the Receiver Mode

The ARVA device, in the receiver mode, can be modeled as an electromagnetic sensor with three antennas aligned along the axes x_r, y_r and z_r. The magnetic field read by the receiver, ${}^{r}h$, is given by the projection of the inertial vector ${}^{i}h_0$ onto the $\mathcal{F}_r$ frame:

$$ {}^{r}h := {}^{r}R_i \, {}^{i}h_0 + Q \, {}^{r}R_i \, {}^{r}w = {}^{r}R_i \frac{\kappa \|m\|}{4\pi \| {}^{t}p_r \|^5} \, {}^{i}R_t \, A({}^{t}p_r) + Q \, {}^{r}R_i \, {}^{i}w \tag{4} $$

where ${}^{i}w$ indicates the noise due to ElectroMagnetic Interference (EMI) expressed in the inertial frame whereas the matrix $Q = \mathrm{diag}(q_1, q_2, q_3)$, $q_i \in (0, 1]$ for $i = 1, 2, 3$, indicates the noise sensitivity of the ith antenna.

2.4 ElectroMagnetic Interference

The ARVA systems is based on an electromagnetic technology thus, by its nature, it is subject to electromagnetic interference (EMI). Even if avalanches usually happen in sites which are far from sources of EMI, the environmental noise is not negligible in the most of operative scenarios. In this context, the effect of EMI can be modeled as a superposition of a disturbance signal ${}^{i}w$, described in the inertial reference frame, on the nominal one (see Eq. (4)). The effect of this EMI on the ARVA receiver signal varies in function of the relative attitude and distance between the transmitter and the receiver.

The structure of the ARVA antennae deeply impacts on the quality of the sensed signal. A field test campaign was performed by measuring the relative position of the receiver with respect to the transmitter (by means of GPS receivers), their relative attitude (by means of Inertial Measurement Units) and the ARVA data. The data were collected in different locations to mediate the effects of the environment electromagnetic noise. The outcome of this test campaign is that, if the distance between the receiver and the transmitter is larger that approximatively 60 m, the ARVA outputs are identically equal to -1. This result is reported in Fig. 4. A reverse engineering process allowed for the determination of the noise floor which is described by the Eqs. 5 and 6. The maximum environment induced EMI are estimated to be equivalent to an ARVA signal emitted by a transmitter approximately located 80 m far from

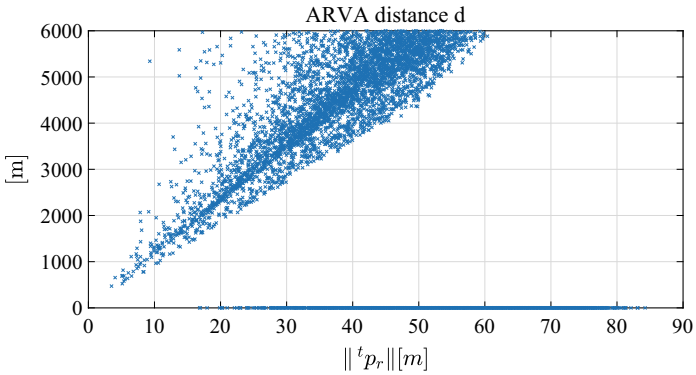


Fig. 4 The distance d provided by commercial ARVAs: the points represent the data d, determined in presence of the environment noise, over the actual distance $\| {}^t p_r \|$

the receiver. Starting from these data an equivalent noise intensity is determined by imposing:

$$\max \| {}^r h \| = \max \| Q \, {}^r \| \| {}^i w \| = \frac{\kappa \| m \|}{2\pi \| {}^t p_{r_{en}} \|^3} \tag{5}$$

with $\| {}^t p_{r_{en}} \| = 80$. The norm $\| {}^i w \|$ solving the equality of Eq. (5), namely $\| {}^i w_{en} \|$, is obtained by the following expression:

$$\| {}^i w_{en} \| = \frac{\kappa \| m \|}{2\pi \max \| Q \, {}^r \| \| {}^t p_{r_{en}} \|^3}. \tag{6}$$

Assuming an EM noise ${}^i w$ described by a randomly oriented vector with constant modulus ($\| {}^i w_{en} \|$), Fig. 5 depicts the effect of the environment EMI on the ARVA output when in presence of one active ARVA transmitter. In particular, the data in Fig. 5 have been obtained by increasing the norm of the relative position between the receiver and the transmitter. The signal to noise ratio has been normalized in the set $[-1, \ 1]$ and is described by

$$\text{SNR} = \frac{\| {}^r h_0 \| - \| {}^r h_{en} \|}{\max(\| {}^r h_0 \|, \| {}^r h_{en} \|)} \tag{7}$$

in which ${}^r h_0 = {}^r R_i \, {}^i h_0$ and ${}^r h_{en} = Q \, {}^r R_i \, {}^i w_{en}$. The Eq. (7) has been introduces to provide an insight of the effect of the noise on the nominal signal. The electromagnetic field h is function of the distance and tends to infinite when the distance between the receiver and the transmitter tends to zero and vice-versa. Thus, in order to provide a representation based on finite values of the effects of the noise on the nominal signal, the authors selected the normalized relative error of Eq. (7) which leads to the Fig. 5. The ratio of Eq. (7) tends to 1 when the environmental noise is negligible with respect to the strength of the ARVA signal (i.e. for $\| {}^t p_r \| \to 0$) and, at the opposite, goes

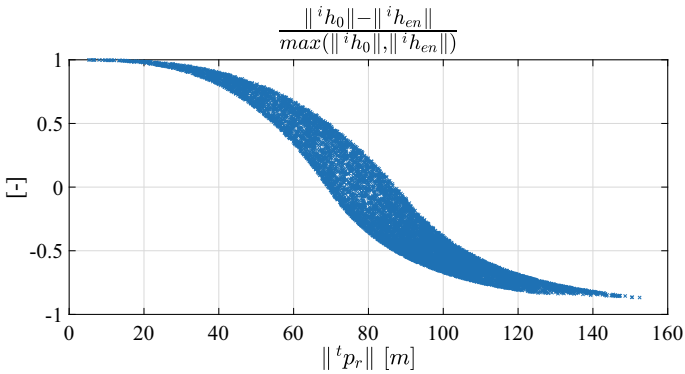


Fig. 5 The effects of electromagnetic interference on the ARVA reading increase as the distance $\|{}^t p_r\|$ increases

to -1 when the electromagnetic field emitted by the transmitter is negligible with respect to the environmental noise (i.e. for $\|{}^t p_r\| \to \infty$).

2.5 Commercial Devices

The factor shape of the ARVA devices available from the market, characterized by a parallelepiped shape specifically studied to improve the transportability and wearability, imposes that the three antennas are necessarily characterized by different ferromagnetic core geometries with, then, different efficiency in reading the magnetic field intensity. In particular, the antenna placed along x_r has the biggest ferromagnetic core and acts as primary antenna with the best Signal to Noise Ratio (SNR). It can be efficiently used also far from the transmitter, where the magnetic field intensity becomes small with respect to the environment noises. The remaining two antennas, placed along y_r and z_r, have decreasing efficiency and act as secondary and thirdly antennas. They can be exploited in processing the output when the magnetic field is sufficiently strong, i.e when the receiver is sufficiently close to the transmitter, according to a strategy explained in the following. The commercial ARVA receivers yield two outputs that are denoted by d and δ, which respectively indicate the estimated distance to the transmitter (namely a proxy of $\|{}^r p_t\|$), and the orientation of ${}^r h$ on the plane $x_r - y_r$. The sensor outputs, of course, are strongly affected by the distance to the victim and by the fact that not all the three components of the magnetic field are measured with the same efficiency (due to the different SNRs). In this respect, the ARVA receivers implement a logic which exploits the measured signal ${}^r h$ to cluster the search area in three zones, which are indicative of the distance to the victim ("very far", "close", "very close"), and to compute the outputs $(d, \delta) \in \mathbb{Z}^2$ according to the actual zone. More specifically:

– *outer zone*: characterized by a SNR under a minimum threshold, indicative of a large distance to the victim (typically $\|^t p_r\| > 60$ m according to the data-sheet of the manufacturer). In this zone the signal ${}^r h$ is too weak to yield a reliable information and the receiver output is set to $(d, \delta) = (-1, -1)$;
– *mid zone*: if the projection of the measured signal on the receiver $x_r - y_r$ axes, namely $L^r_{xy} h$, is larger than a certain threshold reliable data d and δ can be computed. More specifically, the distance d is computed by inverting relation (2) considering only the first two components of ${}^r h$ as relevant signal:

$$d = \left(\frac{\kappa_{eq}}{4\pi \| L_{xy}\ {}^r h \|} \right)^{\frac{1}{3}}, \tag{8}$$

where κ_{eq} is constant suitably tuned by the manufacturers and L_{xy} represents the projector operator on the $x_r - y_r$ axes. The distance d provided in this phase, being determined exploiting only the most sensitive antennae (over the three available), is a function of both the receiver and transmitter orientation, ${}^i R_r$ and ${}^i R_t$, and the actual relative distance, ${}^t p_r$, i.e. $d = d\left({}^i\mathrm{R}_r, {}^i\mathrm{R}_t, {}^t p_r\right)$. When in this operating mode the distance $d \in [300, 5000]$ cm. On the other hand, the angle δ is determined by implementing the following equation:

$$\delta = \tan^{-1} \left(\frac{L_y\ {}^r h}{L_x\ {}^r h} \right), \tag{9}$$

where $L_x\ {}^r h$ and $L_y\ {}^r h$ indicate the projection of ${}^r h$ on the x_r and y_r respectively. Finally, the angle $\delta \in (-90, 90)$ deg. It is worth nothing that, if the receiver orientation ${}^i R_r$ is such that $L_{xy}\ {}^r h = 0$, the output is $(d, \delta) = (-1, -1)$;
– *inner zone*: in this zone the strength of the signal ${}^r h$ is sufficiently high to allow the exploitation of all the three antennas. In the last search phase, the receiver is sufficiently close to the transmitter to exploit the third antenna to estimate, besides the first two, the distance to the victim. In this phase the distance d is not more a function of receiver and transmitter relative orientation, i.e. $d = d\left({}^t p_r\right)$ while the angle δ is not more provided and set to zero. Finally, in the third search phase $d \in [0, 300)$ cm and $\delta = 0$ deg where

$$d = \left(\frac{\kappa_{eq}}{4\pi \| {}^r h \|} \right)^{\frac{1}{3}}. \tag{10}$$

Finally, the effect of the environment noise on the data d provided by the commercial ARVA devices is shown in Fig. 4. In detail, for distances $\| {}^t p_r \| > 60$ m, the strength of the electromagnetic field h_0 is comparable or below the noise power so that the sensed field ${}^r h$ is practically representative of the noise. Moreover, it is worth nothing that, since the clusters are based on the projection $L_{xy}\ {}^r h$ and thus of the receiver orientation, the distance d could be equal to -1 even if the distance $\| {}^t p_r \| \leq 60$ m.

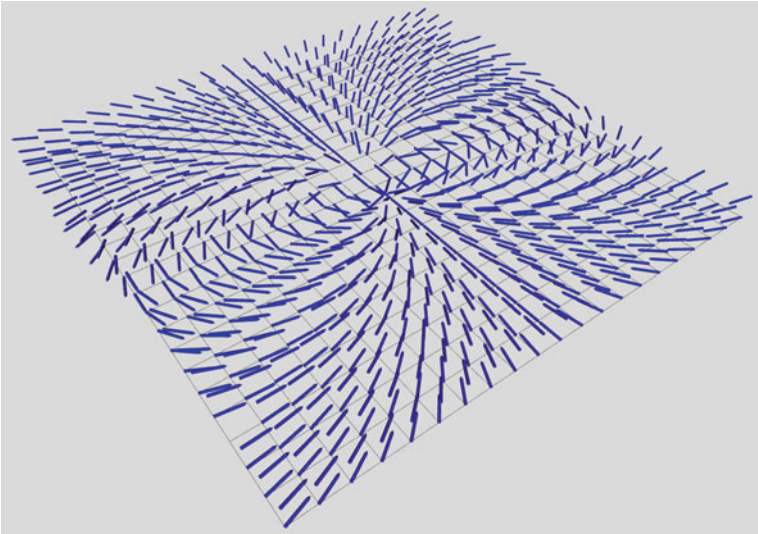


Fig. 6 Direction of the magnetic field detected by the ARVA receiver sensor for each position of the workspace, displayed in the RViz

3 *arva_sim* ROS Package

This Section presents the implementation of the ARVA plugin and demonstrates how to enable it into a Gazebo simulated world. The main scope of the proposed plugins is to replicate the behavior of the ARVA transceiver as depicted in Fig. 3. In particular, the proposed package contains two different plugins: the receiver and transmitter. Considering a specific position for one or more transmitters, the receiver calculates the direction of the flux of the magnetic field. This behavior is represented in Fig. 6, in which an ARVA transmitter is placed in the origin of the simulation scene while ROS visualization tool (RViz) [25] is used to show the direction of the flux for each position of the workspace.

This Section is organized as follows. Initial steps to install and configure the ROS environment from scratch are reported in Sect. 3.1. A first test demonstrating the working principle of the `arva_sim` package is discussed in Sect. 3.2, while in Sect. 3.3 the implementation of the proposed package is detailed.

3.1 Installation Instructions

The first step to take before using the ARVA plugins is to install and configure the ROS environment. The `arva_sim` package supports the latest Long-Term Support (LTS) distributions of ROS, namely *Kinetic Kame* and *Melodic Morenia*. In this

context, the main difference between these ROS distributions concerns the version of Gazebo simulator installed by default. Moreover, from the selected ROS distribution depends the version of Operating System as well. For ROS *Kinetic* is recommended to use Ubuntu 16.04, while Ubuntu 18.04 is recommended for ROS *Melodic*. To use `arva_sim` package the two system setups are equivalent, so the reader can choose the preferred one.

Assuming that an Ubuntu Linux OS is run, the installation steps are listed below:

1. Install ROS: in order to install ROS follow the instruction reported in the official wiki. To install ROS *Kinetic* follow the instruction reported at this link: http://wiki.ros.org/kinetic/Installation/Ubuntu while, to install ROS *Melodic*, follow the instruction listed in this page: http://wiki.ros.org/melodic/Installation/Ubuntu. The full installation of ROS is the only software tool needed to use the ARVA sensor simulator. In fact, it works with the standard ROS libraries and no other external tools or libraries are needed to compile the `arva_sim` package.
2. Setup the ROS workspace: to setup a ROS workspace, create an empty directory in a desired location. To create the workspace *catkin_ws* in the home folder use the following commands:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

3. Automatically load ROS environment: to do this, edit the bash configuration file located in the home folder (i.e. `/.bashrc`): Open the file with the preferred text editor (for example *gedit*):

```
$ gedit ~/.bashrc
```

Add the following lines to the bash file.

```
source ~/catkin_ws/devel/setup.bash
export ROS_WORKSPACE=~/catkin_ws
```

4. Now everything is ready to download and compile the `arva_sim` package. `arva_sim` can be found at the following link: https://github.com/jocacace/arva_sim. In order to build a ROS package, it must be placed in the `src` folder of the ROS workspace. For this reason, in this setup `arva_sim` must be downloaded in the `/catkin_ws/src` folder.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/jocacace/arva_sim
```

5. Select the desired branch: `arva_sim` repository has two main branches, one for each supported version of ROS. The default branch supports Gazebo 9 (and consequently ROS *Melodic*). To use `arva_sim` with ROS *Kinetic* type the following commands:

```
$ roscd arva_sim
$ git checkout gazebo7-devel
```

6. Finally, compile the package:

```
$ cd ~/catkin_ws
$ catkin_make
```

After compilation completed successfully, start to test the `arva_sim` package.

3.2 Testing the ARVA Gazebo Plugins

The ARVA system can be tested using the example launch files contained in the `arva_sim` package. In particular, the aim of this test is to setup a simulation scene with different transmitters and one ARVA receiver. For this reason, a set of convenient files has been prepared to initialize the ARVA system showing the output of the receiver sensor using RViz. The technical details concerning this test will be discussed more deeply in the next section.

Use the following command to start the Gazebo simulator on an empty simulation scene[1]:

```
$ roslaunch gazebo_ros empty_world.launch
```

Once the simulation scene is ready, the ARVA transmitters are initialized. Each transmitter needs to be properly configured with some parameters, mainly the initial pose (position and orientation) and the `id` of the device. Such parameters can be specified using ROS launch files. The setting of such parameters and how them are used in the proposed plugins will discussed in the next section. To initialize the ARVA transmitters use the following command:

```
$ roslaunch arva_sim spawn_arva_transmitters.launch
```

[1]The `roslaunch` is a fundamental tool that manages the launch of the ROS nodes. This command takes as argument a `.launch` file specifying the list of ROS nodes to be executed. Such launch file can be invoked either directly referencing its complete path, or just specifying the package name and the launch file in that package. In the latter case, the syntax of the command is reported in the line `$ roslaunch package_name launch_file`. Since the `roslaunch` command commonly holds the Linux shell, multiple terminals should be used to run the proposed test.

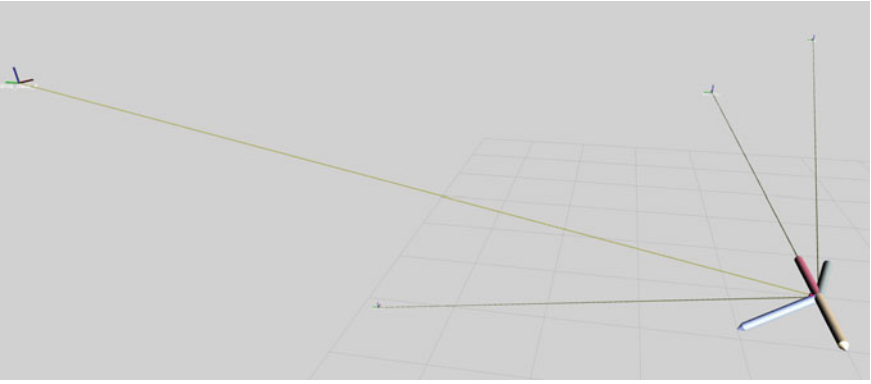


Fig. 7 The ARVA receiver detecting four different transmitters at the same time

The spawning process terminates when all the desired objects (in this case the ARVA transmitters) are loaded in the Gazebo simulation scene. At this point, the reader should be able to see the new object models in the Gazebo Graphical User Interface (GUI). Finally, to spawn the ARVA receiver use the following command:

```
$ roslaunch arva_sim spawn_receiver.launch
```

This command loads the ARVA receiver in the Gazebo environment. In addition, it opens the RViz program in order to show its output using a set of visual markers as reported in Fig. 7. In particular, since four ARVA transmitters have been placed in the Gazebo world, the reader should be able to see on RViz interface four arrows of different colors oriented towards the direction of the magnetic field generated by each transmitter.

3.3 arva_sim

The `arva_sim` ROS package provides Gazebo plugins to simulate the ARVA receiver and transmitter. The software architecture for this package is depicted in Fig. 8 and it is detailed hereafter.

Once the `arva_sim` is compiled, two new libraries, namely `libarva_sim_receiver.so` and `libarva_sim_transmitter.so` are available in the ROS system. They are responsible to emulate the ARVA receiver and transmitter, respectively. These libraries are based on Gazebo plugins and exchange information between them using the ROS communication infrastructure. There are different Gazebo plugins[2] which help the user to control the robot models, the sensors, the world properties, and even the way the simulator runs. Each of them is devoted to

[2] http://gazebosim.org/tutorials/?tut=plugins_hello_world.

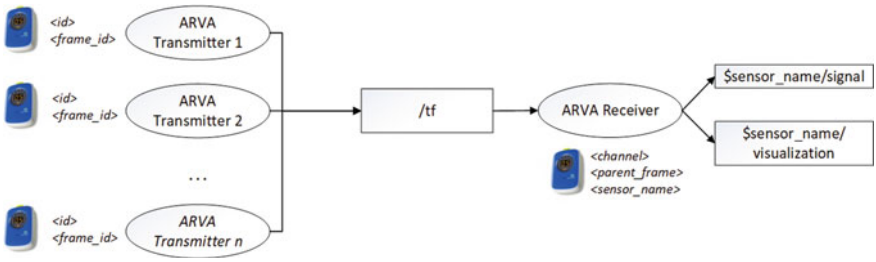


Fig. 8 arva_sim ROS package software architecture

manipulate a given element of the Gazebo simulator. Regarding the plugins developed in the `arva_sim` package, the transmitter module continuously informs the ROS system about its current pose inside the simulated world, while the receiver collects all the ARVA poses and calculates the magnetic flux direction for each of them. The transmitter output consists on a ROS message containing the direction to reach a given transmitter and the distance from it.

In order to setup the ARVA system, it is necessary to attach the devices to physical objects and spawn them in the simulation environment. The common way to do this in the ROS framework is by using *Universal Robotic Description Format* (URDF) [26] files. The URDF is a XML file format used in ROS to describe the kinematics and dynamics of robotic models. In addition, a specific tag is used to define the robotic element related to Gazebo, like plugins. This work uses an enhanced version of the URDF called `xacro` (XML Macros),[3] that is equal to URDF plus some add-ons used to define more complex object structures thanks to mathematical expressions and variable substitutions.

The following paragraphs present the ARVA system implementation showing how to configure the transmitter and receiver in the simulated environment using `xacro` files.

3.3.1 ARVA Transmitter

Each ARVA transmitter module is characterized by the following configuration parameters:

– `<id>`: the identification number of the transmitter.
– `<frame_id>`: point of reference for ARVA data.
– `<pose>`: initial position and orientation of the device in the simulation scene.

The `id` is a positive number used by the ARVA receiver to identify different transmitters. The `frame_id` is a string identifying the frame reference of the data. This frame is also useful to visualize the data using RViz. Finally, the `pose` parameter

[3] http://wiki.ros.org/xacro.

is the initial 3D position and orientation of the device in the simulation scene. This position can be later modified using a proper algorithm or directly moving the device via the Gazebo GUI.

Considering the test case proposed in the previous section, the ARVA transmitters have been imported in Gazebo using the `spawn_arva_transmitters.launch` file that is responsible to spawn in the Gazebo world the object model specified by the `transmitter.urdf.xacro` file. Such file is located in the `urdf` folder of the `arva_sim` package, whereas its content is discussed in the in the following lines:

```
<?xml version="1.0"?>
<robot name="transmitter" xmlns:xacro="http://www.ros.
    org/wiki/xacro">
    <xacro:arg name="id" default="0" />
    <xacro:arg name="frame_id" default="world" />
```

The first section of the description file defines the name of this model, that is *transmitter* and its arguments. As already stated, this model has different parameters, including the `id` and `frame_id`. By default their value is set to 0 and *world*, respectively.

```
    <link name="transmitter">
        <collision>
            <origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
            <geometry>
                <box size="0.3 0.1 0.04"/>
            </geometry>
        </collision>

        <visual>
            <origin xyz="0 0 0.0" rpy="0 0 0"/>
            <geometry>
                <box size="0.3 0.1 0.04"/>
            </geometry>
        </visual>

        <inertial>
            <origin xyz="0 0 1" rpy="0 0 0"/>
            <mass value="1"/>
            <inertia
            ixx="1.0" ixy="0.0" ixz="0.0"
            iyy="1.0" iyz="0.0"
            izz="1.0"/>
        </inertial>
    </link>
```

The second part of this file concerns the definition of the visual and physical shape of the model. It is shaped as a rectangle.

```
    <gazebo reference="transmitter">
        <material>Gazebo/Orange</material>
        <plugin name="arva_transmitter" filename="
            libarva_sim_transmitter.so">
            <id>$(arg id)</id>
            <frame_id>$(arg frame_id)</frame_id>
        </plugin>
    </gazebo>
</robot>
```

The last part of the file includes the elements related to Gazebo, e.g. the plugin to be load and its parameters. The value of such parameters and the initial position of the transmitter can be set including the following lines in a dedicated launch file:

```
<param name="arva_description_1" command="$(find xacro)/xacro --
    inorder $(find arva_sim)/urdf/transmitter.urdf.xacro id:=1
    frame_id:=world"/>
```

This block of code converts the `xacro` file in an URDF file. This conversion is needed because Gazebo only works with URDF robot models. ROS provides a convenient way to convert `xacro` in URDF using the `xacro`[4] ROS package. This work encapsulates this conversion directly in the launch file loading the description file of the ARVA transmitter in the parameter server of ROS. During this conversion process, it is possible to set the desired value for its parameters (`id:=1` and `frame_id:=world` in this example).

Finally, the model loaded in the ROS parameter server is spawned in the Gazebo simulator choosing a desired name, position and orientation. This is made by means of the `spawn_model` node implemented in the `gazebo_ros` package:

```
<node name="spawn_artva_1" pkg="gazebo_ros" type="spawn_model"
    args="-param arva_description_1 -urdf -x 6.0 -y 0.0 -z 1 -R 0
     -P 0 -Y 0 -model artva1" respawn="false" />
```

Further information on how to add plugins to Gazebo models can be found at the following link: http://gazebosim.org/tutorials?tut=ros_gzplugins.

Regarding the implementation of the transmitter plugin, ARVA transmitter module shares its pose using the `tf` ROS package [27]. This package is used to keep track of the relationship between multiple coordinate frames over time. In this way, the transmitter retrieves its pose in the Gazebo environment and updates a dynamic transform between the specified `frame_id` and the transmitter frame. As consequence, the `/tf` topic is automatically published by the ROS transform toolbox carrying out a message of type: `tf2_msgs::TFMessage`.[5] This message is composed by a variable length array of `geometry_msgs::TransformStamped` messages

[4] http://wiki.ros.org/xacro.

[5] http://docs.ros.org/jade/api/tf2_msgs/html/msg/TFMessage.html.

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 28
        nsecs: 814000000
      frame_id: "world"
    child_frame_id: "arva_data_1"
    transform:
      translation:
        x: 6.0
        y: 0.0
        z: 1.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
---
```

Fig. 9 Output ROS message of ARVA transmitter plugin. The *child_frame_id* of the message identifies the transmitter device

describing the pose of the transmitter and a reference to the emitting device. An example the output of the transmitter sensor is reported in Fig. 9. In this case, the transmitter has been spawned using the launch file `spawn_arva_id1.launch` of the `arva_sim` package.

3.3.2 ARVA Receiver

The ARVA system is completed with the receiver plugin. This plugin is characterized by the following parameters:

– `<channels>`: number of transmitters detected by the receiver at same time.
– `<parent_frame>`: point of reference for output ARVA data.
– `<sensor_name>`: device name.

The device name parameter influences the name of the topic where the output of the sensor is published. This is useful when multiple ARVA receivers are spawn in the same simulation scene. The `receiver.urdf.xacro` description file, located in the `urdf` folder of `arva_sim` package is used to spawn the receiver device in Gazebo. This file is quite similar to the one used for the transmitter except for the `<gazebo>` section, that is here reported.

```
<gazebo>
    <plugin filename="libarva_sim_receiver.so" name="receiver">
        <channels>$(arg channels)</channels>
        <frame_id>$(arg frame_id)</frame_id>
        <sensor_name>$(arg sensor_name)</sensor_name>
    </plugin>
</gazebo>
```

The following launch file can be used to spawn in Gazebo a receiver device called *receiver1*, able to read simultaneously four transmitters and with point of reference *arva_receiver*.

```
<launch>
    <param name="arva_receiver_description" command="$(find xacro)
        /xacro --inorder $(find arva_sim)/urdf/receiver.urdf.xacro
         channels:=4 frame_id:=arva_receiver sensor_name:=
        receiver1"/>

    <node name="arva_receiver_description" pkg="gazebo_ros" type="
        spawn_model" args="-param arva_receiver_description -urdf
        -x 0.0 -y 0.0 -z 0.0 -model artva_receiver" respawn="false
        " output="screen" />
</launch>
```

The output of this system is published on the `/$sensor_name/signal` topic. This topic consists in a custom data message defined in the `arva_sim` package and it is organized as follow:

```
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
    arva_sim/arva_data[] arva_signals
        int32 id
        geometry_msgs/Vector3 direction
            float64 x
            float64 y
            float64 z
```

This message contains a variable set of `arva_sim::arva_data`. The size of such set is equal to the number of channels of the receiver. As shown in the architecture depicted in Fig. 8, the receiver module subscribes to the `/tf` topic and calculates the magnetic field flux for all the transmitters spawned in the simulation scene. Considering the ARVA model described in Sect. 2, only the closest transmitters to the given receiver are returned as output. For this reason, the detected transmitter signals are sorted with respect to their distance from the receiver and only the first -`$channels`- signals are included in the output topic.

In the following, an output example of the ARVA system is reported. In particular, an instance of a message published on the `/receiver1/signal` topic after launched the test case discussed in Sect. 3.2 is reported.

```
header:
    seq: 62
    stamp:
        secs: 95
        nsecs: 404000000
    frame_id: "world"
```

```yaml
arva_signals:
    -
    id: 2
    distance: 1046.0
    direction:
        x: 9.99855692498e-05
        y: -6.97877478029e-10
        z: 6.61148733343e-05
    -
    id: 4
    distance: 1374.0
    direction:
        x: 4.37149210484e-05
        y: -0.000132286586449
        z: -0.000109870408778
    -
    id: 1
    distance: 1970.0
    direction:
        x: 1.46662550833e-05
        y: -5.80592723054e-06
        z: 4.81325878354e-06
    -
    id: 3
    distance: 4469.0
    direction:
        x: 1.22545043268e-06
        y: 4.36317304775e-06
        z: 1.73233547685e-06
```

In conclusion, this message contains all the information needed to follow the electromagnetic flux lines emitted by the four transmitters. In fact, the `id` field identifies the device. This is a positive integer greater or equal to zero and, in case of invalid (or non-existent) transmitter signals, it is set to -1. The `distance` field contains the distance between the emitter and receiver in centimeters. As for the framerate, the ARVA device receiver provides a feedback every second (1 Hz). Finally, by following the 3D vector of the *direction* field, the receiver is able to reach the transmitter. Moreover, an additional ROS topic has been set to display the output of the receiver sensor. In particular, the `/$sensor_name/visualization` topic contains a `visualization_msgs::MarkerArray`[6] and allows the visualization of ARVA data on RViz which are represented as arrows. Specifically, `visualization_msgs` is a ROS package implementing a set of messages used to deal in visualization-specific data. The content of this topic has already been show in Fig. 7. In the last example, the ARVA receiver was configured to detect four dif-

[6] http://wiki.ros.org/visualization_msgs.

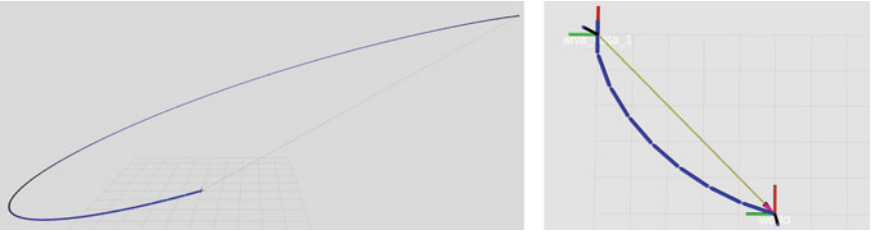


Fig. 10 Generated magnetic flux lines in different transmitter/receiver distances

ferent transmitters simultaneously. When the transmitters are far from the receiver, the detected magnetic fields are very weak. For this reason, the magnitudes of the vectors describing the direction of the field appear to be very small.

Finally, to show how the ARVA receiver works in the simulation scene, Fig. 10 reports two examples of magnetic field connecting the receiver with one transmitter placed at different distances.

4 Toward Automatic Robotic S&R Operations

This section presents an application for autonomous robots, and in particular for quad–copters, exploiting the ARVA system in S&R operations in avalanches.

This section is organized as follows. In the first part, the overall idea of the navigation controller is described, whereas in the last two parts the implementation of the evaluation case study is reported. Moreover, the installation and the configuration of the system setup used to evaluate the ARVA plugin is presented. Then the results of the tests are discussed.

4.1 Search Strategy Using ARVA Receiver

The design of search strategies for improving the S&R operation in avalanche scenarios is still an open topic and the comparison of the obtained performance must be made by means of reliable tools. So, the goal of this section is to provide the implementation of a standard search approach, inspired by the technique adopted by human rescuers, to test the reliability of the proposed ROS package `arva_sim`. Search techniques based on the ARVA system and usually adopted by alpine rescues, consists on a three-step strategy:

– *Primary search:* In this initial phase, the rescuers scan the avalanche site by following square wave or a zig-zag path to look for the first valid ARVA signal;

– *Secondary search:* Once that the first valid ARVA signal is detected, the rescuers start following the electromagnetic flux lines emitted by the ARVA transmitter. Since the flux lines pass through the transmitter, the rescuers use to move along them to reach the source of the electromagnetic field. In this search phase, the commercial receivers use only the two main search antennae (along the x_r-y_r axes) to provide the range d and the angle δ. It is worth observing that, in the secondary search the information exploited for navigating the electromagnetic filed is constituted by the angle δ whereas, the distance d is used to regulate the navigation speed. In particular, the rescuers slowdown while getting closer to the transmitter to make the identification of the transmitter location more accurate;
– *Tertiary search:* When the rescuers are sufficiently close to the transmitter, the electromagnetic field is strong enough to let the use of the third antenna of commercial ARVA receivers. In this working mode only the distance d is provided while the angle δ is set to 0 thus requiring a change of search strategy. Indeed, in this last phase, called *pin-pointing*, the rescuers search the location which minimizes the distance d.

Among the three search phases, the case study proposed in this section focuses on the secondary search phase and proposes an automatic control strategy steering the quad-copter along the ARVA flux lines. This choice is motivated by the availability of ARVA detector which provides the full information, constituted by ${}^r h$, see [28].

The quad-copter is considered equipped with a low level control system which is able to make the robot following an inertial trajectory. In particular, the low level control is fed by a signal representing the inertial speed, ${}^i V_i \in \mathbb{R}^3$, described in the inertial frame $\mathcal{F}_i$. Moreover, the dynamics of the agent is constraint by the presence of the terrain which, make unfeasible the underground trajectories. To accomplish this navigation task, the robotic agent is also equipped with a sensor providing the height above the soil. Given the terrain altitude, namely s, described in the inertial frame $\mathcal{F}_i$ by the map $S : \mathbb{R}^2 \mapsto \mathbb{R}_{>0}$, with $s = S(L_{xy}\, {}^i p_r)$, the constraint dynamics of the robotic agent is described by the system of Eq. (11):

$$
\begin{aligned}
{}^i\dot{p}_r &= {}^i V_i \\
{}^r h &= {}^r R_i\, {}^i R_t\, {}^t h\left({}^t R_i({}^i p_r - {}^i p_t)\right) + Q\, {}^r R_i\, {}^i w \\
y_s &= L_z\, {}^i p_r - S(L_{xy}\, {}^i p_r) + \nu \\
L_z\, {}^i p_r &\geq S(L_{xy}\, {}^i p_r)
\end{aligned}
\tag{11}
$$

where y_s models a device belonging to the family of the range sensors, which are typically corrupted by a white noise, namely ν.

The high level control system, proposed in this paper, is constituted by two independent subsystems: one representing a *height–hold* controller and regulating the vertical position, and one steering the drone along the ARVA flux lines.

4.2 ARVA Navigation Controller

Given the measurement ${}^{r}h$, this controller determines the first two components of the speed ${}^{i}V_i$ as:

$$ {}^{i}V_{xy} = L_{xy} v_{\max} \frac{k_c \|{}^{r}h\|^{-\frac{1}{3}}}{\sqrt{1 + (k_c \|{}^{r}h\|^{-\frac{1}{3}})^2}} \, {}^{i}R_r \frac{{}^{r}h}{\|{}^{r}h\|} \tag{12} $$

where $v_{\max} > 0$ and $k_c > 0$. The term $v_{\max} \frac{k_c \|{}^{r}h\|^{-\frac{1}{3}}}{\sqrt{1 + (k_c \|{}^{r}h\|^{-\frac{1}{3}})^2}} \in [0, \ v_{\max}]$ indicates the speed magnitude whereas the term ${}^{i}R_r \frac{{}^{r}h}{\|{}^{r}h\|}$ determines the orientation, in the inertial frame, of the speed vector.

4.2.1 Height-Hold Controller

Given the measurement y_s and a constant reference height $r_s > 0$ the controller for the management of the height provides the vertical speed, ${}^{i}V_z$, given by the relation of Eq. (13):

$$ {}^{i}V_z = k_z (r_s - y_s) \tag{13} $$

4.2.2 Overall Control System

The composition of the laws of Eqs. (12) and (13) provides the overall control system, thus represented by:

$$ {}^{i}V_i = \begin{bmatrix} {}^{i}V_{xy} \\ {}^{i}V_z \end{bmatrix}. \tag{14} $$

4.3 Case Study

In order to demonstrate the effectiveness of the sensor model proposed in this work, this section presents a simulated test case implementing the control system previously described. The test case consists in a quad–copter equipped with an ARVA receiver and operating in a mountain covered in snow. To demonstrate the effectiveness of the ARVA simulation system, this work adopted the basic search pattern as described by the Eqs. (12)–(13), which constitutes the current strategy exploited by human rescuers.

Regarding the system setup, the core of the UAV is represented by its autopilot, which allows the robot to be remotely operated from a Ground Control Station (GCS). In this work, this manuscript assume that the simulated UAV is equipped with a PX4[7] autopilot. This is a famous open source autopilot system providing both hardware and software stacks, characterized by different features making it particularly suitable to develop autonomous aerial applications.

In relation with the ARVA application, PX4 supports MAVLink (Micro Air Vehicle Link), a standard communication protocol between the UAV and the GCS. In this context, mavros ROS package[8] make very easy to get information from the autopilot (i.e. its status, geographical position, altitude, …) and send different kind of commands to it. In this context, mavros creates a ROS topic for each message supported by the autopilot. In addition, the Gazebo Software-In-The-Loop (SITL) software available with the PX4 suite allows for the emulation of the flight code of the autopilot on a standard Linux machine using Gazebo. The software responsible of such simulation is called `sitl_gazebo`[9] and is able to emulate the autopilot, the sensors and dynamics of flying robots. The `sitl_gazebo` is based on RotorS, a Micro Aerial Vehicles (MAV) Gazebo simulator [29]. In particular, RotorS provides flying robot models and sensors suitable for Gazebo ROS usable to develop custom controllers for UAVs. PX4 SITL uses some pieces from RotorS, but in contrast to this, it directly executes the control code resident on the UAV autopilot. In this way, all the low level controllers and state estimation algorithms for UAV are ready to be used making extremely simple to move the code from the simulated to the real robot.

In the following, the installation of the PX4 software stack and Gazebo SITL is presented in order to replicate the proposed case study.

4.3.1 PX4 SITL Installation

The simulated case study has been implemented using a standard GNU/Linux Ubuntu 16.04 Operating System. Regarding the version of ROS, it is based on ROS *Kinetic Kame*. Moreover, the system supports Ubuntu 18.04 with ROS *Melodic Morenia* as well. To download the source code of PX4 software stack, move to the ROS workspace. This work assumes to use a forked version of the original PX4 repository where a UAV, already equipped with an ARVA receiver, is available:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/jocacace/Firmware --recursive
```

Before to compile the PX4 software suite, some dependencies must be installed by using the following commands:

[7] https://px4.io/.

[8] http://wiki.ros.org/mavros.

[9] https://github.com/PX4/sitl_gazebo.

```
$ sudo apt-get install python-pip
$ sudo apt-get install python-jinja2
$ sudo pip install numpy toml
$ sudo apt-get install protobuf-c*
```

As already stated, to communicate with the autopilot the ROS package called mavros is used. In addition, some sensors of the simulated robots are part of *hector-quadrotor*[10] ROS package. Before to continue, the reader must install these dependencies:

```
$ sudo apt-get install ros-$ROS_DISTRO-mavros-*
$ cd /opt/ros/$ROS_DISTRO/lib/mavros/
$ sudo install_geographiclib_datasets.sh
$ sudo apt-get install ros-$ROS_DISTRO-hector-*
```

Select the correct branch for `sitl_gazebo` package:

```
$ cd ~/catkin_ws/src/Firmware/Tools/sitl_gazebo
$ git checkout master
```

Finally, the PX4 software stack is ready to be compiled:

```
$ cd ~/catkin_ws/src/Firmware
$ make posix_sitl_default gazebo -j4
```

If the compilation process terminates without errors, the Gazebo simulator is launched. Notice that this is a standalone version of Gazebo, without any connection with ROS framework. To use such simulator with ROS, terminate the Gazebo process and continue to follow this tutorial.

The next step regards the setup of the environment before launching the SITL Gazebo simulation. To do this, a configuration file in the `sitl_gazebo` directory has been prepared. Use the `source` command on this file to load the configuration:

```
$ cd ~/catkin_ws/src/Firmware/Tools/sitl_gazebo
$ source load_sitl_conf.sh
```

At this point the Gazebo simulator is ready to be started by spawning the model of the UAV equipped with the ARVA receiver device.

```
$ roslaunch px4 iris_arva.launch
```

This launch file starts mavros too, in order to enable the ROS/autopilot communication.

[10] http://wiki.ros.org/hector_quadrotor.

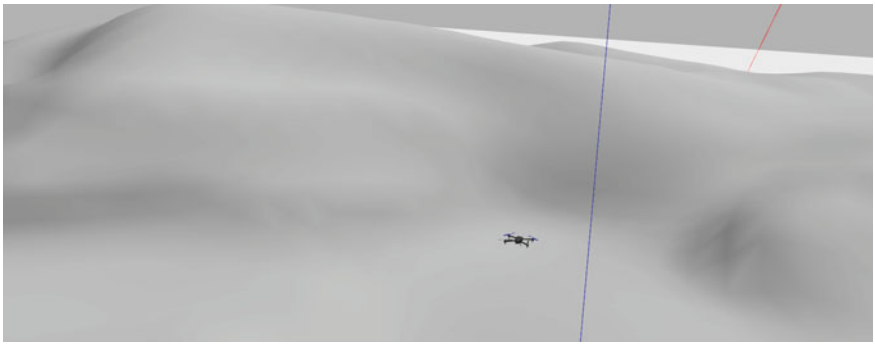


Fig. 11 Simulated environment for system testing

Figure 11 depicts an example simulation scene, where a *3DR Iris* quad–rotor inspects an avalanche site. This scenario is used to validate the proposed ARVA sensor. Regarding the ARVA system, in order to use receiver data to control the motion of the robot, one or more transmitters must be spawned in the Gazebo world. To do this, launch files located in the `arva_sim` package can be used. For example, use the following command to spawn the transmitter with `id:=1`.

```
$ roslaunch arva_sim spawn_arva_id1.launch
```

4.3.2 System Evaluation

After the initialization of simulation scene, the robot is ready for accepting control commands. The whole software architecture deployed to evaluate the system is depicted in Fig. 12 and it is discussed in the remaining of this paragraph.

Gazebo provides the information needed to initialize the sensors of the UAV, as its Inertial Measurement Unit (IMU) and GPS. The mavros package shares these information and enables the control of robot via several ROS services and topics. As already stated, the UAV is equipped with an ARVA receiver, whose output is described in Sect. 3.3. To endow the UAV with the ARVA receiver, this work modified the `xacro` description file model by defining its sensors (the `component_snippets.xacro` file) as shown in the previous section. Such file is located in the PX4 SITL Gazebo folder, precisely in the ∼/catkin_ws/src/Firmware/Tools/sitl_gazebo/models/rotors_description/urdf/ folder. Moreover, the structure of the configuration file of the PX4 SITL simulator is quite complex so, to have deeper knowledge on how modify such configuration files, readers should refer to RotorS package [29] documentation.

Before actuating the UAV by the injection of the desired velocities, some operations must be done. In particular, the robot must be *armed* (the motors must be allow

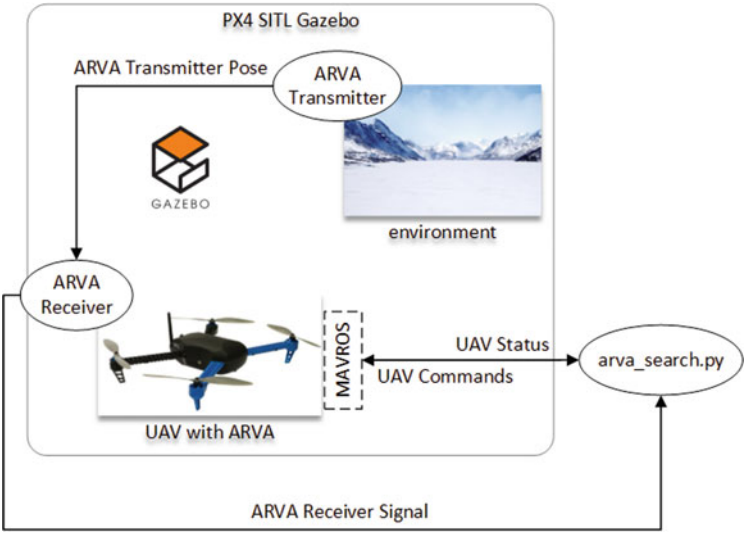


Fig. 12 Software architecture deployed for the testing of the ARVA system

to start spinning) and it must takeoff. Finally, the flight mode of the robot must be switched on `offboard`. Flight Modes define how the autopilot responds to user input and controls the movements of the vehicle. A comprehensive description of the PX4 flight modes can be found at this link: https://docs.px4.io/v1.9.0/en/flight_modes/. In `offboard` flight mode, the UAV is enabled to accept input data from a GCS. Finally, the direction of the magnetic flux generated by the transmitter devices can be taken into account to generate the desired body velocity of the robot.

Such operations are managed by the `arva_search.py` python ROS node located in the example folder of the `arva_sim` package. This code is discussed in the following line:

```
#!/usr/bin/env python
import rospy
import roslib
import time
import numpy
from mavros_msgs.msg import *
from mavros_msgs.srv import *
from arva_sim.msg import arva
from nav_msgs.msg import Odometry

class SIMPLE_ARVA_SEARCH:
    def __init__(self):
    rospy.init_node("arva_search")
    rospy.Subscriber('mavros/state', State, self.stateCb)
    rospy.Subscriber("/receiver1/signal", arva, self.arvaCb)
```

```
    rospy.Subscriber("/mavros/global_position/local", Odometry,
        self.posCb)
    self.vel_pub = rospy.Publisher('mavros/setpoint_raw/local',
        PositionTarget, queue_size=0)
    self.artva_x = self.artva_y = self.artva_z = 0
    self.state = State()
    self.arva_ready = False
```

The ROS node is initialized in the constructor of the `SIMPLE_ARVA_SEARCH` class. In particular, this node subscribes to the following topics:

- `/mavros/state`: this message contains the UAV state, the arming status and the current flight mode.
- `/receiver1/signal`: the ARVA receiver data.
- `/mavros/global_position/local`: the Cartesian position of the UAV. In this context, the altitude is of main interest.

As for the output, the desired velocity of the body of the UAV is published using a `mavros_msgs::PositionTarget`[11] message on the `mavros/setpoint_raw/local` topic:

```
    def arvaCb(self, msg):
        self.artva_x = msg.arva_signals[0].direction.x
        self.artva_y = msg.arva_signals[0].direction.y
        self.artva_z = msg.arva_signals[0].direction.z
        self.dist = msg.arva_signals[0].distance
        if self.dist > -1:
            self.arva_ready = True
        else:
            self.arva_ready = False
    def stateCb(self, msg):
        self.state = msg
    def posCb(self, msg):
        self.z = msg.pose.pose.position.z
```

The callbacks handling the input data of the node are implemented in the above section of code. As already stated, this test case considers an ARVA receiver with only one channel. An ARVA signal is considered valid when the distance value is greater then -1.

```
    def run(self):
        rate = rospy.Rate(10.0)
        max_vel = 1.0
        sp = PositionTarget()
        sp.type_mask = 1 + 2 + 4 + 64 + 128 + 256 + 512 + 1024
        sp.coordinate_frame = 8
        ks = 0.5 #% vertical gain
        kv = 0.1 #% speed gain
```

[11] http://docs.ros.org/api/mavros_msgs/html/msg/PositionTarget.html.

```
        vmax = 1.0 #% [m/s] maximum speed
        rs = 1 #% reference height
        rospy.wait_for_service('mavros/cmd/arming')
        armService = rospy.ServiceProxy('mavros/cmd/arming',
            mavros_msgs.srv.CommandBool)
        armService(True)
        takeoffService = rospy.ServiceProxy('mavros/cmd/takeoff',
            mavros_msgs.srv.CommandTOL)
        takeoffService(yaw = 1.5, latitude = 44.4928127, longitude
            = 11.3299824, altitude = 2.5)
        time.sleep(2)
        flightModeService = rospy.ServiceProxy('mavros/set_mode',
            mavros_msgs.srv.SetMode)
        flightModeService(custom_mode='OFFBOARD')
        while not rospy.is_shutdown():
            if self.arva_ready:
                artva_data = numpy.array([self.artva_x, self.artva_y
                    , self.artva_z])
                vers = artva_data / numpy.linalg.norm(artva_data)
                d = self.dist / 100.0
                k = vmax*(kv*d)/math.sqrt(1+(kv*d)**2)
                vxy = numpy.array( [k*vers[0], k*vers[1]] )
                vz = ks*(rs-self.z);
                sp.velocity.x = -vxy[1]
                sp.velocity.y = vxy[0]
                sp.velocity.z = vz
                sp.yaw_rate = 0.0
            else:
                sp.velocity.x = sp.velocity.y = sp.velocity.z = sp.
                    yaw_rate = 0.0
        self.vel_pub.publish( sp );
        rate.sleep()
if __name__ == '__main__':
    arva_client = SIMPLE_ARVA_SEARCH()
    arva_client.run()
```

In the main loop of the ROS node the control gain are initialized and three ROS services are used to arm the UAV (`mavros/cmd/arming`), perform the takeoff in a desired location (`mavros/cmd/takeoff`) and enable the `OFFBOARD` flight mode (`mavros/set_mode`).

Finally, when the receiver sensor of the UAV engages a valid ARVA signal, the controller makes the drone following the direction of the electromagnetic flux line, as described by the Eq. (14). As for the maximum velocity of the UAV, it has been set to 1 m/s. This velocity is compatible with the one of the human rescuers.

After that the simulation scene is ready and at last one ARVA sensor has been spawned, this ROS node can be ran using the following command:

```
$ rosrun arva_sim arva_search.py
```

As for the ARVA sensor evaluation, the test was repeated considering four different situations. In particular, four ARVA transmitters, placed at different locations in the workspace are configured. For each test case, only one transmitter per time is spawned in the simulation. In each test, the initial position of the UAV has been set to the origin, $x = 0$ and $y = 0$, whereas for the altitude, the initial takeoff height of the quad–copter has been randomly set up to 4 meters. Regarding the height reference, r_s, the presented results have been obtained by imposing a fixed altitude of $r_s = 1$ meter. In this way, the quad–copter is able to safely navigate the environment without crashing on the terrain.

During each test, the most important variables were saved and depicted in the Fig. 13. In particular, the following set of data has been monitored:

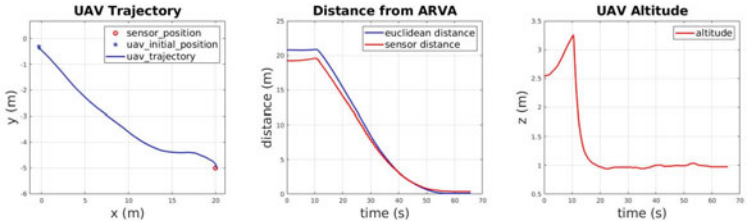
- *UAV Trajectory*: The trajectory navigated by the quad-copter during the rescue mission. These plots report the 2D trajectory, on the inertial $x_i - y_i$ plane, of the transmitter (rigidly attached to the UAV) (blue continuous line) and its initial position, depicted as a blue star. Finally the position of the ARVA transmitter is indicated by a red circle;
- *Distance from ARVA*: The distance between the UAV and the transmitter is investigated by means of two metrics: the euclidean distance on the inertial $x_i - y_i$ plane and the electromagnetic distance d provided as output by the ARVA receiver;
- *UAV Altitude*: The height trajectory of the UAV during the navigation.

The results of the four tests are discussed in the following section.

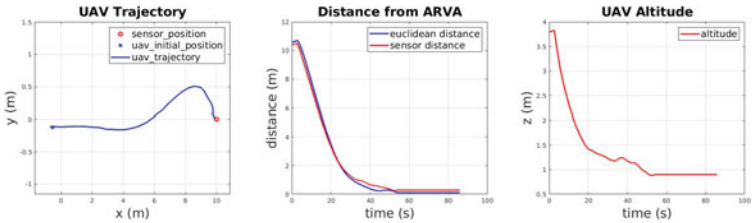
4.4 Discussion

The proposed test case demonstrates the working principle of the ARVA plugins. The data monitored during the four scenarios are reported in Fig. 13a–d. From such graphics it is possible to see that the UAV is able to reach the transmitter following the electromagnetic field flux lines. In particular, the shape of the trajectories performed by the UAV recall the magnetic field shape. Regarding the figures depicting the distance from the ARVA transmitter, can be noticed that following the electromagnetic field lines does not implies that the transmitter is reached with the shortest path. Indeed, for all the experiments the Euclidean distances show a limited growth before decreasing to small values close to zero.

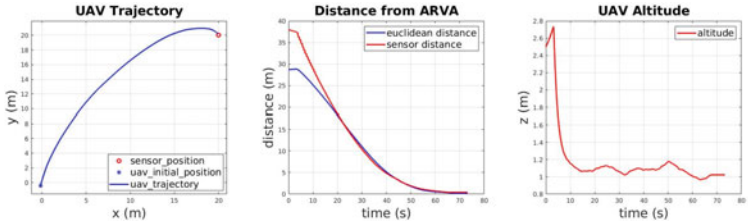
This is particularly evident in the fourth test (see Fig. 13d). Currently, the practical implementation of the proposed test case on a real robot is not feasible. In fact, the manufacturing of a UAV equipped with an ARVA receiver is the main goal of the underway European Project *AIRBorne*. Until now, the main problem with this setup come from the noise generated by the motors of quad–copters. As matter of fact, this noise inhibits the ARVA receiver. In order to replicate this setup on a real UAV, researchers and developers should follow the results of *AIRBorne* project. However, this work proposed one suitable test case for the `arva_sim` package, with the aim to demonstrate its reliability and accuracy. The ARVA Gazebo plugins in fact can



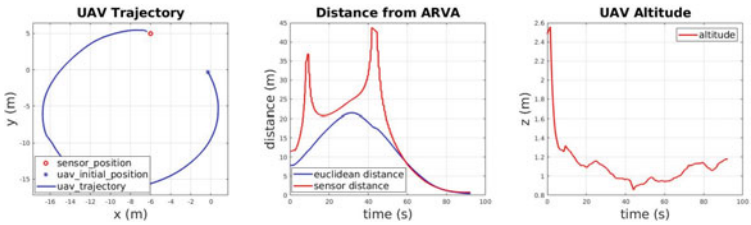
(a) First test case. The transmitter with $id := 1$ has been placed at location: $x := 20.0$, $y := -5.0$, $z := 0$.



(b) Second test case. The transmitter with $id := 2$ has been placed at location: $x := 10.0$, $y := 0.0$, $z := 1.0$.



(c) Third test case. The transmitter with $id := 3$ has been placed at location: $x := 20.0$, $y := -20.0$, $z := -1.0$.



(d) Fourth test case. The transmitter with $id := 4$ has been placed at location: $x := -6.0$, $y := 5.0$, $z := 0.0$.

Fig. 13 Testing of the simulated ARVA system in a S&R missions operated by an UAV. The proposed tests considered different location of the transmitter devices. The thee figures report the UAV trajectory on the inertial plane, the distance between transmitter and receiver, projected on the inertial plane, and the UAV height above the ground during the navigation

be applied to any other robot or object spawnable in Gazebo, such as mobile ground robot or even pedestals thus allowing researchers to develop faster and smarter S&R strategies based on ARVA receiver data.

5 Conclusions

This chapter proposed a new ROS package called `arva_sim` (https://github.com/jocacace/arva_sim) who aims to provide the simulation, the configuration files and libraries to emulate a complete ARVA system. The ARVA represents the forefront technology adopted in S&R operations to localize victims of avalanches buried under the snow. This system is composed by two modules: the transmitter and receiver. The transmitter generates a magnetic field detected by the receiver. The receiver provides the distance and direction to reach a given transmitter following the magnetic field flux lines. The simulation package is based on ROS Gazebo plugins. In this way, developers and researchers can equip simulated robots with ARVA sensors and designing faster and smarter S&R strategies. This work describes the theoretical background of the ARVA transceiver model, its implementation and the configuration of its simulator. Finally, a use case in which an autonomous navigation mission based on ARVA data has been presented in order to validate the accuracy and effectiveness of the package `arva_sim`.

Acknowledgements This research was supported by the European Project AIRBORNE, Grant Agreement no. 780960, web site https://www.airborne-project.eu/.

References

1. M. Hutter, C. Gehring, D. Jud, A. Lauber, C.D. Bellicoso, V. Tsounis, J. Hwangbo, K. Bodie, P. Fankhauser, M. Bloesch, R. Diethelm, S. Bachmann, A. Melzer, M. Hoepflinger, Anymal - a highly mobile and dynamic quadrupedal robot, in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 38–44
2. N. Kashiri, L. Baccelliere, L. Muratore, A. Laurenzi, Z. Ren, E.M. Hoffman, M. Kamedula, G.F. Rigano, J. Malzahn, S. Cordasco, P. Guria, A. Margan, N.G. Tsagarakis, Centauro: a hybrid locomotion and high power resilient manipulation platform. IEEE Robot. Autom. Lett. **4**, 1595–1602 (2019)
3. M. Konyo, Y. Ambe, H. Nagano, Y. Yamauchi, S. Tadokoro, Y. Bando, K. Itoyama, H. Okuno, T. Okatani, K. Shimizu, E. Ito, *ImPACT-TRC Thin Serpentine Robot Platform for Urban Search and Rescue: Results from the ImPACT Tough Robotics Challenge* (2019), pp. 25–76
4. C. Sampedro, A. Rodriguez-Ramos, H. Bavle, A. Carrio, P. de la Puente, P. Campoy, A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. J. Intell. Robot. Syst. **95**, 601–627 (2019)
5. D. Falanga, K. Kleber, S. Mintchev, D. Floreano, and D. Scaramuzza, The foldable drone: a morphing quadrotor that can squeeze and fly. IEEE Robot. Autom. Lett. 1–1 (2018)
6. L. Marconi, C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, A. Kleiner, V. Lippiello, A. Finzi, B. Siciliano,

A. Sala, N. Tomatis, The sherpa project: smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments, in *2012 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Nov 2012, pp. 1–4
7. European Project Airborne. https://www.airborne-project.eu/
8. V. Ferrara, Technical survey about available technologies for detecting buried people under rubble or avalanches. WIT Trans. Built Environ. **150**, 91–101 (2015)
9. Ortovox, Arva ortovox 3+ user manual. https://www.ortovox.com/fileadmin/OX-GAL-3-_web_18_19.pdf, May 2019
10. Barryvox, Arva mammut barryvox user manual. https://www.mammut.com/assets/download/90/barryvox_user_manual_web_v2_it-4890.pdf, May 2019
11. L. Joseph, J. Cacace, *Mastering ROS for Robotics Programming - Second Edition: Design, Build, and Simulate Complex Robots Using the Robot Operating System*, 2nd edn. (Packt Publishing, 2018)
12. `arva_sim` repository. https://github.com/jocacace/arva_sim
13. S. Waharte, N. Trigoni, Supporting search and rescue operations with UAVs, in *2010 International Conference on Emerging Security Technologies* (IEEE, 2010), pp. 142–147
14. T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I.L. Grixa, F. Ruess, M. Suppa, D. Burschka, Toward a fully autonomous UAV: research platform for indoor and outdoor urban search and rescue. IEEE Robot. Autom. Mag. **19**, 46–56 (2012)
15. M.A. Goodrich, B.S. Morse, D. Gerhardt, J.L. Cooper, M. Quigley, J.A. Adams, C. Humphrey, Supporting wilderness search and rescue using a camera-equipped mini UAV: research articles. J. Field Robot. **25**, 89–110 (2008)
16. J. Cacace, A. Finzi, V. Lippiello, M. Furci, N. Mimmo, L. Marconi, A control architecture for multiple drones operated via multimodal interaction in search amp; rescue mission, in *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Oct 2016, pp. 233–239
17. J. Cacace, A. Finzi, V. Lippiello, Implicit robot selection for human multi-robot interaction in search and rescue missions, in *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Aug 2016, pp. 803–808
18. G. Bevacqua, J. Cacace, A. Finzi, V. Lippiello, Mixed-initiative planning and execution for multiple drones in search and rescue missions, vol. 2015, 07 (2015)
19. J. Cacace, R. Caccavale, A. Finzi, V. Lippiello, Attentional multimodal interface for multidrone search in the alps, in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct 2016, pp. 001178–001183
20. G. Strang, *Introduction to Linear Algebra*, 4th edn. (Wellesley-Cambridge Press, Wellesley, MA, 2009)
21. G.A.F. Seber, *A Matrix Handbook for Statisticians*, 1st edn. (Wiley-Interscience, New York, NY, USA, 2007)
22. P.L.E. Uslenghi, Electromagnetic fields. IEEE Antennas Propag. Mag. **52**, 117–118 (2010)
23. R. Featherstone, *Rigid Body Dynamics Algorithms* (Springer, Berlin, Heidelberg, 2007)
24. P. Pinies, J.D. Tardos, Fast localization of avalanche victims using sum of gaussians, in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006*, May 2006, pp. 3989–3994
25. H. Kam, S.-H. Lee, T. Park, C.-H. Kim, Rviz: a toolkit for real domain data visualization, in *Telecommunication Systems*, vol. 60 (2015), pp. 1–9
26. L. Kunze, T. Roehm, M. Beetz, Towards semantic robot description languages, in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 5589–5595
27. T. Foote, tf: the transform library, in *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, April 2013, pp. 1–6
28. C.D. Salos, F. M. Lera, J.L. Villarroel, Digital signal processing in triple antenna arvas, in *2007 IEEE International Conference on Signal Processing and Communications*, Nov 2007, pp. 484–487
29. F. Furrer, M. Burri, M. Achtelik, R. Siegwart, *RotorS - A Modular Gazebo MAV Simulator Framework*, vol. 625 (2016), pp. 595–625

Jonathan Cacace was born in Naples, Italy, on December 13, 1987. He received Master Degree in computer science with honors from University of Naples "Federico II" in 2012 and Ph.D degree in Robotics in 2016 from the same institution. Currently He is post-doc in Naples and his research interest focuses on Human-Robot Interaction in 4.0 Industry, autonomous control of UAV for inspection an maintenance and robotic simulation. He is author of the book: *Mastering ROS for Robotics Programming: Second Edition*, edited by Pack publishing.

Nicola Mimmo graduated in Aerospace Engineering in 2009 at the University of Bologna where he also defended his Ph.D. in Automation. Since 2010 he has been working, at the Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi", on national and European projects for the development of unmanned aerial vehicles for civil applications. His research interests range from fault tolerant controls to nonlinear control systems in aerospace. He is a co-author of tens of scientific papers and he organized technical sessions of international conferences. He worked with the major national and European aircraft designers for the development of both flight systems and flight control laws which are currently protected by an international patent.

Lorenzo Marconi Lorenzo Marconi graduated in 1995 in electrical engineering from the University of Bologna. Since 1995 he has been with the Department of Electronics, Computer Science and Systems at University of Bologna, where he obtained his Ph.D. degree in 1998. From 1999 he has been an assistant professor in the same department where he is now a full professor. He has held visiting positions at various academic/research international institutions. He is a co-author of more than 250 technical publications on the subject of linear and nonlinear feedback designs published on international journals, books and conference proceedings. In 2005, he received the "Outstanding Application Paper Award" from IFAC for a coauthored paper published on Automatica. He is Fellow of IEEE for "contributions to feedback design of nonlinear systems and unmanned aerial vehicles". His current research interests include nonlinear control, output regulation, control of autonomous vehicles, fault detection and control.

Robotic Manipulation and Control

ROS Implementation for Untethered Microrobot Manipulation

Aytaç Kahveci, Nail Akçura, Levent Çetin, Abdulkareem Alasli, and Özgür Tamer

Abstract In this section, a ROS implementation of a system which is utilized for untethered microrobot manipulation is introduced. Untethered microrobot locomotion is an area that is studied globally and magnetic actuation is one of the proven methods for microrobot locomotion. Several types of actuator systems were approved by successful works and became associates in the literature works. These locomotion strategies require real-time and closed-loop control for most of the applications. We developed a custom actuation system using a coaxially aligned electromagnet couple, providing magnetic field for locomotion and the actuator structure was carried by an industrial manipulator, KUKA robot. So that, the complexity of electromagnet current control is divided into simpler sub parts that involve mechanical translation and rotation motions. This structure principle provides the capability of a larger task space. KUKA manipulator and electromagnetic actuator (EMA) as its end-effector form the main system structure. Also, visual feedback is provided by camera systems. Some tasks were distributed to peripheral NVIDIA Jetson TK1 Embedded Development Kits. Here, we focus on the integration of ROS into a microrobot manipulation or alike systems. System components, integration, control, test phases and detailed ROS components are described, an application with the results is presented.

A. Kahveci
Mechanical Engineering Department, Izmir Katip Çelebi University, Izmir, Turkey
e-mail: aytackahveci93@gmail.com

N. Akçura
Mechatronics Engineering Department, Dokuz Eylül University, Izmir, Turkey
e-mail: nail.akcura@deu.edu.tr

L. Çetin (✉)
Mechatronics Engineering Department, Izmir Katip Çelebi University, Izmir, Turkey
e-mail: levent.cetin@ikcu.edu.tr

A. Alasli
Mechanical Systems Engineering Department, Nagoya University, Nagoya, Japan
e-mail: al.asli.abdulkareem@f.mbox.nagoya-u.ac.jp

Ö. Tamer
Electrical and Electronics Engineering Department, Dokuz Eylül University, Izmir, Turkey
e-mail: ozgur.tamer@deu.edu.tr

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_9

Keywords ROS · EMA · Microrobot · Untethered control

1 Introduction

Small-scale robotics is a popular area among the researchers and keeps developing in collaboration with the explorations all over the globe. For proper applications, it still needs progression due to the ultimate challenges in various scopes. Untethered magnetic actuation method is thought to have a great potential for the breakthroughs. It contributes to a concept that external magnetic field is used as the main power source for the robot locomotion. These systems comprise multidisciplinary components for the duties like actuation, sensing and communication in real-time.

ROS is popular among robotics area including daily life solutions as well as critical areas like military and biomedical applications. Multidisciplinary projects may require an integration method and ROS comes to aid the developers with powerful open-source features as a middleware. The people who are beginner in this area may have difficulty in custom-made and complex systems where different components are included.

In this work, an untethered microrobot actuation system is presented that uses ROS for the integration. Microrobot locomotion principle was applied using a system formed of custom designed actuators and a KUKA industrial manipulator. A ROS based structure was developed and integrated to the mechatronics system with various mechanical, hardware and software features. The components and structures are explained, an application is given for the functionality approval.

This chapter covers the following topics:

- The topic background is introduced (Sect. 2),
- The general system layout, actuation principles and ROS-related structure are presented (Sect. 3),
- ROS environment configuration is given (Sect. 4),
- A brief pseudo schematic and a test application is introduced (Sect. 5),
- A deeper ROS packages explanation is presented (Sect. 6),
- An application and its results are studied in (Sect. 7),
- A conclusion is held (Sect. 8),

2 Background

This section provides a coarse overview external magnetic field actuated microrobot manipulation and some proved systems in the literature. Basically, microrobots are the structures with less than 1 mm^3 volume and specialized for a goal. To locomote the microrobot, many actuation methods and different microrobot structures have

been studied. Current technology level makes it possible to manufacture sub millimetres and micrometers scaled microrobots in some desired shapes. Though, it is still not convenient to have the locomotion using on-board power source like a battery. Also, it may lead to many challenges in several scopes like small scaled power circuitry, micro-actuator and microcontroller. Therefore, it drifted to the research methods involving external power source. So that, the focusing area drifted to different methods. Externally applied magnetic field provides promising solutions to the researchers. To be more clear about this, the considered method depends on microrobots with magnetic features locomoted by external magnetic field and it is not related with untethered power transmission that will be harvested by the microrobots for on-board power sources.

There are many researches for microrobot structures, locomotion methods and actuation systems, and field of applications, focusing on different aspects and components. These works were summed up with several reviews [1–3]. The application areas may vary that need of microassembly and micromanipulation operations. In biomedical application, these tiny robots can be used for targeted therapy like drug delivery, material removal like biopsy, controllable structures like implants and stents, and telemetry like image capturing [4]. In the future, these microrobots should be vital in the biomedical applications [5]. Another small scaled application in biomedical can be counted as gastrointestinal or endoscopic applications with especially better visual feedback results comparing to conventional methods [6].

Using biomimicry, microrobot structures in locomotion methods mostly inspired from living creatures and then adapted their motion concepts to the applications. It would be more convenient to acknowledge locomotion methods and microrobot structures as coupled. The locomotion types can be categorized as helical motion, travelling wave, pulling with magnetic gradient and clinical magnetic resonance imaging system. A proved locomotion method, helical motion that e-coli bacteria does, was inspected in [7] and pioneered many other applications. Even if magnetic pulling locomotion does not exist in the nature, it may surpass other methods [8]. It was showed that using magnetic gradient, the microrobot locomotion can be accomplished [9]. This gradient method also leaded to navigation of magnetic beaded systems in-vivo and its feasibility researches [10]. In living animal tests, a cylindrical type microrobot locomoted with magnetic gradient was operated inside a rabbit eye [11].

As the external magnetic field actuators, there were two choices as the actuator type: electromagnets or permanent magnets. Yesin et al. developed an actuator system consisting of Helmholtz and Maxwell coils configurations with the support of a rotary mechanism, having motion with a permanent magnet robot using uniform magnetic field and magnetic field gradient inside a 2D constrained maze [9]. Choi et al. developed a similar system with stationary electromagnets, having a closed-loop control in 2D plane [12]. Go et al. developed the system and proved 3D motion in 3D blood vessel phantom model [13]. OctoMag [14] and MiniMag [15] systems consist of eight and six electromagnets, providing 5 Degrees-of-Freedom (DoF) motion. Diller et al. provided arbitrary control of multiple robots with an actuation system that consist of eight electromagnets [16]. These systems focus on tiny task spaces not

exceeding a few centimetre cubes. Using the same actuator system concept, alumni and follower researchers worked on developing motion and microrobot structures.

In some of the permanent magnet actuator based systems, researchers put a permanent magnet onto the tip of a mechanism like an industrial manipulator. By moving the permanent magnet, the desired magnetic field aspects are applied to manipulate the microrobot. Mahoney et al. attached a cylindrical rotating permanent magnet onto the tip of an industrial manipulator and locomoted a helical shaped capsule using rotating magnetic field [17]. Then, rotating magnetic field characterization was inspected [18] and the capsule was controlled using rotating magnetic field inside a fluid-filled tank [19]. Wright et al. enchanted the system by changing the actuating cylindrical permanent magnet with a spherical rotating permanent magnet and accomplished motion in constrained mock-up test bench [20]. There are also some versions of open-loop controlled systems used for capsule endoscopy [21].

3 Microrobot Manipulation System

3.1 System Components

The manipulation system and its components are given in Fig. 1. Basically, the system consists of a custom designed Electromagnetic Actuator (EMA), a KUKA industrial manipulator and the microrobot.

EMA is the component that generates the magnetic field required for the microrobot motion. It can be inspected as two mechanical subsystems: electromagnets

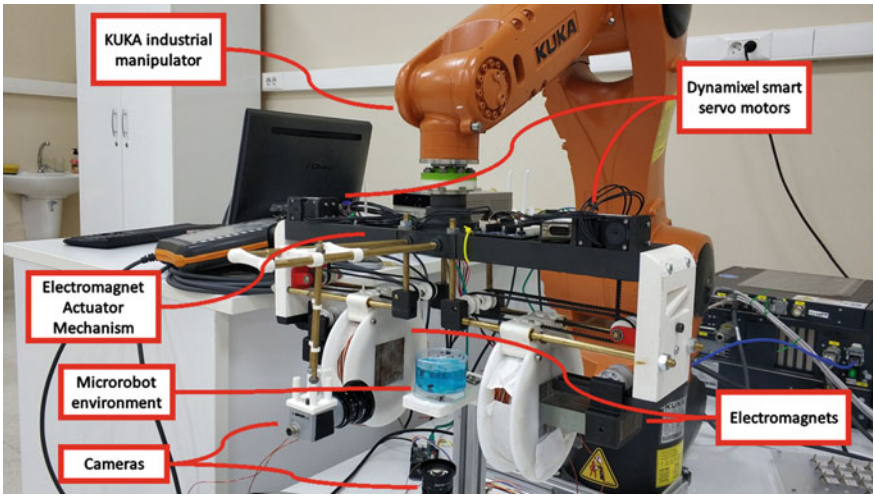


Fig. 1 General system figure

and linear motion mechanism. Electromagnets consists of a pair coaxially aligned facing to each other. When controlled currents flow through them, a resultant and controlled magnetic field occurs between them. Both electromagnets are controlled individually by current drivers. The other subsystem is linear motion mechanism. This component performs the linear motion of each electromagnets along the coaxial axis individually. Meanwhile the actuations are provided by smart servo motors, timing belt systems form the power train assembly. The linear motion mechanism determines the position of electromagnets by their controllers. Electromagnet structure and magnetic field based technical details can be reached in [22].

KUKA industrial robot (KUKA Agilus KR 6 R900 Sixx) is a 6-DoF industrial robot manipulator. It provides translation and rotation motion to EMA, which is attached onto the tip of KUKA. The robot can carry up to 6 kg in secure limits, providing satisfying reach lengths. Normally, the robot includes a controller computer and a Human Machine Interface (HMI) pendant. However, in this work an operator does not teach motions to robot. The industrial manipulator is controlled by a universal controller.

Microrobot is the small-scale robot component that is purposed to accomplish a task in the task space. This task space is admitted as a transparent grounded container filled with a transparent liquid. This liquid shows Newtonian fluid properties. Since the microrobot locomotion method is untethered magnetic actuation, then it should have magnetic properties. Soft and hard magnetic materials show the required needs for the application. Here, permanent magnet robot was preferred as microrobot material by the reason of increasing the force acting onto it and providing easier control. The small-scale robot can have different kinds of structures, basic or complex, for different purposes such as transportation and penetration. In these tests, mostly cylindrical microrobots were used. These permanent magnets can be used in raw or custom designed forms like coating it with different non-magnetic materials to give the shape of a capsule or pill. An example figure for mentioned microrobots is given in Fig. 2.

Visaul feedback system is needed where real-time control task requires microrobot position and orientation data. For this reason, two Basler ace cameras positioned orthogonally determines the position and orientation of the microrobot in 3D.

Further details can be reached in [22].

3.2 Methodology

Fundamentally, the microrobot locomotion is expressed with two general motion equations. The most popular ways to apply these motions are applied by the configurations called Helmholtz and Maxwell.

Fig. 2 A permanent magnet microrobot in cylindrical shape with 1 mm diameter and height

In a configuration that two equivalent electromagnets coaxially aligned facing to each other, when same amount of current flows through them in the same direction of the coaxial axis, magnetic field is exerted in the gap, called Helmholtz configuration. At the centre point of the gap on coaxial axis having the same distances to each electromagnet, the magnetic flux intensity shows homogenous behaviour in a limited region, called Region-of-Interest (ROI). If a particle is placed on this point, it rotates to align its magnetization vector to the magnetic flux lines. Supposing that the electromagnets rotate pivoting the centre point, the particle also rotates according to the same principle. This configuration will be called as symmetrical configuration and it can be used for particle rotation motion. Here, the torque vector $\boldsymbol{\tau}$ is exerted with the external magnetic flux intensity vector $\boldsymbol{B}$ onto the particle with the volume v and particle magnetization vector $\boldsymbol{M}$ as in (1).

$$\boldsymbol{\tau} = v\,(\boldsymbol{M} \times \boldsymbol{B}) \tag{1}$$

Assuming the same configuration, when one of the electromagnets is moved further away (the amount of motion distance is called asymmetry parameter ϵ), a magnetic field gradient occurs onto the particle instead of homogenous magnetic flux intensity. This gradient generates a force and the particle tends to move in the direction of gradient. This configuration will be called asymmetric configuration and it can be used for particle translation motion. Here the force $\boldsymbol{F}$ is exerted with external magnetic field gradient vector $\nabla\boldsymbol{B}$ onto the particle as in (2).

$$\boldsymbol{F} = v\,(\boldsymbol{M} \bullet \nabla)\,\boldsymbol{B} \tag{2}$$

The symmetrical and asymmetrical configuration magnetic field profiles were inspected in finite element methods (FEM) program and then compared with a mock-up model. This mock-up model simply consisted of two electromagnets attached onto carts with sliding rails and a gauss meter. The measurements were made for both symmetrical and asymmetrical configurations. The considered measurements comprised multiple times measuring in multiple points inside ROI by changing symmetry distances and asymmetry parameter with concerned values. The comparisons of FEM and test results were satisfying that the values were overlapping [22].

Using the asymmetric configuration, the particle acts under the force effect. If this motion is compensated with an outer manipulator and the relative particle position stands still referencing to the electromagnets providing same magnetic field gradient vector during the motion, it will proceed under a constant force.

Employing these motions with the system requires some control mechanisms.

- *Electromagnet current control:* The motion control principle considers DC currents with the same amount flowing along with each electromagnet. This motion approach does not require variant current values and if briefly explained, the electromagnets should be turned on at the beginning of the motion in the set current value, then turned off at the end of the motion. The set value should be adjustable, so a basic embedded control system should be adequate to set the current value and keep it steady during the motion.
- *Mechanism control:* As mentioned, force and torque generations are executed in symmetrical and asymmetrical configurations. For adjusting coil distances and transitions between configurations, the electromagnets should make translation motion along the coaxial axis. As rails and timing belt mechanisms provide the motion constraints and power train duty, each electromagnet has a Dynamixel smart servo motor for motion actuation.
- *Task space control:* The actuator system generates forces to the microrobot for rotation and translation. As microrobot rotation DoFs are provided by the EMA rotations, translation motion depends on the EMA translation in global coordinate frame. All those motions are provided by an external manipulator, KUKA. So that, the microrobot motions in task space are directly related with KUKA.
- *Task space feedback:* The microrobot position and orientation (coordinates) should be sensed in real-time for control feedback. Since the microrobot environment is transparent, using camera is a proper solution. The task space consists of three dimensions, so that at least two orthogonally positioned cameras are needed.

Considering these components and control requirements, ROS has the properties for resolve and integration. Powered by the firm-based library support and open-source SDKs, connection of elements in the same network is possible.

3.3 ROS Components and Network

The general ROS framework was built in a distributed package principle where the packages are run in different machines and communicate over network. The reason is that payload was high for a single-handed machine, especially for synchronous multiple image capturing and their post processes. This led to a structure that while one PC runs as Human Machine Interface (HMI), the others process as task space feedback and task space motion control computers.

NVIDIA Jetson TK1 provides good numerical computation power for embedded system applications thanks to its supercomputer architecture. Basically, three stations were planned with two NVIDIA Jetson TK1 and a regular PC in the system. As all the nodes were developed in modular way, basically, each nodes can be run on the desired machine, except HMI package. The monitoring, high-level controller and logging nodes takes place in PC with a monitor, keyboard and mouse hardware, distinct from the other machines. Rest of the packages perform image processing, robot manipulation, servo motor control and current driver controls where some of them require direct physical link to the actuator drivers. Here, one Jetson was attended for sensory feedback and the other one for direct-link required actuation purposes.

The communication network uses a router and all the devices are connected via Ethernet cable, forming a local network. It consists of one PC, two NVIDIA Jetson TK1s, one KUKA robot, one Arduino and two Dynamixel smart servo motors. The general network schematic is given in Fig. 3. The concept of using one PC and two Jetson TK1 is explained in Sect. 6. It was preferred that the default gateway of KUKA was set as its default IP number which was predefined in its factory settings. KUKA network configurations were set using as in `KUKAvarproxy` tutorial. In the next step, router network settings were configured. All the communication network

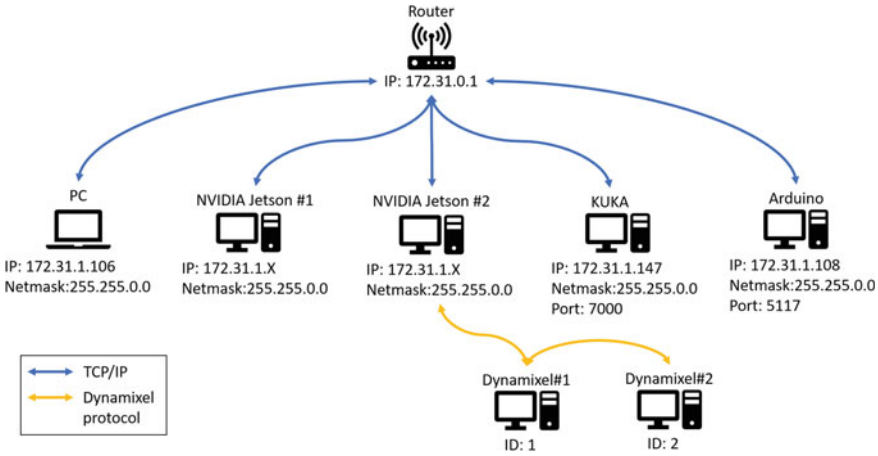


Fig. 3 Network schematic of the network links

use router IP as their gateway. The electromagnet current drivers use an Arduino microcontroller for control and the network connection of Arduino was established using an Ethernet shield. Moreover, Dynamixel servo motors use their protocol of own brand that is converted using a USB-to-Dynamixel converter adaptor hardware for connection to Jetson, providing a serial chain connection option.

In the microrobot locomotion concept, the microrobot is positioned inside a transparent container filled with a transparent liquid. The microrobot coordinates are needed in real-time for the control algorithms. For this reason, two cameras are used, one positioned at the bottom for detection in xy plane and the other camera is always facing parallelly to z axis, eventually position and orientation in 3D are detected.

4 ROS Environment Configuration

The program was organized where three ROS machines run individually on the same local network. Any machine may run the packages of a hardware component which is connected to it. All the nodes were programmed in modular way and it is also possible to run on any of machine and the package repository since the provided files include all of them.

The complete installation script and instructions are given in the following link:

- https://github.com/AytacKahveci/TEME.git

Installation script automatically creates workspaces, clones repositories and builds them. `ros_gui_ws` is optionally created with `build_gui` argument to the installation script. Included repositories are given below:

image_processing_ws:

- image processing https://github.com/AytacKahveci/image_processing.git

ros_control_ws:

- ros-control repo: https://github.com/AytacKahveci/ros_control.git
- ros-controllers repo: https://github.com/AytacKahveci/ros_controllers.git
- control-toolbox repo: https://github.com/AytacKahveci/control_toolbox.git

dynamixel_workbench:

- dynamixel-workbench repo: https://github.com/AytacKahveci/dynamixel-workbench.git

kuka_ws:

- KukaRosOpenCommunication repo: https://github.com/AytacKahveci/KukaRosOpenCommunication.git

ros_gui_ws:

- ros_gui repo: https://github.com/AytacKahveci/ros_gui.git

All of the repositories have $teme - devel$ branch. Developed packages are located in the $teme - devel$ branch.

Before these installation steps, the dependencies should be installed. The required ones are given below.

- https://github.com/ROBOTIS-GIT/DynamixelSDK, for `dynamixel_sdk`
- https://github.com/ROBOTIS-GIT/dynamixel-workbench, for `dynamixel_workbench`
- https://github.com/basler/pypylon, for `pylon_sdk`
- https://github.com/ros-perception/vision_opencv, for `cv_bridge`
- https://github.com/opencv/opencv, for `OpenCV`
- https://www.baslerweb.com, for `pylon 5`
- https://github.com/qt/qt5, for `qt5`

Due to the possible installation instruction changes of these repositories in the future, no steps are included in this chapter and it is recommended to apply as directed in related repositories. The program was developed on Ubuntu 16.04 for PC and 14.04 for NVIDIA Jetson TK1s. The developments were tested with ROS Kinetic in PC machine and ROS Indigo in Jetson machines.

To compile from the source, before cloning the files, catkin workspace should be already set up. If not yet, a catkin workspace should be set with:

```
$ cd ~/catkin_ws/src
```

After that, the repository can be cloned using the following command line. If other higher distributions are expected, switch `indigo-devel` with the related distribution commands. For each machine, the file name should be changed with the related one.

```
$ git clone https://github.com/AytacKahveci/TEME.git
    -b indigo-devel
```

The compilation is completed by the following commands:

```
$ cd ~/catkin_ws
$ catkin_make
```

5 Test of the System

In this section, a general test model is provided. A pseudocode algorithm is given covering the node running sequence and motion control. Working system example motions and some application videos are linked at the end of this section. Also, a specific application and its results are presented in Sect. 7.

The nodes in their related packages are run by *.launch* command. The provided nodes in the repository consist of several fundamental applications. In this testing

section, some specific nodes are run. The related nodes should be run with the following commands where the general notation is given with `roslaunch` command:

```
$ roslaunch <package_name> <file.launch>
```

KUKA

```
$ roslaunch kuka_hw_cart_vel test.launch
```

image_processing

```
$ roslaunch image_processing camera_node_front.launch
$ roslaunch image_processing camera_node_bottom.launch
```

Dynamixel

```
$ roslaunch dynamixel_custom_controller PosCurrentControl.launch
```

Coils

```
$ roslaunch coil_controller coilPidControl.launch
```

HMI (this requires loading $teme - perspective$ after running the command)

```
$ rqt
```

All the node communications are based on *topics*. As the considered nodes are run, the following topics should appear on the *rostopic list*:

```
/camera_node_bottom/image_processed
/camera_node_bottom/image_raw
/camera_node_bottom/image_raw_mouse_left
/camera_node_bottom/image_raw_object_roi
/camera_node_bottom/image_raw_roi
/camera_node_bottom/pose_estimate_2dV4/parameter_descriptions
/camera_node_bottom/pose_estimate_2dV4/parameter_updates
/camera_node_bottom/positionDist
/camera_node_front/image_processed
/camera_node_front/image_raw
/camera_node_front/image_raw_mouse_left
/camera_node_front/image_raw_object_roi
/camera_node_front/image_raw_roi
/camera_node_front/pose_estimate_2dV4/parameter_descriptions
/camera_node_front/pose_estimate_2dV4/parameter_updates
/camera_node_front/positionDist
/coil_pid_controller/CoilController/command
/coil_pid_controller/coil_states
/dynamixel/JointPosGroup/poseffCommand
/dynamixel/joint_states
/joint_position_controller/command
/joint_states
```

Some topics are feedback while others are command intended. The topic data is in *data array* form. The subscription outputs of feedback topics are given with example values:

```
/dynamixel/joint_states
header:
seq: 26441
stamp:
secs: 1559388844
nsecs: 747889857
frame_id: ''
name: ['joint_x', 'joint_y']
position: [2757.0, 3859.0]
velocity: [0.0, 0.0]
effort: [−3.0, −2.0]
```

All these topics share some common structures. *header* part includes topic sequence and time stamp. *frame_id* represents if a reference transformation exists. *data_array* consists of *name* and the rest. Here, Dynamixel motors named *joint_x* and *joint_y* publish their encoder data under the name of *position*. *velocity* and *effort* represent their speed and current.

```
/camera_node_bottom/positionDist
header:
seq: 3026
stamp:
secs: 1559278758
nsecs: 714981661
frame_id: /camera_node_bottom
pose:
position:
x: −0.027310336449
y: −0.0221896584277
z: 0.0
orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0
```

The microrobot position on *xy* plane determined by the bottom camera. This process is performed by some transformations. Raw image consists of pixel addresses and first transformed into length units by manual calibration process. Then, the transformed image is put to use of tracker algorithm for coordinate detection. *position* and *orientation* holds the related microrobot data.

```
/camera_node_front/positionDist
header:
seq: 150
stamp:
secs: 1559278758
nsecs: 914981661
frame_id: /camera_node_front
pose:
position:
x: −0.0273002072435
y: −0.0223365271723
z: 0.0
orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0
```

The microrobot position on yz plane is determined by the front camera. Its transformation process applies the same manners as the other camera does. However, this camera is not stationary and mounted on KUKA. When the actuation mechanism makes rotation, the image of microrobot container also rotates. An additional rotation transformation fixes the image coordinates as the pixel addresses pair with global z axis and z position becomes singularly bounded with one axis of the image.

```
/joint_states
header:
seq: 4493
stamp:
secs: 1559278758
nsecs: 714981661
frame_id: ''
name: ['A', 'B', 'C', 'X', 'Y', 'Z', 'vel_variable']
position: [90.0, −6.73470433e−15, 180.0, 30.0, 2.13162821e−14,
25.0, 1.7500864755892015e+190]
velocity: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 7.71274693504931e−43]
effort: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

joint_state topic provides the tooltip coordinates of KUKA as EMA is its end-effector. This requires some predefinitions in KUKA parameters that include global and local framework coordinates. *name* and *position* arrays pair with their matchings.

```
/coil_pid_controller/coil_states
header:
seq: 4493
stamp:
secs: 1559278758
nsecs:          814981661
frame_id: ''
name: ['coil_x', 'coil_y']
current: [2.01, 2.02]
direction: [0.0, 0.0]
```

Coil driver topics include feedback data as well as reference commands. PID controllers progress low level control inside current drivers. Even if the motion principle is applied using fixed currents, the controller provides stability especially due to parameter changes and disturbances. *name* and *current* arrays pair with the electromagnet couple.

The motion package includes some task files. Here are the brief explanations with the path root *image_processing/image_processing*:

object_positioning.cpp aligns end effector (EMA) center point with microrobot position on *xy* axis.
positioning4DOF.cpp applies the same aligning task referencing to microrobot coordinate *xyz* and α which is the orientation angle of the microrobot on *xy* plane with respect to *x* axis.
objectPolygonPath.cpp was especially developed for testing process. It manipulates the microrobot to follow a circular path.
objectRectanglePath.cpp similarly works applying a microrobot path in rectangle shape.
xy_control.cpp gives a goal target in *xy* plane and executes the motion.
yz_control.cpp similarly executes a movement in *yz* plane.
xyz_control.cpp targets a goal target having 5 DoF and executes the motion.

These task programs can be executed in desired order. After KUKA robot, Dynamixel motors and cameras are activated, those tasks are run in the terminal. A following motion sequence is given like below:

For a translation motion on xy axis, first align the mechanism centre and the microrobot.

```
$ rosrun image_processing object_positioning
```

After aligning, start the motion to target coordinates which are included in the related code file such as the given example below.

```
$ rosrun image_processing xy_control
```

If more briefly described using the user interface, a pseudo-code is given in Alg. 1 (the calibration process and some complex motion algorithms are ignored).

The presented files only include fundamental motion files which are provided on purpose to protect the privacy of some research intended files. An example video link of the system on work is given below:

- https://youtu.be/guE7Ul0bEps

Some working system output videos from the camera image streams are given below:

- https://youtu.be/0xM-JK4nbqs

Algorithm 1: Test application pseudo-code

```
Result: how to run a test application
run roscore;
run rqt and load program perspective;
run KUKA Launch;
run Coil Controller Launch;
run Dynamixel Launch;
set coil currents;
set image filtering using Dynamic Reconfiguration;
set roi regions from both bottom and front camera images;
set Log Section text boxes for logging process;
open terminal and run specified motion algorithm;
% from now on, these processes are done automatically;
while the motion is not completed do
    read feedback data;
    evaluate reference target;
    determine error using motion control node algorithm;
    if KUKA rotation or microrobot torque is needed then
        rotate KUKA;
    end
    if KUKA translation is needed then
        translate KUKA;
    end
    if microrobot force is needed then
        move Dynamixel for force configuration;
        if position error to target position is below the threshold distance value
        then
            decrease the asymmetry parameter inverse proportionately to the
            position error;
        end
    else
        move Dynamixel for torque configuration;
    end
end
```

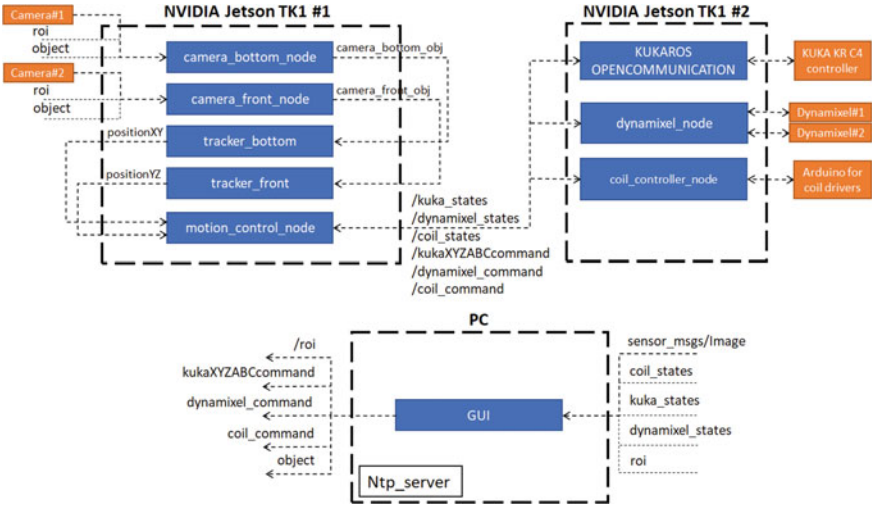


Fig. 4 Software architectures and their input/outputs

6 Description of ROS Packages and Environments

In this section, ROS packages, environments, nodes and algorithms are presented.

A general software architecture including the machines and nodes is given in Fig. 4. It should be noted that the namespaces and topics were simplified in the referred figure to ease the understanding and attract wide mass of readers. The hardware components are presented Sect. 3 if any assistance is needed. Besides, the structure is given in the cases that took part in test and development phases where one camera node and motion control nodes were run on PC to increase the development speed.

All the stations are connected to each other using a local Ethernet network using a hub switch. ROS network configuration requires some extra steps where PC acts as the ROS master while others operate as ROS client (see: http://wiki.ros.org/ROS/Tutorials/MultipleMachines). It should be reminded that in a ROS network, only one master machine can exist. Through the steps, all the clients are coded with master's specific IP address. If those steps are followed, after running the command `roscore` on terminal of the master machine, all other ROS clients share its workspace. Synchronizations between the master and slave machines are ensured with NTP Network Time Protocol. Except these, Arduino ROS integration was developed using the tutorials (see: http://wiki.ros.org/rosserial_arduino).

There are five packages developed or run in the program. These are:

- `dynamixel_custom_controller`
- `image_processing`
- `kuka_hw_cart_vel`
- `rqt`
- `coil_controller`

The package names mention about their functionalities. While the package `dynamixel_custom_controller` is used for Dynamixel motor control, `image_processing` is attended for both image processing and microrobot motion control, `kuka_hw_cart_vel` for KUKA robot control, `rqt` for HMI, and `coil_controller` for electromagnet current control. Except `rqt`, the packages were developed using libraries and support files which are either open-source or proprietary provided by the companies of system hardware. `rqt` is a developed plugin using ROS rqt framework.

The program can generate five working spaces. These are given as:

ros_control_ws workspace includes `ros_control` package (see: http://wiki.ros.org/ros_control) which generates the real-time control loop in ROS for the microrobot control operations. The control approach includes all motion feedbacks as inputs and actuator commands as outputs. The actuators consist of Dynamixel motors, coil current drivers and KUKA robot. In Dynamixel motor controls, the developments were based on the source code provided by the company. For coil current controls, a controller message type was generated. To control KUKA robot, `position_controllers\ JointPositionController` node inside the package `ros_control` was used (see: https://opensource.com/article/18/12/manage-ntp-chrony).

kuka_ws workspace development was inspired from the project `KukaRosOpenCommunication` (see: https://github.com/AytacKahveci/KukaRosOpenCommunication). An interface for KUKA was developed referencing from `OpenShowVar` (see: https://github.com/cyberpro4/openshowvar) and `JOpenShowVar` [23] (see: https://github.com/aauc-mechlab/JOpenShowVar) projects. The communication nodes subscribe and publish topics containing joint feedback and motion command data. `kuka_hw_cart` was developed that can receive reference in Cartesian form. The robot control was accomplished giving `XYZABC` command form.

image_processing_ws workspace includes image processing jobs. The progress purposes to locate the microrobot and track in real-time. For the detection operation, a HSV threshold is applied. Thus, the disturbances are extracted and only the microrobot object is situated inside the microrobot space whose dimensions are constant during the process. OpenCV 3.4 library was used in the development phases, where contribution repository was also downloaded and compiled. Besides, the microrobot motion planning phase for path following in various shapes such as helical, polygonal and circular was developed in this workspace. In microrobot workspace, a goal having up to 5 DoF micororobot motion can be referenced.

dynamixel_ws workspace was developed using the fundamental ROS package for Dynamixel motors provided by Robotis on `Dynamixel SDK` (see: http://wiki.ros.org/dynamixel_workbench). Some modifications were held on this package by first cloning from Git with a different branch. A controller was developed for the current-

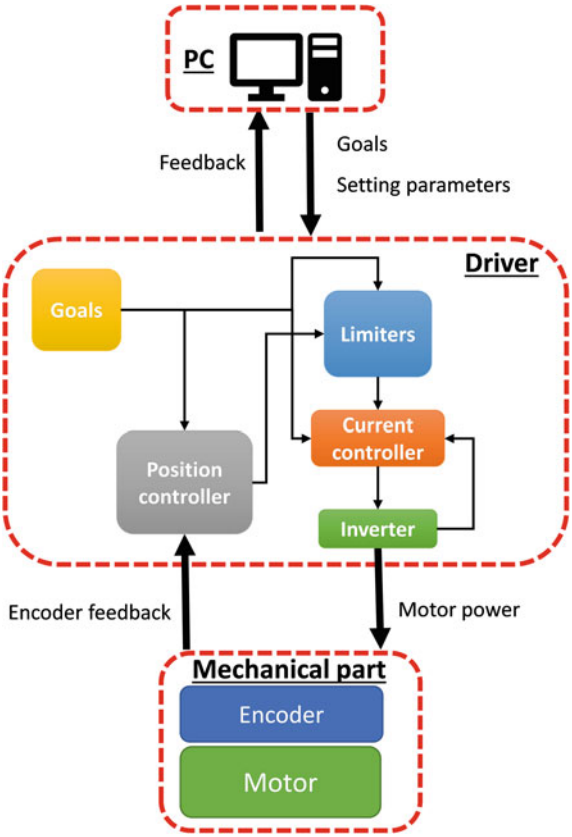


Fig. 5 Dynamixel motor current-based position control mode schematic

based position control mode (see: http://support.robotis.com/en/product/actuator/dynamixel_x/xm_series/xm430-w350.htm). Also, its internal control figure is given in Fig. 5.

coil_controller is not technically a workspace, but can take part here for the relativity with its distinct categorization of the workspaces. This was developed in `dynamixel_workbench`. Its main assignment is the communication with TCP/IP protocol and it involves PID controllers.

ros_gui_ws was designed using `rqt` plug-ins. It is the user interface for human machine interaction, providing an easy control for the user. The interface design is given in Fig. 6.

The user interface consists of five parts: Image view, coil controller, launcher, dynamic reconfigure and log section parts. These were designed and programmed using in `rqt` (see: http://wiki.ros.org/rqt). The parts are called as plug-ins and placed into the main window as desired. The present plug-in parts are briefly like this:

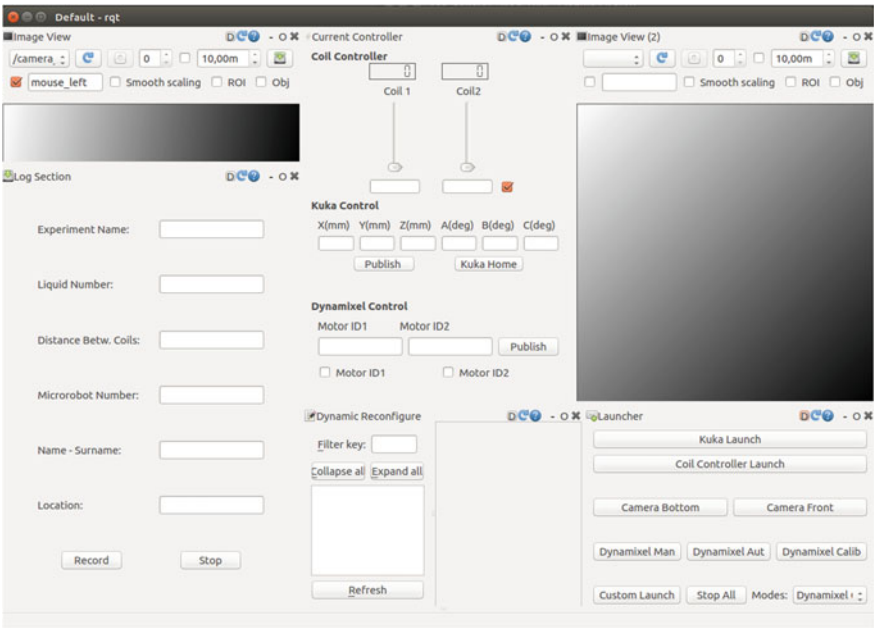


Fig. 6 HMI interface of the system

- **Image view** includes two camera views, one from bottom and the other from front camera that are positioned orthogonally. This part was developed on `rqt_image_view` (see: http://wiki.ros.org/rqt_image_view). In the panel, combo boxes are used for selecting `roi` and `object`. To define the `roi`, ROI combo box is ticked and then a rectangle is marked by the user. Same processes are applied for defining the `object`.
- **Coil controller** provides the current references for both electromagnets and display the feedback sensed by the current control drivers.
- **Launcher** simplifies the start-up of programs.The required launcher files are run by the interface buttons instead of running them on their terminal by commands.
- **Dynamic reconfiguration** provides the HSV threshold adjustments for the image processing parts. The adjustment values can be applied dynamically. This package was not developed but used directly.
- **Log Section** starts recording the streaming video data received from both cameras. Those videos consist of raw video files and video files with overlapping microrobot path trails. Also, logging process logs various kinds of topic data in *.bag* files. The logged data includes of microrobot coordinates determined by bottom and front cameras, KUKA pose, Dynamixel motor commands and coordinates, electromagnet currents, microrobot pose and target pose (if given).

The fundamental nodes are briefly described as follows:

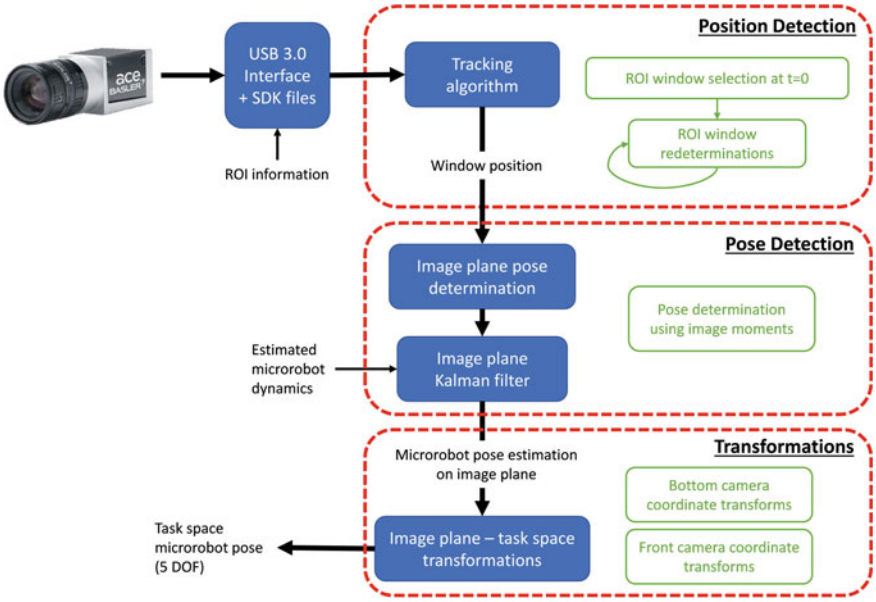


Fig. 7 Image processing block diagram

`motion_control_node` node serves for all the motion planning duties. It subscribes all the feedbacks, receives the commands from the user via GUI and publishes commands to the actuator driver nodes that are linked to the actuators in bidirectional communication manner.

KUKA Agilus KR 6 R900 Sixx industrial manipulator with KCR4 Compact controller running Windows 7 operating system has its individual controller PC provided by the company. This machine does not relay sensor messages to outside network or take any command externally in its factory settings. KUKA brand provides a paid add-on package named `KUKA.RobotSensorInterface` for these mentioned purposes. Instead, a KUKA package for ROS developers was formed. `KUKAvarproxy` is an open-source TCP/IP communication protocol for cross platforms in KUKA applications, which is also acknowledged as a standard by the developers. This protocol was implemented and a node called `KukaRosCommunication` was developed which participates in variable assignments, KUKA command message subscriptions and robot feedback publishs. The topic called `kuka_states` subscribes the feedback data from KUKA and publishes the topic called `kukaXYZABCcommand` which is used for commanding to KUKA. More clearly, it publishes the position and orientation data that KUKA should move in real-time.

Two Basler ace cameras play a significant role in positioning and control. The object tracking phase relates to two topics in the process, `roi` and `object`. As the camera raw images contain very large redundant data, a cropped window, called `roi`, coordinates are given for both cameras, just involving the interested view area. Raw

and `roi` streams of both cameras are published to PC GUI. The image streams are also subscribed by the tracker nodes for detection of object coordinates. The trackers publish the microrobot coordinates for GUI and the controller nodes. Tracker algorithms comes in handy since most of them deals with only the object coordinates and its neighbours for fast and efficient processing. Some of the OpenCV object tracking algorithms are BOOSTING, MIL, KCF, CSRT, MedianFlow, TLD, MOSSE and GOTURN. Since these algorithms have pros and cons depending on the application types, BOOSTING algorithm showed better performance. The orientation detection algorithm was developed using the object characteristics such as moment of inertia and centre of gravity. At the beginning of process, the object is located by the user and then its coordinates are updated in `object` topic in real-time, publishing it to the environment. Also, it should be noted that usage of `roi` is an optional feature. Image processing phases are briefly shown by block diagrams in Fig. 7.

The node working frequencies can be set to any desired values with the constraints of operation load and machine hardware. Specifically speaking, in these microrobot applications, the motion speed is less than 1 cm/s and using trial and error approach, the node frequency for KUKA feedback was set to 10 Hz, for camera feedbacks to 5 Hz and coil controller to 10 Hz.

7 Application and Results

This section includes an academic application using the presented system. For more applications and technical details, one can examine the works in [22, 24, 25].

The microrobot was picked as a Neodymium permanent magnet in a cylindrical shape with a diameter of 1 mm and height of 2 mm. This magnet is embedded into a capsule with 1.25 mm diameter and 5 mm height. The viscosity parameter of the liquid was measured 169 mPa.s and it shows Newtonian characteristics. Also the Reynolds number is less than 1.

In the application, as a 3D motion is established, the microrobot position starts from (0, 0, 0) coordinates to (4, 10, 3.5) using two consequent reference target points. In the liquid filled container, the robot makes a rising motion with the angles of 11.3° and 26.5°, totally 19.3° referencing to the starting point. On the planar orientation, the angle is constant as 68°. In torque configuration the distance between electromagnets is 12 cm, and in force configuration the asymmetry parameter is 2 cm while the current values are set at 2 A. The results are given in Fig. 8.

The results showed that the orientation changes of electromagnet pair were followed by the microrobot with less than a few degrees errors. The asymmetry configuration could provide the translation motion in desired direction. During the first half, the motion speed was around 0.144 mm/s and the second half was 0.237 mm/s. This difference is thought to be occurred due to the microrobot contact with floor in the first part. The error between robotic arm and microrobot positions showed a plot with an average of 0.043 mm while the positioning error in microrobot task space was 0.116 mm. It is thought, even if the perpendicular disturbances cause position

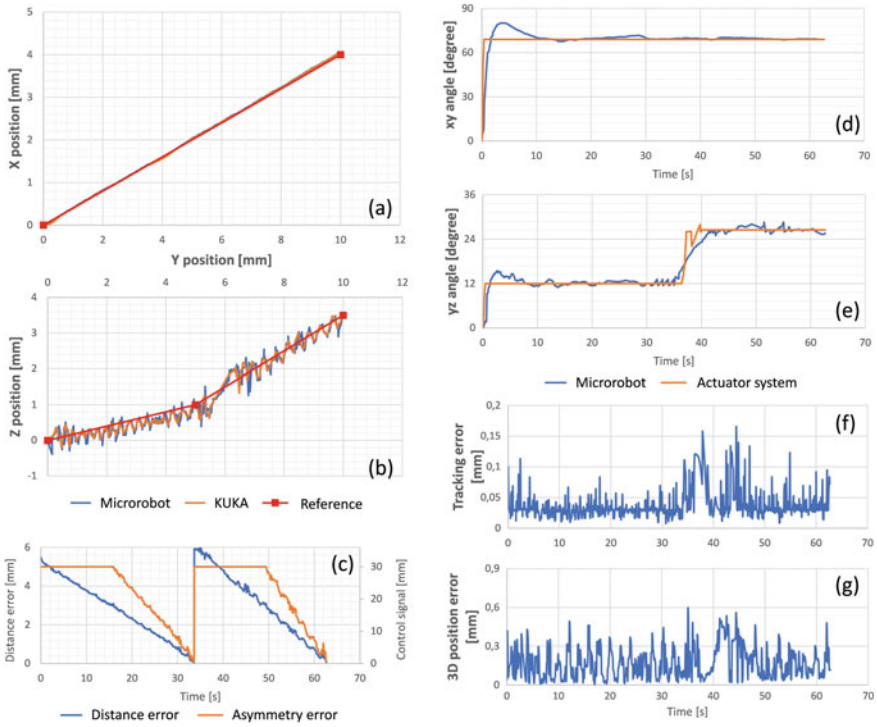


Fig. 8 Microrobot motion in 3D starting from (0,0,0) to (4,10,3.5) with two reference point changes. (a,b) positions on plane during the motion, (c) microrobot task space position controller parameters, (d,e) microrobot task space orientation controller parameters, (f) robot arm-microrobot tracking error, (g) microrobot positioning error in time domain

error, there is a serious contribution of the tracker error by reason of geometrical shape changes in visual.

These results showed that the system is capable of microrobot locomotion inside a task space.

8 Conclusion

In this work, an electromagnetic actuator system for untethered microrobot manipulation applications was developed. It consists of a magnetic actuator system and a KUKA industrial manipulator. It is also supported by Dynamixel smart servo motor, Arduino controlled current driver and Basler ace camera components. This custom-designed system provides a programmable task space while the magnetic field computation complexity is converted into a simple mechanical motion problem while

decreasing the number of electromagnets to two. This system was integrated using ROS.

Developed ROS architecture consists of multiple machines that run ROS at the same time sharing the same ROS network. The developments were run on one PC and two NVIDIA Jetson TK1 embedded development kits. These machines are physically connected to a hub switch forming a local network. Also, the development was made in distributed manner and the user may prefer to run any of the packages (except HMI) on any computer.

There are some dependencies that should be considered. Some of them are provided by the related hardware brands and some are open source dependencies like `OpenCV`. During the development of project, the inspiration sources with the related parts were explained briefly.

For the beginners, using the provided source files, a brief test exercise is explained since it is not an easy task to provide these specific application environment for practice.

An academic application in 3D motion is explained and its results were given. It is concluded that the desired motion goals are reached with acceptable errors that is mostly formed of tracker algorithm error. Also, some other works are cited for further applications.

The main purpose is to show an implementation roadmap for the untethered microrobot manipulation or alike systems using ROS. ROS implementation helps to put together all the system components in middleware level. It is thought that this solution satisfies most of the expectations during this construction process. Even if this application and its components are specific, helpful references are cited and some considered key points are explained. This concept is described in a way to attract other reader who does use any of the components that are used here. Even the system goes into more complex situations as increasing the number of components or giving a highly compelling algorithm in software level, it is believed to be received its convenient support for a long while.

Acknowledgements This work was supported by the Scientific and Technological Research Council of Turkey (TUBITAK, project No. 215M879).

References

1. X.Z. Chen, M. Hoop, F. Mushtaq, E. Siringil, C. Hu, B.J. Nelson, S. Pane, Recent developments in magnetically driven micro-and nanorobots. Appl. Mater. Today **9**, 37–48 (2017)
2. S. Palagi, P. Fischer, Bioinspired microrobots. Nat. Rev. Mater. **1** (2018)
3. H. Ceylan, J. Giltinan, K. Kozielski, M. Sitti, Mobile microrobots for bioengineering applications. Lab Chip **17**(10), 1705–1724 (2017)
4. B.J. Nelson, I.K. Kaliakatsos, J.J. Abbott, Microrobots for minimally invasive medicine. Ann. Rev. Biomed. Eng. **12**, 55–85 (2010)
5. F. Carpi, Magnetic capsule endoscopy: the future is around the corner. Expert Rev. Med. Dev. **7**(2), 161–164 (2010)

6. R. Eliakim, A. Suissa, K. Yassin, D. Katz, D. Fischer, Wireless capsule video endoscopy compared to barium follow-through and computerised tomography in patients with suspected crohn's disease. Digest. Liver Dis. **36**(8), 519–522 (2004)
7. K.E. Peyer, L. Zhang, B.J. Nelson, Bio-inspired magnetic swimming microrobots for biomedical applications. Nanoscale **5**(4), 1259–1272 (2013)
8. J.J. Abbott, K.E. Peyer, M.C. Lagomarsino, L. Zhang, L. Dong, I.K. Kaliakatsos, B.J. Nelson, How should microrobots swim? Int. J. Robot. Res. **28**(11–12), 1434–1447 (2009)
9. K.B. Yesin, K. Vollmers, B.J. Nelson, Modeling and control of untethered biomicrorobots in a fluidic environment using electromagnetic fields. Int. J. Robot. Res. **25**(5–6), 527–536 (2006)
10. S. Martel, J.B. Mathieu, O. Felfoul, A. Chanu, E. Aboussouan, S. Tamaz, P. Pouponneau, L. Yahia, G. Beaudoin, G. Soulez et al., Automatic navigation of an untethered device in the artery of a living animal using a conventional clinical magnetic resonance imaging system. Appl. Phys. Lett. **90**(11), 114105 (2007)
11. F. Ullrich, C. Bergeles, J. Pokki, O. Ergeneman, S. Erni, G. Chatzipirpiridis, S. Pané, C. Framme, B.J. Nelson, Mobility experiments with microrobots for minimally invasive intraocular surgery. Invest. Ophthalmol. Vis. Sci. **54**(4), 2853–2863 (2013)
12. J. Choi, H. Choi, K. Cha, J.O. Park, S. Park, Two-dimensional locomotive permanent magnet using electromagnetic actuation system with two pairs stationary coils, in *2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)* (IEEE, 2009), pp. 1166–1171
13. G. Go, H. Choi, S. Jeong, C. Lee, S.Y. Ko, J.O. Park, S. Park, Electromagnetic navigation system using simple coil structure (4 coils) for 3-d locomotive microrobot. IEEE Trans. Mag. **51**(4), 1–7 (2014)
14. M.P. Kummer, J.J. Abbott, B.E. Kratochvil, R. Borer, A. Sengul, B.J. Nelson, Octomag: an electromagnetic system for 5-dof wireless micromanipulation. IEEE Trans. Robot. **26**(6), 1006–1017 (2010)
15. S. Schuerle, S. Erni, M. Flink, B.E. Kratochvil, B.J. Nelson, Three-dimensional magnetic manipulation of micro-and nanostructures for applications in life sciences. IEEE Trans. Mag. **49**(1), 321–330 (2012)
16. E. Diller, J. Giltinan, M. Sitti, Independent control of multiple magnetic microrobots in three dimensions. Int. J. Robot. Res. **32**(5), 614–631 (2013)
17. A.W. Mahoney, J.J. Abbott, Control of untethered magnetically actuated tools with localization uncertainty using a rotating permanent magnet, in *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)* (IEEE, 2012), pp. 1632–1637
18. A.W. Mahoney, N.D. Nelson, K.E. Peyer, B.J. Nelson, J.J. Abbott, Behavior of rotating magnetic microrobots above the step-out frequency with application to control of multi-microrobot systems. Appl. Phys. Lett. **104**(14), 144101 (2014)
19. A.W. Mahoney, J.J. Abbott, Five-degree-of-freedom manipulation of an untethered magnetic device in fluid using a single permanent magnet with application in stomach capsule endoscopy. Int. J. Robot. Res. **35**(1–3), 129–147 (2016)
20. S.E. Wright, A.W. Mahoney, K.M. Popek, J.J. Abbott, The spherical-actuator-magnet manipulator: A permanent-magnet robotic end-effector. IEEE Trans. Robot. **33**(5), 1013–1024 (2017)
21. X. Duan, S. Zhang, G. Xiao, X. Wang, J. Wang, Apparatus and method for controlling movement of a capsule endoscope in digestive tract of a human body (June 5 2018) US Patent 9,986,898
22. A. Alasli, L. Çetin, N. Akçura, A. Kahveci, F.C. Can, Özgür Tamer: Electromagnet design for untethered actuation system mounted on robotic manipulator. Sens. Actuat. A: Phys. **285**, 550–565 (2019)
23. F. Sanfilippo, L.I. Hatledal, H. Zhang, M. Fago, K.Y. Pettersen, Jopenshowvar: An open-source cross-platform communication interface to kuka robots, in *2014 IEEE International Conference on Information and Automation (ICIA)* (2014), pp. 1154–1159
24. N. Akçura, L. Çetin, A. Kahveci, A.K. Alasli, F.C. Can, O. Tamer, Guided motion control methodology for microrobots, in *2018 6th International Conference on Control Engineering Information Technology (CEIT)* (2018), pp. 1–4

25. N. Akçura, A. Kahveci, L. Çetin, A. Alasli, F.C. Can, E. Gezgin, Ö. Tamer, Close-loop control of microrobot within a constrained environment using electromagnet pairs, in *Interactive Collaborative Robotics*, ed. by A. Ronzhin, G. Rigoll, R. Meshcheryakov (Springer International Publishing, Cham, 2019), pp. 1–9

Aytaç Kahveci graduated in Mechatronics Eng. Dept. at Marmara University in 2016 and received his master's degree in Mechanical Engineering in Izmir Katip Çelebi University, Turkey in 2019. He is currently working on robotics, industrial automation, mobile robot navigation and control systems.

Nail Akçura graduated in Mechatronics Eng. Dept. at Kocaeli University, Turkey, received his M.Sc. in Ege University in 2015 and he is currently a Ph.D. student in Mechatronics Eng. Dept at Dokuz Eylül University in Turkey. His research interests are in mechatronics, robotics and automatic control.

Levent Çetin received his Ph.D. at Dokuz Eylul University Mechanical Eng. Dept in 2008. He is currently working as an associate professor at Izmir Katip Çelebi University Mechatronics Eng. Dept. His main research interests are mechatronics design and control theory applications

Abdulkareem Alasli received his M.Sc. in Mechanical Engineering in 2018 from Dokuz Eylül University, Turkey and his B.Eng. in 2012 from Damascus University, Syria. He is currently pursuing his Ph.D. in the Mechanical System Eng. Dept. in Nagoya University, Japan. His research interests mainly focus on actuating micro/nano robots by electromagnetic fields, heat transfer mechanisms in nanofluids and spin caloritronics.

Özgür Tamer received his B.Eng., M.Eng. in and Ph.D. at Dokuz Eylul University Electrical and Electronics Eng. Dept in 1997, 2001 and 2008 respectively. He is currently working as an assistant professor at Dokuz Eylul University Electrical and Electronics Eng. Dept. His main research interests are focused on smart antenna systems, embedded systems and wireless sensor networks.

ClegS: A Meta-Package to Develop C-Legged Robots

Jorge De León, Raúl Cebolla, Jaime del Cerro, and Antonio Barrientos

Abstract In this chapter we present a tutorial for ClegS, a ROS meta-package for the development of C-legged robots. This is the first meta-package in ROS dedicated to develop robots with this special legs, and allows the user to experiment with different morphologies of these C-legged robots (quadruped, hexapod, octapod). ClegS has been developed to be plug-and-play to the user, he only has to create his walking mode algorithm and the ClegS sets the controllers to realize the task with the best performance. ClegS is portable and is ready to control simulated and real robots. The chapter will explain in detail how the user can choose among different generic models of robots that are included in the meta-package or create new robots. The chapter also describes the software architecture of ClegS and suggest further levels of customization for experienced ROS users. Finally, the advantages and potential of ClegS are demonstrated in a case of study which a C-legged robot is designed and deployed from scratch.

Keywords Legged robots · Robot simulation · ROS for legged robots · C-legged robots

1 Introduction

Mobile robots based on legs as ground locomotion systems requires specific motion coordination of each leg to develop consistent movement of the robot. The coordination of these movements results in a gait pattern (GP), which enables the robot to move on the environment in a specific mode.

This kind of robots also requires different GPs to move in different terrains or perform actions such as jumping or avoiding certain obstacles. As a consequence, managing and coordinating the different GPs implemented on a legged robot turns into a critical topic in the development and research of this type of mobile robots.

J. De León · R. Cebolla (✉) · J. del Cerro · A. Barrientos
Centro De Automática y Robótica, Universidad Politécnica de Madrid,
Calle José Gutiérrez Abascal, 2, 28006 Madrid, Spain
e-mail: raul.cebolla.arroyo@alumnos.upm.es

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_10

Another associate problem regarding GPs is its definition, implementation and debugging. Almost every GP implemented on a robot was deeply tested beforehand, but the code running on each situation is different as commands for actuators and simulation are not the same on most occasions causing translation errors when coding on each case.

In this section the ClegS package collection will be presented to overcome these issues, how to manage the commands and GPs on a c-legged robot and providing a GP C++ programming API for ROS which enables to use the same code for both simulation and hardware.

Design and coding time consumption is outstandingly reduced with this package collection preventing researchers from spending time in unproductive tasks. Thus, this work aims to establish a common framework for GP definition and programming, encouraging the design and development of new GPs that enables a robot to walk, run, jump or stand on various environments or even perform completely new tasks.

This package has been developed for the CLHeRo robot, a robot with 6 "C" shaped legs developed in the Technical University of Madrid. The special leg configuration turns this mobile robot into a great explorer in uneven grounds and unstructured environments.

The ClegS package collection may be found in the following repository:

```
$ git clone https://github.com/grafoeka/clegs_common.git
$ git clone https://github.com/grafoteka/clegs_sim.git
$ git clone https://github.com/grafoteka/clegs_hw.git
```

The rest of the chapter is organized as follows. In Sect. 2 a background for the CLHeRo robot will be presented, describing the kinematics, dynamics and robot's model required to define the GPs. The control strategy for specific movements is gathered in Sect. 3 which is followed by the ROS environment setup in Sect. 4. Section 5 presents the ClegS packages involved, describing their functionality and features along with explanation on how these can be used. Finally, Sect. 6 gathers tutorials for different tasks involving the CLHeRo and ClegS package collection.

2 Background

2.1 Modelling Multilegged Robots

The objective of this first section is to teach the reader how is the behavior of C-legged robots doing a study of the kinematic and dynamic of this kind of robot. Before starting the explain the kinematic, it is necessary to assume the next statements:

- The robot must have an even number of legs.
- Each couple of legs have to be faced and symmetrical respect to the XZ plane and aligned through an axle parallel to the Y axle. This is referring to the system reference of the robot, see Fig. 1.

– Each leg only have one degree of freedom (DOF), which is a rotation through the Y axle referred to the body frame.
– All the legs have the same mechanics properties of elasticity and the "C" shape.

If the robot satisfies the requirements above, then is possible to apply the study of Raibert [11] and we can assert that the displacement of the robot on the horizontal plane is described by the unicycle model with dynamic extension.

To have a better comprehension of the next sections, we will use the CLHeRo V2 robot for all the examples. The robot CLHeRo V2 is explain in detail in the Sect. 2.5, but in a few words, CLHeRo V2 is a hexapod robot with C-legs and meets all the restrictions discussed at the beginning of this section.

2.2 Kinematics

For any mobile robot than can move in the 6 DOFs, it needs 6 coordinates to determine its position and orientation (Table 1). The 3 first coordinates (x, y and z), and their derivatives with respect to time, correspond to the position and translation respect the X, Y and Z axles. On the other hand, the last 3 coordinates (ϕ, θ and ψ), and their derivatives with respect to time, describe the orientation and rotational movement of the robot.

When designing control systems with feedback for mobile robots, it is possible to choose different systems that can give as result a system for acting with 1 DOF, 3 DOF, 4 DOF or 6 DOF. In this study we will focus in the systems with 3 and 6 DOFs.

The 3 DOFs systems represent the model in the horizontal plane (translation in the X and Y axles and rotation on ψ angle) and are used in system for dynamic

Table 1 Notion for the mobile robots systems

DOF	Description	Forces and moments	Linear and angular velocities	Positions and Euler angles
1	Motion in the X direction	X	u	x
2	Motion in the Y direction	Y	v	y
3	Motion in the Z direction	Z	w	z
4	Rotation about the X axis	K	p	ϕ
5	Rotation about the Y axis	M	q	θ
6	Rotation about the Z axis	N	r	ψ

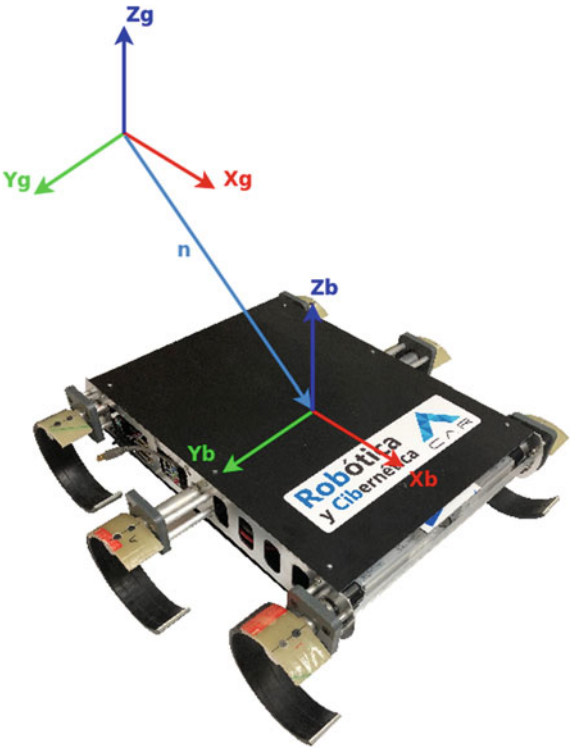


Fig. 1 Global (G) and Body-fixed (B) Coordinate Frames for CLHeRo V2

positioning and navigation and guidance systems. For its part, the 6 DOFs systems are used for the simulation and estimation of the displacement of the robots.

The kinematic equations for the CLHeRo V2 are developed using a global reference system (G) and a reference system fixed to the body of the robot (B), see Fig. 1. With the description of the Fig. 1, the X_b^+ axle represents the positive translation of the robot and the Y_b^+ axle position the displacement to the right of the robot.

The position and orientation vector that sets the position of the robot respect to the world (G) is defined as:

$$\eta = [x, y, z, \phi, \theta, \psi]^T \tag{1}$$

where x, y, z are the position coordinates of the robot, and ϕ, θ, ψ the Euler angles for the orientation. With this vector, now is possible to define the velocity vector refer to the robot frame (B) as:

$$\upsilon = [u, v, w, p, q, r]^T \tag{2}$$

where u, v, w are the linear velocity components and p, q, r the angular velocity components.

With this 2 vectors defined, now we can use the vectorial form to express the equations of the system, so the reader could understand better the physical properties of the models. One advantage of the vectorial form is the possibility of reduce the number of coefficients necessaries to control the robot [5]. Also, is possible to solve the system with truncated Taylor-series expansions [1], but that system will not be employed in this explication.

Thus, the kinematic equations of the robot are expressed in vectorial form as:

$$\dot{\eta} = J(\Theta)\upsilon \tag{3}$$

where $\Theta = [\phi, \theta, \psi]^T$ and the transformation matrix $J(\Theta) \in R^{6x6}$ is

$$J(\Theta) = \begin{bmatrix} R_B^G(\Theta) & 0_{3x3} \\ 0_{3x3} & T(\Theta) \end{bmatrix} \tag{4}$$

with $R_B^G(\Theta)$ and $T(\Theta)$ the linear and the angular velocity transformation matrices from G to B frame, respectively.

$$R_B^G(\Theta) = \begin{bmatrix} c(\gamma)c(\pi) & -sin(\gamma)c(\rho) + c(\gamma)s(\pi)s(\rho) & s(\gamma)s() + c(\gamma)c()s(\pi) \\ s(\gamma)c(\pi) & c(\gamma)c(\rho) + s(\gamma)s(\pi)s(\rho) & -c(\gamma)s(\rho) + s(\gamma)c(\rho)s(\pi) \\ -s(\pi) & c(\pi)s(\rho) & c(\pi)c(\rho) \end{bmatrix}$$

$$T(\Theta) = \begin{bmatrix} 1 & s(\rho)t(\pi) & c(\rho)t(\pi) \\ 0 & c(\rho) & -s(\rho) \\ 0 & s(\rho)/c(\pi) & c(\rho)/c(\pi) \end{bmatrix}$$

where: $s(\alpha) \rightarrow sin(\alpha)$, $c(\alpha) \rightarrow cos(\alpha)$ and $t(\alpha) \rightarrow \tan(\alpha)$.

2.3 Leg Modeling

This section will present a simplified approach to the model of the "C" legs of the CLHeRo V2.

The aim of this section is to explain the effect that legs produce in the robot displacement, instead of develop a complex and very detailed model of one of each legs. To be able to derive the dynamic equations that describe the movement of the CLHeRo V2, we will take in consideration the model of contact forces between the leg and the ground. This is only possible if we use the Newton-Euler formulation, which is based on Newton's second law, which relates mass m, acceleration $\dot{\mathbf{v}}_{G/B}$ and force $\mathbf{f}_G$ according to:

$$m\dot{\mathbf{v}}_{G/B} = \mathbf{f}_G \tag{5}$$

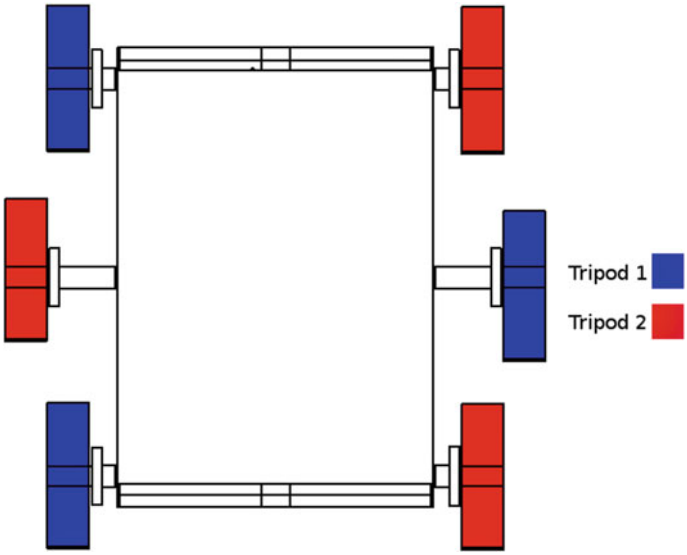


Fig. 2 Tripods for the alternating tripod gait

As was mentioned before and will be described in Sect. 2.5, the CLHeRo robot is a hexapod platform with C-legs, see Fig. 1. The CLHeRo V2 can walk, run, turn in place and turn with radius of curvature, using in each occasion the most efficient GP. For the description of this section, we will assume that the robot is going to move along the X^+ axes and with the alternating tripod pattern.

The alternating tripod gait, is a GP bio-inspired in the cockroaches locomotion [14]. This GP for hexapods needs to divide the 6 legs in 2 subgroups with 3 legs each one (Fig. 2), each tripod includes the two legs of the ends of the same side and the intermediate one of the opposite side. This GP establishes that in every step, one tripod is in contact with the ground, while the other one moves in the air. The tripod that is in contact with the ground is the one that provides the stability to the robot.

Each leg of the robot describes its movement using a spherical coordinate frame $[l_i, \phi_i, \theta_i]^T$ with origin in the point a_i, where a_i represents the translation from the origin of the leg respect the coordinate frame of the robot (B), as you can see in Fig. 3.

Each leg can only move in the ϕ_i revolution DOF and presents a passive flexion from the l_i and θ_i DOFs. This passive flexion is produced due to the mechanical designed in the C shaped leg. Because of this, all the legs can be simplified as a massless linear spring with initial length l_{0i} and stiffness k_i.

Therefore, the movement of the leg can be described as a circle respect to the plane XZ of the frame of the robot.

When a leg i is in contact with the ground, it experiences the ground reaction forces $(F_{x_i}, F_{y_i}, F_{z_i})$ expressed in the frame of the robot, see Fig. 4. The result of the

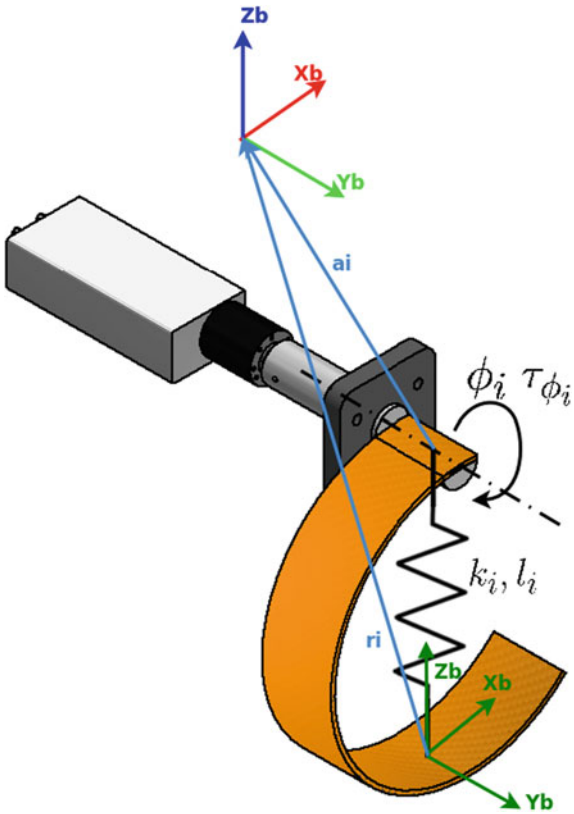


Fig. 3 Modelling of the legs forces

ground forces in the leg i is $F_{a_i} = \frac{\tau_{0_i}}{l_i}$, where τ_{0_i} is the torque of the motor in the DOF ϕ_i and l_i is the length of the spring.

Because of how is the leg fix to the chassis of the robot, we can assert that the lateral force, F_{y_i}, of the robot can be considered as negligible ($F_{y_i} = 0$). Thus, the leg i experiences the ground force with only two components, where $||F_{G_i}|| = ||F_{A_i}||$ (Fig. 4).

Consequently, the forces that the robot experiences are:

- The ground reaction forces and the spring forces for each leg that is in contact with the ground (Fig. 4).
- Gravitational forces and moments.

The spring forces and the gravitational forces can be expressed in the frame of the robot using the next transforms,

$$F_i^B = \begin{bmatrix} F_{x_i} \\ F_{y_i} \\ F_{z_i} \end{bmatrix} = \begin{bmatrix} -sin(\phi_i) & -cos(\phi_i) \\ 0 & 0 \\ -cos(\phi_i) & sin(\phi_i) \end{bmatrix} \begin{bmatrix} F_{S_i} \\ F_{G_i} \end{bmatrix} \tag{6}$$

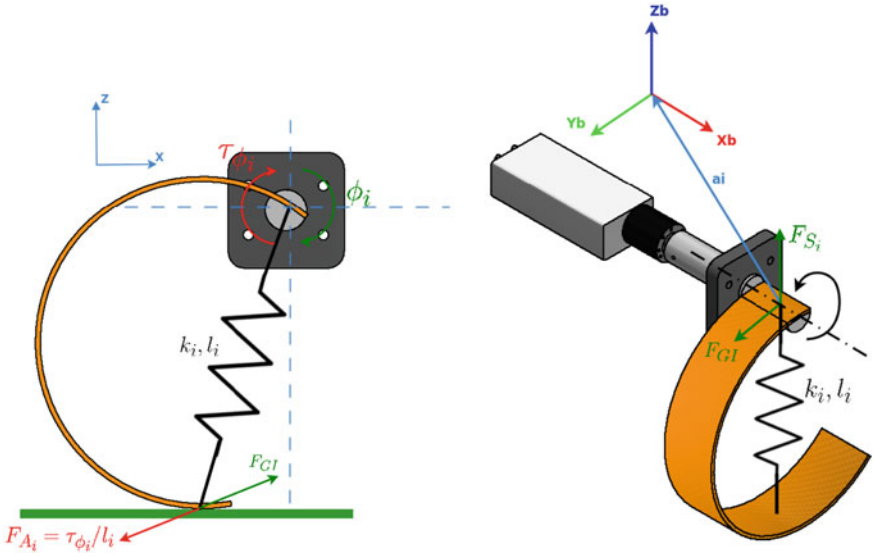


Fig. 4 Leg Forces on the XZ^B plane

where $F_{S_i} = -K_i(l_i - l_{0_i})$ y $F_{G_i} = \frac{\tau_{\phi_i}}{l_i}$, therefore, the vector F_i^B of the forces acting on the leg respect to the frame of the robot are:

$$F_i^B = \begin{bmatrix} F_{x_i} \\ F_{y_i} \\ F_{z_i} \end{bmatrix}^B = \sum_{i=1}^{6} leg_i \cdot F_i^B \tag{7}$$

And the resulting vector of the moments with respect to the center of gravity of the robot is:

$$\tau^B = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix}^B = \sum_{i=1}^{6} leg_i \cdot \tau_i^B \tag{8}$$

where

$$leg_i = \begin{cases} 1 = Leg\ i\ ground \\ 0 = Leg\ i\ air \end{cases} \tag{9}$$

The principle of moments describes that $\tau_i^B = \alpha_i \times F_i^B$. And if we apply the symmetric Skew matrix, is possible to obtain the resultant force in each leg axis of the robot from any force that is applied.

$$\lambda \times F := S(\lambda)F = \begin{bmatrix} 0 & -\lambda_3 & \lambda_2 \\ \lambda_3 & 0 & -\lambda_1 \\ -\lambda_2 & \lambda_1 & 0 \end{bmatrix} F \tag{10}$$

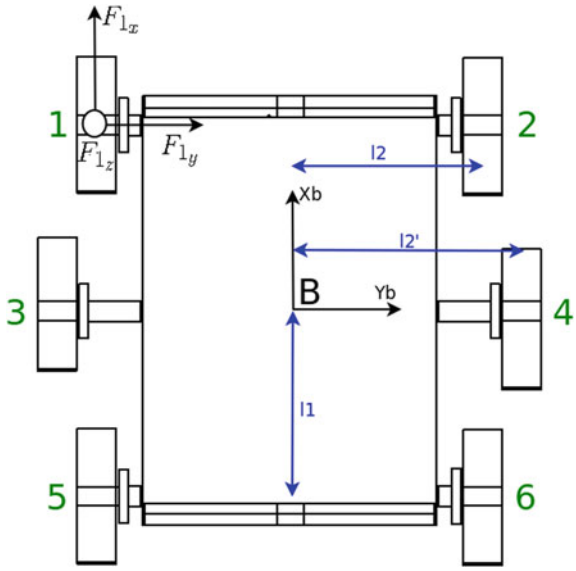


Fig. 5 Localization of the forces in one leg

where $\lambda = \left[\lambda_1 \ \lambda_2 \ \lambda_3\right]^T$.

For the next step, get the τ_i^B moments, is necessary to analyze the robot geometry and localize the actuation forces F_i^B of each leg, where i is the identifier of each leg and B denotes de frame of the robot. With the analyze of the robot geometry, we obtain that the forces along the Z axes, as all they are coplanars, they can be considered as null. And from the schema of the Fig. 5 the following vectors are obtained.

$$\begin{aligned} \lambda_1 &= [l_1 \ -l_2 \ 0]^T, \quad \lambda_2 = [l_1 \ l_2 \ 0]^T, \quad \lambda_3 = [0 \ -l_2^{'} \ 0]^T, \\ \lambda_4 &= [0 \ l_2^{'} \ 0]^T, \quad \lambda_5 = [-l_1 \ -l_2 \ 0]^T, \ \lambda_6 = [-l_1 \ l_2 \ 0]^T, \end{aligned} \tag{11}$$

And the resultant moments are

$$\begin{aligned} \tau_1^B &= \begin{bmatrix} 0 & 0 & -l_2 \\ 0 & 0 & -l_1 \\ l_2 & l_1 & 0 \end{bmatrix} \begin{bmatrix} F_{x_1} \\ 0 \\ F_{z_1} \end{bmatrix}, \ \tau_2^B = \begin{bmatrix} 0 & 0 & l_2 \\ 0 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{bmatrix} \begin{bmatrix} F_{x_2} \\ 0 \\ F_{z_2} \end{bmatrix}, \quad \tau_3^B = \begin{bmatrix} 0 & 0 & -l_2^{'} \\ 0 & 0 & 0 \\ l_2^{'} & 0 & 0 \end{bmatrix} \begin{bmatrix} F_{x_3} \\ 0 \\ F_{z_3} \end{bmatrix}, \\ \tau_4^B &= \begin{bmatrix} 0 & 0 & l_2^{'} \\ 0 & 0 & 0 \\ -l_2^{'} & 0 & 0 \end{bmatrix} \begin{bmatrix} F_{x_4} \\ 0 \\ F_{z_4} \end{bmatrix}, \ \tau_5^B = \begin{bmatrix} 0 & 0 & -l_2 \\ 0 & 0 & l_1 \\ l_2 & -l_1 & 0 \end{bmatrix} \begin{bmatrix} F_{x_5} \\ 0 \\ F_{z_5} \end{bmatrix} \ \tau_6^B = \begin{bmatrix} 0 & 0 & l_2 \\ 0 & 0 & l_1 \\ -l_2 & -l_1 & 0 \end{bmatrix} \begin{bmatrix} F_{x_6} \\ 0 \\ F_{z_6} \end{bmatrix}, \end{aligned} \tag{12}$$

In the case of use that we have describe in this section and continuing the example that the robot is moving with the alternating tripod gait, this means that at the same time, the legs 1, 4 and 5 are in contact with the ground, while the legs 2, 3 and 6 are in the air. Then, in the next step, the state of each leg is the opposite. According to this, the resulting vector of forces and moments is:

$$\tau_A = \begin{bmatrix} F^B \\ \tau^B \end{bmatrix} = \left\{ \begin{array}{l} \text{if tripod 1 ground} \rightarrow \tau_a = \begin{bmatrix} F_1^B + F_4^B + F_5^B \\ \tau_1^B + \tau_4^B + \tau_5^B \end{bmatrix} \\ \\ \text{if tripod 2 ground} \rightarrow \tau_a = \begin{bmatrix} F_2^B + F_3^B + F_6^B \\ \tau_2^B + \tau_3^B + \tau_6^B \end{bmatrix} \end{array} \right\} \tag{13}$$

2.4 Dynamics

As we mentioned before, the definition of a rigid-body system can be defined in several ways (Sect. 2.2) or also in a matrix form [2]. But in this section we will continue the same form as in the previous sections, the vectorial form.

The rigid-body dynamic equation is described as:

$$M_{RB}\dot{v} + C_{RB}(v)v + D_{RB}(v)v + g(\eta) = \tau \tag{14}$$

where M_{RB} is the rigid-body mass matrix, C_{RB} is the rigid-body Coriolis and centripetal matrix, D_{RB} is the damping matrix, g is the vector of gravitational and buoyancy forces and moments, τ is the vector of control inputs and v is the generalized velocity vector expressed in the frame of the robot, as Fossen explains [4]. The subscript X_{RB} make reference to *Rigid-Body*.

Because of the robot that we are using does not belong to the marine environment, the damping matrix can be ignored, and the vector of gravitational and buoyancy forces and moments can be simplified only to the vector of gravitational forces.

In order to obtain a simpler system and be able to solve the simplified equation from Fossen (Eq. 15), from this point onwards the following assumptions will be taken as valid:

- The origin of the reference system of the robot (B) coincides with the center of gravity of the robot, $r_{CG} = 0$
- The axis of the reference system of the robot (B) coincide with the main axis of inertia of the robot.

$$M_{RB}\dot{v} + C_{RB}(v)v + g(\eta) = \tau \tag{15}$$

Under this premises, it is possible to eliminate the antisymmetric part of the inertial matrix of the rigid body (Eq. 16) which, as explained above, relates the mass of the robot with respect to the global position of the robot with respect to the global reference system. Therefore, the matrix M_{RB} remains as:

$$M_{RB} = \begin{bmatrix} mI_{3x3} & -mS(r_G^B) \\ mS(r_G^B) & I_0 \end{bmatrix} \rightarrow M_{RB} = \begin{bmatrix} mI_{3x3} & 0_{3x3} \\ 0_{3x3} & I_0 \end{bmatrix} \tag{16}$$

where m is the CLHeRo V2 mass, I_{3x3} it is the identity matrix $3x3$ and $I_0 = diag(I_X, I_Y, I_Z)$ is the matrix of moments of inertia.

Likewise, the Coriolis matrix (C_{RB}) can be represented as a symmetric Skew matrix

$$C_{RB} = \begin{bmatrix} 0_{3x3} & -mS(v_1) \\ -mS(v_1) & -S(I_0 v_2) \end{bmatrix} \tag{17}$$

with $v_1 = [u, v, w]^T$ y $v_2 = [p, q, r]^T$.

The symmetric matrix of Skew provides the magnitude in which a force applied on one of the axis of the reference system of the robot (R_B) affects the other two components of the reference system. The matrix is defined as:

$$S = \begin{bmatrix} 0 & \lambda_3 & -\lambda_2 \\ -\lambda_3 & 0 & \lambda_1 \\ \lambda_2 & -\lambda_1 & 0 \end{bmatrix} \tag{18}$$

Hence, the submatrices of the Coriolis matrix remain as:

$$-mS(\nu_1) = \begin{bmatrix} 0 & mw & -mv \\ -mw & 0 & mu \\ mv & -mu & 0 \end{bmatrix}, \; -S(I_0\nu_2) = \begin{bmatrix} 0 & I_z r & -I_y q \\ -I_z r & 0 & I_x p \\ I_y q & -I_x p & 0 \end{bmatrix} \tag{19}$$

By last, the gravitational forces vector is expressed as:

$$g(\eta) = \begin{bmatrix} f_G^B \\ r_{GC} \times f_G^B \end{bmatrix} \tag{20}$$

where $f_G^B = R_B^G(\Theta)^{-1}[0, \; 0, \; -mg]^T$.

So that $g(\eta)$ is:

$$g(\eta) = \begin{bmatrix} mgsin(\phi) \\ -mgcos(\phi)sin(\theta) \\ -mgcos(\phi)cos(\theta) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{21}$$

Now, is possible to replace Eq. 15 with $M_{RB}\dot{\nu}$ as:

$$\begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & I_X & 0 & 0 \\ 0 & 0 & 0 & 0 & I_Y & 0 \\ 0 & 0 & 0 & 0 & 0 & I_Z \end{bmatrix} \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix}$$

$C_{RB}(\nu)\nu$ as:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & mw & -mv \\ 0 & 0 & 0 & -mw & 0 & mu \\ 0 & 0 & 0 & mv & mu & 0 \\ 0 & mw & -mv & 0 & I_Z r & -I_Y q \\ -mw & 0 & mu & -I_Z r & 0 & \\ I_X pmv & -mu & 0 & I_Y q & -I_X p & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ p \\ q \\ r \end{bmatrix}$$

and $g(\eta)$ as:

$$\begin{bmatrix} mgsin(\phi) \\ -mgcos(\phi)sin(\theta) \\ -mgcos(\phi)cos(\theta) \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

And finally, Eq. 15, where $\tau = [F_x F_y F_z \tau_x \tau_y \tau_z]^T$ is:

$$\tau = \begin{bmatrix} m\dot{u} + mwq - mvr + mgsin(\phi) \\ m\dot{v} - mwp + mur - mgcos(\phi)sin(\theta) \\ m\dot{w} + mvp - muq - mgcos(\phi)cos(\theta) \\ I_X \dot{p} + I_Z rq - I_Y qr \\ I_Y \dot{q} - I_Z rp + I_X pr \\ I_Z \dot{r} + I_Y qp - I_X pq \end{bmatrix} \tag{22}$$

Now, applying the control model explained by Kelly [6] we can equalize the generalized vector of external forces (15) to the resulting vector of forces and moments of the legs in contact with the ground, we get the next equality

$$\tau = \tau_A \tag{23}$$

Consequently, by clearing from the above equation, the 6 degrees of freedom that form the dynamic equations of CLHeRo V2 movement are obtained analytically:

$$\begin{aligned} m\dot{u} + mwq - mvr + mg\ sin(\pi) &= F_X \\ m\dot{v} - mwp + mur - mg\ sin(\rho)\ cos(\pi) &= F_Y \\ m\dot{w} + mvp - muq - mg\ cos(\rho)\ cos(\pi) &= F_Z \\ I_X\dot{p} + I_Z rq - I_Y qr &= \tau_X \\ I_Y\dot{q} - I_Z rp + I_X pr &= \tau_Y \\ I_Z\dot{r} + I_Y qp - I_X pq &= \tau_Z \end{aligned} \tag{24}$$

Again, the equations of the movement dynamics of CLHeRo V2 can be further simplified considering that:

- The rotational movement on the axis X_B, Y_B (roll, pitch) is virtually void in practice, since in locomotion modes, such as the alternate tripod, the angular velocities p, q are practically void. $\rightarrow p \approx 0, q \approx 0$
- The lateral force F_y in the body of the robot is negligible, this point has been justified in the Sect. 2.1 with what was described by Raibert in [11].
- When the robot performs a movement described in the horizontal plane of the global reference system, roll and pitch angles have values very close to 0, so the criterion established in the first consideration can be applied. $\rightarrow \phi \approx 0, \theta \approx 0$

Therefore, the movement described by the robot in the horizontal plane of the global reference system can be described as a dynamic 3 DOF system.

$$\begin{aligned} m\dot{u} - mvr &= F_x \\ m\dot{v} + mur &= \theta \\ I_z\dot{r} &= \tau_z \end{aligned} \tag{25}$$

Next, the 3 DOF kinematics that describe the movement in the global horizontal plane is derived from the Eq. 3.

$$\begin{aligned} u \cdot cos(\psi) - v \cdot sin(\psi) &= \dot{x} \\ u \cdot sin(\psi) - v \cdot cos(\psi) &= \dot{y} \\ r &= \dot{\psi} \end{aligned} \tag{26}$$

Finally, taking into account that the location of the acting forces, together with the employed walking pattern that is described according to the global horizontal plane, means that the linear velocity on the axis Y_B can be approximated to 0 ($Y_B \approx 0$). Consequently, the movement of CLHeRo V2 on the global horizontal plane can be approximated by the model of a unicycle with dynamic components. In this way the equations that describe the system are the following:

$$
\begin{aligned}
u \cdot cos(\psi) &= \dot{x} \\
u \cdot sin(\psi) &= \dot{y} \\
r &= \dot{\psi} \\
m\dot{u} &= F_x \\
I_Z\dot{r} &= \tau_z
\end{aligned}
\tag{27}
$$

2.5 CLHeRo V2 Platform

The CLHeRo V2 robot is an improved version of the first CLHeRo V1 robot. The inspiration for the CLHeRo design comes from the RHex [12], a hexapod robot with a single rotational actuator without restrictions per extremity and with the characteristics of the C-legs, Each motor shaft engages at the apex of the C-legs. This, apparently simple, design has managed to perform locomotion tasks such as walking and running [12], pronking [8], jumping [13], climbing stairs [9] and even walk on 2 legs [10]. In addition, several versions have emerged on the basis of the RHex: EduBot [15] developed at the University of Pennsylvania for education, Research RHex [12], XRL a light version of the X-RHex to be more maneuver, Desert RHex built in 2009 to evaluate the efficiency in deserts and mountains, SandBot developed to study methods of traversing sand using a legged robot, AQUA [3] an amphibious version, Rugged RHex [8] commercialized by Boston Dynamics. Figure 6 shows all the versions of the RHex platform in order.

The design of the CLHeRo V2 is made in such a way that inside it is able to house a complete computer with capacity for GPU and parallel processing, the storage of batteries that provide power to all the electronic systems on board and the power to the motors, a voltage rectifier to generate the stabilized voltage to the different electronic components, a power distribution board, a RGBD-D RealSense optical sensor and the communication interface with the exterior via WiFi or ethernet.

Fig. 6 All the versions of the RHex platform (RHex V1, EduBot, X-RHex, XRL, Desert RHex, SandBot, AQUA, Rugged RHex)

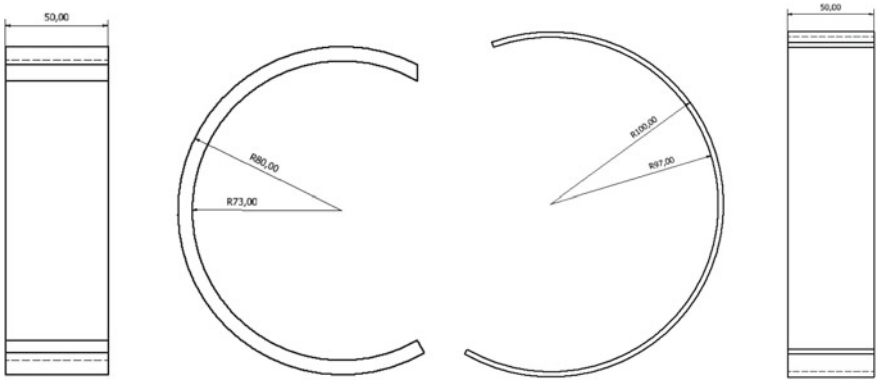


Fig. 7 The two models of legs. At the left the fiberglass leg and at the right the carbon fiber leg

The robot has 2 different (Fig. 7) types of legs that are used according to the needs of the activity to be performed. Some have been constructed from a tubular PVC structure covered in fiberglass, with an external diameter of 160 mm and an internal diameter of 146 mm and another set in carbon fiber constructed from a mold with an outer diameter of 200 mm an inner diameter of 194 mm and a gram weight. Both types of legs have a thickness of 50 mm.

The CLHeRo V2 obtains its movement capacity thanks to a Maxon MCD EPOS 60W Intelligent Compact Drive units together with planetary gearboxes coupled to the motor shaft. The MCD unit features a Maxon EC brushless motor, a digital MR encoder and a fully digital EPOS position controller. The 6 motors are connected internally by the CANopen protocol with a speed of 1MBit/s, however, the communication with the robot CPU is via RS-232 protocol, which limits the speed to 115200 bit/s (schema from Fig. 8). Despite the fact that several packages were used for ROS that support controllers with CAN protocol, none could be operated with this type of controllers, this information is explained more carefully in the Sect. 6.4. The MCD unit offers a maximum torque of 218mNm and an operating range between $12V_{DC}$ and $50V_{DC}$ with an efficiency of 70%. To correctly configure the motors in the CANopen network, it is necessary to add two termination resistors of 120 Ω, see Fig. 9.

The MCD unit has a GP32C planetary gearbox with a 33 : 1 ratio attached to the output of its shaft that raises the continuous working torque to 3 Nm with maximum peaks of 3.75 Nm.

The computer responsible for all the management of the algorithms for moving the robot consists of an Asus Prime H310I-Plus Mini-ITX motherboard, an Intel I7-8700 processor, 8 GB of RAM KINGSTON HYPERX PREDATOR DDR4 2666 MHZ, an

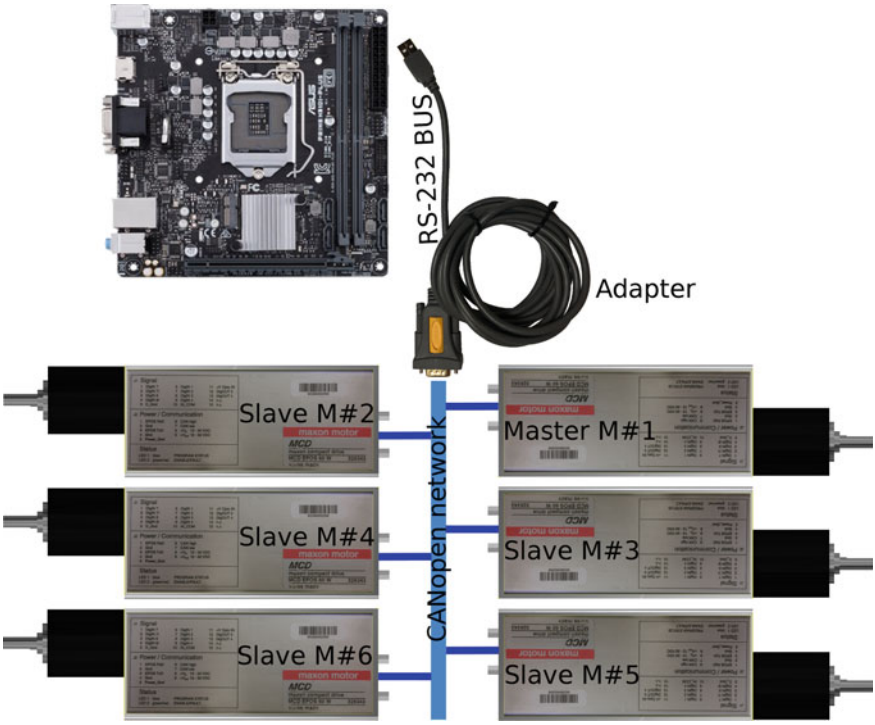


Fig. 8 Communication schema between the CPU and the Maxon MCD of the CLHeRo V2

Fig. 9 Configuration of the resistors for the CANopen network

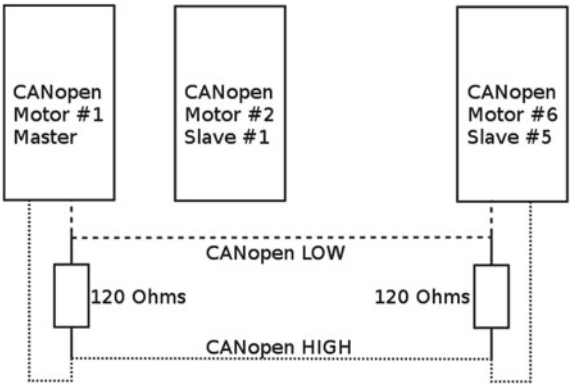


Table 2 Weight of the components of CLHeRo V2

Component	Weight (gr)
Computer	648
Power supply	157
Motors	4140
Legs	2508
Chassis	1024
Batteries	1480
RealSense	72
Total weight	10029

SSD hard drive SSD SAMSUNG SERIE 970 EVO M.2 NVME[1] and a smart power supply Mini-Box M4-ATX-HV 220W DC-DC CAR-PC 6-34V.[2]

In addition, the entire robot is powered by two batteries Turnigy ZIPPY Flightmax 8000mAh 4S1P 30C[3] in series.

The physical characteristics that describes the CLHeRo V2 are mentioned below, the total mass of the robot is 10.497 kg (in the Table 2 you can see the independent mass of each element of the robot), meanwhile, the center of gravity of the robot has been tried to be centered as much as possible to match the axis of inertia of the robot, to achieve this the different devices have been placed in strategic places, leaving the center of gravity of the robot located at the point ($X = -0.976$ mm, $Y = 0.162$ mm and $Z = 1.439$ mm) with respect to the reference system of the robot body and the inertia of the robot body also from the same reference system is:

$$\begin{aligned} &I_{xx} = 179573,933\ \mathrm{kg \cdot mm^2} \\ &I_{xy} = -710,610\ \mathrm{kg \cdot mm^2} \quad I_{yy} = 345391,033\ \mathrm{kg \cdot mm^2} \\ &I_{xz} = 4034,966\ \mathrm{kg \cdot mm^2} \quad I_{yz} = -3457,602\ \mathrm{kg \cdot mm^2} \quad I_{zz} = 504941,769\ \mathrm{kg \cdot mm^2} \end{aligned} \tag{28}$$

Since the terms I_{xy}, I_{xz} and I_{yz} from the Eq. 28 are three orders of magnitude smaller than the other terms, it can be considered that the robot's inertial matrix is only composed of the terms on the diagonal (I_{xx}, I_{yy} and I_{zz}).

[1] https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-250gb-mz-v7e250bw/1.

[2] http://www.mini-box.com/M4-ATX-HV.

[3] https://hobbyking.com/es_es/zippy-flightmax-8000mah-4s1p-30c-xt90.html.

3 ClegS Control Strategy

The CLHeRo V2 does not have external sensors with which it can estimate the state in which it is located. As a consequence, to determine the status of the same it is necessary to know the position in which each motor of the legs is, also knowing the size of the leg, the kinematics of the robot and the landing and takeoff angles of each leg, can be obtained an estimate of the variation in the position change of the robot.

The observation of the internal parameters without having reference of the external world to keep a record of how much the robot has moved is what is known as dead reckoning. This technique is used to know the distance traveled by a car using its odometry, but this measure is only the relative displacement of the same. In nature dead reckoning is known as kinesthesia or idiothetic sensing.

3.1 Walking and Running

The purpose of the robot's progress controller is to calculate the trajectories for each tripod, which are periodic functions in time. As input parameters you need the advance speed of the robot body, the take-off angle of the leg and the angle of landing of the leg (v_{robot}, ϕ_{take_off} and ϕ_{take_land}). The working period of each tripod will stabilize the angle of sweep in the ground (ϕ_{sweep_angle}) which is determined by the difference between the landing angle and the takeoff angle and the forward speed of the robot body.

$$\phi_{sweep_angle} = \phi(v_{robot})_{take_off} - \phi(v_{robot})_{take_land} \tag{29}$$

During a movement cycle, each tripod performs the two phases that make up the sweeping motion, the ground phase and the flight phase. The soil phase was described in the previous paragraph and also with the Eq. (29). On the other hand, the flight phase is the proportional part of the arc that remains of the complete route and that can be expressed as:

$$\phi_{flight_angle} = 2\pi - \phi_{sweep_angle} \tag{30}$$

The desired behavior during walking or running can be achieved by modifying these parameters. However, there are physical limits whose values can not be exceeded, for example both the takeoff angle and the landing angle can not open more than 30° with respect to the line perpendicular to the ground and that cuts the motor shaft. Overcoming this value causes the robot to lose its static stability because the center of gravity of the robot would be displaced from the stability triangle formed by the projections of the supporting tripod.

3.2 Turning

As previously explained, CLHeRo V2 is capable of both turning on itself and turning while traveling. So it has been necessary to develop a different driver for each action.

The controller to turn on itself makes use of the profiles used for running and walking except that the groups of tripods rotate in opposite directions. This enables the hexapod to rotate in place in the direction determined by the direction of rotation of the tripods. It is necessary to remember that the tripods are internally synchronized, always keeping one of these the robot's support with the ground. As it happened with the race, the turn on itself depends on the desired speed of rotation and the angles of landing and takeoff (v_{robot}, ϕ_{take_off} and ϕ_{take_land}).

In order to implement the rotation during locomotion it is necessary to modify the behavior to introduce disturbances in the contralateral legs. This can be modified either by the speed of the contralateral legs or the angle of sweep in the ground, which depends on the landing and take-off angle.

Consequently, 2 new parameters are introduced, $\Delta_{tripod_velocity}$ y Δ_{sweep_angle}. The turn to the right is when $\phi_{sweep_angle_{tripod_2}} > \phi_{sweep_angle_{tripod_1}}$ and the turn to left is when $\phi_{sweep_angle_{tripod_1}} > \phi_{sweep_angle_{tripod_2}}$.

4 Environment Setup

This section explains how to setup the ClegS package, for Ubuntu is shown in the next section and for OS X in Sect. 4.2. The CLHeRo hardware specifications can be found in Sect. 2.5.

4.1 ROS in Ubuntu

Is the recommended operating system to run ROS and the package manager should be used to install the necessary dependencies. You can follow the ROS Kinetic Install Guide[4] to install it.

In addition to the standard desktop package, is necessary to install a few other packages that the ClegS package depends on.

- controller manager: *$ sudo apt install ros-kinetic-controller-manager*
- joint state controller: *$ sudo apt install ros-kinetic-joint-state-controller*
- position controller: *$ sudo apt install ros-kinetic-position-controllers*
- velocity controller: *$ sudo apt install ros-kinetic-velocity-controllers*
- effort controller: *$ sudo apt install ros-kinetic-effort-controllers*
- ros control: *$ sudo apt install ros-kinetic-ros-control*

[4] http://wiki.ros.org/kinetic/Installation/Ubuntu#Installation.

– gazebo ros control: *$ sudo apt install ros-kinetic-gazebo-ros-control*
– gazebo ros plugins: *$ sudo apt install ros-kinetic-gazebo-ros-plugins*
– teleop twist joy: *$ sudo apt install ros-kinetic-teleop-twist-joy*
– joystick: *$ sudo apt install ros-kinetic-joy*.

4.2 ROS in OS X

OS X can run ROS native, and we recommend using Homebrew to install the necessary dependencies. There is also a tutorial[5] that describes how to install ROS Kinetic on OS X and also, you can try Mike Purvis script.[6]

4.3 Create a Workspace

Also is necessary to create a workspace by following the guide in the ROS Wiki.[7] This boils down to:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws
$ catkin_make
```

Just like with the primary ROS install, is necessary to source the newly created setup.*sh file:

```
$ source devel/setup.bash
```

4.4 Install ClegS Repository

Installing the ClegS repository is independent of the operating system and can be done using git tools, as is shown in Listing 10.1.

Listing 10.1 Commands to download and install the ClegS packages

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/grafoeka/clegs_common.git
$ git clone https://github.com/grafoteka/clegs_sim.git
$ git clone https://github.com/grafoteka/clegs_hw.git
$ cd ~/catkin_ws /
$ catkin_make
```

[5] http://wiki.ros.org/kinetic/Installation/OSX/Homebrew/Source.
[6] https://github.com/mikepurvis/ros-install-osx.
[7] http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

5 Package Overview

Before starting this section, is necessary to clarify the syntax that will be used from now on.

```
$ This is some commands in the terminal
```

```
This is some code in a file, it can be C++, XML, ...
```

```
$ And this is the output in the terminal
```

The ClegS simulator is split up in various meta-packages[8] as shown in Fig. 10. In this section, all the packages will be explained so the reader could understand how ClegS works and how to be more efficient developing new algorithms for C-legs robots. The overview of all the meta-packages can be understand with the analyzes of the Fig. 11. This figure shows which nodes are execute in each package and also which are the topics that are used to communicate between all the nodes (via messages or services).

The first meta-package that the user can find is the *clegs_common*. This meta-package contains the packages that can be used by the robots simulation, together to the real robot. Thus, this package contains the package for the description of the robot, the package that includes the core of the GP management, *clegs_gait_controller* and the package *clegs_gait_pattern_definition* that provides the implementation of the patterns, using the ClegS programming API.

The second meta-package is designed to communicate with the real platform CLHeRo V2. It includes the *clegs_hw_control* package with the same objective as *clegs_sim_control*, but in this case, it generate the orders for the Maxon motors. And there is also the *clegs_maxon_controllers*, which establish the connection with the motors and send the orders of movement and received the state of the motors.

The last meta-package is the *clegs_sim*. This meta-package has been create to manage the simulation packages with Gazebo. There are 3 packages in this folder. The package *clegs_sim_controllers* loads the controllers for the actuators in the Gazebo simulation. The *clegs_gazebo* package includes the necessary to launch the robots and the worlds in Gazebo. And *clegs_sim_control* is the package that get the information from the *gait_control* and generate the orders for the motors in the simulation.

The idea behind creating ClegS with a common meta-package is to have better structure in the projects, and so, with that, the developers only have to write their algorithms in the common meta-package. And then, just launching the gazebo package or the clhero_v2 package the algorithm can be tested.

All the meta-package will be explained in detail in the next subsections.

[8] http://wiki.ros.org/Metapackages.

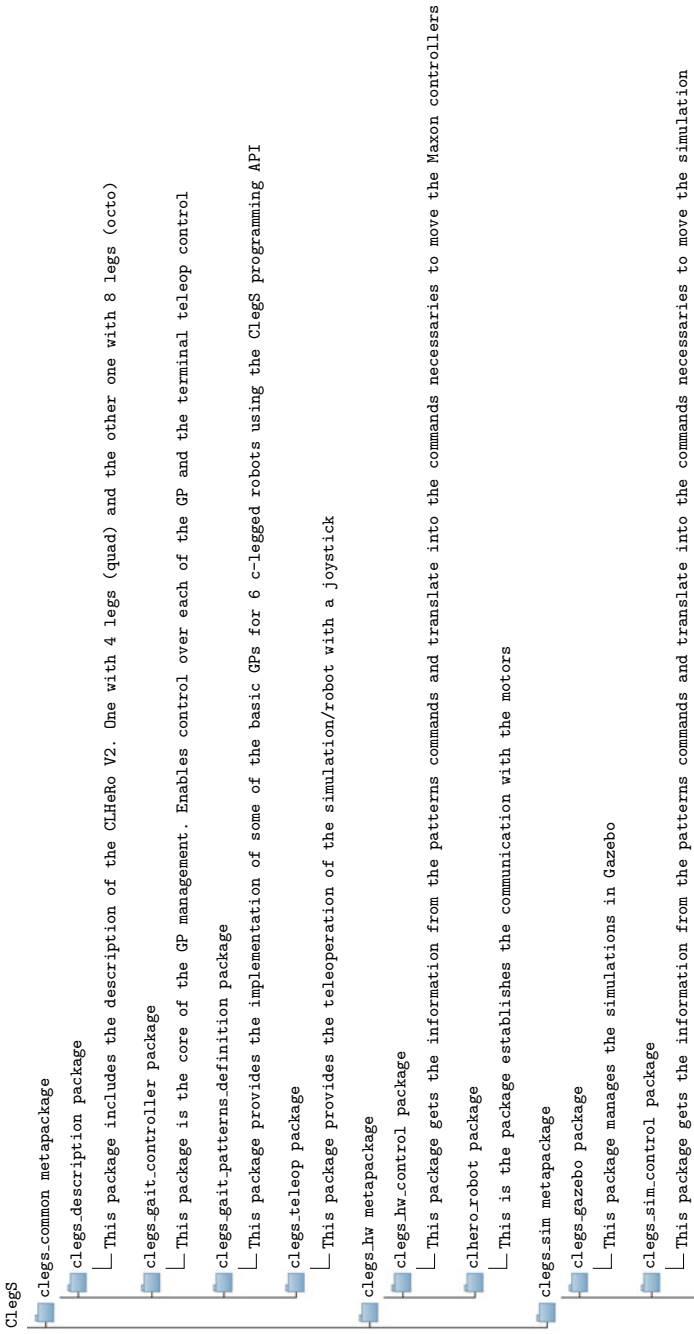


Fig. 10 ClegS tree directory

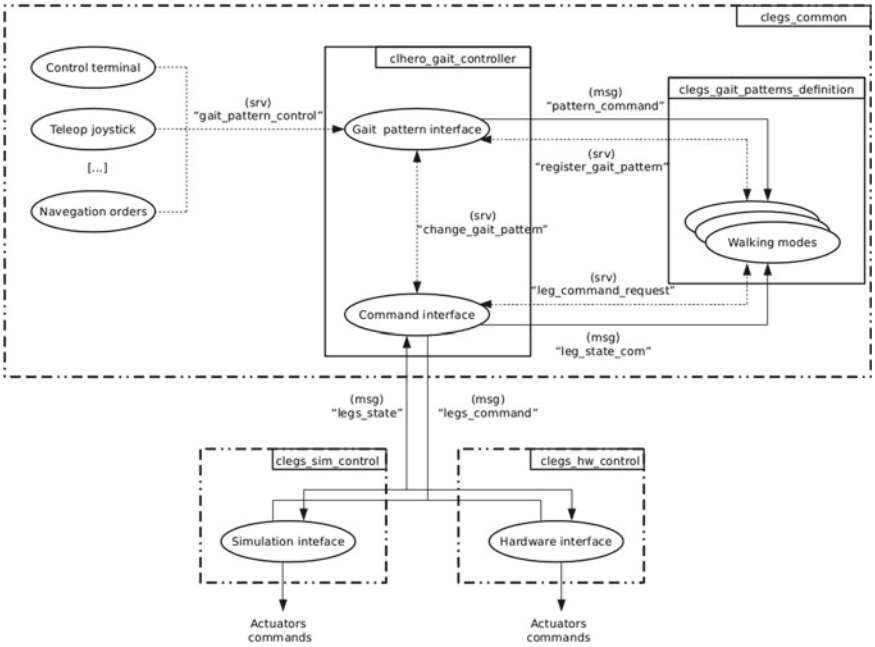


Fig. 11 Structure of ClegS package collection

5.1 ClegS Common Metapackage

5.1.1 ClegS Description Package

The description package is a very common package in the ROS projects. The urdf, meshes, and other elements needed in the description are contained here. This package includes the description of the CLHeRo V2 and 2 prototypes of C legged robots, one with 4 legs (quad) and the other one with 8 legs (octo). The package includes also some launch files to publish the robot state and to test the urdf files in rviz. Figure 12 shows the package structure.

5.1.2 ClegS Gait_controller Package

This package is the core of the GP management, enables control over each of the GP being able to start, stop, pause, resume or pass different arguments to each of the GPs registered. This package also contains the ClegS library which can be used for GP C++ programming.

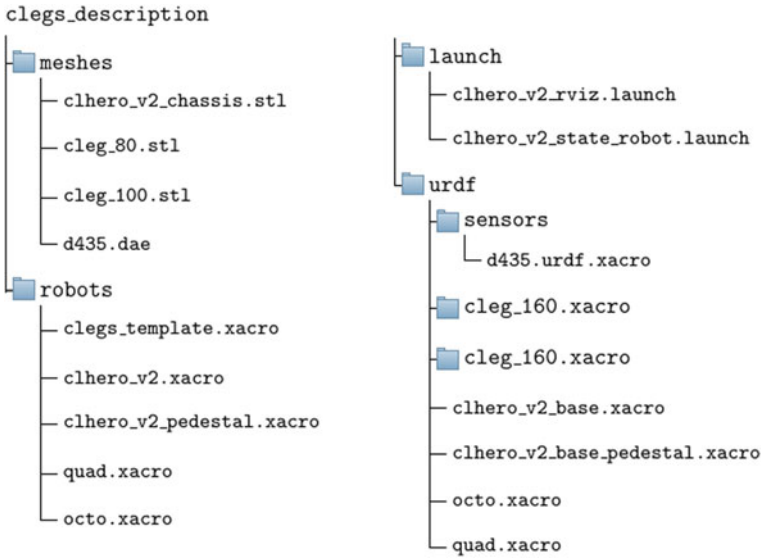


Fig. 12 Structure of ClegS description package

The clegs_gait_pattern_controller provides the functionality for GP control and management. This functionality is based in two ROS nodes: the *gait_pattern_interface* and the *command_interface*.

The *gait_pattern_interface* receives commands from the user or any navigation system or control to manage the state of each of the different GPs registered as individual ROS nodes. This *gait_pattern_interface* enables starting, stopping, resuming, pausing or even passing different parameters of the movement to each of the GPs, as for example, the velocity of each of the actuators in a certain movement or the angle in which the legs are contact the ground.

On the other side, the *command_interface* performs two crucial tasks: providing collision avoidance of the commands given by two or more different GPs as only the active GP's commands shall be passed to the actuators or simulation; and passing the actuators state to each of the GPs.

Along with these nodes, the package is shipped with a C++ library, the *clegs* library which provides an easy-to-use GP programming API integrating all the functionality described for all the packages in the ClegS collection.

Gait pattern interface

Just as stated before, the GP interface's main functionality is to control the state of each of the different implemented GPs.

As coordinating each of the actuators is crucial to perform a consistent gait in legged robots, the GP design and implementation is based on finite state machines (or fsm from now on). Following this principle, the GPs are implemented by states, in which a specific set of movements are performed, and transitions among them, in most occasions the conditions that have to be met to evolve into a different state are

the arrival into a certain position or reaching certain velocity value, this allows an outstanding performance in coordination.

In order to control and add testing and debugging functionalities, the GPs are considered to be in one out of four different modes:

- Active: the GP is currently running and sending the commands to each of the actuators. Only the commands from the active GP will be commanded to the actuators. The GP interface allows that only one GP is active at a time.
- Stopped: the GP is forced into a sleeping state in which does not send any commands and waits to be active to start the gait defined.
- Paused: the GP holds the last command or its last state until a signal from the GP interface is sent which indicates to evolve to the next state.
- Test: the GP is forced into a specific state given as an argument even if the conditions to transition to this state are not met or even if there is no explicit transition to the new state.

Before that any GP is operative, the GP shall be registered. The GP interface handles a ROS parameter with the names of each of the registered GPs. The registration takes place by calling the service */register_gait_pattern* and passing the name of the GP, returning whether the registration has succeeded or failed if the name already exists.

The registration process implies that each GP shall be identified with its own pattern name. This identifier is used for both the GP interface and the command interface and shall be unique among all the other GPs defined.

Once the GP interface is running along the different implemented GPs, the service */gait_pattern_control* allows the user to take control over these patterns. The definition of this service is:

Listing 10.2 */gait_pattern_control* service definition

```
# ==========================================
# GaitPatternControl.srv
#
# Description: this service is set for controlling the pattern manager.
# ==========================================

#Name of the gait pattern
string pattern_name

#Arguments
float32[] args

#Order
# - stop: stop the gait pattern at an initial or emergency state
# - pause: stop the gait pattern at the current state
# - continue: resumes the course of action
# - force_state: transitions to a defined state given in args
# - start: starts the gait pattern
# - update_args: update the gait pattern's arguments
string order

- - -

int8 ans
```

Three are the fields required to control the GPs:

- Patter name: the name identifier of the objective pattern.
- Arguments: array of floats used to pass a GP different movement parameters.
- Order: action to take in the desired GP.

The order field specifies, as the name implies, the action to take in the desired GP. The options given are directly associate with the functionality described up to this point, but there are two options that has not been explained yet: the *force_state* and the *update_args*.

The *force_state* order option enables test mode over a GP, forcing the change into an specific state of the pattern. The *update_args* option allows the user to pass information to the GP by a float array, this option is intended to be used to pass movements parameters to the GP, changing the velocity, acceleration; changing from forward movement to reverse movement; or even modifying the contact angle with the ground of a leg.

Command interface

The command interface is the other part of the foundation for GP management. Just as stated before, the command interface functionality is based on providing a standardized command for actuator which may be used for any destination with the proper interface and passing the actuators state to each of the GPs for their self-coordination.

The command interface prevents non active GPs from sending commands to the actuators. A GP does not command directly to the actuators this is the task of the command interface. In order for a GP to send a command, a request to the command interface shall be fulfilled, this request is carried out by calling the */leg_command_request* service, its definition is the following:

Listing 10.3 */leg_command_request* service definition

```
# ==========================================
# LegCommandRequest.srv
#
# Description: this service is set for requesting a command
# ==========================================

float32[6] pos
float32[6] vel
float32[6] acel
float32[6] decel
std_msgs/Bool[6] new_acel_profile
std_msgs/Bool[6] position_command
int16[6] new_command
string gait_pattern_name
- - -
int8 ans
```

The commands considered may be classified in two different types: position and velocity commands. It has been considered that some designs may support acceler-

ation profile control in position command mode but its implementation may depend on the real application.

If the requesting GP is active, a */leg_command* message is sent which shall be considered as the command that the actuators or simulation interface receive.

Another feature required for movement coordination is to know the state of each actuators at any time. The combination of the position (linear or angular), velocity and effort is considered as the state of an actuator.

The command interface is subscribed to the */leg_state* topic where the state of the actuators shall be published. The command interface delivers this information to the GP so that this status can be consulted through the C++ programming API.

ClegS Library

One of the most outstanding features of this packages is the ClegS library which provides an easy to use GP programming API. The aims of this library are:

- Establish a framework for GP programming based on finite state machines.
- Provide full support for all the functionalities described for GP management.
- Increase performance by including real-time programming features like multi-threading.

The ClegS library is based on the *CLeg_Robot* class which represents the c-legged robot that the GP is being designed for. This class provides methods for commanding, get the actuators' state and attaching new states to the fsm.

Just as stated before, the first step for GP programming with ClegS library is to create an unique pattern name which will be used to be identified. Once this name has been decided, it shall be registered by calling the function:

```
void registerGaitPattern(std::string name);
```

At this point the name has been registered and commands with this name will be accepted by the GP interface. Now, a class' instance can be created by passing the pattern name:

```
CLeg_Robot cl_robot (name);
```

Just after creating the *CLeg_Robot* object, the different states created can be attached which means that the states are recognized and include in the state machine. To attach a new state an id number which uniquely identifies the state and the function prototype are required. Every GP shall identify one of their states as the *starting state*, this state will be the first to be called when the GP turns into an active mode. An example of attaching states is the following:

```

//attach the defined states
cl_robot.attachState(1, state_1, STARTING_STATE);
cl_robot.attachState(2, state_2);
cl_robot.attachState(3, state_3);
cl_robot.attachState(4, state_4);
```

A state shall be defined as a function with void as the returning type and a pointer to *CLeg_Robot* object as the sole argument.

```
//State's prototype
void state_1 (CLeg_robot* clr);
```

In order to command each of the actuators the *CLeg_robot* class includes two methods which can be used for both command types. Their prototypes are:

```

//Set the position for a leg
void setLegPosition (int leg,
float position,
float velocity = DEFAULT_VEL,
bool new_acel_profile = false,
float acel = DEFAULT_ACEL,
float decel = DEFAULT_ACEL);

//Set the Velocity for a leg
void setLegVelocity (int leg,
float velocity,
bool new_acel_profile = false,
float acel = DEFAULT_ACEL,
float decel = DEFAULT_ACEL);
```

An example of position and velocity command is the following:

```

//Position command
clr->setLegPosition(1, TAKE_OFF, GROUND_VELOCITY);

//Velocity command
clr->setLegVelocity(2, 3);
```

Every state shall keep running until that state is no longer the active state. When a transition condition is met, the state shall call the *transition* method which changes the active state with the state that matches the identifier given as argument. The *activeState* method returns the id number of the current active state.

The following example (list 10.4) shows the case of the state 1 transitioning into state 2 when all legs are in position:

Listing 10.4 Transition from state 1 to state 2

```
while(clr->activeState() == 1){

//Code that checks if the legs are positioned

if(legs_in_position == LEG_NUMBER){
clr->transition(2);
}
}
```

To check and retrieve the actuator's state, the class includes the following methods:

```
std::vector<float> getLegsPosition();
std::vector<float>getLegsVelocity();
std::vector<float>getLegsTorque();
```

Once the states are defined and attached and therefore the GP's definition is complete the *CLeg_robot* shall call the *run* method, which starts the GP.

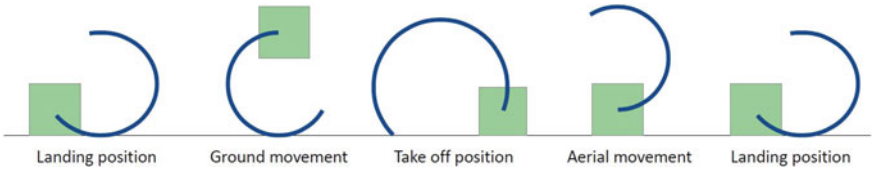


Fig. 13 Movement description of the complete wave GP

```
cl_robot.run();
```

This example shows how the management of an outstanding quantity of messages and services are simplified into simple instances and function calls which prevents the code from turning illegible, being cleaner and clearer.

5.1.3 ClegS Pattern_definition Package

In the ClegS package collection, the *clegs_gait_pattern_definition* package provides the implementation of some of the basic GPs for 6 c-legged robots using the ClegS programming API.

The complete wave and alternating tripod GPs will be described in this section. For each pattern, the movement description will be provided along the fsm diagram and its implementation using the ClegS programming API.

Complete wave

This GP commands all 6 legs with the same velocity command. As a result the robot alternates periods in which the legs are in contact with the ground with others in which the legs are moving through the air.

When the legs have no traction the robot does not move which is the most outstanding disadvantage of this GP.

The Fig. 13 shows the movement description for two legs of this GP.

This GP may be defined with just only one state that commands all the legs with the same velocity but it is highly recommendable that it is defined with two states, one for ground movement and another for aerial movement. The main advantages of defining the GP with two different states are:

- Coordination: setting that all legs are in landing or takeoff position as transition condition forces movement synchronization twice per movement cycle.
- Optimization: differentiating ground and aerial movement may be used to send two different velocity commands, a higher aerial movement velocity may reduce the time in which the robot is not advancing.

After these considerations, the finite state machine of the GP can be defined. This has been represented by a functional GRAFCET diagram which is shown in Fig. 14.

Once the fsm has been defined, the GP can be programmed using the *ClegS library* and the *CLeg_robot* class as seen in the previous section. The following code shows how the state 1 would be implemented (Listing 10.5).

Listing 10.5 Code for the state 1

```
//State 1
//Ground movement
void state_1 (ClegS_robot* clr){

//-----------------------------------------
// State's initial statement
//-----------------------------------------

std::vector<float> state;
int legs_in_position = 0;
ros::Rate loop_rate(STATE_LOOP_RATE);

//Sends the velocity command to each leg
for(int i=1; i <= LEG_NUMBER; i++){
clr->setLegPosition(i, 0, GROUND_VELOCITY);
}

//-----------------------------------------
// State's core loop
//-----------------------------------------

while(clr->activeState() == 1){

//-----------------------------------------
// State's transition checking
//-----------------------------------------

//Gets the legs position
state = clr->getLegsPosition();

//For each leg checks if it has arrived its destination
for(int i=0; i < LEG_NUMBER; i++){
if((fabs(state[i] - TAKEOFF_ANGLE) < ANG_THRESHOLD)){
legs_in_position++;
}
}

//If all the legs has been positioned requests the transition
if(legs_in_position == LEG_NUMBER){
clr->transition(2);
}else{
legs_in_position = 0;
}
loop_rate.sleep();
}
return;
}
```

In order to implement state two, this previous code can be reused changing the GROUND_VELOCITY with AERIAL_VELOCITY and TAKEOFF_ANGLE with LANDING_ANGLE.

Alternating tripod

The alternating tripod finds a solution to the main disadvantage of the complete wave by dividing the legs in two stable tripods and alternating their roles. When one tripod is performing the ground movement, the other tripod is moving through the air, when

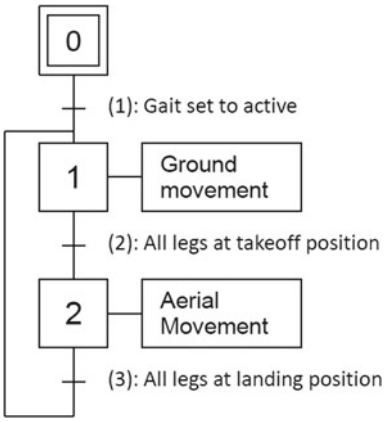


Fig. 14 Functional GRAFCET diagram of the complete wave GP

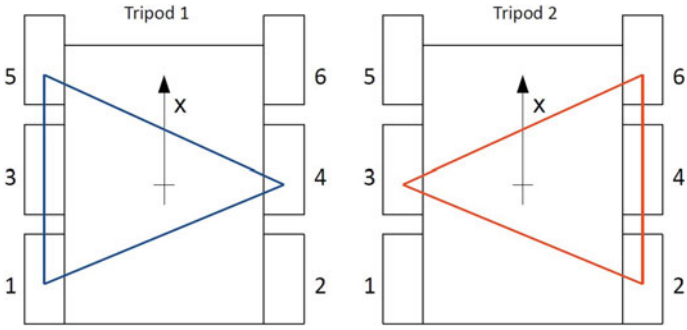


Fig. 15 Tripods in alternating tripod GP

the first tripod reaches the takeoff position and the second the landing position, their roles are switched, creating uninterrupted robot movement.

Figure 15 shows the leg numeration and which legs belongs to a tripod. Thus, legs 1, 4 and 5 belongs to tripod 1 and legs 2, 3 and 6 belongs to tripod 2.

The GP starts forcing the robot to stand up in a neutral position, this first step is required for the following step which is the positioning. In the positioning state, one tripod shall move into the landing position while the other takes the takeoff position. Once the robot is positioned the advance movement starts alternating two states: one in which tripod 1 makes the ground movement and tripod 2 makes the aerial movement and another with their roles switched. The alternating tripod GP movement is shown in Fig. 16.

When the movement is defined, the state machine can be defined as well. Just as how it was done at the complete wave GP definition, the Fig. 17 represents the fsm in a functional GRAFCET diagram.

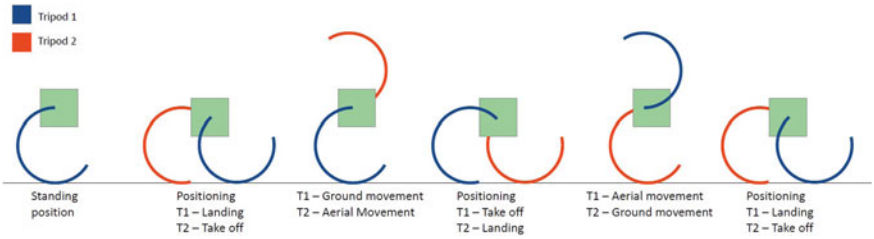


Fig. 16 Movement description of the alternating tripod GP

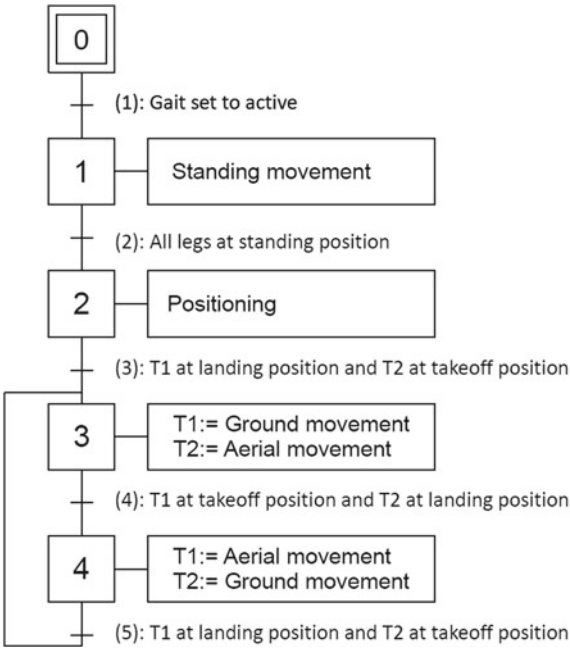


Fig. 17 Functional GRAFCET diagram of the alternating tripod GP

Although the methodology followed when using the ClegS library is the same when any of these states are programmed, which is commanding each leg and check if the condition transition has been met periodically, in the following code is shown how the state 3 has been implemented as an example (Listing 10.6).

Listing 10.6 Code for the state 3

```
//State 3
//Tripod 1 in
void state_3 (CLeg_robot* clr){
//-----------------------------------------
// State's initial statement
//-----------------------------------------
std::vector<float> state;
int legs_in_position = 0;
ros::Rate loop_rate(STATE_LOOP_RATE);
//Commands tripod 1 for ground movement
for(int i=0; i < tripod_1.size(); i++){
clr->setLegPosition(tripod_1[i], TAKEOFF_ANGLE, GROUND_VELOCITY);
}
//Commands tripod 2 for aerial movement
for(int i=0; i < tripod_2.size(); i++){
clr->setLegPosition(tripod_2[i], LANDING_ANGLE, AIR_VELOCITY);
}
//-----------------------------------------
// State's core loop
//-----------------------------------------
while(clr->activeState() == 3){
//-----------------------------------------
// State's transition checking
//-----------------------------------------
//Gets the legs position
state = clr->getLegsPosition();
//Checks if the tripod 1 legs are in position
for(int i=0; i < tripod_1.size(); i++){
if((fabs(state[tripod_1[i]-1] - TAKEOFF_ANGLE) < ANG_THRESHOLD)){
legs_in_position++;
}
}
//Checks if the tripod 2 legs are in position
for(int i=0; i < tripod_2.size(); i++){
if((fabs(state[tripod_2[i]-1] - LANDING_ANG) < ANG_THRESHOLD)){
legs_in_position++;
}
}
if(legs_in_position == LEG_NUMBER){
clr->transition(4);
}else{
legs_in_position = 0;
}
loop_rate.sleep();
}
return;
}
```

State 4 uses the exact same piece of code but changing the ground parameters with the aerial velocity and landing position.

One of the most outstanding features of the ClegS package collection is the code once, run everywhere for GPs. As it was mentioned earlier, a GP programmed with the *ClegS library* and being controlled by the *clegs_gait_pattern_controller* can be used for both simulation and hardware.

However each application uses its own commands and orders to provide similar functionality, each simulator has its own API as well as each motor controller has its

Fig. 18 Actions assigned to each button of the Logitech Gamepad

own. To overcome this issue, the command interface interacts through standardized messages that only shall be translated or adapted to work in any application.

Therefore, for each application, a simulation or hardware interface shall be created to translate or adapt the messages required by the command interface. Thus, the requirements for any interface are:

- Publish at */legs_state* topic with the state information of each of the actuators.
- Subscribe to */legs_command* topic and translate the commands given to the real actuators.

Along with the GP controller and definition, the ClegS package collection includes the *clhero_sim_control* which is the gazebo interface for the clhero robot.

From this section on, the packages for the simulation will be explained. This 3 packages are necessary to run successfully the simulations together with the *clegs_common* package.

5.1.4 ClegS Teleop Package

This package allows the user to use a joystick to control the real robot or the simulation in Gazebo. The user can stand up the robot from the ground or land it again, select the a GP, synchronize the legs or move the robot to any direction. This package is implemented for a *Logitech Gamepad Dual Action*. Figure 18 shows a diagram of the joystick with the function assigned to each button.

5.2 ClegS Sim Metapackage

5.2.1 ClegS Sim_control Package

This gets the information from the patterns commands and translate into the commands necessaries to move the simulation. There is one package equal in the hardware branch but for the real robot. With this package, the developer only has to be focus

on generate a correct GP, then the software do all the necessaries modifications to translate it into the simulation model.

5.2.2 ClegS Sim_controllers Package

All the controllers for Gazebo are defined in this package. The controllers are called from the *.launch* file, but firstly, the controllers have to be parameterized in the *.yaml* file. Lets understand the structure of the file for the parameters. The first line of the file is the name of the robot in the parameter server. And then the configuration of the joint_state_controller (Listing 10.7), with the value of the rate frequency (50 Hz).

Listing 10.7 Joint State Controller configuration

```
clhero_v2: # Publish all joint states ----------------------
joint_state_controller: type:
joint_state_controller/JointStateController publish_rate: 50
```

The next two sections are for customize the controllers. The first one is for the position controllers and the second one for the velocity controllers. The name of each controller is the same that in the .launch file that calls this configuration file. For each motor is necessary to define the type of the controller, it has to be the same one as in the *.xacro* model file. Then, the name of the joint, also in the *.xacro* model file. And finally the PID parameters for the controller. You can see the configuration of the *motor_1_position_controller* and the *motor_1_velocity_controller* in the Listing 10.8. The parameters of each controller can be adjusted by the user, so it can have the same behavior as the real motors of the robot, so the simulation can be reliable.

Listing 10.8 Configuration of the motor 1

```
# Position Controllers -------------------------
motor1_position_controller: type:
effort_controllers/JointPositionController joint:
motor_1_to_base_link_joint pid: {p: 3.0, i: 1, d: 0.0}

[...]

# Velocity Controllers -------------------------
motor1_velocity_controller:
type: effort_controllers/JointVelocityController
joint: motor_1_to_base_link_joint
pid: {p: 3.0, i: 1, d: 0.0}
```

Inside the launch folder there are one *.launch* file for each *.yaml* file. But the unique difference between them is the controllers that are loaded, depending on the robot that the user wants to simulate.

The *.launch* file is divided in two sections. The first one (Listing 10.9) load and start the two types of controllers that the robot can use. The second part runs the robot state publisher node.

In the first section of the script, we have include the variable named *position_controller*. If this variable is equal to *true*, then the position controllers are loaded and started, and the velocity controllers are only loaded. On the other hand, if

the variable is equal to *false*, the position controllers are only loaded and the velocity controllers are loaded and started.

Listing 10.9 Launch file for the controllers

```
<!-- Parameter to select the velocity or position controllers FALSE:
Velocity TRUE: Position --> <arg name="position_controller"
default="false"/>

<!-- Position velocity -->
<group if="$(arg position_controller)">

<!-- Load joint controller configurations from YAML file to parameter server -->
<rosparam file="$(find clegs_sim_controllers)/config/clhero_v2_control.yaml"
     command="load"/>

<!-- load the controllers -->
<node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="
     false" output="screen" ns="/clhero_v2" args="joint_state_controller
motor1_position_controller
[...]
motor6_position_controller"/>

<node name="controller_loader" pkg="controller_manager" type="controller_manager"
      respawn="false" output="screen" ns="/clhero_v2" args="load
motor1_velocity_controller
[...]
motor6_velocity_controller"/>
</group>

<!-- Velocity controllers -->
<group unless="$(arg position_controller)">

<!-- Load joint controller configurations from YAML file to parameter server -->
<rosparam file="$(find clegs_sim_controllers)/config/clhero_v2_control.yaml"
     command="load"/>

<!-- load the controllers -->
<node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="
     false" output="screen" ns="/clhero_v2" args="joint_state_controller
motor1_velocity_controller
[...]
motor6_velocity_controller"/>

<node name="controller_loader" pkg="controller_manager" type="controller_manager"
      respawn="false" output="screen" ns="/clhero_v2" args="load
motor1_position_controller
[...]
motor6_position_controller"/>
</group>
```

Finally, is necessary to execute the *robot_state_publisher* node (Listing 10.10). The robot state publisher broadcasts the state of the robot. The robot state publisher internally has the kinematic model of the robot, it has been loaded from the ROS parameter server; so given the joint positions of the robot, the robot state publisher can compute and broadcast the 3D pose of each link in the robot. For a better comprehension of the topics, they have been renamed from */joint_states* to */clegs/joint_states*.

Listing 10.10 Robot State launcher

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen"> <remap
from="/joint_states" to="/clhero_v2/joint_states" /> </node>
```

5.2.3 ClegS Gazebo Package

The gazebo package manages the simulations in Gazebo. It includes the launch directory with the *.launch* file and the *.world* files that can be loaded in Gazebo.

The first launch file is responsible of start the simulation, the script is shown below (Listing 10.11). The first section is to define some variables, the two most important are *world* and *robot_model*. The *world* variable is used by the user to select the world that will appear in Gazebo, the worlds that are included in the package will be explained in the next paragraphs. The *robot_model* variable is responsible for selecting the model of the robot that the user wants to use in the simulation. This variable calls the robot models included in the directory *clegs_common/description/robots/*, by default it launches the model *clhero_v2.xacro*.

The next section in the script is for launch Gazebo and the chosen world. This section presents some arguments that can be modified if the user need it.

- *paused*: Start Gazebo in a paused state (default false).
- *use_sim_time*: Tells ROS nodes asking for time to get the Gazebo-published simulation time, published over the ROS topic/clock (default true).
- *gui*: Launch the user interface window of Gazebo (default true).
- *recording*: Enable gazebo state log recording.
- *debug*: Start gzserver (Gazebo Server) in debug mode using gdb (default false).
- *verbose*: Run gzserver and gzclient with –verbose, printing errors and warnings to the terminal (default false).

Listing 10.11 clegs_gazebo.launch file

```
<?xml version="1.0"?> <launch>

<!-- roslaunch arguments -->
<arg name="debug" default="false"/>
<arg name="gui" default="true"/>

<!-- worlds available:
empty -> empty world.
stair -> world with a stair.
slopes -> world with different slopes.
uneven_terrain -> world with a terrain with obstacles.
-->
<arg name="world" default="empty.world"/>

<!-- models available:
clhero_v2 -> clhero v2.
clhero_v2_pedestal -> clhero v2 on a pedestal.
quad -> four clegs robot.
octo -> eight clegs robot.
-->
<arg name="robot_model" default="clhero_v2"/>

<!-- We resume the logic in empty_world.launch, changing only
the name of the world to be launched
-->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
<arg name="world_name" value="$(find hexapodo_gazebo)/worlds/$(arg world).world"/
    >
<arg name="paused" value="false"/>
<arg name="use_sim_time" value="true"/>
<arg name="gui" value="$(arg gui)" />
<arg name="debug" value="$(arg debug)" />
<arg name="recording" value="false"/>
</include>

<!-- Load the URDF into the ROS Parameter Server -->
<param name="robot_description"
command="$(find xacro)/xacro --inorder '$(find hexapodo_description)/robots/$(arg
     robot_model).xacro'" />

<!-- Call a python script to the run a service call to gazebo_ros to spawn a URDF
     robot -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
    output="screen"
args="-urdf -model hexapodo -param robot_description"/>

</launch>
```

With Gazebo is possible to create custom worlds (.world files) within your own ROS packages that are specific to your robots and packages. Those new worlds can be find in the directory $clegs_sim/worlds/$. ClegS aims to be a complete simulation package for C legs robots, with that purpose it includes several worlds to improve a better debugging of the algorithms. The worlds can be loaded assigning the variable world with one of this variables: *empty_world.world*, *stairs.world*, *slopes.world*, *uneven_terrain.world*. For a better comprehension a brief description of each world can be find in the next paragraph and in the Fig. 19.

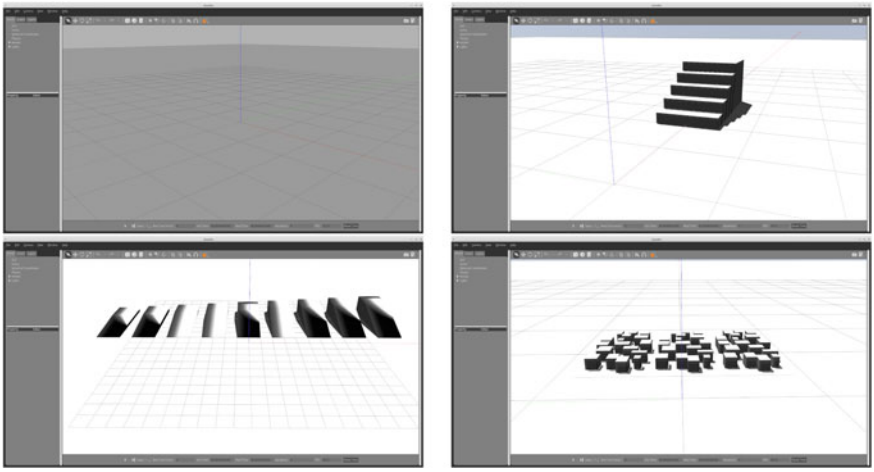


Fig. 19 The 4 worlds available for simulationsi

- *empty_world*: This is the world by default. There is not any obstacle in the scenario. Is ideal to analyze how the robot moves in flat terrains.
- *stairs*: This world includes one stair with X steps at the global point (4, 0, 0). Each step has the dimension of $100 \times 30 \times 17$ mm. It has been designed to develop algorithms for going up and down this kind of obstacles.
- *slope*: This world incorporate 9 different slopes where the percentage of each ramp varies from 10° to 50°. This world is for testing the maximum slope that can achieve the robot, where the 45° slope represents the 100% slope.
- *uneven_terrain*: In this world we have develop a section of terrain the tries to represent the physical conditions after a disaster and that search and rescue robots must complete successfully.

The third part of the script loads the full robot description in the ROS parameter server.

The parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary. Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to protect parameter names from colliding. The hierarchical scheme also allows parameters to be accessed individually or as a tree.

```
<!-- Load the URDF into the ROS Parameter Server --> <param
name="robot_description" command="$(find xacro)/xacro --inorder
'$(find hexapodo_description)/robots/$(arg robot_model).xacro'" />
```

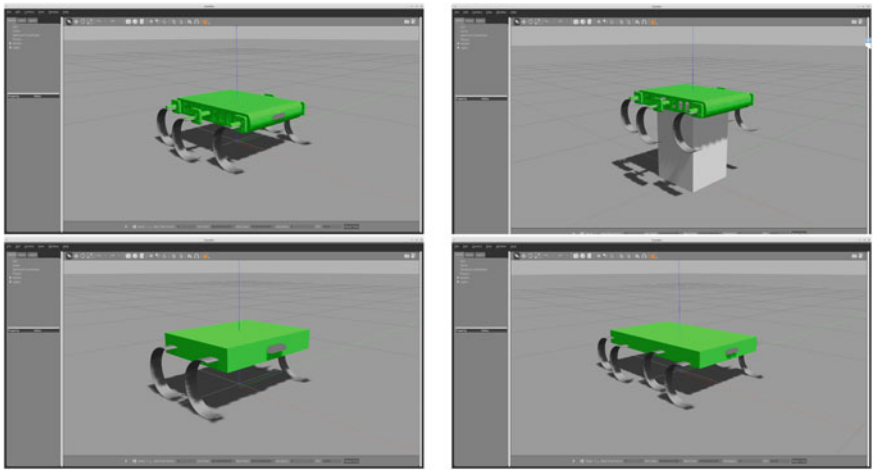


Fig. 20 The 4 models of robots that include ClegS package

As is explained before the variable *robot_description* is use to select the model of the robot that the user wants launch in the simulation. By the default ClegS package have 4 models in the *clegs_common/description/robots/* directory:

- *chlero_v2*: This is the second version of CLHeRo. It is smaller, lighter and includes sensors like the Intel RealSense.
- *chlero_v2_elevate*: This is a version of CLHeRo V2 that is fixed in the position (0, 0, 0.25). This version of the model is useful to test the synchronization of the legs in the first stages of developing an algorithm.
- *quadbot*: This is a low detail model of a quadruped robot.
- *octobot*: This is a low detail model of an eight legs robot.

Figure 20 shows the 4 models in Gazebo simulator.

Finally the script launches the node that spawn the robot into the Gazebo world.

```
<!-- Call a python script to the run a service call to gazebo_ros to
spawn a URDF robot --> <node name="urdf_spawner" pkg="gazebo_ros"
type="spawn_model" respawn="false" output="screen" args="-urdf
-model clhero_v2 -param robot_description"/>
```

5.3 *ClegS Hw Metapackage*

5.3.1 ClegS Hw_control Package

As was explained in Sect. 5.2.1, the aim of ClegS package is to be transparent for the developer. This gets the information from the patterns commands and translate into the commands necessaries to move the Maxon controllers.

5.3.2 CLHeRo Robot Package

This is the package that established the communication with the motors. It also includes the official library from Maxon to get all the functions available for the MCD Epos, but we have create a wrap of it, in order to only have accessible the commands that are needed for controlling the motors (*epos_functions*).

The *epos_functions* also manage the connection with the MCD epos. This connection is established when the constructor is invoke, but firstly is necessary to define the parameters for the connection. For the MCD Epos the parameters are the following:

- g_deviceName_ = "EPOS";
- g_protocolStackName_ = "MAXON_RS232";
- g_interfaceName_ = "RS232";
- g_portName_ = "/dev/ttyS1";
- g_baudrate_ = 115200.

If the user have other models of controllers, like the *EPOS2* or *EPOS4*, some parameters can be different. More information about configuring the communications can be found in the EPOS Communication Guide [7]

- device name: EPOS2, EPOS4.
- protocol stack name: CANopen, MAXON SERIAL V2.
- interface name: USB, CAN_ixx_usb 0, CAN_kvaser_usb 0, …
- port name: COM1, USB0, CAN0.
- baudrate: 115200, 1000000, …

In the Listing 10.12 you can see the functions implemented in the wrapper. Although it seems to have only a few functions, they allow to have a complete control of the robot. And, of course, the wrapper can be extended to achieve the necessities of the user.

Listing 10.12 EPOS functions wraper

```
public:
// Open Epos
int OpenDevice(unsigned int*, int);
// Clear errors
int ClearErrors(unsigned int* );

// Show errors and info
void LogError(std::string , int , unsigned int );
void LogInfo(std::string );

// Motor controlling
int ActivateProfilePosition(int);
int SetPositionProfile(int, int, int, int);
int MoveToPosition(int, int, bool, bool);
int GetPosition(int);
bool HaltPositionMovement(int);
```

Finally, the *.launch* file manages to launch the *clegs_hw_control* and together with the *epos_functions* wrapper.

6 Tutorials

6.1 Launching a Simulation

To check that everything is working correctly, this section will explain a short example. In this first tutorial the reader will learn how to launch the CLHeRo V2 robot with the necessary controllers to apply movement to the legs of the robot. This tutorial has also a video[9]video showing all the steps.

Although the CLHeRo V2 incorporates some sensors like the Intel RealSense D435, in this tutorial we are only focusing just in the movement of the robot.

To execute correctly this tutorial the reader needs to have two terminal windows opened. The first terminal is necessary to launch Gazebo with the selected world, load the robot description of the robot in the ROS parameter server and spawn the robot into the Gazebo world. The second terminal is for launching the controllers for the robot and publish the state of the robot.

Now, the user can launch in the first terminal the file with *world* = *empty_world* and *robot_model* = *chlero_v2_elevate*.

```
$ roslaunch hexapodo_gazebo clegs_gazebo.launch
robot_model:="clhero_v2_elevate"
```

If everything is correct, these are the topics that are accessible:

[9]https://youtu.be/qEIgASd-oCk.

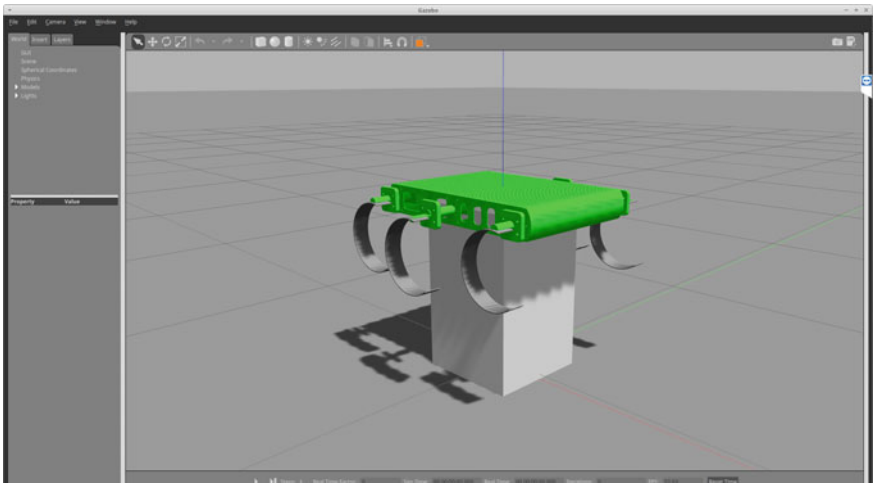


Fig. 21 CLHeRo V2 rise at the point (0, 0, 0.25)

Listing 10.13 List of topics

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/rosout /rosout_agg
```

You can see the CLHeRo V2 in the elevated position, Fig. 21. Also, to check that the model is loaded correctly and all the links are correctly respect the frame of the robot is possible to visualize the transformation tree. To get this, is necessary to run the command $ $rosrun\ rqt_gui\ rqt_gui$. Once, the new window appears, you have to select the option $Plugins \rightarrow Visualization \rightarrow TFTree$ and it will show a transformation tree like the one in Fig. 22.

Now, is time to launch the second script, for this tutorial we are going to select the velocity controllers.

```
$ roslaunch hexapodo_control hexapodo_control.launch
position:="false"
```

If there is not any error, the topic list will have increased considerably because there are now 2 controllers for each leg, and the configurable parameters for each one and also the topic to get the state of the robot. For a better visualization of the state of the controllers, from the rqt_gui you can select the option $Plugins \rightarrow RobotTools \rightarrow ControllerManager$ (Fig. 23). The controller manager is a graphic tool visualizes the state of the controllers and change their state between: stop, start load/unload.

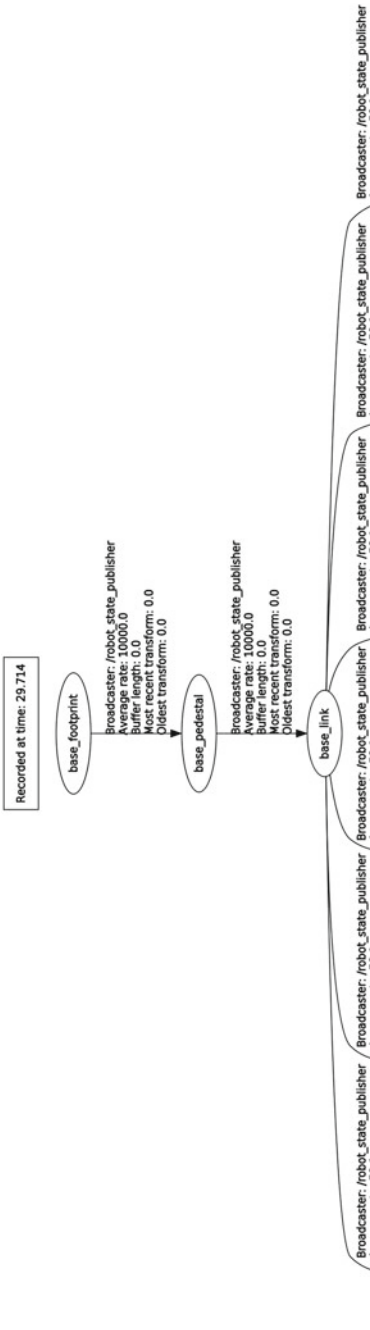


Fig. 22 CLHeRo V2 TF Tree

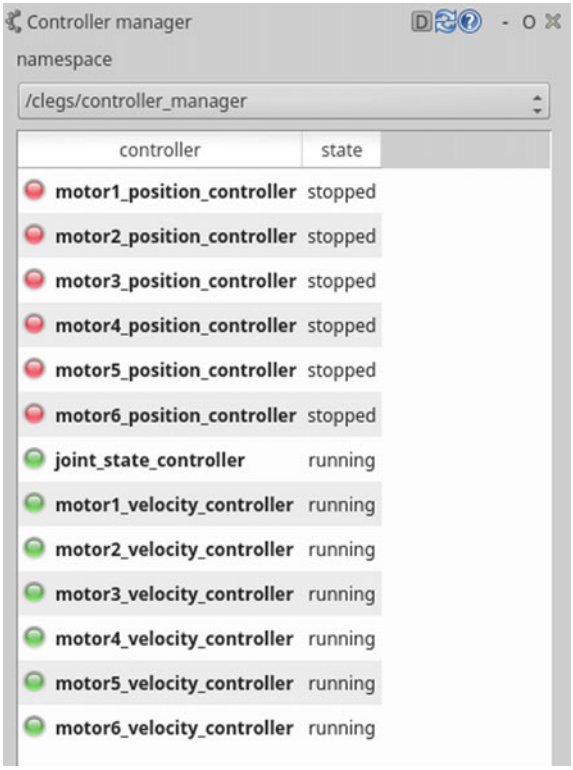


Fig. 23 Controller manager for the CLHeRo V1

Listing 10.14 topic list with the controllers loaded

```
/clegs/joint_states /clegs/motor1_position_controller/command
/clegs/motor1_position_controller/pid/parameter_descriptions
/clegs/motor1_position_controller/pid/parameter_updates
/clegs/motor1_position_controller/state
/clegs/motor1_velocity_controller/command
/clegs/motor1_velocity_controller/pid/parameter_descriptions
/clegs/motor1_velocity_controller/pid/parameter_updates
/clegs/motor1_velocity_controller/state [...]
/clegs/motor6_position_controller/command
/clegs/motor6_position_controller/pid/parameter_descriptions
/clegs/motor6_position_controller/pid/parameter_updates
/clegs/motor6_position_controller/state
/clegs/motor6_velocity_controller/command
/clegs/motor6_velocity_controller/pid/parameter_descriptions
/clegs/motor6_velocity_controller/pid/parameter_updates
/clegs/motor6_velocity_controller/state
```

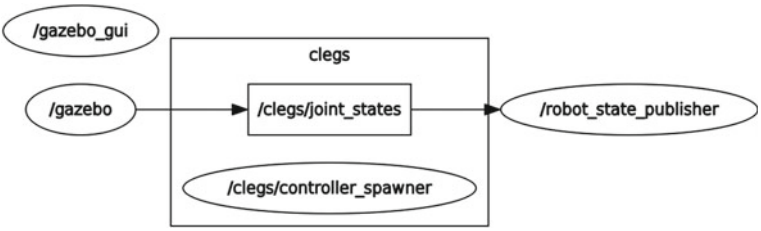


Fig. 24 Nodes and topics actives to control CLHeRo V1 motors

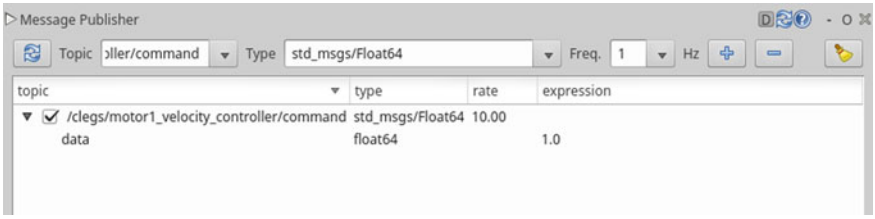


Fig. 25 RQT_gui for publishing the velocity of the motor 1

In Fig. 24 you can see the output of rqt_graph.[10] This tool gives an overview of all ROS nodes that are running, and the topics on which the nodes are communicating, which is very helpful for debugging and understanding the connection between nodes.

At this point the reader can start moving the legs of the robot. You can use the traditional command via terminal:

```
$ rostopic pub -r 10 /clegs/motor1_velocity_controller/command
std_msgs/Float64 "data: 0.0"
```

Or, you can go back to the *rqt_gui* window and add the $Plugins \rightarrow Topics \rightarrow Message: \ Publisher$. With the new plugin (Fig. 25) you can select the topic, the rate and the value of the desired velocity to be publish.

6.2 Teleoperating a Simulation

This second tutorial goes a bit further than the first one. It will show how to launch a robot on an empty world, load the the GP that the user select and teleoperate the robot around the Gazebo world. This tutorial has also a video[11] video showing all the steps.

Firstly, we are going to launch Gazebo with the empty world and with the CLHeRo V2 robot on the ground. Once Gazebo has loaded the robot, we can launch the *clegs_control* just as it was explained in the previous tutorial. Then we have to

[10] http://wiki.ros.org/rqt_graph.

[11] https://youtu.be/en61xOmp7mI.

launch the *gait_controller*, which was explained in section (5.1.2). Every command, must be in a different terminal.

```
$ roslaunch clegs_gazebo clegs_gazebo.launch $ roslaunch
clegs_control clegs_control.launch $ roslaunch
clhero_gait_controller gait_controller.launch
```

Finally, we have to launch the node *control_terminal* that is included in the package *gait_controller*. This new node, is a terminal interface to control the robot. The first menu (Listing 10.15) gives the option to start moving the robot, pause or stop the movement, force one state and update the movement arguments. To start the movement, is necessary to insert option #1. Then, the second menu appears (Listing 10.16), offering the move options for the robot. If for example, we want the CLHeRo V2 to walk in alternating tripod, we push option #5.

```
$ rosrun clhero_gait_controller control_terminal
```

Listing 10.15 Options of the first menu

```
$ rosrun clegs_gait_controller control_terminal
-----------------------------------
Terminal for clegs gait control
-----------------------------------
Select a command:
\cite{ch10Abkowitz1964} Start gait pattern
\cite{ch10Antonelli2013} Stop gait pattern
\cite{ch10Dudek2007} Pause gait pattern execution
\cite{ch10Fossen2002} Continue paused execution
\cite{ch10Fossen2011} Force gait state
\cite{ch10Kelly2005} Update movement arguments
[h] Show main menu
[q] Exit
-----------------------------------
[Command]:
```

Listing 10.16 Options of the second menu

```
-----------------------------------
[Command]: 1
-----------------------------------
-- Start gait pattern --
Available patterns:
\cite{ch10Abkowitz1964} complete_wave
\cite{ch10Antonelli2013} test_gait_pattern
\cite{ch10Dudek2007} position_control_test
\cite{ch10Fossen2002} reverse_position_control_test
\cite{ch10Fossen2011} alternating_tripod
[6] turn_right_tripod
\cite{ch10maxon2006} turn_left_tripod
Introduce the desired pattern to start (press q to cancel):
```

6.3 Create a New Robot Model

In this tutorial, we are going to explain how the reader can create a new robot model, so he can be able to use the ClegS package with his own robot.

Whit this purpose we have create some files with the subfix *_template* in the *clegs_description* package. In this tutorial, the reader will be able to generate the model of a new hexapod robot.

First of all, we go to *clegs_description/robots* folder and select the file *clegs_template.xacro*. The first action is to modify the file name, in this case we are going to use *example_robot.xacro*, also, this name can be our robot name, which has to be modified in the second line of the file (Listing 10.17).

Listing 10.17 Name of the robot fot the tutorial

```
<?xml version="1.0" ?> <robot name="example_robot"
xmlns:xacro="http://ros.org/wiki/xacro">
```

Then, in the section *Files to include for the robot* we have to select the model of the leg that we want to use (*cleg_100* or *cleg_80)*, the name of the base of the robot, for this example will be *example_base.xacro* and if we want to have any sensor, but in this tutorial we will let it comment (Listing 10.18).

Listing 10.18 Name of the robot fot the tutorial

```
<!-- Files to include for the robot --> <xacro:include
filename="$(find clegs_description)/urdf/cleg_80.robot.xacro"/>
<xacro:include filename="$(find
clegs_description)/urdf/base_example.xacro"/> <!--xacro:include
filename="$(find
clegs_description)/urdf/sensores/d435.urdf.xacro"/-->
```

The next section is for including the Gazebo ROS control plugin. We have to modify the *namespace* to */example_robot*. Listing 10.19

Listing 10.19 Gazebo ROS control plugin

```
<gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/example_robot</robotNamespace>
    </plugin>
</gazebo>
```

And now, we are going to use the macros included in the file to include the legs of the robot. This section of the code is modular, that means, that if the user wants a different number of legs in his robot, he only have to modify the number of times that he has the macro in the file.

This macro needs as parameter, one identifier that can be the number of leg or motor, the parent link, which always is the *base_link*. One boolean value that specifies if the leg is in the right or left side of the robot, and the position and orientation of the leg (Listing 10.20). This macro, is used by the legs file.

Listing 10.20 Macro for defining the legs of the robot

```
<motor name="motor_1" parent_link="base_link" right="0">
    <origin xyz="${-235/1000} ${233/1000} ${0/1000}" rpy="0 0 0"/>
</motor>

<motor name="motor_2" parent_link="base_link" right="1">
    <origin xyz="${-235/1000} ${-233/1000} ${0/1000}" rpy="0 0 0"/>
</motor>

<motor name="motor_3" parent_link="base_link" right="0">
    <origin xyz="${0/1000} ${289/1000} ${0/1000}" rpy="0 0 0"/>
</motor>

<motor name="motor_4" parent_link="base_link" right="1">
    <origin xyz="${0/1000} ${-289/1000} ${0/1000}" rpy="0 0 0"/>
</motor>

<motor name="motor_5" parent_link="base_link" right="0">
    <origin xyz="${235/1000} ${233/1000} ${0/1000}" rpy="0 0 0"/>
</motor>

<motor name="motor_6" parent_link="base_link" right="1">
    <origin xyz="${235/1000} ${-233/1000} ${0/1000}" rpy="0 0 0"/>
</motor>
```

The last section of this file, that is comment for this example, is for adding the sensors.

Now, we are going to modify the *chassis_temple* file, is much simpler than the template for the defining the robot. This file manages to generate the chassis of the robot. The first variable that we have to modify is the leg length, by default is for the smaller leg, as in this example, but if the user prefers the other leg, he has to change this value to 100, Listing (10.21).

Listing 10.21 Variable for the radius of the leg

```
<xacro:property name="leg_radius" value="${80/1e3}"/>
```

Then, we have to move to the section *chassis*. In this section the reader have to complete with the information of his robot. He needs to complete the origin of inertia of the robot, the mass of the chassis, and the inertia matrix of the robot. If he does not have this information, we can assume, that as happens with the CLHeRo V2, that the mass is centered in the origin of the robot. So he can assume that the origin of the inertia is located at (0, 0, 0), the mass of the robot can be any value in kilograms and the inertia matrix we will fill it as an identity matrix (Listing 10.22). Finally the user can import the *.stl* file of his chassis, or just create a box with the dimensions of the robot as in this example.

Listing 10.22 Data for the chassis of the robot

```
<!-- CHASIS --> <link name="base_link">

    <inertial>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <mass value="5"/>
        <inertia ixx="1"
```

```
            ixy="0" iyy="1"
            ixz="0" iyz="0" izz="1"/>
    </inertial>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="${600/1000} ${460/1000} ${80/1000}"/>
        </geometry>
        <material name="base_link_material">
            <color rgba="${51/255} ${102/255} ${0/255} 1"/>
        </material>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="${600/1000} ${460/1000} ${80/1000}"/>
        </geometry>
    </collision>

</link>
```

Finally in the *clegs_leg* files the user does not have to modify anything, except is he wants to create his own leg, change the motor parameters. For changing the motor parameters he only has to modify the variables with this parameters: reduction of the motor, maximum torque of the motor and maximum revolutions of the motor, Listing 10.23

Listing 10.23 Parameters for the motor

```
<xacro:property name="motor_reduction" value="${529.0/16}"/>
<xacro:property name="motor_max_effort" value="${3.75}"/>
<xacro:property name="motor_max_rev"
value="${1256.637/motor_reduction}"/>
```

And with this, the creation of a new robot has ended. Now, the user can launch his own new robot as is described in the Sect. 6.1.

6.4 Install MAXON Drivers

This section is only valid for the Maxon Epos MCD drivers. Before creating all the ClegS package, we tried to use some packages available in the ROS repositories (epos_hardware[12] and epos_driver[13]) and also we explore other communication packages for CANopen (ros_canopen[14] and canopen_master[15]). Only with *epos_hardware* we get some control on the motors, but there was a big latency, so we could not do a very precise control.

Right now, with the *CLHeRo_robot* package (Sect. 5.3.2) the user can control the Maxon motors without any problem. But firstly, is necessary to install the Maxon

[12] http://wiki.ros.org/epos_hardware.

[13] http://wiki.ros.org/epos_driver.

[14] http://wiki.ros.org/ros_canopen.

[15] http://wiki.ros.org/canopen_master.

official libraries. When we start the project, this process was a manually process, but with the last update of the Maxon drivers (Version 6.5.1.0), they have include a script that realizes the process automatically.

The new script is include in the drivers folder of Maxon, you can download it from here.[16]

Once you download the *.zip* file. Check first, that in the unzipped Folder */EPOS_Linux_Library* the file *install.sh* is existing:

```
jorge@silth:~/Downloads/EPOS_Linux_Library$ ls examples include
install.sh lib misc jorge@silth:~/Downloads/EPOS_Linux_Library$
```

After that, start the script (You have to be in the folder where the install.sh lies). If everything is correct, you will see an output as in Listing 10.24

```
$ sudo bash ./install.sh
```

Listing 10.24 Install of the script for the Maxon drivers

```
jorge@silth:~/Downloads/EPOS_Linux_Library$ sudo bash ./install.sh
[sudo] password for jorge:
-----------------------------------------
EPOS Command Library 6.5.1.0 installation started
-----------------------------------------
- Remove existing installation [OK]
- Install library into directory: /opt/EposCmdLib_6.5.1.0 [OK]
- Install examples into directory: /opt/EposCmdLib_6.5.1.0 [OK]
- Library system integration [OK]
- Configure device access rights [OK]
- Configure user access rights [OK]
-----------------------------------------
EPOS Command Library 6.5.1.0 installed
-----------------------------------------
jorge@silth:~/Downloads/EPOS_Linux_Library$
```

Finally, if you are using a Maxon controller that is connected via RS232-USB, you have to create a symbolic link every time you restart the computer (Listing 10.25), or also, generate a script that can do it automatically, so the ROS package can established the connection.

Listing 10.25 Creation of the symbolic link for the TTYUSB connection

```
jorge@silth:~$ sudo rm /dev/ttyS1 [sudo] password for jorge:
jorge@silth:~$ sudo ln -s /dev/ttyUSB0 /dev/ttyS1 jorge@silth:~$
```

7 Conclusions and Future Work

In this chapter, we presented an overview of the necessary background to understand the basics for C-legs robots. We then explained what is included in the meta-package,

[16] https://www.maxonmotor.com/medias/sys_master/root/8831822430238/EPOS-Linux-Library-En.zip.

and finally we explained step-by-step how to run a simulation with existing models, and after, how to teleoperate the simulation. As advanced tutorials, we presented instructions on how to implement a custom c-legged robot, and how to install the necessary drivers to control the real robot with the ClegS. We believe that this meta-package will be very helpful for both research and teaching, as it gives an easy start into the topic.

One of the most important contributions of this package is high level of abstract for the user when programming the GPs for the robot. While in other ROS packages you have to deal with low level programming, for example reading encoders or configuring PWM signals for the motors, with this package the reader only have to program the sequence of the legs with finite state machines which are very intuitive to use.

Another good point of this simulation package is its versatility. We have focus in create an environment that can be used by any type of C-legged robot, not only for the CLHeRo platform. So, for example, the research groups that works with similar platforms (Carniege Mellon or Middle East Technical University) can adapt it to their platforms. And that give to everyone the possibility to develop the same GP for different robots and test the efficiency of it.

For the next versions of the ClegS we are working in create a graphical user interface, so the user will not need to send the all the orders via nodes or commands in the terminal and also to improve the compatibility with Maxon USB controllers and Dynamixel motors.

Acknowledgements Work result of research activities carried out at the Centre for Automation and Robotics, CAR (UPM-CSIC), within the Robotics and Cybernetics research group (RobCib). Supported by the RoboCity2030-DIH-CM, Madrid Robotics Digital Innovation Hub, S2018/NMT-4331, and the UPM Program project VJIDOCUPM18JCG.

References

1. M.A. Abkowitz, Lectures on ship hydrodynamics-steering and maneuvering. Technical report, Hydro-and Aerodynamics Laboratory Report Hy-5, Lyngby, Denmark (1964)
2. G. Antonelli, *Underwater Robots*, 3rd edn. (Springer Publishing Company Incorporated, Berlin, 2013)
3. G. Dudek, P. Giguere, C. Prahacs, S. Saunderson, J. Sattar, L. Torres-Mendez, M. Jenkin, A. German, A. Hogue, A. Ripsman, Aqua: an amphibious autonomous robot. Computer **40**(1), 46–53 (2007)
4. Th.I. Fossen *Marine Control Systems, Marine Cybernetics* (2002)
5. Th.I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control* (Wiley, New York, 2011)
6. R. Kelly, V. Santibanez, A. Loria, *Control of Robot Manipulators in Joint Space* (Springer, London, 2005)
7. Maxon Motor Ag, Brunigstrasse 220. P.O. Box 263 CH-6072 Sachseln. *EPOS Positioning Controller. Communication Guide* (2006)
8. D. McMordie, M. Buehler, Towards pronking with a hexapod robot, in *International Conference on Climbing and Walking Robots*, Karlsruhe, Germany (2001)

9. E.Z. Moore, D. Campbell, F. Grimminger, M. Buehler, Reliable stair climbing in the simple hexapod 'rhex'. Proc. IEEE Int. Conf. Robot. Autom. **3**, 2222–2227 (2002)
10. N. Neville, M. Buehler, I. Sharf, A bipedal running robot with one actuator per leg, in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006* (2006), pp. 848–853
11. H. Marc, *Raibert* (Legged Robots That Balance. Massachusetts Institute of Technology, Cambridge, MA, USA, 1986)
12. U. Saranlı, M. Buehler, D.E. Koditschek, Rhex: a simple and highly mobile hexapod robot. Int. J. Robot. Res. **20**(7), 616–631 (2001)
13. U. Saranlı, A. Rizzi, D. Koditschek, Model-based dynamic self-righting maneuvers for a hexapedal robot. Int. J. Robot. Res. **23**(9), 903 (2004)
14. T. Weihmann, P. Brun, E. Pycroft, Speed dependent phase shifts and gait changes in cockroaches running on substrates of different slipperiness. Front. Zool. **14**(1), 54 (2017). Dec
15. J.D. Weingarten, D.E. Koditschek, H. Komsuoglu, C. Massey, Robotics as the delivery vehicle: A contexualized, social, self paced, engineering education for life-long learners, in *Robotics Science and Systems Workshop on Research in Robots for Education* (2007)

Jorge De León Rivas received the Bachelor Degree in Industrial Electronics from University of La Laguna in 2013 and the MsC in Automation and Robotics from the Technical University of Madrid (UPM) in 2015. His main research interests are focused on designing new mobile robots and and algorithms for searching and surveillance missions. He is currently a researcher at the Robotics and Cybernetics research group of the Centre for Automatic and Robotic (CAR UPM-CSIC) where he is developing his PhD thesis.

Raúl Cebolla Arroyo received his bachelor degree in industrial technologies engineering from Technical University of Madrid in 2017. Since 2017, he is studying for two master's degrees in Industrial Engineering and in Automation and Robotics in the Technical University of Madrid. His research interests are systems control, mobile robotics, machine learning and artificial intelligence.

Jaime del Cerro Giner received his Ph.D. degree in Robotics and Computer vision at Polytechnic University of Madrid at 2007. He has been working in Robotics for more than 20 years, participating in several local and European Framework projects, ESA (European Space Agency) and EDA (European Defense Agency) as well as commercial agreements with relevant national companies. Currently is a member of the CAR (Center for Automation and Robotics) UPM-CSIC and teaches Robotics and control in the Escuela Técnica Superior de Ingenieros industriales of Madrid.

Antonio Barrientos Cruz received the MsC Engineer degree by Automatic and Electronic from the Polytechnic University of Madrid in 1982, and the PhD in Robotics from the same University in 1986. In 2002 he obtained de MSc Degree in Biomedical Engineering by Universidad Nacional de Educación a Distancia. Since 1988 he is Professor of robotics, computers and control engineering at the Polytechnic University of Madrid. He has worked for more than 30 years in robotics, developing industrial and service robots for different areas. Currently, he is the head of the Robotics and Cybernetics research group of the Centre for Automatic and Robotic in the Technical University of Madrid - Spanish National Research Council. The main interests of his research group are in the field robotics (ground and aerial), seeking its application in two areas: agriculture and security, search and rescue scenarios.

Emerging Topics

Video Frames Selection Method for 3D Reconstruction Depending on ROS-Based Monocular SLAM

Yasin Maan Yousif and Iyad Hatem

Abstract In this research chapter, a new method for building a point cloud for an object in a static scene is presented. The method uses images taken by an RGB camera mounted on a controllable robot while moving around that object. During the estimation of the pose of every video frame, a selection method is applied to extract the best frames from the video. Based on these selected images and their estimated rotation and transition vectors, a sparse 3D reconstruction process is conducted. The estimation of these vectors is done by applying Extended Kalman Filter to solve the Simultaneous Localization and Mapping (SLAM) problem with ROS (Robotics Operating System) as a framework. Covariance information provided by Kalman filter is utilized as additional selection criterion. Then, a ROS-based sparse bundle adjustment (SBA) process is performed on both of the new point cloud and the estimated pose vectors. Finally, a dense 3D reconstruction is performed on the optimized values of the rotations and transitions vectors to get a denser point cloud. This method is tested using simulation in Gazebo framework and the results are discussed. All the experiments are explained in details in this chapter. The source code of this project is available online and divided to two public repositories, one for the filtering phase (https://github.com/engyasin/EKF-MonoSLAM_for_3D-reconstruction) and the other for the 3D reconstruction phase (https://github.com/engyasin/3D-reconstruction_with_known_poses).

Keywords 3D reconstruction · Covariance matrix · Extended Kalman filter · Simultaneous Localization and Mapping · Sparse Bundle Adjustment · Structure from motion

Y. M. Yousif (✉)
Al-Baath University, Homs, Syria
e-mail: yyousif@albaath-univ.edu.sy

I. Hatem
University of Tishreen, Lattakia, Syria
e-mail: iyadhatem@tishreen.edu.sy

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_11

1 Introduction

There are many applications for the process of 3D reconstruction from images. For example, making 3D models or virtual environments for video games, or generating digitalized 3D models for a heritage site or a precious piece [1]. However, using only 2D images to do this is considered a hard task, but at the other hand, the availability and the low cost of RGB cameras as used here, give advantages over other solutions such as 3D scanners

Also mounting cameras on robots maybe used for an automated way to take images and enhances the ability to scan vast or hard-to-reach areas. One important usage of this way is in building a 3D map for the robot itself, so that it can remember places it passed through previously [2].

Other possible implementation of using monocular data for 3D reconstruction is in the medical field, where the endoscope camera gives live video feed of the human inner organs [3, 4].

The main goal of this research is to enhance the 3D reconstruction process, by providing additional information from the covariance matrices. The covariance is estimated from the Simultaneous Localization and Mapping (SLAM) process.

In the Visual SLAM stage, a group of images is selected, depending on the regular geometrical distribution among the images and on a minimum covariance of their poses. A subset of all the video frames must be selected, otherwise using all frames requires heavy computations and may not always increase the accuracy [5]. From these images, an initial point cloud is built, then a SBA problem is solved for that point cloud. Finally, a dense 3D reconstruction program of the central object is performed.

Visual SLAM is the first stage in this research, and as a real time process, there is a limit to the density and accuracy of its results. However, with the second stage applied here, which is 3D reconstruction followed by Sparse Bundle Adjustment (SBA), the accuracy and density can be increased. Unlike SLAM, 3D reconstruction is an offline process which allows the usage of all the images in reconstructing the 3D structure at the same time and not necessarily in its acquisition order.

In the next section, a review of the previous studies of the problem addressed in this chapter is presented. The third section is an overview of the complete method. The fourth is a general background of the Extended Kalman Filter equations and its implementation in the Mono-SLAM problem.

After that, there are two main sections. The first is about ROS environment configurations and tools used to implement the proposed method. The second presents several tests performed in this work using the described method. The tests are followed by a discussion and explanation of the results. Lastly, a general conclusion is made on the efficiency and the performance of the method.

2 Related Work

In the literature, some methods have used visual SLAM for 3D reconstruction. For example, the paper in [6] uses the visual SLAM process to make a final point cloud in an offline process. In that paper, images are selected according to the geometrical positions of the camera. This work also [6] runs a real-time enhancement process which increases the accuracy of the selected poses subset by smoothing past poses. This method, however, fails to avoid blurred or noisy images.

Another method for selecting frames from a video for 3D reconstruction is proposed in [5] where the camera poses are computed offline instead of using visual SLAM, and the image selection process is performed in two steps. First, it takes an initial group of images according to regular distribution, then it adds more images to maximize the total number of matched features.

At the other hand, in the field of Visual SLAM, a wide variety of methods exists. The paper in [7] is considered one of the first researches that uses Extended Kalman filter for monocular SLAM problems to track the camera poses and the environment map simultaneously.

After that research, few improvements were made on this method. For example, a better point representation method for the initialization is used, which is called inverse depth representation [8]. Another improvement is about using the active search method for matching [9].

The usage of the Extended Kalman filter has an advantage of a small computation time which is suitable for real time programs, but it suffers from the linear approximation and other problems when working in a dynamic environment that is assumed to be static. Other improvements are made in [9], such as dealing with occluding objects.

Additionally, the method in [9] predicts the blur resulting from the fast movement of the camera. This prediction increases the matching accuracy. The last version of Mono-SLAM in [9] is the closest to what is implemented here, although it still suffers from the constant speed model errors and non-linearity errors.

However, there are other recent approaches that use Kalman Filter to solve the localization problem such as [10]. In this approach only the camera pose is updated based on the matching results between the current frame and the previous one. This method does not build a point cloud or a map in real time. Such a map may be useful for other improvements, like loop closing or recovering pose estimation after an occluding object passes in front of the camera.

All other approaches, where the problem is only concerned with finding the poses of the camera without the 3D map, are considered solutions for the Visual Odometry problem [11] and not for SLAM.

Generally, in the field of visual SLAM there are other non-filtering approaches. These approaches use an error minimization technique called Local Bundle Adjustment (LBA), instead of filtering.

LBA methods are proven to have more accuracy than filtering in some cases, as shown by [12]. However, filtering methods that use Kalman Filter are known for their

ease of implementation, especially in the case of sensors with different models. In addition, Kalman Filter gives covariance information for the estimated parameters directly.

Another classification of Visual SLAM solutions considers two categories. The first is called Indirect Methods, which is also known as Feature Based Methods. It includes the EKF-SLAM method. The second category is called Direct Methods such as [13, 14]. One difference between the two categories is that the direct methods use all the pixels in the image for tracking and building the map, which makes the computations slower, but as a result, the point cloud will be very dense in comparison.

In the 3D structure computation stage in this work, a well-known method in photogrammetry, detailed in Chap. 12 of [15] is used. After that, a Sparse Bundle Adjustment step is performed to minimize the errors in the result as in [16].

3 System Overview

Figure 1 shows a general model for the different parts of the presented system here. As mentioned earlier, this system consists of two parts. The first part is done in real-time and it receives its input from a simulation of a remotely controlled robot with a camera. This part takes video frames and velocity sensor readings as inputs to the kalman filter. The filter estimates the state vector and its covariance for every frame. After each reading of a new frame and updating the state, a set of conditions is tested for that frame to determine whether it will be selected. Its final output is the selected images group with its covariance matrices.

The second part is done offline. It starts by building a sparse point cloud from the saved images and its poses. These points and their 2D projections in addition to camera poses and covariance matrices are passed as inputs to a Sparse Bundle Adjustment process (SBA) that minimizes the total error in the estimates.

Finally, a dense 3D reconstruction is performed on the SBA optimized output to get the requested 3D model.

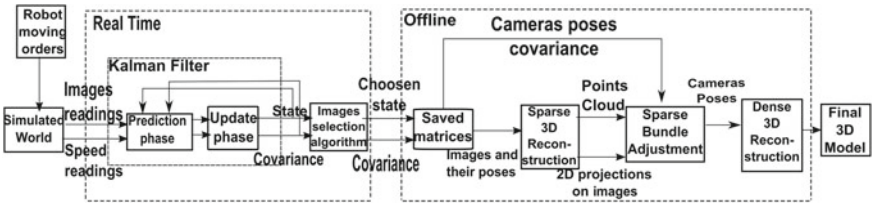


Fig. 1 Overview of the system

Algorithm (1): Prediction step for Extended Kalman Filter

input: $\mu_{\mathbf{k|k}}, \mathbf{\Sigma_{k|k}}, \mathbf{u_k}$
output: $\mu_{\mathbf{k+1|k}}, \mathbf{\Sigma_{k+1|k}}$

$\mathbf{G_k} = (\frac{\partial g}{\partial \mu}|_{\mu_{k|k}})$

$\mathbf{F_k} = (\frac{\partial g}{\partial n}|_{\mu_{k|k}})$

$\mu_{\mathbf{k+1|k}} = g(\mu_{\mathbf{k|k}}, \mathbf{u_k})$

$\Sigma_{k+1|k} = G_k \Sigma_{k|k} G_k^T + F_k P_n F_k^T$

4 Background

4.1 Extended Kalman Filter

In this subsection, a general review of the mathematical background of Extended Kalman Filter is presented. This filter processes sensors data in non-linear systems. Its work consists of two phases, predication phase and update phase.

This filter is the most common type of kalman filters for non-linear systems. This popularity is a result of its obvious application manner. It uses the same linear kalman relations but for an approximated model of the system, where the derivatives of all the non-linear relations are used instead of the original formulas.

The Kalman filter was introduced first in [17]. Later, the Extended Kalman Filter was proposed and used by many researches to solve the Mono-SLAM problem such as [7, 9]. The following algorithm presents the general equations of the update and predict steps for the extended kalman filter.

If $\mu_{k|k}$ is the state vector in the step k based on the information from step k, Σ_k is its covariance matrix, $\boldsymbol{u}$ is the control vector, *g()* is the process function, P_n is the noise covariance, and $\boldsymbol{n}$ is the noise vector, the prediction step will be as in Algorithm 1.

While the update step, is shown in Algorithm 2. Where $\boldsymbol{z_k}$ is the measurement vector, $h()$ is the transformation function between the state and measurement spaces, **W** is the measurement covariance matrix and **w** is the measurement noise vector.

4.2 State Vector Representation in Mono-SLAM

The state vector for the Mono-SLAM problem consists of the position of the camera center, the camera rotation, the camera transitional and rotational velocity, as well as the points positions. The covariance of the state is a square matrix estimated by the filter and used for representing the Gaussian noise of the state vector.

Algorithm (2): Update step for Extended Kalman Filter

input: $\mu_{\mathbf{k|k-1}}, \mathbf{\Sigma_{k|k-1}}, \mathbf{z_k}$
output: $\mu_{\mathbf{k|k}}, \mathbf{\Sigma_{k|k}}$

$\mathbf{H_k} = (\frac{\partial h}{\partial \mu}|_{\mu_{k|k-1}})$
$\mathbf{M_k} = (\frac{\partial h}{\partial w}|_{\mu_{k|k-1}})$
$\mathbf{e_k} = z_k - h(\hat{\mu}_{k|k-1})$
$S_k = H_k \Sigma_{k|k-1} H_k^T + M_k W M_k^T$
$K_k = \Sigma_{k|k-1} H_k^T \ S_k^{-1}$
$\mu_{\mathbf{k|k}} = \mu_{\mathbf{k|k-1}} + \mathbf{K_k} e_k$
$\Sigma_{k|k} = (\mathbf{I} - K_k H_k) \Sigma_{k|k-1}$

If $\boldsymbol{x_k}$ is the vector containing the camera pose and velocity in the time step k, and $f_0 \rightarrow f_m$ are the points positions from point 0 to point m, the state $\boldsymbol{\mu}$ will be given by Eq. 1.

$$\boldsymbol{\mu}_k = \begin{bmatrix} \boldsymbol{x}_k \\ f_0 \\ \cdot \\ \cdot \\ \cdot \\ f_m \end{bmatrix} \in \mathbb{R}^{n \times 1} \tag{1}$$

$\boldsymbol{x_k}$ is given by Eq. 2.

$$\boldsymbol{x_k} = [\boldsymbol{r_k} \ \boldsymbol{q_k} \ \boldsymbol{v_k} \ \boldsymbol{w_k}]^{\boldsymbol{T}} \tag{2}$$

Where $\mathbf{r}_k = (\mathrm{x}_c \ \ \mathrm{y}_c \ \mathrm{z}_c)$ is the camera center position relative to the world coordinates and $\boldsymbol{q_k}$ is the camera rotation relative to the world coordinates represented as a vector of four elements called Quaternion. $\boldsymbol{v_k}$ is the transitional velocity of the camera and $\boldsymbol{w_k}$ is the rotational velocity of the camera.

The prediction model assumes that the scene is static, therefore the points positions don't change in the prediction step.

An additional element in the state vector $\boldsymbol{\mu}$ is a scale factor for all other elements. This factor can be adjusted and updated when an object with known dimensions is observed in the scene.

4.3 Prediction Step in Mono-SLAM

Prediction step in Mono-SLAM predicts the state vector for the next step, depending on the information from the current step and the system model. Our system model, and also the model in [9] is the constant speed model, with additional noise vectors affecting the speed prediction.

If x_k is the state vector in the k step, then the system model function $g()$ is given by Eq. 3.

$$x_{k+1} = g_v(x_k) = \begin{bmatrix} r_k + (v_k + \Delta V_k)\,\Delta t \\ q_k \times quat\,((\omega_k + \Delta\Omega_k)\,\Delta t) \\ v_k + \Delta V_k \\ \omega_k + \Delta\Omega_k \end{bmatrix} \tag{3}$$

Where r_k is the camera center coordinates, q_k is the quaternion representing the camera rotation, x_{k+1} is the state vector in the step $k+1$ and $\boldsymbol{\Delta V_k, \Delta\Omega_k}$ are the noise vectors for the transitional and rotational speed respectively. These last vectors represent the information from the readings of the odometry sensor. The operation $q_k \times quat\,((\omega_k + \Omega_k)\,\Delta t)$ represents the quaternion cross product that gives the equivalent rotation of the two multiplied rotations. $quat()$ is the transforming function from vector rotation to quaternion rotation representation.

4.4 Update Step in Mono-SLAM

This step uses the difference between the measurement vector and the prediction vector of the features to find the innovation vector ($\boldsymbol{e_k}$ in Algorithm 2).

After calculating the kalman matrix K, the state estimation is updated. In order to apply EKF to the problem here, the matrices derivatives (jacobians) are calculated to be used for the covariance calculation. This is done to approximate the nonlinear model to a linear one. Fortunately, once the formulas of the jacobians are found, all that is left to do is to evaluate these formulas for every filter step, without the need to re-calculate them.

The mathematical relations are calculated in [9, 18]. However, a Jupyter Notebook is provided with the source code of the filter[1] to show how these formulas are found in symbols using Sympy[2] python library. These formulas include the jacobians of the Image Distortion Model, Covariance Prediction, and Normalizing Formula of Quaternions.

4.5 Sparse Bundle Adjustment (SBA)

Images are noisy data in general. This will result in a low accuracy for 3D reconstruction process from images. One solution to this is to perform an optimization process, known as Bundle Adjustment [19]. It optimizes jointly both of the point cloud and the images poses (external parameters) by minimizing a cost function related to the

[1] https://github.com/engyasin/EKF-MonoSLAM_for_3D-reconstruction.

[2] https://www.sympy.org/.

errors in the system. This cost function uses the sum of the differences between the detected position and the reprojected position of every point in the point cloud.

This process is iterative and maybe slow for a large number of points and images. Fortunately, the sparse nature of the matrices involved in the solution helps in accelerating the process [16]. A sparse matrix here is a matrix where a number of rows and columns are all zeros. This variation of Bundle Adjustment is abbreviated as SBA.

4.6 1-Point RANSAC Method

The existence of any outlier in the matching result will cause more errors and bigger covariance of the filter estimates. Therefore, another noise filtering operation is needed.

The RANSAC method (Random Sample Consensus) is used to help in such problems by filtering measurements data from inconsistent data points [20].

RANSAC is an iterative method used to estimate the parameters of a mathematical model from a group of measurements that may include outliers. First, many hypotheses are put for the model using the minimum number of measurements points required to define such a model.

For each hypothesis, a consensus set is computed and the largest consensus set is taken as the solution Inliers. Lastly, the model parameters are updated for all the Inliers.

For the Mono-SLAM case, a variant of RANSAC called 1-point RANSAC is proposed in [21]. This method uses the information of the EKF to put hypotheses and compute their consensus sets. As a result, two types of Inliers are found. The first is called Low-Innovation Inliers and they are usually the far points in the scene; they give information about the rotation of the camera pose. These Inliers are annotated by red squares in the Rviz window for the image feed topic in this project. The second type of Inliers is High-Innovation Inliers which give information about the transition of the camera pose. These points are the near points to the camera and are annotated by green squares in Rviz image viewer window.

1-point RANSAC uses only one random point to define hypotheses. The selected consensus set is the Low-Innovation Inliers set. The High-Innovation Inliers are selected in a second step according to the compatibility with the Low-Innovation Inliers.

4.7 Feature Representations

The first few versions of Mono-SLAM used only one method for representing 3D points, and it is called the Euclidean method. This representation has problems related

to features covariance initialization, so another representation was proposed in [8]. It was called Inverse Depth representation.

- **The Euclidean representation** represents each point **Y** by three coordinates [x, y, z] in the world coordinate frame. This is a clear and easy way for representing points, but it has some issues like the big covariance of the far points.
- **The Inverse Depth representation** For a point $\boldsymbol{\psi}$, this representation is given by:

$$\Psi = (x_c \; y_c \; z_c \; \theta \; \phi \; \rho) \tag{4}$$

Where $\mathbf{r}_c = (x_c \; y_c \; z_c)$ is the position of the camera center at the first detection of the Ψ point. ϕ and θ are the angles formed between the arrow pointing from the camera center to the point and the image coordinate axis (Vertically ϕ, horizontally θ).

Finally, $\rho = \frac{1}{d}$ is the inverse of the depth d of the point. This solves the previous covariance problem, because a small range of covariance for ρ will represent a wide range of distances from the camera now.

The transformation from the Inverse Depth to the computationally easier Euclidean representation can be done using the relation:

$$\boldsymbol{y} = \boldsymbol{r}^C + \frac{1}{\rho}\boldsymbol{\eta}\,(\theta, \phi)\,, \tag{5}$$

Where:

$$\boldsymbol{\eta}(\theta, \phi) = \left[\, sin\theta cos\phi \;\; -sin\phi \;\; cos\theta cos\phi \,\right]^T \tag{6}$$

The transformation between the two representations is important process in Mono-SLAM, especially when a point has a low covariance, then it is transformed to the Euclidean representation, which is easier computationally to update than the inverse depth representation.

The code in this work and also in [9] transforms between Inverse Depth and Euclidean according to a linearity criteria proposed in [22], which is tested with every cycle of the filter and for each feature as explained in the next subsection.

4.8 Transforming Between Inverse Depth Representation and Euclidean Representation

A method for using both of the representations in the same program is presented in [22]. It relies on a linearity factor L_d which represents how much the transform will be useful to the accuracy of the filter estimations.

The condition to transform a feature from Inverse Depth to Euclidean according to L_d is given by:

$$L_d = \frac{4\sigma_d}{d_1}|cos\alpha| < 10\% \tag{7}$$

Where d_1 is the distance between the point and the camera center, and it is given by:

$$d_1 = \| \mathbf{d} \|, \quad \boldsymbol{d} = \boldsymbol{y} - \boldsymbol{r_k} \tag{8}$$

α is the angle between the vector $\boldsymbol{\eta} = \boldsymbol{\eta}(\theta, \phi)$ and the vector $\mathbf{d}$.

$$\cos(\alpha) = \frac{\boldsymbol{d}^T \cdot \boldsymbol{\eta}}{\| \boldsymbol{d} \| \| \boldsymbol{\eta} \|} = \frac{\boldsymbol{d}^T \cdot \boldsymbol{\eta}}{d_1} \tag{9}$$

The value of σ_d can be found from the standard division of ρ, which is σ_ρ as follows:

$$\sigma_d = \left| \frac{\partial}{\partial \rho} \frac{1}{\rho} \right| \sigma_\rho = \frac{\sigma_\rho}{\rho^2}. \tag{10}$$

The previous L_d will be found for all the points in the state vector then a transformation is applied only to the points that fulfill the previous condition in Eq. 7.

5 ROS Environment Configuration and the General Workflow

The ROS version used here is Kinect on Ubuntu 15.10. Source code[3] from [9] was mainly utilized in this work, specifically for the Mono-SLAM part which uses OpenCV and Eigen C++ libraries.

The speed sensor readings were entered in the prediction step of Kalman Filter as a noise vector in the constant speed model.

The 3D-reconstruction phase (which runs offline) uses the python language with libraries such as Numpy and Matplotlib. In addition, it was sped up with Cython compiler.

At the end of the sparse 3D-reconstruction step, the point cloud can be viewed in Meshlab.[4] Later, this point cloud is refined along with the poses of all the images involved in the 3D-reconstruction using Sparse Bundle Adjustment (SBA). SBA is an optimization process, that was run as ROS package,[5] and its result is shown in Rviz.

[3] https://github.com/rrg-polito/mono-slam.

[4] https://www.meshlab.net.

[5] https://code.ros.org/svn/ros-pkg/stacks/vslam/trunk.

The experiments were done in simulation using Gazebo as framework, where a simulated Turtlebot was controlled using the keyboard.[6] The models in the simulated scene were downloaded from the web.[7] All the configuration of this environment can be recreated using the `.world` file included with the same repository for Mono-SLAM.

In the following subsections, a detailed overview of the contents and running of the packages is provided in the same order of the general work-flow of the experiments.

5.1 Mono-SLAM Program

The Mono-SLAM phase is a real-time process that starts by taking images from the camera as its only input to estimate the 3D positions of the camera and the world. To simplify this task we assume the movement of the camera to be at a constant speed.

The accompanying code implements Extended Kalman Filter on SLAM problems in C++. In the following, an explanation of the general structure of that code is given with a focus on the new improvements made in this work.

The real-time program initiates an `ImageConverter` class instance with an input that represents the name of a file containing the camera internal parameters and other constants. Some of them are shown in Table 2.

This file is a text file with `.cfg` extension. It is parsed by the Libconfig library that can be installed on Ubuntu by running the command:

```
$ sudo apt–get install libconfig
```

All of the filtering work is contained in one ROS node. This node will subscribe to the image topic `/camera/image_raw` and the odometry sensor readings topic `/pose` using that class methods. The Odometry sensor in the simulation measures the transitional and rotational speed of the robot in the coordinates of the robot base.

With the arrival of every new frame from the robot camera, a complete cycle of Kalman Filter is performed. Then, the new camera position and the new point cloud in the scene is rendered in the Rviz window along with the images. These images contain marks annotating the tracked features.

The prediction step takes velocity noise vectors as inputs. Since the vectors from the odometry sensor are relative to the robot base coordinates frame, they need to be transformed to the camera coordinates frame before using them in the prediction equations. The relative position of these two coordinate frames is shown in Fig. 2.

[6] https://github.com/ros-teleop/teleop_twist_keyboard.

[7] https://bitbucket.org/osrf/gazebo_models.

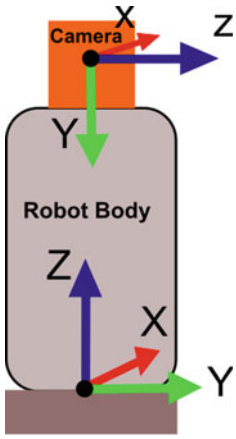


Fig. 2 The relative position between the camera and the robot base coordinate frames used in this work

5.1.1 Prediction

This step follows the standard Extended Kalman Filter computations. It predicts the new state and covariance of state according to the constant speed model.

The time difference between each subsequent camera frames time stamp is recalculated in each filter step. Therefore the two stages of the filter use a value `dT` indicating the difference between the time stamps of the previous frame and the current one.

The prediction function needs "correction vectors" as input to find the predicted result. This is represented by the noise vector $\boldsymbol{n} = \begin{bmatrix} \boldsymbol{\Delta V} \\ \boldsymbol{\Delta \Omega} \end{bmatrix}$ of Eq. 3. This vector can be easily computed from the measured odometry.

5.1.2 Speed Calculation

The previous noise vector $\boldsymbol{n}$ has a constant covariance matrix that represents how much this vector must be followed instead of using the constant velocity model when predicting the next state. The values of this covariance matrix are related to the odometry sensor drift value.

The filter cycle (prediction and update) is triggered by a reading of a new video frame. The images may be scaled down before filtering in order to speed up the calculations. However, once an image is selected, it will be saved in memory at full resolution, for the 3D reconstruction

After reading the velocity values from the simulated sensor, which experiences a simulated drift, two vectors need to be computed, $\Delta \boldsymbol{V}$ and $\Delta \boldsymbol{\Omega}$. Their values are considered as the changes in transition and in rotation of the robot respectively. They can be found using the following equations:

$$\mathbf{\Omega_{cam}} = \omega_z * (\boldsymbol{R}_{state} * \overrightarrow{Z}) \tag{11}$$

$$\Delta\mathbf{\Omega} = \mathbf{\Omega}_{cam} - \mathbf{\Omega}_{state} \tag{12}$$

The third element of the rotation vector represents the rotation around Z:

$$\theta_z = rotation_matrix2vector(\boldsymbol{R}_{state})[2] \tag{13}$$

$$\Delta\boldsymbol{V} = \begin{bmatrix} V_x * cos(\theta_z) \\ V_x * sin(\theta_z) \\ 0 \end{bmatrix} - \mathbf{V}_{state} \tag{14}$$

Where $\boldsymbol{R}_{state}$ is the current rotation matrix, $\mathbf{V}_{state}$ is the current transitional velocity vector, ω_z is the measured rotation around Z axis, V_x the measured velocity on X axis, $\mathbf{\Omega}_{state}$ is the current rotation velocity, $\overrightarrow{Z}$ is the unit vector on Z axis and the function $rotation_matrix2vector()$ is the transforming function between the matrix representation of the rotation (3×3 matrix) and the vector representation (3 angle elements).

After calculating the new predicted state, the positions of the new patches in the new frame need to be determined along with each patch ellipse of 2D covariance.

The patches covariance matrices are computed from the system covariance matrix, the latter is computed in the update function.

An additional process [9] is applied on patches to reduce the noise during the matching step. This process is based on filtering the patch to remove the moving camera effect (blur) which in turn gives better matching accuracy especially when the camera speed is high or when it is shaking.

5.1.3 Feature Matching in Mono-SLAM

The matching step is performed between the patch from the previous frame and all the possible locations of the same patch in the current frame. The matching is done using a template matching method which implements a correlation operation to find the most probable position of the patch.

After the matching, results may still contain outliers. Therefore, 1-point RANSAC is performed on these results.

From the filtered matching results, an innovation vector $\boldsymbol{e}$ and kalman matrix $\boldsymbol{K}$ are calculated according to the filter update equations. Then, an update for the state and covariance matrix is performed, leading to more confidence and less covariance in the state vector.

At the end of the update function, a check is made on the number of features to keep it in a determined range. This is done because a big number of features will slow down the computations, while a small number will make the estimated values less accurate. Therefore, features deletion is performed when the number of features raises above a threshold. Another case of deletion happens when any feature fails to match correctly on a specified number of subsequent frames.

The deletion happens in the state vector and in the corresponding elements of the covariance matrix.

A new modification in this work is done by changing the noise vector covariance values $\boldsymbol{P_n}$ between filtering steps. When a new image frame is read and no new odometery readings are available (due to sensor failure or delay), the last speed readings are taken but with bigger variance to indicate its bigger inaccuracy. This helps to get more relatively accurate estimations of the filter.

5.1.4 Features Detection and Appending for Mono-SLAM

A minimum number of features must be matched in each camera frame, if this is not the case, new features need to be detected, as in the work of [9].

This is simply done for the first frame. But for frames where past patches are still present, the detection process run only on the vacant areas of the image. Otherwise, the same feature may be detected twice. This detection in the accompanying code is done by the OpenCV function `goodfeaturestoTrack`, which accept a boolean mask as an optional input to extract features only from regions allowed by the mask, i.e. not occupied with previous patches. Then, these features are appended to the state vector and the covariance matrix is altered accordingly.

When a feature is detected for the first time, it is represented by Inverse depth. Later, when the feature's linearity increases, it can be transformed to Euclidean representation.

5.2 Selecting Images for 3D Reconstruction

After each predict and update cycle of the filter, performed for every frame, a selection process is run on a range of recent frames to select one frame satisfying certain conditions.

The proposed method here depends on calculating a value which represents the transitional and rotational movement of the robot between two selected locations. Another selection criterion is the sum of the diagonal elements in the covariance matrix of each frame which is considered as an indicator for the accuracy of the respective pose vector. Therefore, a minimum value for this sum is an optimal option when selecting the frames for 3D reconstruction.

In Algorithm 3, as shown next, a method for selecting images is presented. That method tries to achieve a balance between geometrical distribution and minimum covariance of the selected group of poses.

An important point here is that the value of the rotation weight (λ_2) is much bigger than the transition weight (λ_1). This is done because the reconstructed point from two images with a big rotational movement difference has less covariance and more accuracy than the other relative positions. As shown in Fig. 3, where the shaded area represents the covariance of the 3D position. The left part of the figure is the best case where the rotational change is the biggest. The transitional change only, as in

the right part, gives bigger covariance values. The middle part of the figure shows the case of small rotational change between the two images. This also has a big covariance.

Algorithm (3): Image Selection Method with Consideration of Movement and Covariance (symbols are explained in Tables 1 and 3)

Input: $C_{current}$, $q_{current}$, $\Sigma_{current}$, $\text{State}_{current}$, I
Output: $\text{State}_{\text{selected}}$, $\Sigma_{selected}$, $I_{selected}$

initialization:
$State_{selected} = [0\,,0\,,0\,,0\,,0\,,0\,,0\,]$; $old_pose = 0$
$C_{prev} = State_{selected}[0:3]$
$q_{prev} = State_{selected}[3:7]$
$T_2 = 1000,000$; $\lambda_3 = 0.00008$

for each image **I**:
  $a = C_{current} - C_{prev}$
  $b = q_{current}\,[:3] - q_{prev}[:3]$
  $dist = \lambda_1 a + \lambda_2 b$ (λ_1, λ_2 are constants)
  $passed = old_pose - dist$

  if $passed > \frac{MoveThresh}{2}$ and $passed < MoveThresh$:
    $P = Sum\,(diag\,(\Sigma_{current}))$
    if $T_2 > P$:
      $T_2 = P$
      $State_{temp} = State_{current}$
      $I_{temp} = I$
      $\Sigma_{temp} = \Sigma_{current}$
      $dist_{temp} = dist$

  if $MoveThresh \leq passed$:
    if $Sum\,(diag\,(\Sigma_{current})) - T_2 > \lambda_3$:
      $\text{State}_{selected} = \text{State}_{temp}$
      $\Sigma_{selected} = \Sigma_{temp}$
      $I_{selected} = I_{temp}$
      $old_pose = dist_{temp}$
      $T_2 = 1000,000$
    else:
      $\text{State}_{selected} = \text{State}_{current}$
      $\Sigma_{selected} = \Sigma_{current}$
      $I_{selected} = I_{current}$
      $old_pose = dist$
      $T_2 = 1000,000$
end for loop

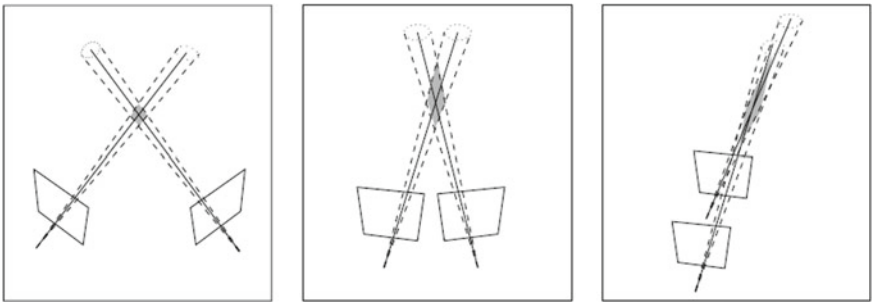


Fig. 3 Covariance of the triangulated point from two views shown as a shaded area for three different relative positions. The rotational change has the best effect in decreasing the covariance

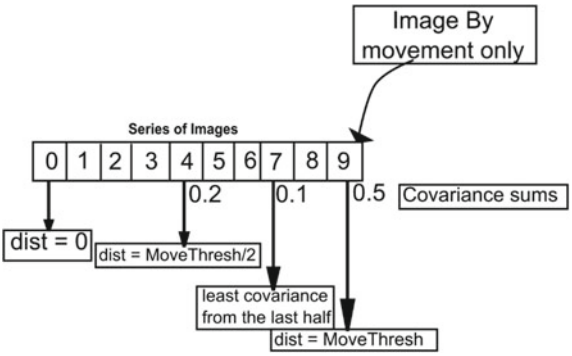


Fig. 4 Image Selection Method with Consideration of Covariance and Movement. The first 5 images didn't have a space change from the first image in the sequence (image 0) above a threshold ($MoveThresh/2$). The last 5 images are tested using the covariance matrix trace value, and the one with the least value is selected (image 7)

Algorithm 3 implements the process of selecting images through two stages. As Fig. 4 shows, that when a certain parameter related to movement (*dist*) is less than half the threshold (*MoveThresh*) no image is taken. Later, when this parameter passes half of the threshold and is still less than the whole threshold, the image with the least covariance trace value is taken.

The inputs to Algorithm 3 are the current state vector and its covariance matrix. After each cycle of the filter, the current image is tested using the trace of the current covariance matrix, as well as, the amount of transition and rotation made starting from the first image in the sequence. When a movement threshold is exceeded, an image is selected and saved with its covariance and pose. These saved values are the outputs of the algorithm.

Constants in this method have experimental values and they depend on some factors such as the velocity of the robot and the required number of images.

The travelled distance (parameter `passed` in Algorithm 3) calculation is done as shown in the code snippet below taken from the code accompanying this work:

```
float poses_diff(VectorXf state_old , VectorXf state_new,
    Vector3f last_rot){

  float a = (state_old.segment<3>(0)-state_new.segment<3>(0)).
```

```
    norm()*3.33
  Vector3f b = (last_rot - quat2vec(state_new.segment<4>(3)))
    *57.295779;
  //std::cout << " a, and b for poses_diff" << std:: endl;
  //std::cout << a << std::endl;
  //std::cout << b << std::endl;
  return a+abs(b(0))+abs(b(1))+abs(b(2));
}
```

While the next code snippet implements the conditions for selecting images and saving their corresponding poses and covariance matrices (as in Algorithm 3 and Fig. 4)

```
float DistWalked = poses_diff(last_image_pose,stat14.segment<7>(0)
    ,last_vrot);
  if ( DistWalked>(MoveThresh/2)   &&   DistWalked<MoveThresh )
    {
    double some_var =  slam.Covariance_Parameter();
    if(some_var <min_cov_for_pose){
      min_cov_for_pose = some_var;
      Pose_id = frameId;
      //replace traj1
      min_stat = stat14.segment<7>(0);
      min_camscov = slam.getSigma();
      min_projs = slam.Point4sba;
      //will add here the image assignment
      Selected_Pose = cv_ptr->image.clone();
    }
  } else if ( DistWalked>=MoveThresh) {
      if(( slam.Covariance_Parameter()- min_cov_for_pose )< 0.000085){
        node_proj << "P" << frameId << endl;
        node_proj << stat14.segment<7>(0)<< endl;
        node_proj << "0  0  0" << endl;
        // will add here the image write
        min_camscov2 = slam.getSigma();
        cov_cams2 << min_camscov2.block<7,7>(0,0) << endl;
        cov_cams << min_camscov2.block<7,7>(0,0) << endl;
        string i_p = ToString(frameId);
        cv::imwrite(i_p +".png",cv_ptr->image);
      }else{
        node_proj << "P" << Pose_id << endl;
        node_proj << min_stat << endl;
        node_proj << min_projs << endl;

        cov_cams << min_camscov.block<7,7>(0,0) << endl;
        // will add here the image write
        string i_p = ToString(Pose_id);
        cv::imwrite(i_p +".png",Selected_Pose);
      last_vrot = quat2vec(stat14.segment<4>(3));
      last_image_pose = stat14.segment<7>(0);
    min_cov_for_pose = 10000000;
  }
```

Table 1 The meanings of the variables in Algorithm 3

Parameter	Definition
I	The image
$q_{current}$	The Quaternion representing the current rotation of the camera
a	A number representing the transitional change
b	A number representing the rotational change
$C_{current}$	The current position of the camera center
$\Sigma_{current}$	The current covariance matrix for the camera pose
$State_{current}$	The current state vector
$I_{selected}$	The selected image
$\text{State}_{\text{selected}}$	The state vector of the selected image
$\Sigma_{selected}$	The pose covariance of the selected image
old_pose	The last chosen pose sum of the quantity of rotation and transition (a, b)

5.3 Running Simulation Environment

The method here is tested using simulation in Gazebo for Turtlebot simulated model. The simulation is done using a virtual environment consisted of different 3D objects as shown in the video[8] accompanying this work.

It is worth noting that the turtlebot model used here is modified. First the camera is turned around its Y axis, as in Fig. 2. Second, the resolution of the RGB frames coming from the Kinetic camera is increased 16 times, while the depth information is discarded.

These changes can be done by editing the corresponding tags in the `.xacro` files for the kinect camera and the turtlebot properties in `turtlebot_description` ROS package. Usually those files are found in:

`$ros_root_directory/kinect/share/turtlebot_description`.

The environment file which is provided with the code accompanying this work can be used to simulate the world and the robot using the command:

```
$roslaunch turtlebot_gazebo turtlebot_world.launch
           world_file:="/some_world9.world"
```

The previous command requires Gazebo simulation framework to be installed. For controlling the robot in the simulation, another ROS package[9] is used which allows the usage of a keyboard or a joystick to guide the robot.

To achieve the manual control using the previous package, the following command must be executed in a new terminal:

```
$roslaunch turtlebot_teleop keyboard_teleop.launch
```

[8] https://youtu.be/gxz2JRf8LvQ.

[9] http://wiki.ros.org/turtlebot_teleop.

Lastly, the filter nodes for filtering and visualizing can be run by opening a third terminal and running the following commands:

```
$ source catkin_ws/devel/setup.bash
$ roslaunch mono_slam_sim.launch
```

This launch file runs the Rviz interface and the filter node with an appropriate remapping of topics.

When all of the three commands are run, then the robot can be moved in a circular path around a central object in the simulation environment.

However, it is possible to move the robot in a straight line or any random path, but this will give poor experiment data for 3D reconstruction of objects, as shown in Fig. 3. The result of running the previous commands are shown in the videos accompanying this work.[10]

In the next step, these selected images will be used to extract a point cloud of the central object.

5.4 3D Reconstruction Program

The code used for this step in this work[11] is in Python language. It is accelerated with Cython,[12] outside of ROS framework. The goal of the program is to take a group of images along with a text file containing their internal and external parameters, and to return the 3D point cloud of the object shown in these images. This method may be inefficient if the images overlapping areas are small or if the images resolutions are low.

The 3D reconstruction is done using OpenCV function `cv2.triangulate` which applies a minimization solution to compute the 3D position from the 2D matches. The computations performed by this function use well-known formulas in the Multi-view Geometry field [15].

The program has two modes of work, sparse and dense. For both modes, the ORB [23] features detector and descriptor is used, which is a fast feature detector compared to SIFT [24] detector. However, this speed comes at a cost of less accuracy for ORB. Although the program is run offline, but it still needs to be fast enough to process many high-resolution images.

5.4.1 Sparse 3D Reconstruction

The sparse mode is used here just to produce a starting point cloud so it can be optimized jointly with the poses. Matching is done between each two consecutive images, using FLANN library function in OpenCV `cv2.FlannBasedMatcher`.

[10] https://youtu.be/XabY1OyT6_4.

[11] https://github.com/engyasin/3D-reconstruction_with_known_poses.

[12] https://cython.org.

After the matching step, a group of tracks of features matched through many images is built. Each track will give only one 3D point. To further filter the matching result from errors, a re-projection error is calculated and its value is tested against a threshold for every matched pair.

When every track is taken to find its 3D point, it is tested further for errors by finding the differences between the resulting points from each pair of features.

For example, if a track is made of 3 matched features across 3 images, then from each unique pair of these features, a 3D point can be found. The first and the second features will be triangulated first, then the second and the third, and finally the first and the third. This will produce three different points in 3D space. These points should coincide in the optimal case. However, the big differences among them indicate big noise in the tracks values. Using this as a test parameter, noisy measurements in the tracks can be discarded. Finally, a point cloud is built for the object or the objects apparent in the scene.

5.4.2 Dense 3D-Reconstruction

For the dense mode, a gird that samples the pixels using a constant sampling step is applied to extract features. Although this is less accurate than most of corners detection methods, it gives more features for a denser point cloud even when starting from a small number of images.

As an additional filtering step for the detected features in a dense detection, only a group of descriptors with the highest variance are taken. This starts by extracting ORB descriptors from every feature then calculating their variances. From these values, only the features with the highest variance are taken as the strongest features. This, however, is not the only way to measure the strength of a feature, but it is fast and accurate relatively. This filtering step assures getting a better corners for matching and for reconstruction as shown in Fig. 5.

In the dense mode, this grid of features is taken only from the object of interest and not all over the entire image. The latter case would produce a huge number of features for high-resolution images, making the process very slow and inefficient. Therefore, in order to further limit the number of features, only a rectangular window defined by the user is sampled with this grid.

The user draws this window for the first image, then a template matching method using OpenCV `matchingTemplate` function is used to determine the window position in all of the following images.

The rectangle should be centered on the object of interest in the image, as shown in the video.[13] The user maybe requested to draw a new rectangle if the template matching methods fails in any of the images.

The rest of the matching and triangulation steps in the dense mode will run as in the sparse mode without any major differences.

[13] https://youtu.be/BRDCWAJDuM0.

Fig. 5 Matching between the features of two images produced by the corners of a grid on the object of interest (the fountain) and filtered by the strength expressed as the variance of the respective descriptor vector. Much bigger number of matches is found than the standard detectors case

5.5 Sparse Bundle Adjustment Program

The result from the last step is a system of a 3D point cloud and its camera poses. This system of parameters has noise coming from both the images and the errors in calibration, beside other factors. To further enhance the accuracy of the whole system containing the poses and the 3D points, a Bundle Adjustment process is performed as in [19].

A Sparse Bundle Adjustment (SBA) is an improvement on Bundle Adjustment targeted to speed up the performance of Bundle Adjustment [16] by exploiting the sparse nature of the Jacobian matrices used in the computations.

A ROS package that implements SBA is adopted with some modifications. The point cloud is passed along with the camera poses to a `SysSBA` class instance as input arguments.

This accomplished by running the following command from a new terminal:

```
$ roslaunch sba_1.launch
```

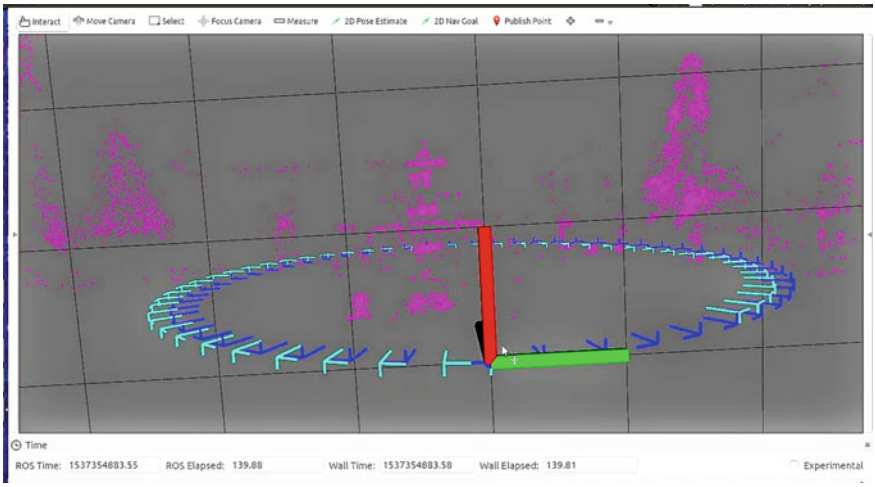


Fig. 6 The point cloud after performing SBA (in pink color) with the coordinates from all the poses around it (in bright blue color)

This also open an Rviz window as shown in Fig. 6 and in the video accompanying this work.[14] The images are added group by group increasingly and the SBA function is called for multiple iterations after adding each group.

A Root Mean Square Error (RMSE) is calculated after each run of SBA on the system, then the results are published as topics and written to a text file. The results are the optimized poses and the 3D point cloud with minimum total RMSE error.

5.6 Calculating the Projection Covariance

It is possible to use the projection covariance matrices in the cost equation of SBA, as explained in [16]. This is done in order to enhance the performance of the optimization process. However, since only the poses covariance are available, a formula, such as Covariance Propagation [25], is needed to extract the projection covariance, as shown in Fig. 7.

One factor affecting the projection covariance is the covariance of the triangulation of the 3D points. However, In the experiments in this work, this factor is ignored.

What also limits this method of calculation beside using only the covariance of the camera pose, is the approximation in the Covariance Propagation formula itself.

The theory behind Covariance Propagation states that for a given function $\boldsymbol{Y} = \boldsymbol{f(X)}$ where $\mathbf{X}$ is the input and $\mathbf{Y}$ is the output, if the covariance of $\mathbf{X}$ is known as $\boldsymbol{X}_{cov}$ and $\boldsymbol{J}_f$ is the derivative (jacobian) of $\boldsymbol{f()}$ relative to the input $\mathbf{X}$, then the covariance

[14] https://youtu.be/ucTHoqFaFig.

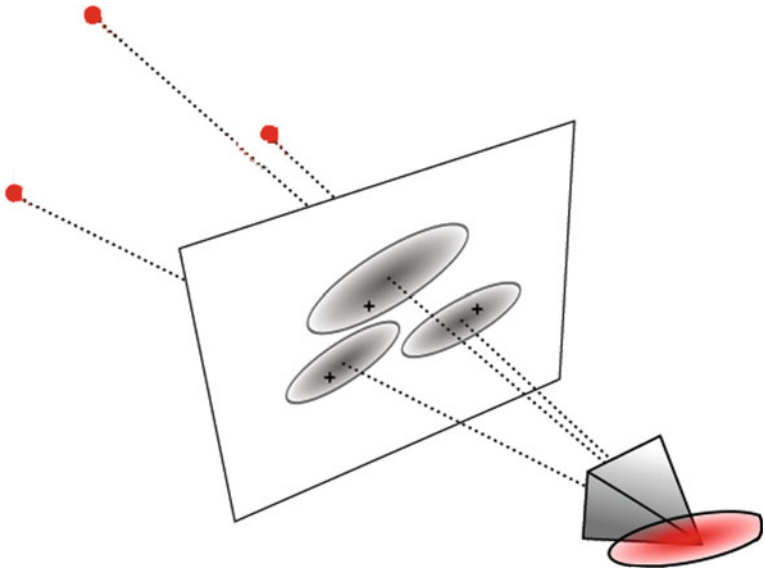


Fig. 7 Covariance Propagation from the camera center to the 2D points projections on the image plane (the 2D ellipses). The 3D points covariance matrices (The triangulation covariance) are ignored

of **Y** is given approximately as in [25] by:

$$\boldsymbol{Y_{cov}} = \boldsymbol{J_f} * \boldsymbol{X_{cov}} * \boldsymbol{J_f^T} \tag{15}$$

Here $\boldsymbol{J_f^T}$ is the transpose of the jacobian matrix.

If **X** is the camera pose, and ***f()*** is the projection function of a fixed 3D point, then the previous formula can be used to find the projection covariance approximately.

After finding each 2D projection covariance, it can be used with the SBA system in the optimization process.

However, because the point cloud is sparse, it can be discarded with relatively small loss of information, and the next dense 3D reconstruction uses only the new optimized poses.

This dense 3D reconstruction will run using the same method explained earlier in the Sect. 5.4.2.

Finally, the resulting point cloud can be viewed on a program such as Meshlab. This final result represents the outcome of the whole process.

6 Computational Performance

All the presented experiments are implemented using Lenovo ideapad 110 laptop with Intel core i5 and 4 Gigabytes of RAM. Many of the program parameters are fixed in the experiments, as shown in the next section.

For the Mono-SLAM part, the frame rate of the camera video feed is 7 frames per second. This low frame rate is a result of rendering high-resolution images from the simulation, which takes more time than the lower resolution case. The operating of this step (MonoSLAM) is shown in the accompanying videos.[15]

7 Results and Discussion

This section presents the numerical results of many experiments performed for our method. The results show the performance of the method which helps with the concluding to the final findings.

The following experiments are performed for a comparison purpose between two groups of selected images for the same simulation and filtering conditions. Each group elements are selected from the same video feed of the robot camera, but with different factors governing the selection process.

The first group of images is selected according to the method proposed here, as shown in Fig. 4. The second group of images is selected according to the difference in movement only, ignoring the covariance information. For example, in the image sequence shown in Fig. 4, the first group will contain image number 7 while the second group will contain image number 9. It's worth noting that the number of images is the same for both groups of the same experiment.

None of the previous methods, as discussed earlier, uses the covariance explicitly for selecting frames from video for 3D reconstruction, therefore no comparison with previous method is done here. The comparison is between the two images groups and it shows the importance of considering the poses covariance when selecting images. The best group will be the one that gives more points in the point cloud, for the same parameters and thresholds, than the other group. This is done using the sparse 3D-reconstruction program which calculates such a point cloud.

For the Mono-SLAM part of the experiments, the constants in Table 2 were used. The Image Selection program uses the parameters of Table 3 (shown in Algorithm 3).

The comparison is made between images selected according to movement only and images selected according to movement and covariance. The purpose of this is to show the importance of the covariance factor.

The comparison conclusion depends on the number of points in the point cloud built by each group of images as the comparison factor. This number represents the

[15] https://youtu.be/XabY1OyT6_4.

Table 2 The diagonal elements of the noise covariance matrix and other values for feature extraction and matching (used in MonoSLAM)

Definition	Parameter name in code	Value in the tests
Standard division of transitional velocity noise on X	`sigma_vx`	2×10^{-7} m/sec
Standard division of transitional velocity noise on Y	`sigma_vy`	2×10^{-7} m/sec
Standard division of transitional velocity noise on Z	`sigma_vz`	4×10^{-11} m/sec
Standard division of rotational velocity noise on X	`sigma_wx`	2×10^{-9} rad/sec
Standard division of rotational velocity noise on Y	`sigma_wy`	2×10^{-7} rad/sec
Standard division of rotational velocity noise on Z	`sigma_wz`	2×10^{-11} rad/sec
Patch size	`window_size`	30 pixels
Image resizing factor	`Scale`	10
Minimum number of features in image	`min_features`	20
Maximum number of features in image	`max_features`	35
Scale factor of the area of possible matches	`sigma_size`	4

Table 3 Constants in the Image Selection program

Parameter definition	Value
Transition factor λ_1	3.33
Rotation factor λ_2	57.29
Movement threshold (MoveThresh)	26
Initialization value for the minimum covariance T_2	100000
Covariance difference threshold λ_3	$10^{-5} \times 8.5$

accuracy of the poses, and the quantity of the matches in each group of images with a level of noise below a certain threshold.

According to this standard, the covariance-guided group should give more points for the same parameters than the other group. This is true, because less covariance means more accurate poses, and eventually more accepted points in the point cloud, but for the second group, the number of points will be smaller (using the same thresholds and parameters).

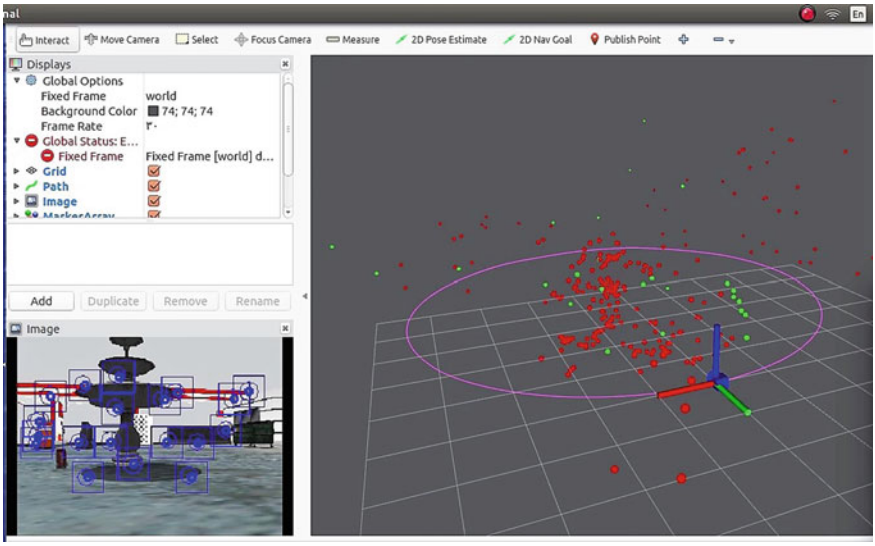


Fig. 8 Rviz window as it appears at the end of Mono-SLAM program, where each square in the grid in the right side is 1×1 m. The bottom left window is showing the video feed from the camera. The points in the right side are the estimated environment positions from the filter

Table 4 3D-reconstruction results for both sets of images (the higher number of points is given in bold)

Exp. number	Number of images for the first group	Number of points for the first group (with covariance)	Number of images for the second group	Number of points for the second group
1	31	**1613 points**	31	1478 points
2	43	**3589 points**	43	3011 points
3	43	**3431 points**	45	2951 points
4	52	4754 points	51	**4783 points**
5	50	**5237 points**	51	5169 points
6	50	**6913 points**	51	6558 points

The experiments were performed by making the simulated robot complete a closed path around a central object. The path length is about 26 m in the simulation program, Gazebo [26], as shown in Fig. 8.

The camera is rotated in a manner making it always facing the object of interest (the fountain in Fig. 8).

All of the experiments were performed in the same virtual scene. Table 4 shows some of the results, where the exact same set of parameters were used across all of the experiments, except for experiment 5. In that experiment, the thresholds of errors in the 3D reconstruction program were higher than the rest.

Table 5 3D reconstruction results for both image groups with the additional condition for covariance change (the higher number of points given in bold)

Exp. number	Number of images for the first group	Number of points for the first group (with covariance)	Number of images for the second group	Number of points for the second group
1	42	**3387 points**	42	3120 points
2	43	**5023 points**	43	3399 points
3	43	**3081 points**	43	2865 points

In the first group, where the covariance is taken into account, the result contains more points. However, in experiment 4, this is not the case because of the fact that the covariance difference between the two groups of images poses is too small to increase the quantity of the results (number of points).

To confirm this explanation, a final condition is added to the image selection algorithm. This condition puts a minimum threshold (λ_3 in Algorithm 3) for the accepted covariance difference between each image selected according to covariance, and the respectful one selected according to movement.

In the experiments, λ_3 was set to $10^{-5} \times 8.5$. This is an experimental value, and it is related to the noise covariance matrix.

Table 5 shows three more experiments after applying the last condition. The first one was done in the same conditions as experiment 4 in Table 4, where the robot path was less circular-like.

In all of the experiments in Table 5, we notice an increase in the number of points for the first group. This improvement in the results is due to avoiding to select images with poor pose estimates. An example of such estimates are the blurred images resulted from the sudden movement of the camera.

A few hundreds of points were added to the result, using about 40–50 images when considering covariance of each selected image. These additional points are covering the same objects surfaces, which means a denser point cloud as a result.

Therefore, considering covariance as a factor when selecting images increases the density and accuracy of the generated point cloud.

In other experiments—for the SBA program—a comparison is performed between the operations with and without 2D projection covariance for the same data. The result of this is the same in both cases, both in terms of the RMSE error and the number of poor points. This is explained by the fact that the covariance matrices may not be accurate enough, mainly, because of two reasons, the approximated formula used to compute these matrices and the discarding of the triangulation covariance.

As explained in [27], the covariance itself has estimation errors, and the improvement from using it may be lost if its values were too noisy. This is probably the case of the results of the SBA experiments performed here.

The final point cloud in one of these experiments is shown in the videos[16] accompanying this work. Additional processing steps may be applied to the point cloud, such as building the continuous surface and adding colors and textures according to the real object surface.

8 Conclusion and Future Work

The main result of this work states that using the covariance of the poses as a factor when selecting images for 3D reconstruction leads to an increase in the number of 3D points in the point cloud. Another result is that the covariance of the camera pose alone is not sufficient to compute the accurate 2D covariance of the features positions.

This method can be useful when applying kalman filter to a robot with a camera and then trying to build a 3D model of the objects in the scene. If the video from the robot is sampled for frames regularly in time or in space, many repeated and blurry frames maybe selected. This may affects the quality and quantity of the resulting point cloud.

As a future work, a more accurate balance can be made between this new factor and the geometrical factor, by using one formula for both, instead of applying an additional condition on the covariance difference.

In the SBA step, a more accurate projection covariance may be found by considering the triangulation covariance as well as the covariance of the poses, then the same tests may be repeated to prove the assumed improvement.

References

1. S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S.M. Seitz, R. Szeliski, Building Rome in a day. Commun. ACM **54**(10), 105–112 (2011)
2. B. Williams, M. Cummins, J. Neira, P. Newman, I. Reid, J. Tardós, A comparison of loop closing techniques in monocular SLAM. Robot. Auton. Syst. **57**(12), 1188–1197 (2009)
3. O.G. Grasa, J. Civera, A. Guemes, V. Munoz, J.M.M. Montiel, EKF monocular SLAM 3D modeling, measuring and augmented reality from endoscope image sequences, in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 2 (2009)
4. N. Mahmoud, I. Cirauqui, A. Hostettler, C. Doignon, L. Soler, J. Marescaux, J.M.M. Montiel, ORBSLAM-based endoscope tracking and 3d reconstruction, in *International Workshop on Computer-Assisted and Robotic Endoscopy* (Springer, 2016), pp. 72–83
5. G. Pavoni, M. Dellepiane, M. Callieri, R. Scopigno, Automatic selection of video frames for path regularization and 3D reconstruction, in *Proceedings of the 14th Eurographics Workshop on Graphics and Cultural Heritage, GCH'16*, Goslar Germany, Germany (Eurographics Association, 2016), pp. 1–10

[16] https://youtu.be/jrnJ9LEAygE.

6. A. Rachmielowski, N. Birkbeck, M. Jägersand, D. Cobzas, Realtime visualization of monocular data for 3d reconstruction, in *2008 Canadian Conference on Computer and Robot Vision* (IEEE, 2008), pp. 196–202
7. A.J. Davison, I.D. Reid, N.D. Molton, O. Stasse, MonoSLAM: real-time single camera SLAM. IEEE Trans. Pattern Anal. Mach. Intell. **29**(6), 1052–1067 (2007)
8. J. Civera, A.J. Davison, J.M.M. Montiel, Inverse depth parametrization for monocular SLAM. IEEE Trans. Robot. **24**(5), 932–945 (2008)
9. L. Russo, S. Rosa, B. Bona, M. Matteucci, A ros implementation of the mono-slam algorithm. Int. J. Comput. Sci. Inf. Technol. **6**(1), 339–351 (2014)
10. M. Li, A.I. Mourikis, High-precision, consistent EKF-based visual-inertial odometry. Int. J. Robot. Res. **32**(6), 690–711 (2013)
11. B. Williams, I. Reid, On combining visual SLAM and visual odometry, in *2010 IEEE International Conference on Robotics and Automation* (IEEE, 2010), pp. 3494–3500
12. H. Strasdat, J.M.M. Montiel, A.J. Davison, Visual SLAM: why filter? Image Vis. Comput. **30**(2), 65–77 (2012)
13. J. Engel, T. Schöps, D. Cremers, LSD-SLAM: large-scale direct monocular SLAM, in *European Conference on Computer Vision* (Springer, 2014), pp. 834–849
14. J. Engel, V. Koltun, D. Cremers, Direct sparse odometry. IEEE Trans. Pattern Anal. Mach. Intell. **40**(3), 611–625 (2018)
15. R. Hartley, A. Zisserman, *Multiple View Geometry in Computer Vision* (Cambridge University Press, Cambridge, 2003)
16. M.I.A. Lourakis, A.A. Argyros, SBA: a software package for generic sparse bundle adjustment. ACM Trans. Math. Softw. (TOMS) **36**(1), 2 (2009)
17. Rudolph Emil Kalman, A new approach to linear filtering and prediction problems. J. Basic Eng. **82**(1), 35–45 (1960)
18. J. Civera, A.J. Davison, J.M.M. Montiel, *Structure from Motion Using the Extended Kalman Filter*, vol. 75 (Springer Science & Business Media, Berlin, 2011)
19. B. Triggs, P.F. McLauchlan, R.I. Hartley, A.W. Fitzgibbon, Bundle adjustment–a modern synthesis, in *International Workshop on Vision Algorithms* (Springer, 1999), pp. 298–372
20. R.C. Bolles, M.A. Fischler, A RANSAC-based approach to model fitting and its application to finding cylinders in range data, in *IJCAI*, vol. 1981 (1981), pp. 637–643
21. J. Civera, O.G. Grasa, A.J. Davison, J.M.M. Montiel, 1-Point RANSAC for extended Kalman filtering: application to real-time structure from motion and visual odometry. J. Field Robot. **27**(5), 609–631 (2010)
22. J. Civera, A.J. Davison, J.M.M. Montiel, Inverse depth to depth conversion for monocular SLAM, in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, April 2007, pp. 2778–2783
23. E. Rublee, V. Rabaud, K. Konolige, G.R. Bradski, ORB: an efficient alternative to SIFT or SURF, in *ICCV*, vol. 11 (Citeseer, 2011), p. 2
24. D.G. Lowe et al., Object recognition from local scale-invariant features, in *ICCV*, vol. 99 (1999), pp. 1150–1157
25. B. Ochoa, S. Belongie, Covariance propagation for guided matching, in *Proceedings of the Workshop on Statistical Methods in Multi-Image and Video Processing (SMVP)*, vol. 83 (2006)
26. N. Koenig, 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
27. M.J. Brooks, W. Chojnacki, D. Gawley, A. Van Den Hengel, What value covariance information in estimating vision parameters? in *Proceedings 8th IEEE International Conference on Computer Vision. ICCV 2001*, vol. 1 (IEEE, 2001), pp. 302–308

Yasin Maan Yousif M.Sc. in Mechatronics Engineering, and a lecturer at the Department of Mechatronic Engineering, Al-Baath University. His research interests include Robotics, Machine Vision and Deep learning.

Dr. Iyad Hatem Associate Professor at the Department of Mechatronic Engineering, Tishreen University. His research interests include Image analysis, Computer vision and Pattern classification.

ROS Rescue: Fault Tolerance System for Robot Operating System

Pushyami Kaveti and Hanumant Singh

Abstract In this chapter we discuss the problem of master failure in ROS 1.0 and its impact on robotic deployments in the real world. We address this issue in this tutorial chapter where we outline, design and demonstrate a fault tolerant mechanism associated with a ROS master failure. Unlike previous solutions which use primary backup replication and external checkpointing libraries which are resource demanding, our mechanism adds a lightweight functionality to the ROS master to enable it to recover from failure. We present a modified version of the ROS master which is equipped with a logging mechanism to record the meta information and network state of ROS nodes as well as a recovery mechanism to go back to the previous state without having to abort or restart all the nodes. We also implement an additional master monitor node responsible for failure detection on the master by polling it for its availability. Our code is implemented in Python and preliminary tests were conducted successfully on a variety of land, aerial and underwater robots and a teleoperated computer running ROS Kinetic on Ubuntu 16.04. The code is publicly available under a Creative Commons license on Github at https://github.com/PushyamiKaveti/fault-tolerant-ros-master.

Keywords ROS1.0 · Fault-tolerance · Failure detection · Master recovery

1 Introduction

The Robot Operating System (ROS) [8] is the most widely used framework [3] that provides libraries and tools to create robotic applications. It allows us to build and execute distributed applications across multiple machines within the context of a publish/subscribe master-slave architecture. ROS supports both synchronous

P. Kaveti (✉) · H. Singh
Northeastern University, 360 Huntington Avenue, Boston, MA 02130, USA
e-mail: kaveti.p@husky.neu.edu

H. Singh
e-mail: ha.singh@northeastern.edu
URL: https://www.northeastern.edu/robotics

© Springer Nature Switzerland AG 2021

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_12

communication via services and asynchronous communication via publish/subscribe. It maintains a peer to peer runtime graph of processes communicating via XMLRPC.

The ROS master has a unique place in the ROS architecture. It acts as a central authority for the nodes to setup communication links with each other. This uniqueness is a major issue as the system is prone to a single point of failure when the ROS master crashes. Even though a ROS master crashing may be a very low probability event, a requirement for resiliency to faults is critical for implementing real world robotics applications.

We have experienced ROS master crashes in the environments that we focus on—long deployments on missions in challenging environments with extremes of temperature and adverse environmental conditions as are found in underwater robotic applications and in ice/snow covered areas. Battery limitations in case of high power consumption on Unmanned Aerial Systems (UAS or drones), software failures associated with memory or processing overload can also be contributing factors. The impact of failure may be dramatic and catastrophic leading to the loss of the robot, property damage and possible human injury or death when dealing with mission critical systems on large robotic systems such as unmanned aerial systems and autonomous cars. Thus, it is extremely important to detect and recover from such failures, and to do so quickly and without the need to abort or reboot the mission entirely.

1.1 Related Work

There have been previous attempts towards fault tolerance in ROS [6, 7]. Both these cases propose a transparent approach where there is a separation between functional code and the fault tolerance mechanism.

Lauer et al. [7], propose an adaptive fault-tolerance technique related to a component based approach. Their design to fault-tolerance uses primary-backup replication and the introduction of, *before-proceed-after*, interceptor nodes between the clients (ROS nodes) and the master. A back-up replica of the ROS master is maintained to tolerate faults associated with crashes. A request to the master is first handled by a proxy server which is then passed on to the *before-proceed-after* nodes. Once it passes through the *after* node, the backup replica is updated to match the primary. There is a crash detector node on the master which periodically reports master's status to the crash detector on the slave. There is also a recovery node which is notified in the event of a primary crash to stop and remove all nodes from master and bind them to the slave after which all the client requests are forwarded to the slave.

They provide a nice component-based extensible design and show how multiple fault-tolerant mechanisms can be introduced into the computation graph. A strong feature of this approach is high availability of the ROS master, but it comes with the cost of adding an extra layer of communications between the nodes and the master.

Although this approach shows promise, the results for a ROS implementation, as reported by the authors themselves, were not satisfactory. Unfortunately, neither the

source code nor the results of any experiments that were conducted with the proposed FTM in ROS are available for us to do a comparative study against our method.

Another approach by Jain et al. [6] uses DMTCP checkpointing [4] software to deal with the single point of failure of ROS master. DMTCP checkpointing is a multi threaded program which works by taking a snapshot of the application state which can be used later to restart the application from checkpoint. It is mainly designed for long running applications on a large cluster, where if the program crashes at the last minute, it needs to be started from the beginning.

In the DMTCP checkpointing [4] method an application snapshot which consists of the user space, the libraries, the processor state etc., are all saved. This is useful in case of long running applications where the entire application state needs to be recovered to continue processing. In ROS, the master acts as a service registry, where recovering just the meta-data is sufficient.

Also, DMTCP works by spawning a DMTCP coordinator thread and one checkpointing thread per node which adds additional load on the processor. It saves one checkpointing image file per node. The checkpointing files are comparatively large in size, about 16 MB for a simple string publisher node. This is due to the fact that the checkpoint contains the snapshot of the application state including the libraries, memory contents etc. This limits the scalability of this method. In addition, since, DMTCP is a separate application from ROS it needs to checkpoint periodically, each time suspending the nodes, saving the checkpoint and only then resuming the operation. This would entail a severe limitation for usage in robotics applications in highly dynamic environments.

In the adaptive fault-tolerance solution, a replica master is maintained which replaces the primary in case of a failure. While this provides high availability of the ROS master, it adds the overhead of dynamic binding (unregistering/re-registering) all the nodes to the replica and vice versa for the case when the master is recovered. It also requires a suspension of all the nodes when changing the bindings to the new master.

For DMTCP, we note that this case is quite different from that of ROS, where the master acts as a service registry. This implies that ROS can be restarted at any point without running the entire program from the beginning. Saving its meta-data is sufficient for recovery and for re-establishing connections between nodes. No experimental evaluation of using DMTCP with ROS is provided in the paper to understand its applicability (Fig. 1).

In contrast to these methods, our work presents a simple and light-weight fault-tolerance mechanism as an added functionality within the ROS framework. We solve the problem of fault tolerance by maintaining the state of ROS master and its nodes via a logging mechanism. This not only helps us to store the meta information of the system but also helps us to recover the system state after a failure. It is simple as it is just a feature addition to ROS. It is light-weight in so far as we are only recording the meta-data and when there is a change to the configuration as opposed to periodic checkpointing. We believe our "ROS Rescue" is a very useful feature to have in ROS for carrying out real-world robotics missions/applications safely and without interruptions. A strong fundamental requirement for our work was to enable ROS

Fig. 1 ROS is the dominant software framework [8] used in variety of robotic platforms including aerial systems, autonomous driving and underwater & surface vehicles. The requirement for a ROS master can be a critical point of failure, albeit with low probability. We propose a very lightweight method that addresses this issue

master recovery in a fraction of a second, a number determined by the tolerance of our robotic systems in real world applications. The main contributions of this chapter include software for the ROS master with fault tolerance, and a detailed step by step tutorial on how to obtain, build and use ROS Rescue into standard applications while meeting our fundamental requirement.

In the upcoming sections of the chapter we discuss the following topics as they relate to our problem and its implementation

- Section 2 covers an overview of the structure and various components of ROS.
- Section 3 describes the impact of master failure and the need for fault tolerance in ROS.
- Section 4 covers fault tolerance using our package which we call ROS Rescue.
- Section 5 covers the implementation details of our fault tolerance technique and a step by step guide to install and use ROS rescue from an application standpoint.

2 ROS Overview and Architecture

An overview of the ROS framework is shown in the Fig. 2. The main technical components include a ROS master, a parameter server and ROS nodes. Each of these components implement an XMLRPC server for communication.

As a peer to peer topology ROS requires some sort of look up service like a DNS server so that processes can find each other. The ROS master facilitates this and acts as the registration and lookup service which maintains a registration API for nodes to register with it as publishers, subscribers or service providers. The ROS master has a URI stored in the environment variable ROS_MASTER_URI on which the XMLRPC server is running.

Every node is essentially an XMLRPC server and has a URI. This server is used to receive call backs from the master and negotiate connections with other nodes via a slave API. Nodes communicate with each other asynchronously using topics via a publish/subscribe mechanism, and synchronously via service calls.

A subscriber node connects with the master on startup. The master saves the information about the subscriptions and sends the publisher information which con-

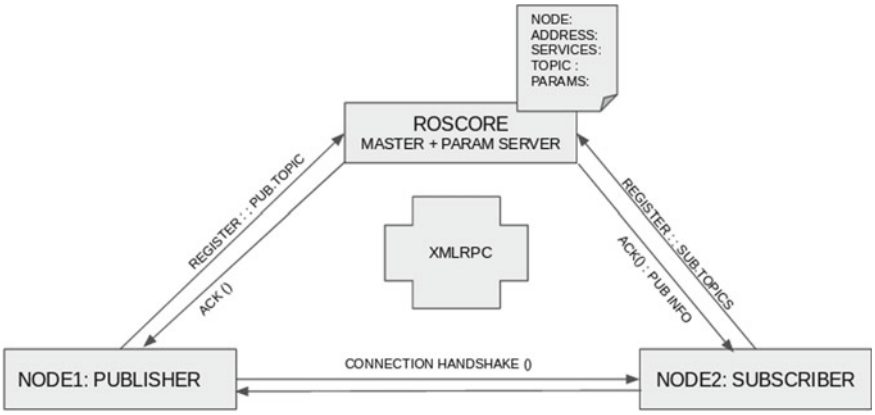


Fig. 2 The ROS software architecture in current state

tains the subscribed topic names and the list of URIs of the nodes that publish those topics. The master also sends the updated publisher list to the subscriber in case of a change. After receiving publisher updates, the subscriber node will connect to any new publishers by requesting a topic connection using the publisher's XMLRPC server. The nodes then communicate with each other on an agreed protocol (TCPROS or UDPROS).

Similarly, a publisher node establishes connection with the master first which saves the information about the advertised topics and relays it to the subscribers. Publisher then receives calls from the nodes looking for topic connections for subscription. The topics are transported using libraries called TCPROS or UDPROS depending on the protocol.

In case of services, a node providing that service registers with the master. A client node requesting that service contacts the master and looks up the information. It then makes a serialized request call and gets the response from the service. In case of services there is no callback on the master which informs the clients about any changes. Thus, the clients poll and wait for the service to appear.

In addition to the above mentioned components there is a parameter server, which is a shared global dictionary consisting of the configuration parameters. The parameter server runs inside the ROS master and nodes can access and modify these parameters via remote procedure calls to the ROS master.

Thus, we can see that the master is an important entity which facilitates the communication between various nodes and stores the configuration state of the runtime graph of a ROS network. Hence, it is crucial to keep the master up and running for system integrity.

3 Fault Tolerance in ROS

The ROS architecture described above is characterized by single point of failure with an inadvertent loss of the ROS master. The master is the central server which keeps the metadata about all the nodes. The metadata is the information regarding the publisher nodes and subscriber nodes, their URIs, services and configuration parameters in the parameter server. Every node communicates with the master on startup and registers itself. When the master fails this metadata is lost and the system is left with no way of connecting nodes. As it is a peer to peer system, nodes which are already connected still communicate. But, the problem arises when a new node comes up or goes down, or an existing node makes a service call; the nodes need to contact the master for metadata information in these cases. Without recovering the lost metadata nodes are forced to restart even after master comes back alive and the system is reinitialized.

Consider a scenario with a publisher and a subscriber node. As explained in the previous section, after they register with master they communicate with each other directly. Now if the master crashes the nodes still talk to each other over the topic. However they cannot make any new connections or publish/subscribe new topics. When the master recovers, it starts afresh and does not have the knowledge of the current system state. This leaves the nodes hanging and isolated from any other nodes that may come up after reinitialization.

At present there is no logging or checkpointing mechanism in ROS to record the master metadata so that we can recover the last uncorrupted state.

3.1 Importance of Fault Tolerance in Robotics

Fault tolerance is a crucial property to have when implementing real time applications, especially in the field of robotics where the systems are highly prone to failures. There can be physical failures due to long deployments of missions in challenging environments such as at disaster areas or construction sites and in harsh conditions such as is the case for robotics underwater, under-ice or in polar regions. There are additional issues such as battery limitations and power consumption, software failures including those associated with memory or processing overload. Moreover, in robotics the impact of a failure can have serious consequences including loss of the robot, property and lives. Due to the active and dynamic nature of these systems, the time to recovery is a critical factor. However, the time to recovery from failure depends on the type of robot, the underlying computing hardware and reaction time of sensors and/or actuators, all of which vary with the type of robot and its application area.

The main goal of this chapter is to provide an implementation of a fault tolerant mechanism in ROS [8], which is the most widely used software framework [3] for building robotics applications both in academia and industry in order to facilitate scalable and reliable robotic systems. In the following sections, we lay out our method including the logging and recovery solution, and its implementation.

4 ROS Rescue

We have designed, developed and tested our software which we call ROS Rescue, that deals with the two related problems of system failure detection and recovery. The fundamental idea is to equip the ROS master with a logging mechanism to save its state to persistent memory whenever there is a change to its metadata. When the master recovers from failure it can return to the previous working state by reading the log. The log contains the complete metadata information about various nodes including URIs, port numbers, published/subscribed topics and services, and the parameters in the parameter server. After recovering the working state, the master can resume its operation.

We developed our approach to be as simple and light weight as possible and at the same time keep it suitable for robotics applications.

Our solution is a simple feature addition to ROS master as opposed to running another multi-threaded application or having interceptor nodes and a replica master. We save only the meta-data as a log file as opposed to the application snapshot consuming less space. We log if and when a change occurs to the ROS master instead of periodically saving a checkpoint. And, we are not suspending the nodes when we are writing to persistent disk, in fact this can be parallely done using a writing thread for complete separation. Finally, we can recover the configuration by just restarting the master while all the other nodes are active unlike the checkpoint recovery using DMTCP.

The architectural changes to the original ROS framework can be visually depicted as shown below in Fig. 3. The important aspects of the fault tolerance mechanism are:

1. Failure detection by continuous monitoring of the ROS master.
2. Detecting changes in the network configuration and continuous update of meta information to persistent memory.
3. Recovery of the most recent state after master crash and updates to the state to reflect the network configuration in a timely manner.
4. The code simply provides added functionality, while all the key components, their functions, and APIs remain the same within ROS. There is no overhead associated with external libraries.
5. The software is lightweight and is a simple implementation when compared to previously proposed solutions.

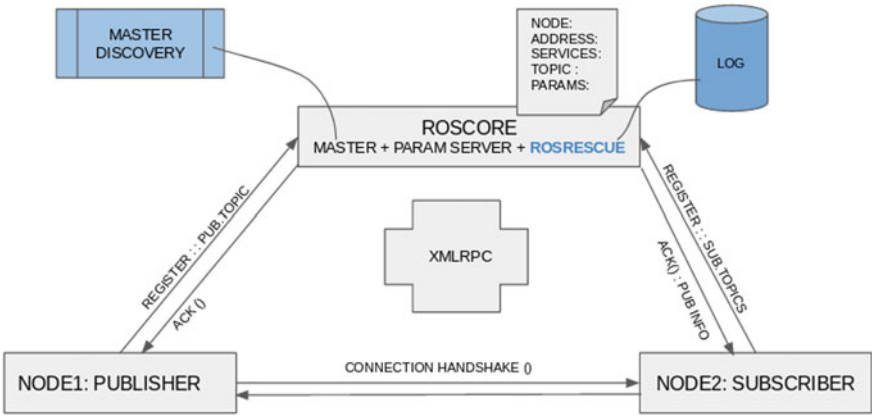


Fig. 3 ROS Software architecture with fault tolerance. In comparison to Fig. 2 note the addition of ROSRESCUE inside ROSCORE along with an associated logging mechanism

4.1 Implementation Details

The architecture of ROS is very similar to that of the Google File System (GFS) [5] where there is a single master which saves the meta information and chunk servers which host the data. When needed the clients and chunk-servers establish connections after contacting the master for look up information. The implementation of logging and recovery mechanism is inspired from GFS, but we record the latest state of the master as opposed to an operational log which needs to be replayed for recovery. In case of GFS an operational log is maintained not just for metadata but also to record the logical timeline of the concurrent operations on the data and maintain version information. In case of ROS, this is not necessary as it is not a database or filesystem but a framework of processes which only needs the information about process communication. The fault tolerance is implemented in two stages (Fig. 4).

Failure Detection: A master discovery node is implemented which keeps track of the availability of the master. It starts up with the master and detects master failure using a pull design by periodically polling the master node. Finding the master state is done by repeatedly making a XMLRPC function call to the master server. When it detects that the master has not responded for a specified number of polls it declares that the master has failed and issues command to restart the master.

Recovery: Master logging and recovery is implemented inside the ROS Rescue process which exists within the master and can be activated by passing a command line argument to the rosmaster startup script. Once this feature is enabled, ROS Rescue repeatedly logs the metadata information of the master onto the persistent disk whenever it changes. This change can happen when a new node registers/unregisters itself with the master, a node crashes or stops, when a new topic is published or subscribed to, or changes are made to the parameter server. When the master recovers from failure, it restores its last seen state by reading the log. It then checks to see

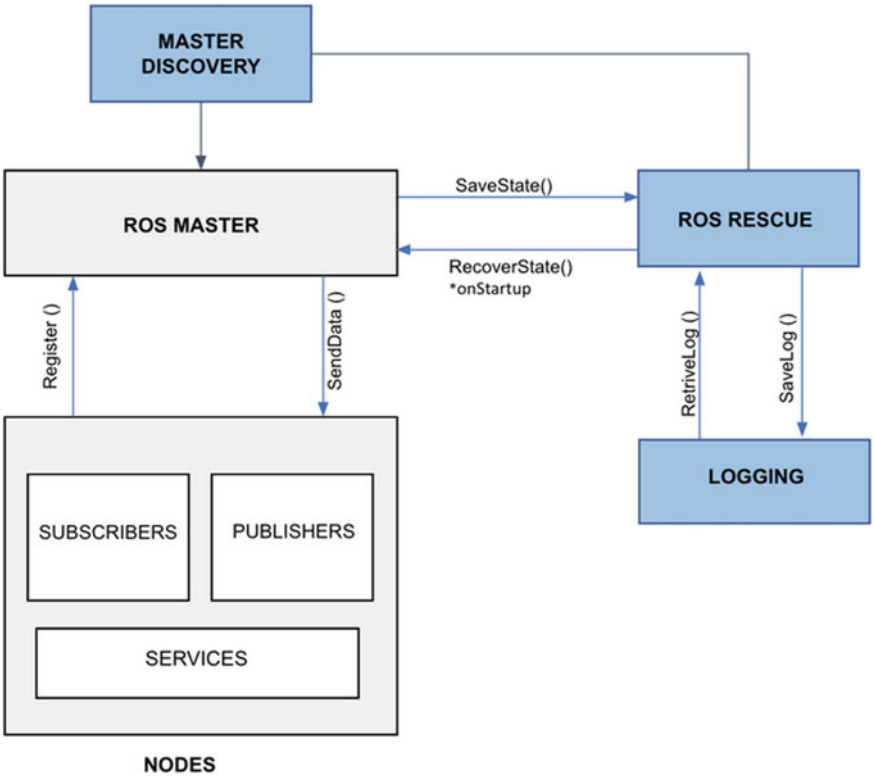


Fig. 4 Process flow diagram of the Fault tolerance mechanism

if there are any changes to the recovered state. For example, some nodes might have crashed while the master is down but their information will be present in the log. Hence, master goes through each node and makes sure that they exist and their metadata remains consistent. If there are any changes it updates its state and re-establishes connections with the nodes that are running.

4.2 Logging Metadata

As mentioned in the previous section the state of the ROS runtime graph is written to a log whenever there is a change. The ROS master maintains the information about publishers, subscribers and the available services in the form of dictionaries. The structure of the log that is written to disk is maintained in a manner similar to the way it is stored in the ROS master for easier recovery. The YAML format is a natural choice to save the metadata of ROS master and the written log has the structure shown in Fig. 5 and illustrated for a real process with all its complexity in Fig. 7.

```
Nodes :
        Node1 : URI
        Node2 : URI
        ...
Parameters :
        Param1 : Value 1
        Param2 : Value 2
Publishers:
        Node1 :
            Topic1, Type1
            Topic2, Type2
            ...
    ...
Subscribers:
        Node1 :
            Topic1, Type1
            Topic2, Type2
            ...
    ...
Services :
        Node1 :
            service1, service1 URI
            service2, service2 URI
            ...
    ...
```

Fig. 5 Log file structure. Our log file is in YAML format and contains all the metadata associated with the ROS master. In comparison to this simple illustration, we have also included the partial contents of an actual log file in Fig. 7

4.3 Testing

The following cases are considered for testing the implemented fault tolerance and recovery mechanism.

Case 0—Master failure and no configuration changes: Fault tolerance mechanism in this case is demonstrated by spawning three nodes- ROS master, a publisher node advertising a topic and a subscriber listening to the same topic. We can then abruptly crash the master and upon restart the master should be able to get back its previous state and ready to connect new nodes. We can check this by:

1. Inspecting the meta data and the configuration parameters
2. Spawning a new subscriber to the topic advertised by the publisher before the crash and checking on recovery, to ensure that the subscription was successful.

Case 1—Master failure and node configuration changes: This case is demonstrated similar to case 0, but now we crash one of the nodes while the master is down. Upon restart, the master should not only recover its last seen state, but also update the latest node configuration changes. This can be tested in a way similar to the last case.

Case 2—Master failure during the registration process: By default ROS deals with this issue as a node typically keeps trying periodically to see if the master is available during the registration process. However, there can be a situation where

the master crashes after it unregisters a node with itself, but before it can let other nodes know about the change. This case is currently not being handled in the code although we are working on the implementation. Our method would go through the topics published by the nodes and notify subscribers that are alive but are not privy to this information as observed in their registration data.

Case 3—Master failure while writing to persistent disk: If the master fails while it is saving the state it is important to save either the entire meta-data or none at all to avoid inconsistent state recovery. We have implemented a two stage commit for saving the system state. Whenever the master saves the state, it writes to a temporary checkpoint file which will be later renamed as the latest checkpoint upon successful write. That way, if the master dies/crashes while writing the state it will be in the temporary file and we can still recover the last fully saved state from the old checkpoint.

All the above mentioned cases have been tested using ROS Rescue and in each case we were able to recover the meta-data of the ROS runtime graph. In order to evaluate the effectiveness of our fault-tolerant approach we also performed quantitative analysis in terms of time taken for master recovery for a varying number of nodes and on different robotic systems. The first set of experiments was conducted on a Ubuntu 16.04, amd64 desktop where the ROS master was run with and without the rescue option. For each test, we spawned a set of n nodes (both publishers and subscribers), killed the master and brought it back up. We recorded the time taken for recovering the saved state from the log file. This test was performed multiple times for each value and the results averaged. The final averaged results are tabulated in Table 1. We can see that time to recovery changes linearly with the number of nodes. We also compare these results with a vanilla version of ROS (that is ROS without ROS Rescue compiled in).

We have successfully tested and have actively been using ROS rescue on multiple robotic platforms including the turtlebot3 [9], DJI matrice M100 [2] and a Lincoln MKZ autonomous car equipped with the drive-by-wire ADAS kit from Dataspeed Inc [1]. These robotic platforms use the Raspberry PI, Nvidia Tx2 and an Intel Xeon processor (running Ubuntu) respectively. We have tabulated the results of these experiments in Table 2. For each robot we include the average number of nodes while running, the average time taken for ROS master recovery and the start time for ROS with and without rescue option. We can see that the recovery time is different on different platforms, and unlike Table 2, is not linear. We note that this is to be expected as in these cases, the computational burden on the differing underlying computer hardware is different. However, in each case the time to recovery is well under a second which was our required figure of merit based on the dynamic constraints for these robotic systems.

Table 1 ROS Rescue experimental results showing the total time taken to start the master and time taken to recover the master in ROS with and without ROS Rescue

	Time to recovery (s)	Total time to start master (s)
Vanilla ROS	N/A	0.0579
ROS Rescue 1 node	0.00512	0.0638
ROS Rescue 5 nodes	0.0593	0.1173
ROS Rescue 10 nodes	0.112	0.175
ROS Rescue 20 nodes	0.218	0.276
ROS Rescue 40 nodes	0.436	0.499
ROS Rescue 80 nodes	0.9	0.97

Table 2 Experimental results on real robotic platforms—turtlebot3, DJI matrice M100 drone and autonomous car showing the average time taken to recover the master state and total time to start the master with and without ROS Rescue for a typical application. As explained in the text the results are not linear with the number of nodes due to different underlying computer hardware. However, full recovery is well within the sub second figure of merit

	Vanilla ROS (s)	Avg time to start master (s)	Avg time to recover (s)
Turtlebot3 (7∼10 nodes)	0.23	0.62	0.36
Drone (15–20 nodes)	0.12	0.45	0.35
Autonomous Car (20–25 nodes)	0.057	0.33	0.27

5 Step-by-Step Guide to ROS Rescue

This section provides the necessary guidelines for building, testing and implementing ROS Rescue in custom ROS applications.

5.1 System Setup

Ros Rescue is included as an additional functionality inside the rosmaster package for light-weight implementation. We have implemented the rescue feature in the bare bones source code of ROS which is available on github at the repository shown below.

https://github.com/PushyamiKaveti/fault-tolerant-ros-master.git

We need to prepare the system to execute roscore with rescue feature enabled, as a first step we make sure to have the prerequisites. Open the terminal and execute the following commands which take cares of Python dependencies, initialize the ROS dependencies and update the ROS dependency packages.

```
$ sudo apt-get install python-rosdep python-rosinstall
    -generator python-wstool python-rosinstall build-
    essential
$ sudo rosdep init
$ rosdep update
```

To use ROS Rescue the code needs to be built from source. Open terminal and navigate to the folder of your choice and clone the git repository mentioned above

```
$ git clone https://github.com/PushyamiKaveti/fault-
    tolerant-ros-master.git
```

Now, issue the following commands in the terminal. This will build the code locally in the current folder and run ROS with the fault tolerant feature enabled.

```
$ cd fault-tolerant-ros-master
$ ./src/catkin/bin/catkin_make_isolated --install -
    DCMAKE_BUILD_TYPE=Release
```

Open the ros_env_setup.sh file present in the fault-tolerant-ros-master folder in a text editor. Find and replace all <your_path_folder> with the your local path. For example, if you have cloned our repository in home folder then the <your_path_folder> will be replaced as "~/fault-tolerant-ros-master" the absolute path of the file location.

Also, the environment setup file has the list of environment variables which are required for execution. Sourcing the file will keep the current shell aware of the environment variables. If you want to utilize the ROS for other users in the system, place the copy of the file in "/etc/profile.d/ros_env_setup.sh". This assumes you are running ROS on a linux/unix platform. This will facilitate loading the file globally for all users in the system during login/shell spawning.

```
$ source ros_env_setup.sh
```

The system setup instructions are needed to build ROS Rescue from source for immediate use. However, we plan to limit the ROS Rescue functionality only to rosmaster package and make a pull request to ROS's official github repository. Once approved the Rescue functionality will be available in official ROS packages and one can skip to Sect. 5.2 for running ROS with rescue option.

5.2 *Running ROS Rescue*

Once the environment is setup as outlined above we can run roscore with the rescue feature enabled. ROS master with our rescue option will function and operate as

```
started roslaunch server http://pushyamik-u:36779/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [2854]
Set TOPIC MANAGER
Ros Rescue enabled. Master is now fault tolerant!!!
ROSRescue ENABLED
ROS_MASTER_URI=http://pushyamik-u:11311/
```

Fig. 6 Console output of roscore with fault tolerance enabled

normal with the additional capabilities associated with logging and rescue. The only change is to initiate the roscore with "–rescue" option as shown below. Before running roscore it is advisable to ensure that the correct version of roscore is being called by issuing the "which roscore" command in the terminal.

```
$ roscore --rescue
```

By default logs are saved in "~/.ros/log/latest-chkpt.yaml" with the system default's umask value which enables everyone to read the logs. To keep the logs locked or to avoid manual intervention you can set the file permission to '655'. Owner can read and write, others can only read and execute by "chmod 655 ~/.ros/log/latest-chkpt.yaml". This will help us keep the master state secure and stable.

If the ROS master is running with fault tolerance enabled you should see "ROS Rescue enabled. Master is now fault tolerant!!" printed to the console where roscore –rescue was executed as shown in the Fig. 6. Another check that can be done is to make sure the master state is being saved to latest-ckhpt.yaml file mentioned above.

The ROS Rescue package also comes with a master discovery node to detect master failure. Master monitoring is required for failure detection and master restart. This node is made a part of roscore.xml file alongside rosout node and is launched when roscore command is issued. We can also run it separately by issuing the following commands.

```
$ cd src/ros_comm/rosmaster/scripts/
$ ./master_monitor --rescue
```

```yaml
NODES:
  /listener_3458_1559421274831: !!python/object:rosmaster.registrations.NodeRef
    api: http://pushyamik-u:39427/
    id: /listener_3458_1559421274831
    param_subscriptions: []
    services:
    - /listener_3458_1559421274831/get_loggers
    - /listener_3458_1559421274831/set_logger_level
    topic_publications:
    - /rosout
    topic_subscriptions:
    - /chatter
  /rosout: !!python/object:rosmaster.registrations.NodeRef
    api: http://pushyamik-u:42791/
    id: /rosout
    param_subscriptions: []
    services:
    - /rosout/get_loggers
    - /rosout/set_logger_level
    topic_publications:
    - /rosout_agg
    topic_subscriptions:
    - /rosout
  /talker_3409_1559421265672: !!python/object:rosmaster.registrations.NodeRef
    api: http://pushyamik-u:37353/
    id: /talker_3409_1559421265672
    param_subscriptions: []
    services:
    - /talker_3409_1559421265672/get_loggers
    - /talker_3409_1559421265672/set_logger_level
    topic_publications:
    - /chatter
    - /rosout
    topic_subscriptions: []
---
PUBLISHERS:
  /chatter:
  - !!python/tuple
    - /talker_3409_1559421265672
    - http://pushyamik-u:37353/
  /rosout:
  - !!python/tuple
    - /talker_3409_1559421265672
    - http://pushyamik-u:37353/
  - !!python/tuple
    - /listener_3458_1559421274831
    - http://pushyamik-u:39427/
  /rosout_agg:
  - !!python/tuple
    - /rosout
    - http://pushyamik-u:42791/
---
```


Fig. 7 Screenshot of an actual run showing the latest-chkpt.yaml created with fault tolerance enabled

To make sure ROS Rescue logging is working correctly we can run a simple publisher/subscribe test as shown below.

```
$ cd src/node_test/scripts/
$ python talker.py
$ python listener.py
```

The talker node publishes a dummy string message to a topic called '/chatter'. The listener node subscribes to that string message on topic "/chatter". If everything is working well the messages published by publisher in its console should relay on subscriber console. The log file should now contain the information about the nodes, publishers, subscribers and services available. Figure 7 below shows a small portion of a log file that was created with ROS Rescue enabled when talker and listener nodes were running.

6 Conclusion and Future Work

As part of this project, a failure detection and recovery mechanism has been implemented for the Robot Operating System 1.0 (ROS). The implemented code has been tested successfully on custom publisher and subscriber nodes as well as on a real robot systems. We also conducted tests and used ROS Rescue on a Turtlebot3, an autonomous car and on drones. The logging mechanism has implemented for publishers, subscribers, services and the parameter server. We have made our code and demo videos available on the project's Github repository for use by researchers and developers in the ROS community as they see fit. As mentioned earlier we plan to limit ROS Rescue functionality to rosmaster package and make a pull request to the official ROS repository for integration and porting across multiple versions. As an extension to this work we are in the process of conducting testing to cover cases of highly dynamic and scalable environments, including a multi-robot swarm system.

References

1. Dataspeed inc drive-by-wire adas kit, https://bitbucket.org/DataspeedInc/dbw_mkz_simulation/src/default
2. Dji matrice m100 quadcopter for developers, https://www.dji.com/matrice100
3. The rise of the robot operating system, https://roboticsandautomationnews.com/2019/05/16/the-rise-of-the-robot-operating-system/22485/
4. J. Ansel, K. Arya, G. Cooperman, DMTCP: transparent checkpointing for cluster computations and the desktop, in *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium* (2009). https://doi.org/10.1109/IPDPS.2009.5161063
5. S. Ghemawat, H. Gobioff, S.T. Leung, The google file system. Oper. Syst. Rev. (ACM) (2003). https://doi.org/10.1145/1165389.945450
6. T. Jain, G. Cooperman, Dmtcp: fixing the single point of failure of the ros master (2017)

7. M. Lauer, M. Amy, J.C. Fabre, M. Roy, W. Excoffon, M. Stoicescu, Resilient computing on ros using adaptive fault tolerance. J. Softw.: Evol. Process. **30**(3), e1917 (2018)
8. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, ROS: an open-source Robot Operating System, in *ICRA Workshop on Open Source Software* (2009)
9. P. Yoonseok, J. Leon, Turtlebot3 (2016)

Pushyami Kaveti is now a Ph.D. student at the Khoury College of Computer Sciences at Northeastern University. She received her M.S. degree in Computer Science in 2014 at University of Florida, and B. Tech degree in Computer Science and Engineering from JNTU, India in 2011. Her research interests lie at the intersection of robotics, computer vision and machine learning. Her work at Field Robotics Laboratory focuses on robot perception and navigation in real world and dynamic environments.

Hanumant Singh is professor in the department of ECE at Northeastern University and the Director of the Center for Robotics there. He received his Ph.D. from the MIT/WHOI Joint Program in 1995 after which he worked on the Staff at WHOI until 2016 when he joined Northeastern. Prior to that, he graduated with dual degrees in Computer Science and Electrical and Computer Engineering from George Mason University. While at WHOI, his group designed and built the Seabed AUV, as well as the Jetyak Autonomous Surface Vehicle, dozens of which are in use for scientific and other purposes across the globe. His interests lie in the area of Marine, Land and Aerial robotics especially as they relate to sensing, imaging and navigation.